1

Pushdown Automata*

Hendrik Jan Hoogeboom and Joost Engelfriet

Institute for Advanced Computer Science,
Universiteit Leiden, the Netherlands

1.1 Introduction

Recursive functions in a computer program can be modelled by suitable grammatical rules. As an example, cf. Figure 1.1, the recursive function `Hanoi`, moving n disks from pin `s` to pin `t` using additional pin `v` can be represented by productions like $H_{stv}(n) \to H_{svt}(n-1)\ m_{st}\ H_{vts}(n-1)$ and $H_{stv}(0) \to \lambda$ —with terminal symbols m_{xy}, $x, y \in \{s, t, v\}$. Of course, context-free grammars do not have attributes to their nonterminals and we could abstract from them by writing $H_{stv} \to H_{svt} m_{st} H_{vts}$, $H_{stv} \to \lambda$.

Such a recursive function will lead to a stack of recursive function calls during execution of the program. Thus the LIFO ('last-in-first-out') stack is another face of recursion.

In this chapter we consider a machine model based on an abstraction of this very natural data type, the pushdown automaton [23, 25]. Of course this chapter should be read in conjunction with Chapter **??** on context-free grammars, and Chapter **??** regarding finite state automata. In the abstract world of formal languages the models of context-free grammar and pushdown automaton are again two faces of the same phenomenon: recursion. In particular, this means the models are equivalent in their language defining power, as shown in Theorem 6 below.

As expected, this chapter contains much of the main theory of pushdown automata as treated in the various introductory books on formal language the-

* Fall 2003. 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.

```
function Hanoi (n, s, t, v)
   if (n > 0)
   {  Hanoi (n − 1, s, v, t);
      movedisk (s, t);
      Hanoi (n − 1, v, t, s);
   }
```

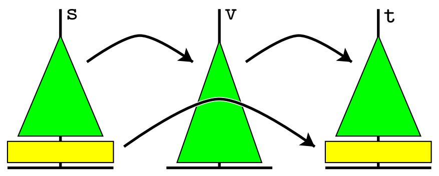

Fig. 1.1. Recursive solution to the Towers of Hanoi

ory, as [19, 15], or in the relevant handbook chapters [1] and [3]. We show in Section 1.3 the equivalence between context-free grammars and pushdown automata, and apply this to the family CF of context-free languages: we present as a consequence closure properties, normal forms, and a characterization of CF in terms of Dyck languages, all having a natural interpretation in terms of pushdown automata.

In Section 1.4 we study determinism for pushdown automata, just some highlights of an otherwise extensive theory [15]. Of course, we are happy to mention the decidability of the equivalence of deterministic pushdown automata, a result that was only recently obtained.

It is impossible to avoid overlap with the available introductory texts in the area, but we have managed to find a niche of our own, studying the language of pushdown stack contents during computations in Section 1.5. We apply the results to build 'predicting machines' [18, Section 12.3], automata that use knowledge about their stack contents to decide on the future steps in their computations. This approach, also called 'look-ahead on pushdowns', allows an elegant solution to several closure properties of (deterministic) context-free languages as was made explicit in [8]. Moreover, it can be applied to the theory of $LL(k)$ and $LR(k)$-grammars. In Section 1.6 we show how to build a deterministic pushdown automaton for a given $LR(k)$-grammar, in an abstract fashion, avoiding the introduction of the usual, more 'practical', item sets.

We close our presentation in Section 1.7 by a bird's eye view on some of the many machine models related to the pushdown automaton.

1.2 Basic Notions

A pushdown automaton, cf. Figure 1.2, is a finite state device equipped with a one-way input tape and with a pushdown stack which is a LIFO external memory that can hold unbounded amounts of information; each individual stack element contains finite information.

Definition 1. *A* pushdown automaton*,* pda *for short, is 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*.*

A *configuration* (or instantaneous description) of a pda $\mathcal{A}$ is given by the contents of the part of the input tape that has not been read, the current state, and the current contents 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.

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

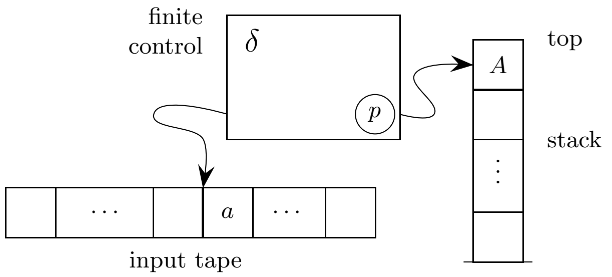

Fig. 1.2. Pushdown automaton

the pda —valid in state p, with a next on the input tape, and A as topmost symbol of the stack (as in Figure 1.2 for $a \in \Delta$)— 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., $(q, \alpha) \in \delta(p, a, A)$. 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.

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

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

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

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

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 1.4). 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 6.

Example 3. Consider the exchange language for the small euro coins, which has the alphabet $\Delta = \{\mathtt{1}, \mathtt{2}, \mathtt{5}, \mathtt{=}\}$:

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

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

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, $N(\mathcal{A}) = L_{ex}$. □

Note that in our previous 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 4. *Given a pda $\mathcal{A}$ one can effectively construct a pda $\mathcal{A}'$ such that $N(\mathcal{A}) = L(\mathcal{A}')$, and vice versa.*

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 $\mu \in \Gamma^*$, $\mu \neq \lambda$.*

1.3 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).

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 [5, 11, 25].

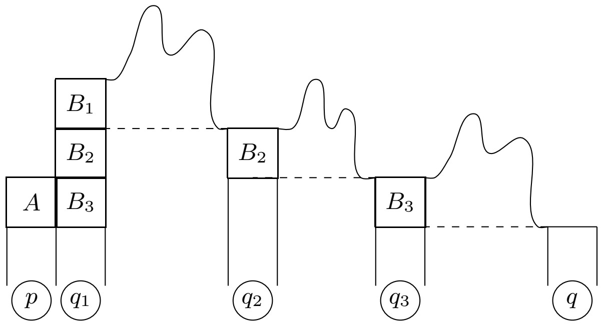

Fig. 1.3. Computation of type $[p, A, q]$

Theorem 6. *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 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 $[q_1, B_1, q_2]$, $[q_2, B_2, q_3], \ldots, [q_n, B_n, q]$, respectively, where $q_2, \ldots, q_n$ are intermediate states (cf. Figure 1.3 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 $q_2, \ldots, q_n$.

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 $q, q_2, \ldots, q_n \in Q$, $n \geq 1$,
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., $(x, p, A) \vdash_{\mathcal{A}}^{*} (\lambda, q, \lambda)$. □

Grammars with storage. In fact, this result can be generalized to context-free grammars *with storage.* As a simple example of such a type of storage, we return to the Hanoi function in the Introduction, and we consider context-free grammars where each nonterminal carries a natural number which is inherited by its children in a derivation, after decrementing the value by one. The axiom is initiated with an arbitrary number; on reaching zero only terminal symbols can be produced. For instance, the grammar with productions $S(n) \to S(n-1)S(n-1)$ for $n \geq 1$, and $S(0) \to a$ generates the (non-context-free) language $\{ a^{2^n} \mid n \in \mathbb{N} \}$.

An *excursion* of a pda is a computation of the form $(xz, p, A\gamma) \vdash_{\mathcal{A}}^{+} (z, q, B\gamma)$, where each intermediate stack is of the form $\mu\gamma$, $|\mu| \geq 2$. For a bounded excursion pda we put a fixed upper bound k on the number of excursions starting from any stack element. We can implement this restriction by assigning the number k to each newly introduced stack symbol, and decrementing it when the symbol is replaced, viz., instructions that push symbols back on the stack are of the form $(p, a, \langle A, i\rangle, q, \langle B_n, k\rangle \cdots \langle B_2, k\rangle\langle B_1, i-1\rangle)$, for $1 \leq i \leq k$.

Note that for every context-free grammar $G = (N, T, S, P)$ there is a bounded excursion (single state) pda $\mathcal{A} = (\{q\}, T, \Gamma, \delta, q, A_{in}, \varnothing)$ such that $L(G) = N(\mathcal{A})$. The stack alphabet Γ consists of all 'dotted productions' of G (also called 'items'), i.e., all productions of G with a position marked in their right-hand side: $\Gamma = \{ [A \to \beta \bullet \gamma] \mid A \to \beta\gamma \text{ in } P \}$. The initial stack symbol A_{in} is $[S \to \bullet S]$, where we assume w.l.o.g. that $S \to S$ is in P. The instructions of $\mathcal{A}$ in δ are of the form

- $(q, \lambda, [A \to \alpha \bullet B\gamma]), q, [B \to \bullet \beta][A \to \alpha B \bullet \gamma])$ 'expand'
- $(q, a, [A \to \alpha \bullet a\gamma], q, [A \to \alpha a \bullet \gamma])$ 'match'
- $(q, \lambda, [A \to \alpha \bullet], q, \lambda)$ 'reduce'

Clearly, the upper bound on the number of excursions of $\mathcal{A}$ is the maximal length of the right-hand sides of the productions of G.

With this terminology, context-free grammars with some storage type are equivalent to bounded excursion pda with stack symbols carrying the same storage, see [9, Theorem 6.3]. This generalizes a result of van Leeuwen that the family E0L (from which the above example is taken) equals the languages of so-called preset pushdown automata. If the storage type allows the identity operation on storage (as in the next example) then the bounded excursion restriction can be dropped [9, Corollary 5.21].

If the nonterminals of the context-free grammars themselves carry a pushdown stack, then we obtain the *indexed grammars*, see [19, Chapter 14.3]. These are equivalent to so-called nested stack automata, or following the general result of [9] to pushdown-of-pushdowns automata. This process can be iterated and leads to Maslov's hierarchy of language families generated by k-iterated indexed grammars and recognized by k-iterated pushdown automata, which starts with REG, CF, and the indexed languages (for $k = 0, 1, 2$, respectively); see, e.g., [7, 8].

Applications to CF. 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. In its state the new pda keeps track of the state of both machines, the 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.

Another closure application of the main equivalence we treat explicitly.

Lemma 7. 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 6 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 $h(b) \in \Delta^*$. The simulated input string $h(b)$ is temporarily stored in a 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 $\mathrm{Buf} = \{w \in \Delta^* \mid w \text{ is a suffix of } h(b) \text{ for some } b \in \Sigma\}$. 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. □

Theorem 8 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 6 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 4, 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. This feature, presented under the name *predicting machines* in [18, 19], is treated in Section 1.5.

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.

A *transducer* $\mathcal{A} = (Q, \Delta_1, \Delta_2, \delta, q_{in}, F)$ is a finite state automaton with both input and output tape, with tape alphabet Δ_1 and Δ_2 respectively. Transitions in δ are of the form $\langle p, u, v, q\rangle$, where $p, q \in Q$ are states, $u \in \Delta_1^*$, and $v \in \Delta_2^*$. With computations from initial state q_{in} to final state in F as usual, these machines define a *rational relation* $\tau_{\mathcal{A}} \subseteq \Delta_1^* \times \Delta_2^*$ rather than a language. 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: $h(\langle p, u, v, q\rangle) = u$, $g(\langle p, u, v, q\rangle) = v$.

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 $B_1, \dots,$ 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 [14]:

Theorem 8. $\mathsf{CF} = \widehat{\mathcal{M}}(D_{\{A,B\}})$*, 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.

1.4 Deterministic Pushdown Automata

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 9. *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* $(p, \lambda, A, q, \alpha)$ *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 $(p, \lambda, A, q, \alpha)$ 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 3 is deterministic.

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 4) can be made to work in the deterministic case; the converse construction can easily be adapted for prefix-free languages.

Lemma 10. *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 DCF. The strict inclusion $\mathsf{REG} \subset \mathsf{DCF}$ 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 $L_{\mathrm{pal}} = \{ x \in \{a,b\}^* \mid x = x^{\mathrm{R}} \}$, 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 DCF do not share the same closure properties (as opposed to using some kind of pumping property).

For a language L we define $\mathrm{pre}(L) = \{xy \mid x \in L, xy \in L, y \neq \lambda\}$, in other words, $\mathrm{pre}(L)$ 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 $\mathrm{pre}(L_d) = \{ a^n b a^m b a^n \mid m \geq n \geq 1 \}$.

Lemma 11. DCF *is closed under* pre.

Proof. Let $\mathcal{A} = (Q, \Delta, \Gamma, \delta, q_{in}, A_{in}, F)$ be a deterministic pda. The new deterministic pda $\mathcal{A}' = (Q', \Delta, \Gamma, \delta', q'_{in}, A_{in}, F')$ with $L(\mathcal{A}') = \mathrm{pre}(L(\mathcal{A}))$ 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'.

□

As an immediate consequence we have the strict inclusion $\mathsf{DCF} \subset \mathsf{CF}$, and in fact it follows that the language L_d above is an element of the difference $\mathsf{CF} - \mathsf{DCF}$. Additionally we see that DCF 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 $\min(L) = L - \mathrm{pre}(L)$ 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 12. DCF *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 DCF: otherwise, $L_d = L_{\mathrm{pal}} \cap (a^+ba^+ \cup a^+ba^+ba^+)$ would also be in DCF. 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 DCF 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 listed in Chapter **??**. 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 [26]. 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 [15].

The decidability of the equivalence of k-iterated pda's remains open.

1.5 Predicting Machines

In this section we study the behaviour of the pda with respect to the stack rather than to the input. It leads to a powerful technique where the pda $\mathcal{A}$ is able to make decisions on the continuation of its computation based on whether any other given pda (usually closely related to $\mathcal{A}$) is able to empty the current stack of $\mathcal{A}$. This works even when $\mathcal{A}$ is deterministic, while the other pda is nondeterministic.

Let $\mathcal{A} = (Q, \Delta, \Gamma, \delta, q_{in}, A_{in}, F)$ be a pda. The *store language* of $\mathcal{A}$ is defined as

$$\mathrm{SN}(\mathcal{A}) = \{ \alpha \in \Gamma^* \mid (x, q_{in}, \alpha) \vdash_{\mathcal{A}}^* (\lambda, q, \lambda) \text{ for some } x \in \Delta^* \text{ and } q \in Q \},$$

i.e., consisting of stacks that can be completely popped when a suitable input is given to the pda.

Note that the string $B_1 B_2 \cdots B_n$ belongs to the store language if the symbols B_1 to B_n can be consecutively popped from the stack. Hence, we are in a situation similar to the construction of a context-free grammar for a given pda. So, $B_1 B_2 \cdots B_n \in \mathrm{SN}(\mathcal{A})$ iff there exist states $q_1, q_2, \ldots, q_n, q_{n+1}$ such that $q_1 = q_{in}$ and for each $1 \leq i \leq n$, $(x, q_i, B_i) \vdash_{\mathcal{A}}^* (\lambda, q_{i+1}, \lambda)$ for some $x \in \Delta^*$. This means that we can build a finite state automaton for $\mathrm{SN}(\mathcal{A})$ using the states of $\mathcal{A}$ (initial state q_{in}, all states final) and adding an edge from state p to q with label B iff $(x, p, B) \vdash_{\mathcal{A}}^* (\lambda, q, \lambda)$ for some $x \in \Delta^*$. This is equivalent to requiring $[p, B, q] \Rightarrow_G^* x$, where $[p, B, q]$ is a nonterminal of G as introduced in the proof of Theorem 6. Note that this property of nonterminal symbols (is the symbol *productive*, i.e., does it derive a terminal string?) is decidable for context-free grammars.

Thus, we have a rather surprising result for store languages [12].

Lemma 13. *For each pda $\mathcal{A}$ the language $\mathrm{SN}(\mathcal{A})$ is regular, and can be effectively constructed from $\mathcal{A}$.*

This result has several extensions that are easily seen to follow. We may additionally require (in the definition of $\mathrm{SN}(\mathcal{A})$) that the string x read belongs to R for a given regular R or that the last state q belongs to the set F of final states. For these extensions consider a new pda that simulates both $\mathcal{A}$ and a finite state automaton for R (accepting the intersection of their languages, cf. Section 1.3), or a new pda that has a new bottom-of-stack symbol that can be popped only in states in F.

Also, we may be interested in the reverse process, asking for stacks that can be pushed during computations, rather than those that can be popped, i.e., we may wish to use

$$\mathrm{SF}(\mathcal{A}) = \{ \alpha \in \Gamma^* \mid (x, q_{in}, A_{in}) \vdash_{\mathcal{A}}^* (\lambda, q, \alpha) \text{ for some } x \in \Delta^* \text{ and } q \in F \}.$$

Basically this follows by considering the pda that simulates $\mathcal{A}$ in reverse, interchanging pops and pushes.

A direct application of this result is to Büchi's *regular canonical systems* [4]. Such a system is similar to a type-0 (unrestricted) Chomsky grammar, where the productions may only be applied to a prefix of the string. This means that the derivation process is much like the LIFO behaviour of the pushdown stack. A production $\alpha \to \beta$ applied to the prefix α of a string can be simulated by a pda $\mathcal{A}$ on its stack in a sequence of steps, popping α and pushing β. As a consequence, by taking only those stacks that appear in between these simulation sequences, i.e., by using an appropriate F in $\mathrm{SF}(\mathcal{A})$, the set of strings obtained by prefix rewriting starting from a given initial string forms a regular language.

A second application is in the theory of pushdown automata itself. A pda can make decisions on the continuation of its computation by inspecting the topmost symbol of its stack. In some circumstances it is convenient to have the possibility to inspect the stack in some stronger way, to answer questions like: 'with the present stack will the pda be able to read the next symbol, or will it diverge into an infinite λ-computation?'; this question is particularly important in connection with the closure of DCF under complementation.

As a consequence of Lemma 13, many relevant queries are actually of the type 'is $\alpha \in R$?', where $\alpha = B_n \cdots B_2 B_1$ is the current stack (for convenience we have numbered the symbols bottom-up), and R is a fixed regular language. We show that by adding suitable data to the stack it is possible to keep track of this information while the stack grows and shrinks.

For fixed regular R as above, we consider a deterministic finite state automaton $\mathcal{A}_R$ with state set Q_R for the reverse $\{x^{\mathrm{R}} \mid x \in R\}$ of R. We extend the stack alphabet Γ of the pda under consideration to $\Gamma \times Q_R$, and we replace the stack $B_n \cdots B_2 B_1$ by $\langle B_n, q_n\rangle \cdots \langle B_2, q_2\rangle\langle B_1, q_1\rangle$, where q_i is the state assumed by $\mathcal{A}_R$ on input $B_1 B_2 \cdots B_i$, a prefix of the reverse of the stack. Of course, $\alpha \in R$ iff q_n is a final state, and this can be decided by inspecting the top of the new stack. Obviously, after popping the stack the relevant state of $\mathcal{A}_R$ is again available, and push instructions can be adapted to contain the new state information, i.e., the simulation of $\mathcal{A}_R$ on the symbols pushed. Note that this construction preserves determinism of the pda.

This technique can, for instance, be used to avoid infinite computations and to signal acceptance by the first state after reading the input — typical technical problems in considerations on deterministic pda's [25]. Thus we obtain the following normal form for deterministic pda's.

Lemma 14. *For each deterministic pda we can construct an equivalent deterministic pda that can read every input string and has no λ-instructions entering final states.*

Proof. We assume that the given pda $\mathcal{A}$ never empties its stack, cf. the discussion preceding Lemma 4. To obtain the second property, we adapt $\mathcal{A}$ as follows. Just before executing a non-λ-instruction $\rho = (p, a, A, q, \alpha)$, first query the stack to see whether after execution of ρ it is possible to reach a final state using λ-instructions only (including the case when q is final). If not, ρ is executed. Otherwise, the new instruction $(p, a, A, \bar{q}, \alpha)$ is executed, where $\bar{q}$ is a new final state with instructions $(\bar{q}, \lambda, B, q, B)$ for all B. The only final states of the adapted pda are the barred ones.

It remains to show that the test on the stack contents is of the regular type discussed above. We do this by indicating how to construct the pda $\mathcal{A}_\rho$, for each non-λ-instruction ρ of $\mathcal{A}$, for which the store language $\mathrm{SN}(\mathcal{A}_\rho)$ is the regular language we are looking for.

Let $\mathcal{A}_\rho$ be a copy of $\mathcal{A}$ with a new initial state s and a new instruction (s, a, A, q, α). We remove all non-λ-instructions (except the one for s), and we add λ-instructions to empty the stack for each final state. Obviously, $\mathcal{A}_\rho$ empties a given initial stack $A\gamma$ iff $\mathcal{A}$ reaches a final state using λ-instructions only, starting in state q with initial stack $\alpha\gamma$. This is the query we want to make.

To obtain the first property, we may clearly assume that if $\mathcal{A}$ has no instruction of the form $(p, \lambda, A, -, -)$ then it has an instruction of the form $(p, a, A, -, -)$ for every input symbol a: just introduce a (non-final) 'garbage' state g that reads the remainder of the input. The only other reason that $\mathcal{A}$ might not read its entire input is that it may get stuck in an infinite computation with λ-instructions only. To avoid this, we always query the stack whether it is possible to reach a non-λ-instruction, i.e., whether it is possible to read another symbol from the tape. If not, we move to garbage state g, as it is of no use to continue. $\square$

Predicting techniques like this (or look-ahead on pushdowns [8]) lead to the closure of DCF under complementation. In fact, to construct a deterministic pda $\mathcal{A}_c$ that accepts the complement of $L(\mathcal{A})$, where $\mathcal{A} = (Q, \Delta, \Gamma, \delta, q_{in}, A_{in}, F)$ is in the normal form of Lemma 14, change every instruction (p, a, A, q, α) of $\mathcal{A}$ with $a \in \Delta$ and $q \notin F$ into the instructions $(p, a, A, \tilde{q}, \alpha)$ and $(\tilde{q}, \lambda, B, q, B)$ for all $B \in \Gamma$; the state set and set of final states of $\mathcal{A}_c$ are $Q_c = Q \cup F_c$ and $F_c = \{\tilde{q} \mid q \notin F\}$.

Also, as originally shown in [18, 19], one can show the closure under right quotient with a regular language R in a deterministic fashion by inspecting the stack in each current state p, to see whether the pda has an accepting continuation for the current stack on input from R. The query language $\mathrm{SN}(\mathcal{A}')$

is obtained from the pda $\mathcal{A}$ under consideration by changing the initial state to p, intersecting its language with R, and emptying the stack in each final state. Closure under max can be shown in a similar way.

Using Lemma 14 it is easy to show that if $L \in \mathsf{DCF}$ then $L\$ \in \mathsf{DCF}$, where \$ is a new symbol. The reverse implication holds by the closure of DCF under right quotient with $\{\$\}$. This shows that, in the model of the deterministic pda, we may provide the input tape with an endmarker without changing the expressive power of the model.

1.6 $LR(k)$ Parsing

The *expand-match* construction from Section 1.3 yields a pda for an arbitrary given context-free grammar G. We can use the resulting pda $\mathcal{A}$ as a recognizer to verify that a given string x belongs to the language of G, or as a parser of G once $\mathcal{A}$ is equipped with proper output facilities. As we have seen, $\mathcal{A}$ simulates the leftmost derivations of G, following the nodes in the derivation tree in pre-order. This is a top-down approach to recognition, starting with the axiom S and choosing productions, working towards the terminal string x. In general it is a nondeterministic process, which needs backtracking or clever guesswork. For some families of grammars however, the productions that have to be chosen in the derivation can be predicted on the basis of a *look-ahead* on the input x, a fixed window of k symbols not yet read. This has lead to the study of suitable classes of context-free grammars, most notably the $LL(k)$-grammars.

Another approach for building recognizers works bottom-up, trying to reconstruct the derivation tree of x by reducing x to S, i.e., by applying the productions backwards. This leads to the *shift-reduce* construction of a pda for G, where we use an extended model for the pda, popping an arbitrary number of symbols at a time, and starting with the empty stack. Moreover, we now assume the top of the stack *to the right*.

- (q, a, λ, q, a) for each terminal a 'shift'
- $(q, \lambda, \alpha, q, A)$ for each production $A \to \alpha$ 'reduce'

Now these instructions correspond to *rightmost* derivations of the grammar, reconstructed *backwards*. Formally, for $x \in T^*$ and $\alpha, \beta \in (N \cup T)^*$, if $(x, q, \alpha) \vdash_{\mathcal{A}}^{*} (\lambda, q, \beta)$ then $\beta \Rightarrow_{G,r}^{*} \alpha y$. The reverse implication is valid for $\alpha \in (N \cup T)^* N \cup \{\lambda\}$, where according to the convention on the direction of the stack, the last nonterminal of α is the topmost symbol of the stack.

Again, using look-ahead on the input, this can be made into a deterministic process for suitable classes of grammars. We present here a little theory of $LR(k)$-grammars, following [18], which means we omit the customary construction of 'item sets'. Moreover, we use the results of Section 1.5.

Let $G = (N, T, S, P)$ be a context-free grammar. In order to simplify notation we pad the end of the input string by a sequence of \$'s, which is a new symbol that we add to T. Let $k \in \mathbb{N}$ be a fixed natural number.

Formally, G is an *$LR(k)$ grammar* if

- $S\$^k \Rightarrow^*_{G,r} \alpha A x_1 x_2 \Rightarrow_{G,r} \alpha\beta x_1 x_2$, and
- $S\$^k \Rightarrow^*_{G,r} \gamma B x \Rightarrow_{G,r} \alpha\beta x_1 x_3$

imply that $\alpha = \gamma$, $A = B$, and $x = x_1 x_3$, for all $A, B \in N$, $x_1, x_2, x_3, x \in T^*$ with $|x_1| = k$, $\alpha, \beta, \gamma \in (N \cup T)^*$, and $A \to \beta \in P$.

Intuitively this means that if $\alpha\beta$ is on the stack of the shift-reduce pda $\mathcal{A}$ for G and the first k remaining symbols on the input tape (the look-ahead) form x_1, then $\mathcal{A}$ necessarily has to reduce the production $A \to \beta$. Thus, the instruction that $\mathcal{A}$ has to execute next is uniquely determined. But how can $\mathcal{A}$ determine that instruction? Answer: by querying its stack.

Motivated by the definition above, for a production $A \to \beta$ and a terminal string x_1 of length k let $R(A \to \beta, x_1)$ consist of all situations in which a reduction from β back to A can occur in G with look-ahead x_1; more precisely, $R(A \to \beta, x_1)$ contains the strings of the form $\alpha\beta x_1$ for which there exists a derivation $S\$^k \Rightarrow^*_{G,r} \alpha A x_1 x_2 \Rightarrow_{G,r} \alpha\beta x_1 x_2$ for some $x_2 \in T^*$.

Our first claim is that these sets are regular, which makes recognition of reduction sites feasible as a finite state process.

Lemma 15. *$R(A \to \beta, x_1)$ is a regular set, effectively constructible from G.*

Proof. Rightmost derivations are turned into leftmost derivations by considering all sentential forms in reverse, while also reversing every right-hand side of productions in the grammar. These leftmost derivations can be executed by a pda $\mathcal{A}$, nonterminals represented on its stack, using the expand-match construction. However, $\mathcal{A}$ should also keep the k most recently matched terminals in its state and check that they equal x_1 when $\mathcal{A}$ halts (which is just after expanding $A \to \beta$). Thus, by Lemma 13 for the variant SF($\mathcal{A}$), these stacks form a regular language. □

It is a straightforward exercise to characterize the $LR(k)$ property in terms of these sets: if $R(A \to \beta, x_1)$ and $R(A' \to \beta', x_1')$ contain strings, one a prefix of the other, then this must imply that these strings are equal, and moreover that the productions $A \to \beta$ and $A' \to \beta'$ are equal. As this can easily be tested effectively, we have the immediate corollary that, for given k, the $LR(k)$ property is decidable for context-free grammars.

Finally, we build the deterministic recognizer-parser for a given $LR(k)$ grammar. The approach is somewhat abstract, as we assume the regular languages $R(A \to \beta, x_1)$ to be represented by their finite state automata. In practice these automata have to be found explicitly. Usually their states are represented by so-called *item sets*, each item consisting of a dotted production together with a look-ahead string of length k. Then the stack contains the usual nonterminal symbols but interleaved with item sets to give information on which reduction to choose. This is a solution equivalent to the one based on the notion of predicting machines which we use here.

Lemma 16. *If G is an $LR(k)$ grammar, then $L(G) \in \mathsf{DCF}$.*

Proof. For convenience we again reverse the stack, and write its top to the right. Given an $LR(k)$ grammar G, we construct a deterministic pda $\mathcal{A}$ for $L(G)\$^k$. The result for $L(G)$ follows from the closure of DCF under quotient with regular languages. The pda $\mathcal{A}$ is similar to the shift-reduce pda for G, but stores the look-ahead symbols on its stack.

First, $\mathcal{A}$ shifts k symbols from the input to its stack. Then it repeats the following steps.

If there exists a production $A \to \beta$ and a look-ahead x_1 of length k such that the top of the stack is of the form βx_1, and the stack itself belongs to $R(A \to \beta, x_1)$, then the reduction defined by the production is applicable, and the topmost βx_1 is replaced by Ax_1. Note that by Section 1.5 we may assume that $\mathcal{A}$ can test this property of its stack. By the $LR(k)$ property, at most one production can be reduced.

Otherwise, if no reductions are applicable, then another input symbol is shifted to the stack. The pda accepts when its stack assumes the value $S\$^k$, i.e., when it has completely reduced the input. $\square$

Since the standard construction that converts a pda into a context-free grammar can be shown to yield $LR(1)$ grammars for deterministic pda's (under some additional precautions), we obtain the following grammatical characterization of DCF [20].

Theorem 17. *A context-free language is deterministic iff it has an $LR(1)$ grammar iff it has an $LR(k)$ grammar for some $k \geq 1$.*

1.7 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. [15, 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 3. 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 [19, Theorem 7.9] for further details.

Blind and partially blind counters. A counter is *blind* if it cannot be tested for zero [13]. 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 multicounter 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 $B_k = \{x \in \Sigma_k^* \mid |x|_{a_i} = |x|_{b_i} \text{ for each } 1 \le i \le k\}$. 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 8, 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 [17] 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 1.3) 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 [15, 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 [22]. 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 [21] for a historical account.

Finally, multi-head pda's, either deterministic or non-deterministic (!), characterize the family P of languages recognizable in deterministic polynomial time. An introduction to automata theoretic complexity is given in [19, Chapter 14], while more results are collected in [27, Sections 13 and 20.2]. Multi-head k-iterated pda's characterize the deterministic $(k-1)$-iterated exponential time complexity classes [7].

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.

Final Pop. In a recent edition of the conference Developments in Language Theory [10] we find (at least) three contributions that feature pda's and variants. *Restarting automata* are finite state automata that model reduction techniques from linguistics. In [24] an overview of the theory is given. Certain subclasses of restarting automata recognize DCF, and have connections to $LR(0)$ grammars. *Distributed pushdown automata systems* consisting of several pda's that work in turn on the input string placed on a common one-way input tape are introduced in [6]. Finally, in [16] *flip-pushdown automata* are studied, pda's that may 'flip' their stack, bringing the bottom up. Even after forty years the pushdown automaton still proves to be a versatile tool!

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.
2. J. Berstel. *Transductions and Context-Free Languages.* Teubner Studienbücher, Stuttgart, 1979.
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.
4. J.R. Büchi. Regular Canonical Systems. Arch. Math. Logik Grundlagenforschung, 6 (1964) 91–111.
5. N. Chomsky. Context Free Grammars and Pushdown Storage. Quarterly Progress Report, Vol. 65, MIT Research Laboratory in Electronics, Cambridge, MA, 1962.

6. E. Csuhaj-Varjú, V. Mitrana, G. Vaszil. Distributed Pushdown Automata Systems: Computational Power. In: [10], pages 218–229.
7. J. Engelfriet. Iterated Stack Automata. Information and Computation, 95 (1991) 21–75.
8. J. Engelfriet, H. Vogler. Look-Ahead on Pushdowns. Information and Computation, 73 (1987) 245–279.
9. J. Engelfriet, H. Vogler. Pushdown Machines for the Macro Tree Transducer. Theoretical Computer Science, 42 (1986) 251–368.
10. Z. Ésik, Z. Fülöp (Eds.). *Developments in Language Theory*, 7th International Conference, DLT 2003, Proceedings. Lecture Notes in Computer Science, Vol. 2710, Springer 2003.
11. J. Evey. Application of Pushdown Store Machines. Proceedings of the 1963 Fall Joint Computer Conference, Montreal, AFIPS Press, 1963.
12. S. Greibach. A Note on Pushdown Store Automata and Regular Systems. Proceedings of the American Mathematical Society, 18 (1967) 263–268.
13. S. Greibach. Remarks on Blind and Partially Blind One-way Multicounter Machines. Theoretical Computer Science, 7 (1978) 311–324.
14. S. Ginsburg. *Algebraic and Automata-theoretic Properties of Formal Languages*. Fundamental Studies in Computer Science, Vol. 2, North-Holland, 1975.
15. M.A. Harrison. *Introduction to Formal Language Theory*. Addison-Wesley, Reading, Mass., 1978.
16. Markus Holzer, Martin Kutrib. Flip-Pushdown Automata: Nondeterminism is Better than Determinism. In: [10], pages 361–372.
17. 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.
18. J. Hopcroft, J. Ullman. *Formal Languages and their Relation to Automata*. Addison-Wessley, Reading, Mass., 1969.
19. J. Hopcroft, J. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Reading, MA: Addison-Wesley, 1979.
20. D.E. Knuth. On the Translation of Languages from Left to Right. Information and Control, 8 (1965) 607–639.
21. D.E. Knuth, J.H. Morris, V.R. Pratt. Fast Pattern Matching in Strings. SIAM Journal on Computing, 6 (1977) 323–360.
22. R.E. Ladner, R.J. Lipton, L.J. Stockmeyer. Alternating Pushdown and Stack Automata. SIAM Journal on Computing 13 (1984) 135–155.
23. A.G. Oettinger. Automatic Syntactic Analysis and the Pushdown Store. Proceedings of Symposia on Applied Mathematics, Vol. 12, Providence, RI, American Mathematical Society, 1961.
24. F. Otto. Restarting Automata and Their Relations to the Chomsky Hierarchy. In: [10], pages 55–74.
25. M. Schützenberger. On Context Free Languages and Pushdown Automata. Information and Control, 6 (1963) 246–264.
26. G. Sénizergues. $L(A) = L(B)$? A Simplified Decidability Proof. Theoretical Computer Science, 281 (2002) 555–608.
27. K. Wagner, G. Wechsung. *Computational Complexity*. Reidel, Dordrecht, 1986.