Abstract. This chapter introduces *Pushdown Automata*, and presents their basic theory. The two language families defined by pda (acceptance by *final state*, resp. by *empty stack*) are shown to be equal. The pushdown languages are shown to be equal to the *context-free* languages. *Closure properties* of the context-free languages are given that can be obtained using this characterization. *Determinism* is considered, and it is formally shown that deterministic pda are less powerfull than the general class.

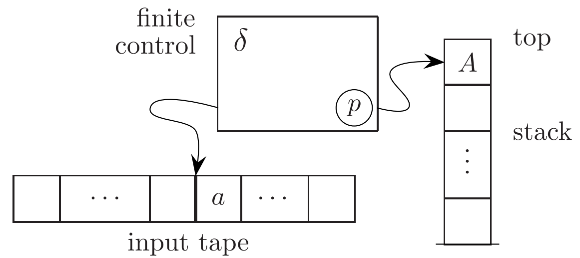


Figure 1: Pushdown automaton

Pushdown Automata

A pushdown automaton is like a finite state automaton that has a pushdown stack as additional memory. Thus, it is a finite state device equipped with a one-way input tape and a 'last-in first-out' external memory that can hold unbounded amounts of information; each individual stack element contains finite information. The usual stack access is respected: the automaton is able to see (*test*) only the topmost symbol of the stack and act based on its value, it is able to *pop* symbols from the top of the stack, and to *push* symbols onto the top of the stack.

Denoting the reversal of a string w by w^R, the language $L = \{ \, w\,c\,w^R \mid w \in \{a,b\}^* \, \}$ consists of palindromes, the middle of which is marked by the special symbol c. Using the pumping lemma for regular languages it is easily shown this language is not regular. However, it can be recognized by a pushdown automaton as follows. Starting in the initial state the pda reads the letters from the input, copying them to the stack, until the special input marker c is encountered. Then it switches to the next state, where the tape is read, and its letters are matched with the stack, popping a stack symbol for each letter read. When topmost stack symbol and letter on the input do not match, the input does not belong to the languages, and the automaton blocks, rejecting the input. Otherwise, it reaches the end of the input, which is then accepted only if the stack is empty.

[0]http://www.liacs.nl/home/hoogeboo/praatjes/tarragona/ Based on: Pushdown Automata, Hendrik Jan Hoogeboom and Joost Engelfriet. Chapter 6 in: Formal Languages and Applications (C. Martín-Vide, V. Mitrana, G. Paun, eds.), Studies in Fuzziness and Soft Computing, v. 148, Springer, Berlin, 117-138, 2004.

1 Formal Definition

The formal specification of a pushdown automaton extends that of a finite state automaton by adding an alphabet of stack symbols. For technical reasons the initial stack is not empty, but it contains a single symbol which is also specified. Finally the transition relation must also account for stack inspection and stack operations and is changed accordingly.

Definition. A *pushdown automaton*, *pda* for short, is specified as a 7-tuple $\mathcal{A} = (Q, \Delta, \Gamma, \delta, q_{in}, A_{in}, F)$ where

- Q is a finite set (of *states*),
- Δ an alphabet (of *input symbols*),
- Γ an alphabet (of *stack symbols*),
- δ a finite subset of $Q \times (\Delta \cup \{\lambda\}) \times \Gamma \times Q \times \Gamma^*$, the *transition relation*,
- $q_{in} \in Q$ the *initial state*,
- $A_{in} \in \Gamma$ the *initial stack symbol*, and
- $F \subseteq Q$ the set of *final states*.

An element (p, a, A, q, α) of δ is called an *instruction* (or transition) of $\mathcal{A}$. If a is the empty string it is a λ-instruction.

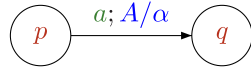
The instruction (p, a, A, q, α) of the pda is valid in state p, with a next on the input tape, and A as topmost symbol of the stack (as in Figure 1 for $a \in \Delta$), and it specifies a change of state from p into q, reading a from the input, popping A off the stack,

and pushing α onto it. When one wants to distinguish between the pre-conditions of an instruction and its post-conditions, δ can be considered as a function from $Q \times (\Delta \cup \{\lambda\}) \times \Gamma$ to finite subsets of $Q \times \Gamma^*$, and one writes, e.g., $(q, \alpha) \in \delta(p, a, A)$.

A transition may read λ from the input, but it always pops a specific symbol A from the stack. Pushing a string α to the stack regardless of its current topmost symbol has to be achieved by introducing a set of instructions, each popping a symbol $A \in \Gamma$ and pushing αA. In particular, when $\alpha = \lambda$ we have a set of instructions that effectively ignores the stack by popping the topmost symbol and pushing it back.

Convention. We introduce our personal pictorial representation for pda, including shortcuts for actions 'push A' and 'read a' that do not touch the stack.

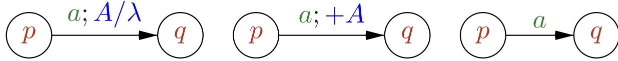
The general instruction (p, a, A, q, α) is depicted by an arrow from node p to node q labelled by the input read (like for finite automata) and by the operation on the stack (replace topmost A by α).



There are three intuitive actions we consider, some of which have to be encoded using several instructions. We list them, and their representations.

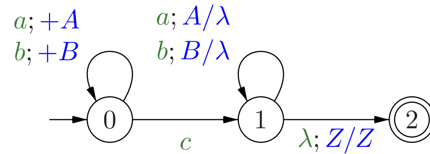
intuitive	*formalized as*	
pop A	(p, a, A, q, λ)	$\alpha = \lambda$
push A	(p, a, X, q, AX)	for all $X \in \Gamma$
read a	(p, a, X, q, X)	for all $X \in \Gamma$

These three operations are depicted as follows. In this way a single arrow may represent a set of related instructions.



Example 1 The pushdown automaton $\mathcal{A} = (Q, \Delta, \Gamma, \delta, q_{in}, Z_{in}, F)$ is given in the figure below. The intended meaning is that in initial state 0 symbols a and b are read and pushed onto the stack (as A and B). When a c is encountered in the input regardless of the stack contents the pda moves to state 1. In that state symbols of the input are matched against the topmost stack symbol, which is popped as we read. When the original bottom-of-stack symbol is reached, the pda can move to final state 2 without reading input.

In an explicit specification we have $Q = \{0, 1, 2\}$, $\Delta = \{a, b, c\}$, $\Gamma = \{A, B, Z\}$, $q_{in} = 0$, $Z_{in} = Z$, and $F = \{2\}$.



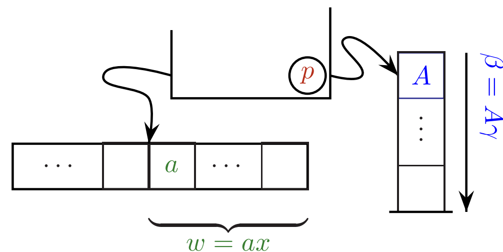
Written out in full, δ consists of the instructions $(0, a, Z, 0, AZ)$, $(0, a, A, 0, AA)$, $(0, a, B, 0, AB)$, $(0, b, Z, 0, BZ)$, $(0, b, A, 0, BA)$, $(0, b, B, 0, BB)$, $(0, c, Z, 1, Z)$, $(0, c, A, 1, A)$, $(0, c, B, 1, B)$, $(1, a, A, 1, \lambda)$, $(1, b, B, 1, \lambda)$, and $(1, \lambda, Z, 2, Z)$

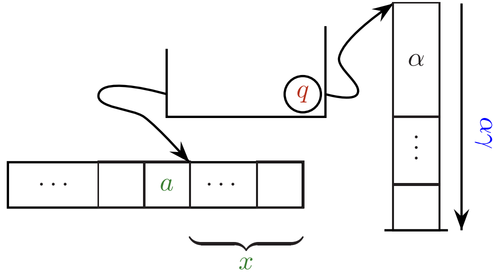
Computations

A *configuration* (w, p, β) of a pda $\mathcal{A}$ is given by the contents w of the part of the input tape that has not been read, the current state p, and the current contents β of the stack. Hence the set of configurations of $\mathcal{A}$ is the set $\Delta^* \times Q \times \Gamma^*$. In denoting the stack as a string of stack symbols we assume that the topmost symbol is written first.

According to the intuitive semantics we have given above, δ defines a *step relation* $\vdash_{\mathcal{A}}$ on the set of configurations, modelling a single step of the pda:

$(ax, p, A\gamma) \vdash_{\mathcal{A}} (x, q, \alpha\gamma)$ iff
$(p, a, A, q, \alpha) \in \delta$, $x \in \Delta^*$, and $\gamma \in \Gamma^*$.





As a consequence of the definitions, a pda cannot make any further steps on an empty stack, as each instruction specifies a stack symbol to be removed.

A *computation* of the pda is a sequence of consecutive steps; the *computation relation* is the reflexive and transitive closure $\vdash_{\mathcal{A}}^*$ of the step relation. That is, $c \vdash_{\mathcal{A}}^* c'$ for two configurations c, c' if there is a sequence $c_0, c_1, \ldots, c_n$ such that $c_0 = c$, $c_i \vdash_{\mathcal{A}} c_{i+1}$ $(0 \leq i < n)$, and $c_n = c'$.

Example 2 A configuration for the pda from the previous example is for instance the tuple $(abbcabc, 1, AAZB)$, where $abbcabc$ is the part of the input not yet read, 1 is the state, and $AAZB$ is the stack (read from top A to bottom B).

For this pda we have the step $(aabcbaa, 0, Z) \vdash_{\mathcal{A}} (abcbaa, 0, AZ)$ which applies the instruction $(0, a, Z, 0, AZ)$.

The following sequence is a computation which starts with this step $(aabcbaa, 0, Z) \vdash_{\mathcal{A}} (abcbaa, 0, AZ) \vdash_{\mathcal{A}} (bcbaa, 0, AAZ) \vdash_{\mathcal{A}} (cbaa, 0, BAAZ) \vdash_{\mathcal{A}} (baa, 1, BAAZ) \vdash_{\mathcal{A}} (aa, 1, AAZ) \vdash_{\mathcal{A}} (a, 1, AZ) \vdash_{\mathcal{A}} (\lambda, 1, Z) \vdash_{\mathcal{A}} (\lambda, 2, Z)$

The computation starts with $aabcbaa$ on the input, in state 0 en with a stack consisting of the single symbol Z. Then the symbols aab are read from the input, and represented on the stack. The c is read and the automaton moves to state 1. In that state baa is read from the input and the symbols are compared to the contents of the stack, popping a stacksymbol each step. Finally the automaton moves to final state 2 not reading any input. This step is possible as the topmost stack symbol is now Z.

There are two natural ways of defining the language of a pda, basing the acceptance condition either on internal memory (the states) or on the external memory (the stack). A pda accepts its input if it has a computation starting in the initial state with the initial stack symbol on the stack, completely reading its input, and either (1) ending in a final state, or (2) ending with the empty stack. In general, for a fixed pda, these languages are not equal. Note that in the latter case the final states are irrelevant.

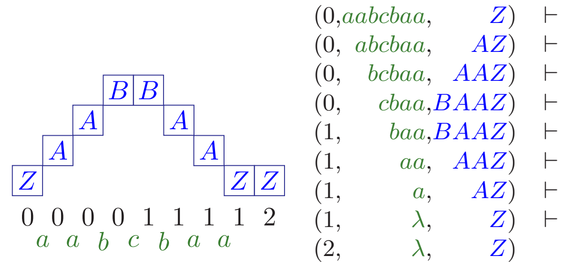
Definition. Let $\mathcal{A} = (Q, \Delta, \Gamma, \delta, q_{in}, A_{in}, F)$ be a pda.

1. The *final state language* of $\mathcal{A}$ is
   $L(\mathcal{A}) = \{\ x \in \Delta^* \mid (x, q_{in}, A_{in}) \vdash_{\mathcal{A}}^* (\lambda, q, \gamma)$ for some $q \in F$ and $\gamma \in \Gamma^*\ \}$.
2. The *empty stack language* of $\mathcal{A}$ is
   $N(\mathcal{A}) = \{\ x \in \Delta^* \mid (\ x, q_{in}, A_{in}) \vdash_{\mathcal{A}}^* (\lambda, q, \lambda)$ for some $q \in Q\ \}$.

We stress that we only accept input if it has been completely read by the pda, however, the pda cannot recognize the end of its input (and react accordingly). This is especially important in the context of determinism (see Section 5). We can remedy this by explicitly providing the input tape with an end marker \$, recognizing a language $L\$$ rather than L.

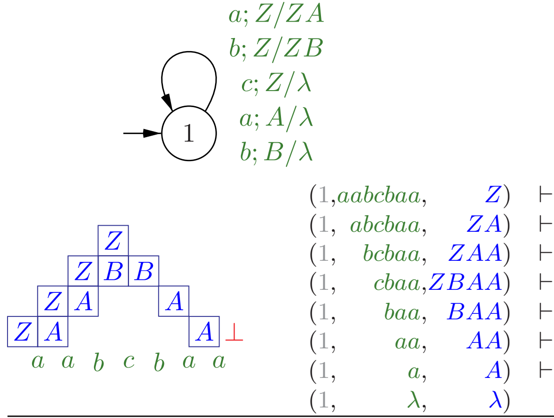
We also stress that, due to the presence of λ-instructions, a pda can have infinite computations. Thus, it is not obvious that the membership problem '$x \in L(\mathcal{A})$?' is decidable. This will follow from Theorem 8.

Example 3 The automaton from the example accepts the string $aabcbaa$ using the computation given before. To the left is a pictorial representation, the successive stacks as columns. Below the state sequence, and the letters read from the input. To the right the corresponding computation is repeated as a sequence of consecutive steps.



As a matter of fact $L(\mathcal{A}) = \{ \, w\,c\,w^R \mid w \in \{a,b\}^* \, \}$, while $N(\mathcal{A}) = \varnothing$, as the bottom stack symbol is never popped.

The same language can be accepted using empty stack by a pda that has only a single state and only five instructions.



The language accepted in the previous examples, palindromes with a middle marker, are easily seen to be context-free. The linear grammar with single nonterminal S and productions $S \to aSa$, $S \to bSb$, $S \to c$ generates the same language.

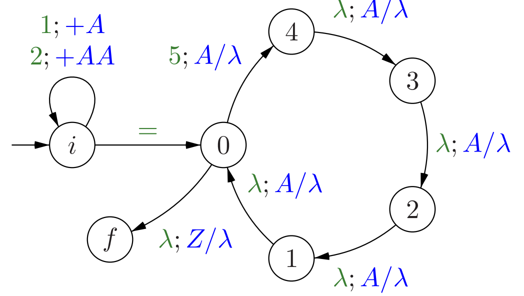
In the next example the stack is programmed to count. One can also design a context-free grammar for the language of this example, but this is more complicated than the simple approach of the pda.

Example 4 Consider the exchange language L_{ex} for the small euro coins, which has the alphabet $\Delta = \{\mathtt{1}, \mathtt{2}, \mathtt{5}, =\}$:

$$\{ \, x{=}y \mid x \in \{\mathtt{1},\mathtt{2}\}^*, y \in \{\mathtt{5}\}^*, |x|_{\mathtt{1}} + 2 \cdot |x|_{\mathtt{2}} {=} 5 \cdot |y|_{\mathtt{5}} \, \},$$

where $|z|_a$ denotes the number of occurrences of a in z. For instance, the language contains $\mathtt{12(122)^3 11 = 5555}$. It is accepted using empty stack acceptance by the pda $\mathcal{A}$ with states $Q = \{i, 0, 1, 2, 3, 4, f\}$, initial state i, input alphabet Δ, stack alphabet $\Gamma = \{Z, A\}$, initial stack symbol Z, and the following instructions:

- *pushing the value of* $\mathtt{1}$ *and* $\mathtt{2}$ *on the stack*: $(i, \mathtt{1}, Z, i, AZ)$, $(i, \mathtt{1}, A, i, AA)$, $(i, \mathtt{2}, Z, i, AAZ)$, $(i, \mathtt{2}, A, i, AAA)$;
- *reading the marker*: $(i, =, Z, 0, Z)$, $(i, =, A, 0, A)$;
- *popping* $\mathtt{5}$ *cents from the stack*: $(0, \mathtt{5}, A, 4, \lambda)$, and $(k, \lambda, A, k-1, \lambda)$ for $k = 4, 3, 2, 1$;
- *emptying the stack*: $(0, \lambda, Z, f, \lambda)$.

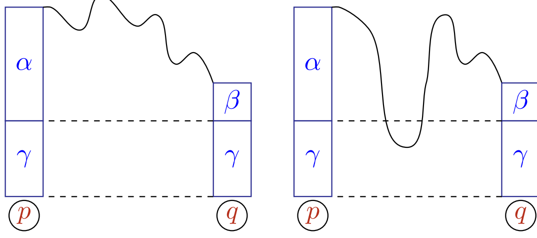


While reading $\mathtt{1}$'s and $\mathtt{2}$'s the automaton pushes one or two A's onto the stack to represent the value of the input. We have to provide two instructions for each of the two input symbols as the topmost stack symbol may be A or Z (when no input has been read). When reading $\mathtt{5}$ a total of five A's is removed in a sequence of five consecutive pop instructions. The stack can only be emptied when the value represented on the stack is zero (there are no A's left) and when we are in state 0 (we are finished processing the input symbol $\mathtt{5}$). Thus, $N(\mathcal{A}) = L_{ex}$.

A technical lemma. Context-free grammars owe their name to the property that derivations can be cut-and-pasted from one context into another. For pushdown automata, the part of the input that is not consumed during a computation, as well as the part of the stack that is not touched, can be omitted without effecting the other components of the computation. This leads to a technical result that is of basic use in composing pda computations.

Lemma 5 *Let* $\mathcal{A} = (Q, \Delta, \Gamma, \delta, q_{in}, A_{in}, F)$ *be a pda. If* $(x, p, \alpha) \vdash_{\mathcal{A}}^* (\lambda, q, \beta)$ *then* $(xz, p, \alpha\gamma) \vdash_{\mathcal{A}}^* (z, q, \beta\gamma)$, *for all* $p, q \in Q$, *all* $x, z \in \Delta^*$, *and all* $\alpha, \beta, \gamma \in \Gamma^*$. *The reverse implication is also valid, provided every stack of the given computation (except possibly the last) is of the form* $\mu\gamma$ *with* $\mu \in \Gamma^*$, $\mu \neq \lambda$.

The following picture illustrates the stack of the computation. At the right we see why the extra condition for the reverse implication is necessary.



2 Two Families

We have given two ways of defining a language for a pda. For each individual pda these may be two different languages. Over all pda, the two families of final state languages and of empty stack languages are the same.

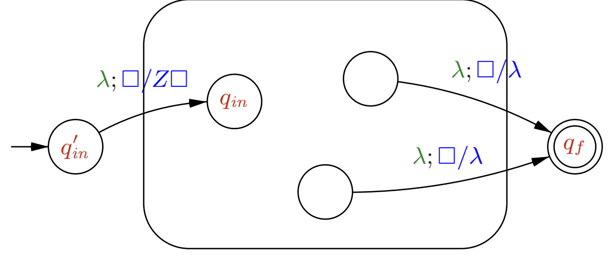
Note that in our first example the pda recognizes the moment when it reaches the bottom of its stack. This is achieved by reserving a special symbol Z that takes the bottom position of the stack, i.e., during the computation the stack has the form $\Gamma_1^* Z$ with $Z \notin \Gamma_1$.

This trick is also the main ingredient in the proof of the following result stating that final state and empty stack acceptance are equivalent. By putting a special marker at the bottom of the stack (in the first step of the computation) we can recognize the empty stack and jump to a final state (when transforming empty stack acceptance into final state acceptance) and we can avoid reaching the empty stack before the input has been accepted by final state (when transforming final state acceptance into empty stack acceptance).

Lemma 6 *Given a pda $\mathcal{A}$ one can effectively construct a pda $\mathcal{A}'$ such that $N(\mathcal{A}) = L(\mathcal{A}')$, and vice versa.*

Proof. (i) Given a pda $\mathcal{A}$ one can effectively construct a pda $\mathcal{A}'$ such that $N(\mathcal{A}) = L(\mathcal{A}')$.

Intuitively the new automaton moves to a final state when the original has emptied the stack. We add a new symbol at the bottom of the stack to make this move possible.



Let $\mathcal{A} = (Q, \Delta, \Gamma, \delta, q_{in}, Z_{in}, F)$ be a pda. We adapt it and constuct the new pda $\mathcal{A}' = (Q \cup \{q'_{in}, q_f\}, \Delta, \Gamma \cup \{\Box\}, \delta', q'_{in}, \Box, \{q_f\})$, where q'_{in} and q_f are two new states, and $\Box$ is a new pushdown symbol.

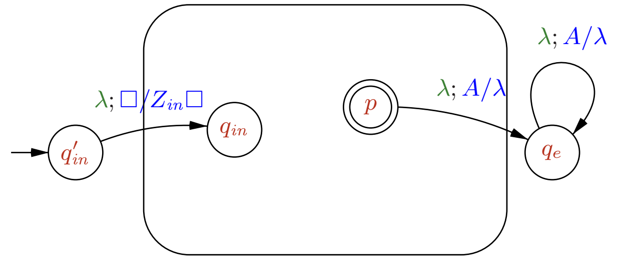
The transition relation δ' equals the original δ except that we add new transitions:

- $(q'_{in}, \lambda, \Box, q_{in}, Z_{in}\Box)$,
- $(q, \lambda, \Box, q_f, \lambda)$, for all $q \in Q$.

If $(w, q_{in}, Z_{in}) \vdash^* (\lambda, w, \lambda)$ is a computation of $\mathcal{A}$ (accepting w by empty stack) then it also is a computation for $\mathcal{A}'$, and remains a computation by adding $\Box$ at the bottom of the stack. This can be extended to the computation $(w, q'_{in}, \Box) \vdash (w, q_{in}, Z_{in}\Box) \vdash^* (\lambda, q, \Box) \vdash (\lambda, q_f, \lambda)$, now acepting w by final state for $\mathcal{A}'$. This also works the other way. Any computation of $\mathcal{A}'$ can be split into the initial step $(w, q'_{in}, \Box) \vdash (w, q_{in}, Z_{in}\Box)$, a computation leaving only $\Box$ on the stack $(w, q_{in}, Z_{in}\Box) \vdash^* (\lambda, q, \Box)$ and a final step to the new final state $(\lambda, q, \Box) \vdash (\lambda, q_f, \lambda)$. The midle part is also valid for $\mathcal{A}$, even when $\Box$ is removed from the stack as $\Box$ came never on top (there are no instructions for $\Box$ other than moving to q_f), see Lemma 5.

(ii) Given a pda $\mathcal{A}$ one can effectively construct a pda $\mathcal{A}'$ such that $L(\mathcal{A}) = N(\mathcal{A}')$.

Intuitively the new automaton moves (nondeterministically) to a new special state when the original has reached a final state (assuming it has read all its input). In the new state the stack is emptied. We add a new symbol at the bottom of the stack to avoid emptying the stack when we do not want to accept.



Let $\mathcal{A} = (Q, \Delta, \Gamma, \delta, q_{in}, Z_{in}, F)$ be a pda. We adapt it and constuct the new pda $\mathcal{A}' = (Q \cup \{q'_{in}, q_e\}, \Delta, \Gamma \cup \{\Box\}, \delta', q'_{in}, \Box, \{q_f\})$, where q'_{in} and q_e are two new states, and $\Box$ is a new pushdown symbol.

The transition relation δ' equals the original δ except that we add new transitions:

- $(q'_{in}, \lambda, \Box, q_{in}, Z_{in}\Box)$,
- (q, λ, A, q_e, A), for all $q \in F$, $A \in \Gamma \cup \{\Box\}$,
- $(q_e, \lambda, A, q_e, A)$, for all $A \in \Gamma \cup \{\Box\}$.

The relation between computations of $\mathcal{A}$ and $\mathcal{A}'$ is similar to that for the other implication. A word w is accepted by $\mathcal{A}$ using final state iff it is accepted by $\mathcal{A}'$ using empty stack. $\Box$

3 Context-Free Languages

Each context-free grammar *generating* a language can easily be transformed into a pda *recognizing* the same language. Given the context-free grammar $G = (N, T, S, P)$ we define the single state pda $\mathcal{A} = (\{q\}, T, N \cup T, \delta, q, S, \varnothing)$, where δ contains the following instructions:

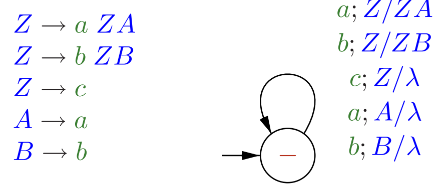
- $(q, \lambda, A, q, \alpha)$ for each $A \to \alpha \in P$ 'expand'
- (q, a, a, q, λ) for each $a \in T$ 'match'

The computations of $\mathcal{A}$ correspond to the *leftmost* derivations $\Rightarrow^*_{G,\ell}$ of G; the sequence of unprocessed nonterminals is stored on the stack (with intermediate terminals). Formally, for $x \in T^*$ and $\alpha \in (N \cup T)^*$, if $(x, q, S) \vdash^*_{\mathcal{A}} (\lambda, q, \alpha)$ then $S \Rightarrow^*_{G,\ell} x\alpha$. The reverse implication is valid for $\alpha \in N(N \cup T)^* \cup \{\lambda\}$. This correspondence is easily proved by induction, using Lemma 5.

By taking here $\alpha = \lambda$, we find that $S \Rightarrow^*_{G,\ell} x$ iff $(x, q, S) \vdash^*_{\mathcal{A}} (\lambda, q, \lambda)$, for all $x \in T^*$. This means that $L(G) = N(\mathcal{A})$, as leftmost derivations suffice in generating the language of a context-free grammar.

If the given context-free grammar G has only productions of the form $A \to a\alpha$ with $a \in T \cup \{\lambda\}$ and $\alpha \in N^*$ —this is satisfied both by grammars in Chomsky normal form and by those in Greibach normal form— then the construction is even more direct, as we can combine an expand instruction with its successive match instruction. The pda, with stack alphabet N, is constructed by introducing for each production $A \to a\alpha$ the instruction (q, a, A, q, α).

Example 7 Consider a context-free grammar and the single state pda constructed as indicated above.



The correspondence between (leftmost) derivations of the grammar and computations of the pda is clear from the diagram below. The tail consisting of nonterminals in the sentential forms matches the stack of the pda in each step.

	S		$(-, abbcbba, S)$
$\Rightarrow$	$a\,SA$	$\vdash$	$(-, bbcbba, SA)$
$\Rightarrow$	$ab\,SBA$	$\vdash$	$(-, bcbba, SBA)$
$\Rightarrow$	$abb\,SBBA$	$\vdash$	$(-, cbba, SBBA)$
$\Rightarrow$	$abbc\,BBA$	$\vdash$	$(-, bba, BBA)$
$\Rightarrow$	$abbcb\,BA$	$\vdash$	$(-, ba, BA)$
$\Rightarrow$	$abbcbb\,A$	$\vdash$	$(-, a, A)$
$\Rightarrow$	$abbcbba$	$\vdash$	$(-, \lambda, \lambda)$

This correspondence shows the equivalence of single state pda's (under empty stack acceptance) and context-free grammars. Keeping track of the states during the derivations requires some additional effort. The full equivalence of context-free grammars and pda's is the central result in the theory of context-free languages; it is attributed to Chomsky, Evey, and Schützenberger [4, 6, 16].

Theorem 8 *A language is recognized by a pda, either by final state or by empty stack, iff it is context-free.*

Proof. It suffices to demonstrate that every language recognized by a pda using empty stack acceptance is context-free.

Let $\mathcal{A} = (Q, \Delta, \Gamma, \delta, q_{in}, A_{in}, F)$ be a pda. A computation $(xz, p, A\gamma) \vdash^*_{\mathcal{A}} (z, q, \gamma)$ of $\mathcal{A}$ is said to be of *type* $[p, A, q]$ if the symbols from γ are not replaced during the computation, i.e., each of the intermediate stacks is of the

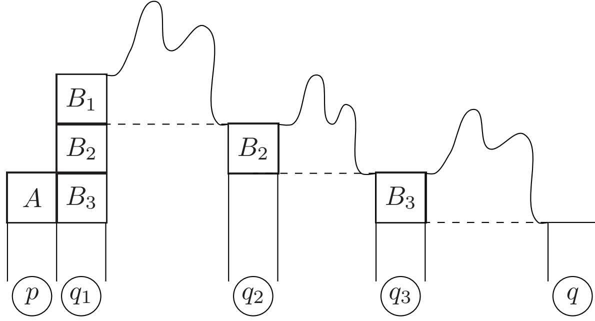


Figure 2: Computation of type $[p, A, q]$

form $\mu\gamma$ with $\mu \in \Gamma^*$, $\mu \neq \lambda$, cf. Lemma 5. Such a computation starts in state p, ends in state q, and effectively removes the topmost A from the stack. If the first instruction chosen is $(p, a, A, q_1, B_1 \cdots B_n)$ then A is replaced by $B_1 \cdots B_n$, and these symbols in turn have to be removed from the stack, one by one, before the computation of type $[p, A, q]$ ends. This means that the remainder of the $[p, A, q]$ computation is composed of computations of type $[q_1, B_1, q_2]$, $[q_2, B_2, q_3], \dots, [q_n, B_n, q]$, respectively, where $q_2, \dots, q_n$ are intermediate states (cf. Figure 2 where $n = 3$).

Now we construct a context-free grammar $G = (N, \Delta, S, P)$ such that $L(G) = N(\mathcal{A})$. The nonterminals represent the types of computations of the pda: $N = \{ [p, A, q] \mid p, q \in Q, A \in \Gamma \} \cup \{S\}$. The productions simulate the pda by recursively generating computations following the decomposition sketched above. The first production nondeterministically guesses the last state. The second production nondeterministically guesses the intermediate states $q_2, \dots, q_n$.

1. $S \to [q_{in}, A_{in}, q]$ for all $q \in Q$,
2. $[p, A, q] \to a[q_1, B_1, q_2][q_2, B_2, q_3] \cdots [q_n, B_n, q]$ when $(p, a, A, q_1, B_1 \cdots B_n)$ in δ, for all $q, q_2, \dots, q_n \in Q$, $n \geq 1$,
3. $[p, A, q] \to a$ when (p, a, A, q, λ) in δ.

Formally, the construction can be proved correct by showing inductively the underlying relation between computations and derivations: $[p, A, q] \Rightarrow_G^* x$ iff there is a computation of type $[p, A, q]$ reading x from the input, i.e., $(x, p, A) \vdash_{\mathcal{A}}^* (\lambda, q, \lambda)$.

Use induction on the length of the derivation/ computation to prove $[p, A, q] \Rightarrow_G^* w \iff (p, w, A) \vdash_{\mathcal{A}}^* (q, \lambda, \lambda)$

$\Leftarrow$: Assume $(p, w, A) \vdash^* (q, \lambda, \lambda)$. Consider the first instruction ι used by $\mathcal{A}$.

$*$ $\iota = (p, a, A, p', \lambda)$. This empties the stack; hence we have a single step computation, $(p, a, A) \vdash (p', \lambda, \lambda)$, so $p' = q$, $w = a$. By construction there exists the rule $[p, A, p'] \to a$. This yields the derivation $[p, A, q] \Rightarrow_G w = a$ as required.

$*$ $\iota = (p, a, A, q_1, B_1 \dots B_n)$, $n \geq 1$. The computation starts like $(p, w, A) \vdash (q_1, w', B_1 \dots B_n) \vdash^* (q, \lambda, \lambda)$, $w = aw'$. These B_i must be popped from the stack. We can split the computation $(q_1, w_1 w_2 \dots w_n, B_1 \dots B_n) \vdash^* (q_2, w_2 \dots w_n, B_2 \dots B_n) \vdash^* (q_n, w_n, B_n) \vdash^* (q_{n+1}, \lambda, \lambda)$, such that q_i is the first position where B_i appears as top of stack, $q_{n+1} = q$, and and $w' = w_1 w_2 \dots w_n$.

By the Computation Lemma [previous section], we can remove common parts of the input and stack at the start and end of a computation. We obtain separate computations $(q_i, w_i, B_i) \vdash^* (q_{i+1}, \lambda, \lambda)$.

As these are shorter than the original computation (even when $n = 1$ we separated the first step) we know that $[q_i, B_i, q_{i+1}] \Rightarrow_G^* w_i$.

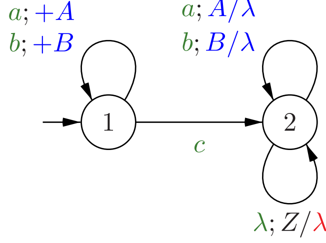
Given ι we know G has the rule $[p, A, q] \to a[q_1, B_1, q_2][q_1, B_1, q_2] \dots [q_n, B_n, q_{n+1}]$, which can be combined with the computations we found $[p, A, q] \Rightarrow a[q_1, B_1, q_2][q_1, B_1, q_2] \dots [q_n, B_n, q_{n+1}] \Rightarrow^* aw_1 w_2 \dots w_n = aw' = w$. As $q_{n+1} = q$ this is as required.

$\Rightarrow$: For the other direction use a similar technique. Given a derivation in G consider its first production rule, use induction on the remaining subtrees of the derivation to find computations, and join the pieces into a full computation.

□

The proof gives an explicit construction that transforms a pda (with empty stack acceptance) into an equivalent context-free grammar. This construction in general introduces non-terminals that are not useful in the grammar: they cannot generate a terminal string. Such non-terminals can be removed by standard constructions on cfg, or directly, by analyzing the computations of the original pda. We give a small example of this.

Example 9 The following pda accepts the language $L = \{ w c w^R \mid w \in \{a, b\}^* \}$ by empty stack. We transform it into an equivalent cfg.



Its twelve transitions lead to a total of 33 productions, where in the list below the variable X ranges over $\{A, B, Z\}$. The underlined states are 'guesses' made by the grammar on the intermediate states assumed by the pda during the computation that is simulated.

initial	$S \to [1, Z, \underline{1}] \mid [1, Z, \underline{2}]$
$(1, a, X, 1, AX)$	$[1, X, \underline{1}] \to a\, [1, A, \underline{1}][\underline{1}, X, \underline{1}]$
	$[1, X, \underline{1}] \to a\, [1, A, \underline{2}][\underline{2}, X, \underline{1}]$
	$[1, X, \underline{2}] \to a\, [1, A, \underline{1}][\underline{1}, X, \underline{2}]$
	$[1, X, \underline{2}] \to a\, [1, A, \underline{2}][\underline{2}, X, \underline{2}]$
$(1, b, X, 1, BX)$	$[1, X, \underline{1}] \to b\, [1, B, \underline{1}][\underline{1}, X, \underline{1}]$
	$[1, X, \underline{1}] \to b\, [1, B, \underline{2}][\underline{2}, X, \underline{1}]$
	$[1, X, \underline{2}] \to b\, [1, B, \underline{1}][\underline{1}, X, \underline{2}]$
	$[1, X, \underline{2}] \to b\, [1, B, \underline{2}][\underline{2}, X, \underline{2}]$
$(1, c, X, 2, X)$	$[1, X, \underline{1}] \to c\, [2, X, \underline{1}]$
	$[1, X, \underline{2}] \to c\, [2, X, \underline{2}]$
$(2, a, A, 2, \lambda)$	$[2, A, 2] \to a$
$(2, b, B, 2, \lambda)$	$[2, B, 2] \to b$
$(2, \lambda, Z, 2, \lambda)$	$[2, Z, 2] \to \lambda$

Recall the relation between derivations and computations: $[p, A, q] \Rightarrow_G^* w \iff (p, w, A) \vdash^* (q, \lambda, \lambda)$, which, informally, is a computation from p to q that pops A.

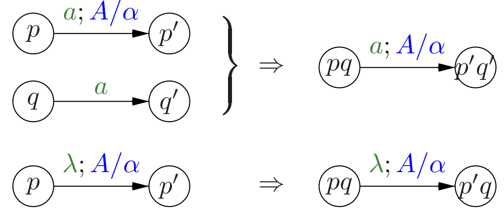
In the present automaton we see that popping computations must end in state 2. This means that the third component of the non-terminals only takes this value in successful derivations. Thus, from four productions for each of the three instructions $(1, a, X, 1, AX)$ we only need one each:

$(1, a, X, 1, AX)$	$[1, X, \underline{2}] \to a\, [1, A, \underline{2}][\underline{2}, X, \underline{2}]$

4 Application: Closure Properties

Pushdown automata are machines, and consequently they can be 'programmed'. For some problems this leads to intuitively simpler solutions than building a context-free grammar for the same task. We present two examples of closure properties of the family CF that can be proved quite elegantly using pushdown automata.

As a first example consider the closure of CF under intersection with regular languages: given a pda and a finite state automaton, one easily designs a new pushdown automaton that simulates both machines in parallel on the same input. The construction extends tha classical product construction for simulating two finite state automata in parallel. States are pairs of states, one for each of the simulated automata. In this way the new pda keeps track of the state of both machines, while its stack mimics the stack of the given pda. When simulating a λ-instruction of the given pda it does not change the state of the finite state automaton.



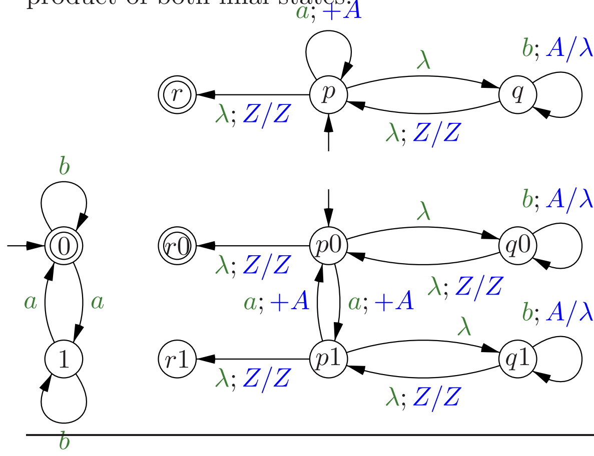
Thus, we have given the construction that can be used to prove the following closure property. The construction will be illustrated in an example.

Lemma 10 CF *is closed under intersection with regular languages.*

Example 11 We apply the construction to obtain a pda for the intersection $\{\, a^n b^n \mid n \geq 1 \,\}^* \cap \{\, w \in \{a, b\}^* \mid \#_a x \text{ even} \,\}$.

The picture to the left is the fsa, while the upper picture is the original pda; it accepts by final state. The pda that is obtained by the construction is shown as the 'product' of both atomata. Its initial state is

the product of both initial states, its final state the product of both final states.



Another closure application of the main equivalence we treat explicitly. If $h : \Sigma \to \Delta^*$ is a morphism, and K a language over Δ, then the inverse image $h^{-1}(K) \subseteq \Sigma^*$ is defined as $\{ w \in \Sigma^* \mid h(w) \in K \}$.

Lemma 12 CF *is closed under inverse morphisms.*

Proof. Let $K \subseteq \Delta^*$ be a context-free language, and let $h : \Sigma \to \Delta^*$ be a morphism. We show that the language $h^{-1}(K) \subseteq \Sigma^*$ is context-free. According to Theorem 8 we assume that K is given as the final state language of a pda $\mathcal{A} = (Q, \Delta, \Gamma, \delta, q_{in}, A_{in}, F)$.

The newly constructed pda $\mathcal{A}'$ for $h^{-1}(K)$ simulates, upon reading symbol $b \in \Sigma$, the behaviour of $\mathcal{A}$ on the string $h(b) \in \Delta^*$. So for input w the new pda checks whether $h(w)$ is accepted by $\mathcal{A}$, or equivalently, whether $h(w)$ belongs to K. The simulated input string $h(b)$ is temporarily stored in a (bounded) buffer that is added to the state. During this simulation $\mathcal{A}'$ only follows λ-instructions, 'reading' the input of the original automaton $\mathcal{A}$ from the internal buffer. Now let $\mathrm{Buf} = \{w \in \Delta^* \mid w$ is a suffix of $h(b)$ for some $b \in \Sigma\}$. The pda $\mathcal{A}'$ is given as follows.

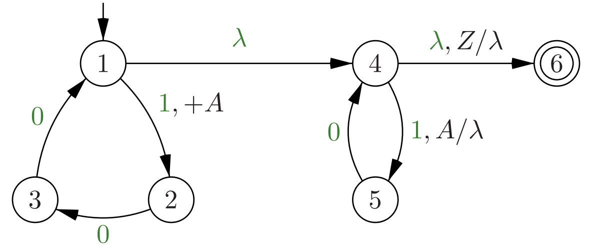
$\mathcal{A}' = (Q \times \mathrm{Buf}, \Sigma, \Gamma, \delta', \langle q_{in}, \lambda\rangle, A_{in}, F \times \{\lambda\})$, where δ' contains the following instructions (for clarity we denote elements from $Q \times \mathrm{Buf}$ as $\langle q, w\rangle$ rather than (q, w)):

- (*input & filling buffer*) For each $b \in \Sigma$, $p \in Q$, and $A \in \Gamma$ we add $(\langle p, \lambda\rangle, b, A, \langle p, h(b)\rangle, A)$ to δ'.
- (*simulation of* $\mathcal{A}$) For each $a \in \Delta \cup \{\lambda\}$ and $v \in \Delta^*$ with $av \in \mathrm{Buf}$ we add $(\langle p, av\rangle, \lambda, A, \langle q, v\rangle, \alpha)$ to δ' when (p, a, A, q, α) belongs to δ.

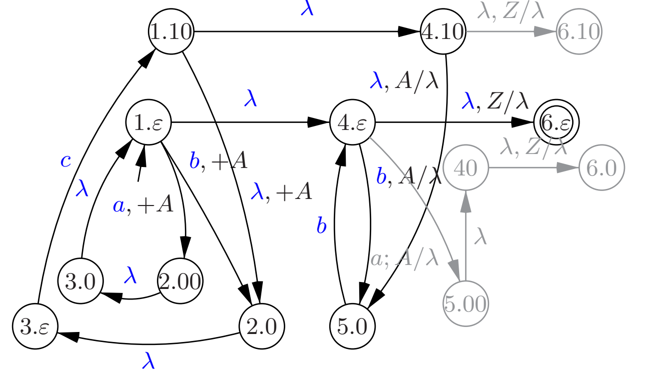
The pda $\mathcal{A}'$ obtained in this way accepts $L(\mathcal{A}') = h^{-1}(K)$ and consequently, as pda's accept context-free languages, $h^{-1}(K)$ is context-free. □

Example 13 Consider the homomorphism $h : \{a, b, c\}^* \to \{0, 1\}^*$ defined by $h : \begin{cases} a \mapsto 100 \\ b \mapsto 10 \\ c \mapsto 010 \end{cases}$

We construct a pda for $h^{-1}(K) \subseteq \{a, b, c\}^*$ where K is the context-free language $\{ (100)^n (10)^n \mid n \geq 0 \}$; K is accepted by the following pda with final states.



Note that *aaabbb*, *abccbb*, and *bcccbb* belong to $h^{-1}(K)$ as their image under h equals $(100)^3(10)^3$, which belongs to K. Following the 'buffer construction' we obtain a pda for $h^{-1}(K)$. The picture shows some states in gray. These states do not lead to a terminal state and can be omitted form the automaton without changing its language.



Theorem 19 below, and the discussion preceding it provide an alternative view on this closure property.

Normal forms and extensions. We have seen in the beginning of this section that context-free grammars in Greibach normal form can be disguised as

single state pushdown automata (under empty stack acceptance). Together with Theorem 8 this shows that these single state automata constitute a *normal form* for pda's. More importantly, these automata are *real-time*, that is they do not have any λ-instructions. Additionally we can require that each instruction pushes at most two symbols back on the stack, i.e., in (q, a, A, q, α) we have $|\alpha| \leq 2$.

For final state acceptance we need two states in general, in order to avoid accepting the prefixes of every string in the language.

Another normal form considers the number of stack symbols. An elementary construction shows that two symbols suffice. On the stack the element B_i of $\Gamma = \{B_1, B_2, \ldots, B_n\}$ can be represented, e.g., by the string $A^i B$ over the two symbol stack alphabet $\{A, B\}$.

An *extension* of the model can be obtained by allowing the pda to move on the empty stack. As we have seen in connection with Lemma 6, this can be simulated by our standard model by keeping a reserved symbol on the bottom of the stack. A second extension is obtained by allowing the model to push symbols without popping, or to pop several symbols at once, making the general instruction of the form (p, a, β, q, α) with $\beta, \alpha \in \Gamma^*$. Again this is easily simulated by the standard model.

A useful extension is to give the pda access to any relevant finite state information concerning the stack contents (i.e, does the stack contents belong to a given regular language) instead of just the topmost symbol.

5 Finite State Transductions

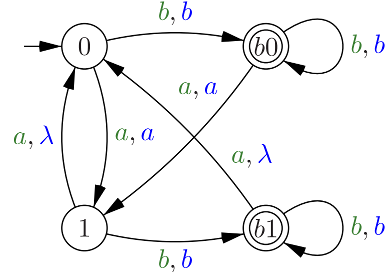
This section deals with a very general mechanism to translate lanuages, the finite state transducer. We show that the context-free languages are closed under its translations. This result generalizes the closure under morphism, inverse morphism, and intersection under regular languages (and in fact any composition of them). This theory is not always part of textbooks on formal language theory, although it is not very deep and at the same time rather elegant.

A finite state transducer (fst) $\mathcal{M}$ is a finite state machine having both input and output. It defines a *rational relation* $T_\mathcal{M}$ in $\Sigma^* \times \Delta^*$ for two alphabets Σ and Δ. Thus, for (u, v) to be in $T_\mathcal{M}$ the fst must be able to read $u \in \Sigma^*$ while writing $v \in \Delta^*$.

A fst is specified as 6-tuple $\mathcal{M} = (Q, \Sigma, \Delta, \delta, q_{in}, F)$ with finite state set Q, alphabets Σ and Δ, initial state $q_{in} \in Q$, final state set $F \subseteq Q$, and instructions δ, a subset of $Q \times \Sigma^* \times \Delta^* \times Q$. The pair $(u, v) \in \Sigma^* \times \Delta^*$ belongs to $T_\mathcal{M}$ if there exists a sequence of instructions from the initial state to a final state, such that the two sequences of labels concatenate to u and v respectively. While we say that $\mathcal{M}$ reads u and writes v, mathematically the two alphabets are equivalent.

For a language $K \subseteq \Sigma^*$, the image (translation) of K by $\mathcal{M}$ equals $T_\mathcal{M}(K) = \{v \in \Delta^* \mid (u, v) \in T_\mathcal{M} \text{ for some } u \in K\}$.

Example 14 The following finite state transducer accepts only strings ending with the letter b. All other input is rejected. In the transduction every second symbol a in the input is erased.



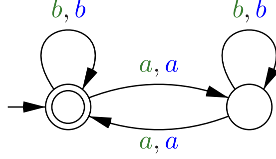
A morphism $h : \Sigma^* \to \Delta^*$ can be realised by a fst in a simple way. Just introduce a single state q, and for every symbol $a \in \Sigma$ add a loop $(q, a, h(a), q)$. Now every path in the fst reads a string $w \in \Sigma^*$ while writing the image $h(w) \in \Delta^*$.

By swapping a and $h(a)$ we obtain a fst for the inverse morphism $h^{-1} : \Delta^* \to \Sigma^*$.

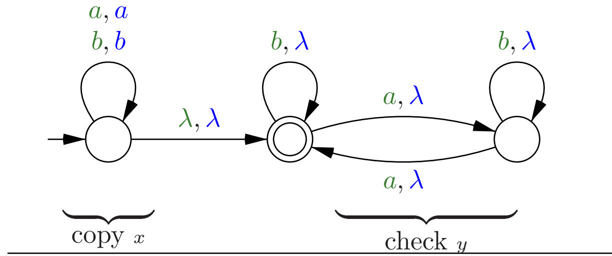
Additionally fst can also realise other operations, like intersection with a regular language, and quotient with a regular language.

Example 15 Let R be the regular language $\{ x \in \{a, b\}^* \mid \#_a x \text{ even} \}$. It is easy to transform a finite

stae automaton for R into a transducer $\mathcal{A}$ that realizes the intersection $T_{\mathcal{A}}(K) = K \cap R$, by just copying the input to the output.



One also may build a transducer for the quotient operation $T(K) = \{ x \mid xy \in K \text{ and } y \in R \}$ copying a prefix of the input, and checking whether the remaining suffix belongs to R.



We have he following general closure property of the context-free languages.

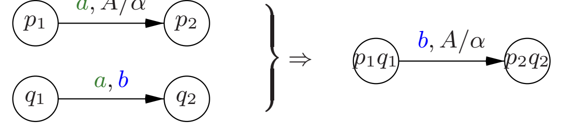
Lemma 16 CF *is closed under finite state transductions.*

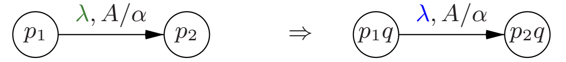
Proof. Assume the context-free language K is given by a pushdown automaton $\mathcal{A} = (Q_{\mathcal{A}}, \Sigma, \Gamma, \delta_{\mathcal{A}}, q_{\mathcal{A}}, Z, F_{\mathcal{A}})$, such that $L(\mathcal{A}) = K$ (using final state acceptance).

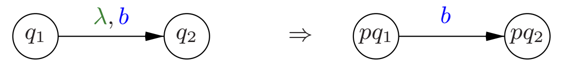
Given a fst $\mathcal{M} = (Q_{\mathcal{M}}, \Sigma, \Delta, \delta_{\mathcal{M}}, q_{\mathcal{M}}, F_{\mathcal{M}})$, we build a pda $\mathcal{A}'$ for $T(\mathcal{M})(K)$ that simulates $\mathcal{A}$ and $\mathcal{M}$ in parallel, not reading the original symbol, but rather its translation by $\mathcal{M}$. We take some special care to deal with λ: when either the pda $\mathcal{A}$ or the fst $\mathcal{M}$ read λ, the other 'component' does not make a step.

The new pda is specified as $\mathcal{A}' = (Q_{\mathcal{A}} \times Q_{\mathcal{M}}, \Delta, \Gamma, \delta, \langle q_{\mathcal{A}}, q_{\mathcal{M}}\rangle, Z, F_{\mathcal{A}} \times F_{\mathcal{M}})$, where the transition relation δ is constructed as follows.

- For each $(p_1, a, A, p_2, \alpha) \in \delta_{\mathcal{A}}$, and each $(q_1, a, b, q_2) \in \delta_{\mathcal{M}}$ (with $a \neq \lambda$), we have $(\langle p_1, q_1\rangle, b, A, \langle p_2, q_2\rangle\alpha) \in \delta$,
- for each $(p_1, \lambda, A, p_2, \alpha) \in \delta_{\mathcal{A}}$ and each $q \in Q_{\mathcal{M}}$, we have $(\langle p_1, q\rangle, \lambda, A, \langle p_2, q\rangle\alpha) \in \delta$, and
- for each $(q_1, \lambda, b, q_2) \in \delta_{\mathcal{M}}$ and each $p \in Q_{\mathcal{A}}$, $X \in \Gamma$, we have $(\langle p, q_1\rangle, b, X, \langle p, q_2\rangle, X) \in \delta$.







In this way a successful computation of $\mathcal{A}$ on $u \in \Sigma^*$ and a accepting computation of $\mathcal{M}$ on $(u, v) \in \Sigma^* \times \Delta^*$ is combined into a successful computation of $\mathcal{A}'$ on v. □

Thus we can generalize the results from Lemma 10 and Lemma 12 to all finite state transductions (regular relations).

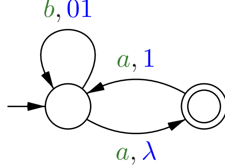
Corollary 17 CF *is closed under morphisms, inverse morphisms, intersection, quotient & concatenation with regular languages, prefix, suffix, ...*

Chomsky-Schützenberger. There are several elementary characterizations of CF as a family of languages related to the Dyck languages, i.e., languages consisting of strings of matching brackets (see Section II.3 in [2]). We present here one of these results, claiming it is directly related to the storage behaviour of the pushdown automaton being the machine model for CF.

Many common operations, most notably intersection with a regular language and (inverse) morphisms, are in fact rational relations. Moreover, the family of rational transductions is closed under inverse and under composition. A famous result of Nivat characterizes rational transductions τ as a precise composition of these operations: $\tau(x) = g(h^{-1}(x) \cap R)$ for every $x \in \Delta_1^*$, where g is a morphism, h^{-1} is an inverse morphism, and R is a regular language. There is a clear intuition behind this result: R is the regular language over δ of sequences of transitions leading from initial to final state, and h and g are the

morphisms that select input and output, respectively: $h(\langle p,u,v,q\rangle) = u$, $g(\langle p,u,v,q\rangle) = v$.

Example 18 Consider the following fst, with input alphabet $\{a,b\}$ and output alphabet $\{0,1\}$. Its domain (input language) is the regular language $\{aa,b\}^*a$, and it changes every aa into 1, changes every b into 01, and deletes the last a.



K	$\ni$	b	b	a	a	b	a
			$\uparrow h$				
$R_{\mathcal{M}}$	$\ni$	b:01	b:01	a:λ	a:1	b:01	a:λ
			$\downarrow g$				
$T_{\mathcal{M}}(K)$	$\ni$	01	01	λ	1	01	λ

$$x \xleftarrow{h} R \xrightarrow{g} y$$

input   computation   output

A pda $\mathcal{A}$ can actually be seen as a transducer mapping input symbols to sequences of pushdown operations. Assuming stack alphabet Γ we interpret Γ as a set of push operations, and we use a copy $\bar{\Gamma} = \{\bar{A} \mid A \in \Gamma\}$ to denote pop operations. The pda instruction $(p,a,A,q,B_n\cdots B_1)$ can thus be re-interpreted as the transducer transition $\langle p,a,\bar{A}B_1\cdots B_n,q\rangle$, mapping input a to output $\bar{A}B_1\cdots B_n$ (pushdown operations 'pop A, push B_1, ..., push B_n'). Now input x is accepted with empty stack by the pda $\mathcal{A}$ if the sequence of pushdown operations produced by the transducer is a legal LIFO sequence, or equivalently, if transduction $\tau_{\mathcal{A}}$ maps x to a string in D_Γ, the *Dyck language* over $\Gamma \cup \bar{\Gamma}$, which is the context-free language generated by the productions $S \to \lambda$, $S \to SS$, $S \to AS\bar{A}$, $A \in \Gamma$. Thus, $N(\mathcal{A}) = \tau_{\mathcal{A}}^{-1}(D_\Gamma)$.

Since we may assume that $\Gamma = \{A,B\}$, it follows from this, in accordance with the general theory of Abstract Families of Languages (AFL), that CF is the *full trio generated by* $D_{\{A,B\}}$, the Dyck language over two pairs of symbols; in the notation of [8]:

Theorem 19 $\mathsf{CF} = \widehat{\mathcal{M}}(D_{\{A,B\}})$, *the smallest family that contains* $D_{\{A,B\}}$ *and is closed under morphisms, inverse morphisms, and intersection with regular languages (i.e., under rational relations).*

This is closely related to the result attributed to Chomsky and Schützenberger that every context-free language is of the form $g(D_\Gamma \cap R)$ for a morphism g, alphabet Γ, and regular R; in fact, $D_\Gamma = h^{-1}(D_{\{A,B\}})$, where h is any injective morphism $h : \Gamma \to \{A,B\}^*$, extended to $\bar{\Gamma}$ in the obvious way.

6 Deterministic PDA

From the practical point of view, as a model of recognizing or parsing languages, the general pda is not considered very useful due to its nondeterminism. Like for finite state automata, determinism is a well-studied topic for pushdown automata. Unlike the finite state case however, determinism is not a normal form for pda's.

In the presence of λ-instructions, the definition of determinism is somewhat involved. First we have to assure that the pda never has a choice between executing a λ-instruction and reading its input. Second, when the input behaviour is fixed the machine should have at most one applicable instruction.

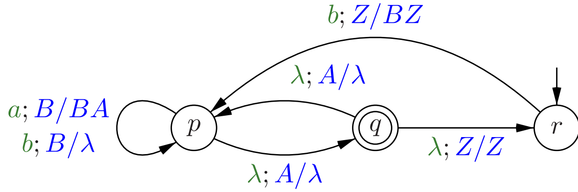
Definition 20 The pda $\mathcal{A} = (Q,\Delta,\Gamma,\delta,q_{in},A_{in},F)$ is *deterministic* if

- for each $p \in Q$, each $a \in \Delta$, and each $A \in \Gamma$, δ does not contain both an instruction (p,λ,A,q,α) and an instruction (p,a,A,q',α').
- for each $p \in Q$, each $a \in \Delta \cup \{\lambda\}$, and each $A \in \Gamma$, there is at most one instruction (p,a,A,q,α) in δ.

We like to stress that it *is* allowed to have both the instructions (p,λ,A,q,α) and (p,a,A',q',α') in δ for $a \neq \lambda$ provided $A \neq A'$. That is, the choice between these two instructions is determined by the top of the stack in otherwise equal configurations. The pda from Example 4 is deterministic.

Example 21 The pda in the picture below is deterministic. There are two outgoing transitions in state q, both λ-transitions: $(q, \lambda, A, p, \lambda)$ and (q, λ, Z, r, Z). These does not conflict as the stack symbols differ.

Then there are three transitions leaving state p. Two of these, (p, a, B, p, BA) and (p, b, B, p, BA), have the same stack symbol, but determinism is enforced by the two different input symbols. The third $(p, \lambda, A, q, \lambda)$ is a λ-transition, a possible cause for non-determinism, but here the stack symbol differs from the two transitions reading a and b.



Keep in mind that a pda can engage in a (possibly infinite) sequence of λ-steps even after having read its input. In particular, this means that acceptance is not necessarily signalled by the first state after reading the last symbol of the input.

Again, we can consider two ways of accepting languages by deterministic pda: either by final state or by empty stack. Languages from the latter family are prefix-free: they do not contain both a string and one of its proper prefixes. As a consequence the family is incomparable with the family of regular languages. The pda construction to convert empty stack acceptance into final state acceptance (cf. Lemma 6) can be made to work in the deterministic case; the converse construction can easily be adapted for prefix-free languages.

Lemma 22 *A language is accepted by empty stack by a deterministic pda iff it is prefix-free and accepted by final state by a deterministic pda.*

Here we will study languages accepted by deterministic pda by final state, known as *deterministic context-free languages*, a family denoted here by DPDA. The strict inclusion REG $\subset$ DPDA is obvious, as a deterministic finite state automaton can be seen as a deterministic pda ignoring its stack, and a deterministic pda for the non-regular language $\{ a^n b a^n \mid n \geq 1 \}$ can easily be constructed.

Intuitively, the deterministic context-free languages form a proper subfamily of the context-free languages. In accepting the language of palindromes $L_{\text{pal}} = \{ x \in \{a, b\}^* \mid x = x^{\text{R}} \}$, where x^{R} denotes the reverse of x, one needs to guess the middle of the input string in order to stop pushing the input to the stack and start popping, comparing the second half of the input with the first half. However, this is far from a rigorous proof of this fact. We establish the strict inclusion indirectly, by showing that CF and DPDA do not share the same closure properties (as opposed to using some kind of pumping property).

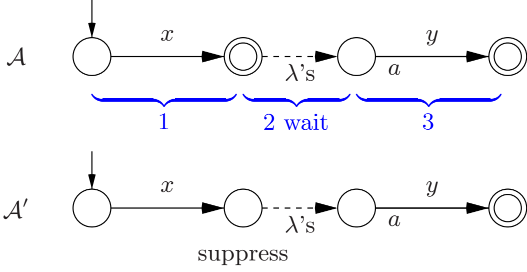
For a language L we define $\text{pre}(L) = \{xy \mid x \in L, xy \in L, y \neq \lambda\}$, in other words, $\text{pre}(L)$ is the subset of L of all strings having a proper prefix that also belongs to L. Observe that CF is not closed under pre, as is witnessed by the language $L_d = \{ a^n b a^n \mid n \geq 1 \} \cup \{ a^n b a^m b a^n \mid m, n \geq 1 \}$ for which $\text{pre}(L_d) = \{ a^n b a^m b a^n \mid m \geq n \geq 1 \}$.

Lemma 23 DPDA *is closed under* pre.

Proof. Let $\mathcal{A} = (Q, \Delta, \Gamma, \delta, q_{in}, A_{in}, F)$ be a deterministic pda, with final state acceptance. The new deterministic pda $\mathcal{A}' = (Q', \Delta, \Gamma, \delta', q'_{in}, A_{in}, F')$ with $L(\mathcal{A}') = \text{pre}(L(\mathcal{A}))$ simulates $\mathcal{A}$ and additionally keeps track in its states whether or not $\mathcal{A}$ already has accepted a (proper) prefix of the input. Let $Q' = Q \times \{1, 2, 3\}$. Intuitively $\mathcal{A}'$ passes through three phases: in phase 1 $\mathcal{A}$ has not seen a final state, in phase 2 $\mathcal{A}$ has visited a final state, but has not yet read from the input after that visit, and finally in phase 3 $\mathcal{A}$ has read a symbol from the input after visiting a final state; $\mathcal{A}'$ can only accept in this last phase. Accordingly, $F' = F \times \{3\}$, and $q'_{in} = \langle q_{in}, 1\rangle$ whenever $q_{in} \notin F$ and $\langle q_{in}, 2\rangle$ when $q_{in} \in F$. The instructions of $\mathcal{A}'$ are defined as follows:

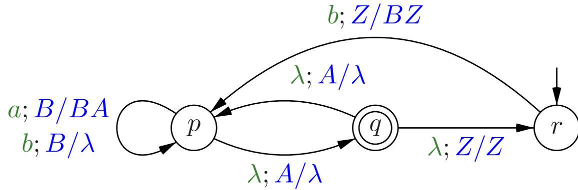
- for (p, a, A, q, α) in δ and $q \notin F$, add $(\langle p, 1\rangle, a, A, \langle q, 1\rangle, \alpha)$ to δ',
- for (p, a, A, q, α) in δ and $q \in F$, add $(\langle p, 1\rangle, a, A, \langle q, 2\rangle, \alpha)$ to δ',
- for $(p, \lambda, A, q, \alpha)$ in δ, add $(\langle p, 2\rangle, \lambda, A, \langle q, 2\rangle, \alpha)$ to δ',
- for (p, a, A, q, α) in δ with $a \in \Delta$, add $(\langle p, 2\rangle, a, A, \langle q, 3\rangle, \alpha)$ to δ', and

- for $(p, a, A, q, \alpha) \in \delta$, add $(\langle p, 3\rangle, a, A, \langle q, 3\rangle, \alpha)$ to δ'.



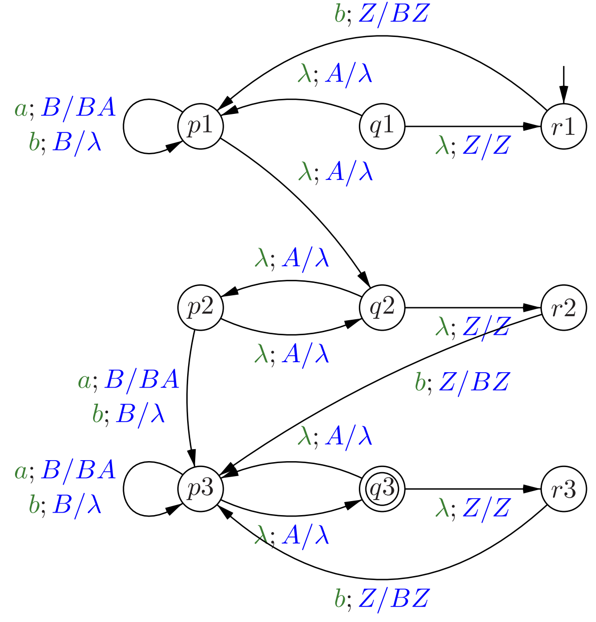
□

Example 24 We apply the construction of the previous proof to the deterministic pda from Example 21.



Make three copies of the automaton. For the first level, the instruction $(p, \lambda, A, q, \lambda)$ enters the final state, and its copy will go to the second level $(p1, \lambda, A, q2, \lambda)$. For the second level we keep only λ-instructions. The instructions that read a letter will end in the third copy.

In this we we get the following pda.



As an immediate consequence we have the strict inclusion $\mathsf{DPDA} \subset \mathsf{CF}$, and in fact it follows that the language L_d above is an element of the difference $\mathsf{CF} - \mathsf{DPDA}$. Additionally we see that DPDA is not closed under union.

Without further discussion we state some basic (non)closure properties. Note that these properties differ drastically from those for CF. By $\min(L) = L - \text{pre}(L)$ we mean the set of all strings in L that do *not* have a proper prefix in L; $\max(L)$ is the set of all strings in L that are not the prefix of a longer string in L.

Theorem 25 DPDA *is closed under the language operations complementation, inverse morphism, intersection with regular languages, right quotient with regular languages,* pre, min, *and* max*; it is* not *closed under union, intersection, concatenation, Kleene star, (λ-free) morphism, and mirror image.*

We just observe here that closure under min is obtained by removing all instructions (p, a, A, q, α) with $p \in F$, and that closure under inverse morphisms and under intersection with a regular language is proved as in the nondeterministic case. The latter closure

property allows us to prove rigorously that L_{pal} is not in DPDA: otherwise, $L_d = L_{\text{pal}} \cap (a^+ba^+ \cup a^+ba^+ba^+)$ would also be in DPDA. We return to the proof of the remaining positive properties in the next section.

Real-time. For deterministic automata, real-time, i.e., the absence of λ-instructions is *not* a normal form. However, it is possible to obtain automata in which every λ-instruction pops without pushing, i.e., is of the form $(p, \lambda, A, q, \lambda)$. This is explained in [1].

Decidability. Partly as a consequence of the effective closure of DPDA under complementation, the decidability of several questions concerning context-free languages changes when restricted to deterministic languages. Thus, the questions of completeness '$L(\mathcal{A}) = \Delta^*$?', and even equality to a given regular language '$L(\mathcal{A}) = R$?', are easily solvable. Also regularity 'is $L(\mathcal{A})$ regular?' is decidable, but its solution is difficult.

The questions on complementation and ambiguity — 'is the complement of $L(\mathcal{A})$ (deterministic) context-free?' and 'is $L(\mathcal{A})$ inherently ambiguous?'— are now trivially decidable, while undecidable for CF as a whole.

The *equivalence problem* '$L(\mathcal{A}_1) = L(\mathcal{A}_2)$?' for deterministic pda's has been open for a long time. It has been solved rather recently by Sénizergues, and consequently it is not mentioned in most of the textbooks on Formal Language Theory. The problem is easily seen to be semi-decidable: given two (deterministic) pda's that are not equivalent a string proving this fact can be found by enumerating all strings and testing membership. The other half of the problem, establishing a deduction system that produces all pairs of equivalent deterministic pda's was finally presented at ICALP'97. A more recent exposition of the decidability is given in [17]. Many sub-cases of the equivalence problem had been solved before, like the equivalence for *simple* context-free languages, accepted by single state deterministic (real-time) pda's. For an exposition of the theory of simple languages see [9].

7 Related Models

There are really many machine models having a data type similar to the pushdown stack. Some of these were motivated by the need to find subfamilies of pda's for which the equivalence is decidable, others were introduced as they capture specific time or space complexity classes. We mention a few topics that come to our mind.

Simple grammars. A context-free grammar is *simple* if it is in Greibach normal form, and there are no two productions $A \to a\alpha$ and $A \to a\beta$ with terminal symbol a and $\alpha \neq \beta$. Via a standard construction we have given before, these grammars correspond to single state, deterministic, and real-time pda's. But in fact the real-time property can be dropped, cf. [9, Section 11.9].

Two stacks. Finite state devices equipped with two stacks are easily seen to have Turing power. Both stacks together can act as a working tape, and the machine can move both ways on that tape shifting the contents of one stack to the other by popping and pushing.

Counter automata. When we restrict the stack to strings of the form A^*Z, i.e., a fixed bottom symbol and one other symbol, we obtain the *counter automaton*, cf. Example 4. The stack effectively represents a natural number ($\mathbb{N}$) which can be incremented, decremented, and tested for zero.

As such an automaton can put a sign in its finite state, while keeping track of the moments where the stack 'changes sign' this can be seen to be equivalent to having a data type which holds an integer ($\mathbb{Z}$) which again can be incremented, decremented, and tested for zero.

With a single counter, the counter languages form a proper subset of CF, as L_{pal} cannot be accepted in this restricted pushdown mode, see [2, Section VII.4]. Automata having two of these counters can, by a clever trick, code and operate on strings, and are

again equivalent to Turing machines. See [12, Theorem 7.9] for further details.

Blind and partially blind counters. A counter is *blind* if it cannot be tested for zero [7]. The counter keeps an integer value that can be incremented and decremented. It is tested only once for zero, at the end of the computation as part of the (empty stack) acceptance condition.

The family of languages accepted by blind multi-counter automata, i.e., automata equipped with several blind counters, is incomparable with CF. Let Σ_k be the alphabet $\{a_1, b_1, \ldots, a_k, b_k\}$. Define $B_k = \{x \in \Sigma_k^* \mid |x|_{a_i} = |x|_{b_i} \text{ for each } 1 \leq i \leq k\}$. Observe that B_k models the possible legal operation sequences on the blind counter storage, interpreting a_i and b_i as increments and decrements of the i-th counter. Of course, B_k can be recognized by an automaton with k blind counters, while it can be shown that it cannot be recognized by a pda (for $k > 1$) or by a blind $(k-1)$-counter automaton. In fact, in the vein of Theorem 19, the family of languages accepted by blind k-counter automata equals the full trio generated by B_k.

A counter is *partially blind* if it is blind and holds a natural number; on decrementing zero the machine blocks as the operation is undefined. Partially blind multicounters form the natural data type for modelling Petri nets.

Valence grammars. Valence grammars associate with each production of a context-free grammar a vector of k integers, and consider only those derivations for which these valences of the productions used add to the zero vector. An equivalent machine model for these grammars consists of a pda equipped with k additional blind counters. Consequently, their language family is characterized as the full trio generated by the shuffle of $D_{\{A,B\}}$ and B_k, from which closure properties follow. Greibach normal form (for grammars) and real-time normal form (for automata) can be shown to hold. See [10] for an AFL approach and further references.

Finite turn pda's. A pda is *finite turn* if there is a fixed bound on the number of times the machine switches from pushing to popping. Like for bounded excursions (Section 3) such a bound can be implemented in the machine itself. The restriction to a single turn leads to the *linear languages*, whereas finite turn pda's are equivalent to ultralinear context-free grammars, as explained in [9, Section 5.7]. A context-free grammar $G = (N, T, S, P)$ is *ultra linear* if there is a partition of the nonterminals $N = N_0 \cup \cdots \cup N_n$ and each production for $A \in N_i$ is either of the form $A \to \alpha$ with $\alpha \in (T \cup N_0 \cup \cdots \cup N_{i-1})^*$ —A introduces only nonterminals of lower 'levels' of the partition— or of the form $A \to uBv$ with $u, v \in T^*$ and $B \in N_i$ —the only nonterminal introduced by A is from the same 'level'.

Alternation. A nondeterministic automaton is successful if it has a computation that reads the input and reaches an accepting configuration. Thus, along the computation, for each configuration there exists a step eventually leading to success. A dual mode —all steps starting in a configuration lead to acceptance— is added in *alternating* automata; states, and hence configurations, can be existential (nondeterministic) or universal. The alternating pda's accept the family $\bigcup_{c>0} \mathsf{DTIME}(c^n)$ of languages recognizable in exponential deterministic time [14]. Note that alternating finite automata just accept regular languages.

Two-way pda's. Considering the input tape as a two-way device, we obtain the *two-way pushdown automaton*; it is customary to mark both ends of the input tape, so that the two-way pda detects the end (and begin) of the input. These machines can scan their input twice, or in the reverse direction, etcetera, making it possible to recognize non-context-free languages like $\{ a^n b^n c^n \mid n \geq 1 \}$ (easy) and $\{ ww \mid w \in \{a, b\}^* \}$ (try it). Hence, just as for alternation, the two-way extension is more powerful than the standard pda, unlike the case for finite automata where both variants define the regular languages.

Languages of the *deterministic* two-way pda can be recognized in *linear time*, which has led to the discovery of the pattern matching algorithm of

Knuth-Morris-Pratt, as the pattern matching language $\{ v\#uvw \mid u,v,w \in \{a,b\}^* \}$ can be recognized by such an automaton. See Section 7 in [13] for a historical account.

Finally, multi-head pda's, either deterministic or nondeterministic (!), characterize the family P of languages recognizable in deterministic polynomial time. An introduction to automata theoretic complexity is given in [12, Chapter 14], while more results are collected in [18, Sections 13 and 20.2]. Multi-head k-iterated pda's characterize the deterministic $(k-1)$-iterated exponential time complexity classes [5].

Stack automata. A *stack automaton* is a pda with the additional capability to inspect its stack. It may move up and down the stack, in read-only mode, i.e., without changing its contents. This makes the stack automaton more powerful than the pda. The family of languages recognized by stack automata lies properly between CF and the indexed languages. Stack automata that do not read input during inspection of the stack are equivalent to pda's.

A *nested* stack automaton has the possibility to start a new stack 'between' the cells of the old stack. This new stack has to be completely removed before the automaton can move up in the original stack. These automata are equivalent to pushdown-of-pushdowns automata, i.e., to indexed grammars. More generally, k-iterated nested stack automata correspond to $2k$-iterated pda's.

Again, variants of the corresponding two-way and multi-head automata characterize complexity classes; see the references mentioned above. Let us mention that the families accepted by the nondeterministic two-way stack (or nested stack) and nonerasing stack automata coincide with $\bigcup_{c>0} \mathsf{DTIME}(c^{n^2})$ and $\mathsf{NSPACE}(n^2)$, respectively (where a stack automaton is *nonerasing* if it never pops a symbol). The nondeterministic multi-head k-iterated stack (or nested stack) and nonerasing stack automata define deterministic $(2k-1)$-iterated exponential time and $(k-1)$-iterated exponential space.

Bibliography

Early studies in this subject are [15, 16]. The main text of this introductory text is taken from our book chapter [11]. Examples from the Tarragona lectures were added. Two in-depth introductions to pushdown automata (and context-free grammars) are the handbook chapters [1] and [3].

References

[1] J.-M. Autebert, J. Berstel, L. Boasson. Context-Free Languages and Pushdown Automata. In: *Handbook of Formal Languages*, Vol. 1 (G. Rozenberg, A. Salomaa, eds.) Springer, Berlin, 1997. pages 15, 17

[2] J. Berstel. *Transductions and Context-Free Languages*. Teubner Studienbücher, Stuttgart, 1979. pages 11, 15

[3] J. Berstel, L. Boasson. Context-Free Languages. In: *Handbook of Theoretical Computer Science*, Vol. B: Formal Models and Semantics (J. van Leeuwen, ed.) Elsevier, Amsterdam, 1990. pages 17

[4] N. Chomsky. Context Free Grammars and Pushdown Storage. Quarterly Progress Report, Vol. 65, MIT Research Laboratory in Electronics, Cambridge, MA, 1962. pages 6

[5] J. Engelfriet. Iterated Stack Automata. Information and Computation, 95 (1991) 21–75. pages 17

[6] J. Evey. Application of Pushdown Store Machines. Proceedings of the 1963 Fall Joint Computer Conference, Montreal, AFIPS Press, 1963. pages 6

[7] S. Greibach. Remarks on Blind and Partially Blind One-way Multicounter Machines. Theoretical Computer Science, 7 (1978) 311–324. pages 16

[8] S. Ginsburg. *Algebraic and Automata-theoretic Properties of Formal Languages*. Fundamental Studies in Computer Science, Vol. 2, North-Holland, 1975. pages 12

[9] M.A. Harrison. *Introduction to Formal Language Theory*. Addison-Wesley, Reading, Mass., 1978. pages 15, 16

[10] H.J. Hoogeboom. Context-Free Valence Grammars – Revisited. In: Developments in Language Theory, DLT 2001 (W. Kuich, G. Rozenberg, A. Salomaa, eds.), Lecture Notes in Computer Science, Vol. 2295, 293-303, 2002. pages 16

[11] Hendrik Jan Hoogeboom and Joost Engelfriet. Pushdown Automata. Chapter 6 in: Formal Languages and Applications (C. Martín-Vide, V. Mitrana, G. Paun, eds.), Studies in Fuzziness and Soft Computing, v. 148, Springer, Berlin, 117-138, 2004. pages 17

[12] J. Hopcroft, J. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Reading, MA: Addison-Wesley, 1979. pages 16, 17

[13] D.E. Knuth, J.H. Morris, V.R. Pratt. Fast Pattern Matching in Strings. SIAM Journal on Computing, 6 (1977) 323–360. pages 17

[14] R.E. Ladner, R.J. Lipton, L.J. Stockmeyer. Alternating Pushdown and Stack Automata. SIAM Journal on Computing 13 (1984) 135–155. pages 16

[15] A.G. Oettinger. Automatic Syntactic Analysis and the Pushdown Store. Proceedings of Symposia on Applied Mathematics, Vol. 12, Providence, RI, American Mathematical Society, 1961. pages 17

[16] M. Schützenberger. On Context Free Languages and Pushdown Automata. Information and Control, 6 (1963) 246–264. pages 6, 17

[17] G. Sénizergues. $L(A) = L(B)$? A Simplified Decidability Proof. Theoretical Computer Science, 281 (2002) 555–608. pages 15

[18] K. Wagner, G. Wechsung. *Computational Complexity*. Reidel, Dordrecht, 1986. pages 17

Leiden, October 13, 2008. *Work in progress.*

Preliminaries

A context-free grammar is specified as a 4-tuple $G = (N, T, S, P)$, where N is the set of non-terminals, T is the set of terminals ($N \cap T = \varnothing$), $S \in N$ is the start symbol (or axiom), and $P \subseteq N \times (N \cup T)^*$ is a finite set of productions (or rules).

If (A, α) is a production, then we usually write $A \to \alpha$.

Using the productions of G strings over $(N \cup T)^*$ can be rewritten. In a rewrite step an occurrence in a string of the left hand side of a rule is replaced by the right hand side of the rule. One defines a rewriting relation accordingly. For $x, y \in (N \cup T)^*$ and $A \to \alpha$ we have $xAy \Rightarrow x\alpha y$.