An innovative finite-state concept for recognition and parsing of context-free languages[1]

Eberhard Bertsch
Ruhr University, Bochum
Germany
bertsch@lpi.ruhr-uni-bochum.de

Mark-Jan Nederhof[2]
University of Groningen
The Netherlands
markjan@let.rug.nl

Abstract. We recall the notion of regular closure of classes of languages. We present two important results. The first one is that all languages which are in the regular closure of the class of deterministic (context-free) languages can be recognized in linear time. This is a non-trivial result, since this closure contains many inherently ambiguous languages.

The second one is that the class of deterministic languages is contained in the closure of the class of deterministic languages with the prefix property, or stated in an equivalent way, all LR(k) languages are in the regular closure of the class of LR(0) languages.

1 INTRODUCTION

In a series of recent articles [3, 9], the authors have studied recognition and parsing of context-free languages by means of previously unknown simulations of nondeterministic techniques. The motivation for this work came from error detection problems, and as a matter of fact an open problem of long standing could be solved in that area.

Most notably, a core concept that turned out to be helpful in the course of this research can be interpreted as a two-level parser whose upper level is a finite automaton with nonterminal labels at its edges and whose lower level consists of languages associated to each such label.

If all lower-level languages are assumed to be deterministic, the class of languages characterized in this new way can be shown to be parsable in linear time. This constitutes a genuine surprise because some of the languages included are not deterministic, in fact inherently ambiguous. Furthermore, even if languages at the lower level are restricted to the properly smaller LR(0) class, the language-generating capability of our two-level devices stays the same.

Natural-language parsing cannot be implemented by *exclusive* use of deterministic techniques, since many constructs in natural languages are inherently nondeterministic. A consequence of our findings is that this fact does not necessarily preclude the possiblity of natural-language parsing in linear time.

[1] The full version of this paper is available as: Regular closure of deterministic languages. Bericht Nr. 186, Fakultät für Mathematik, Ruhr-Universität Bochum, August 1995. It has been accepted at SIAM Journal on Computing.

[2] Supported by the Dutch Organization for Scientific Research (NWO), under grant 305-00-802.

Proceedings of the ECAI 96 Workshop
Extended Finite State Models of Language
Edited by A. Kornai.

The article may be outlined as follows: After introducing the definitional framework in Section 2, we show that all deterministic context-free languages can be constructed from prefix-free deterministic languages by regular operations (Section 3). This follows from a detailed decomposition of pushdown computations into sequences of moves that do not discard more than one element of the stack they started with.

The most important proposition of Section 4 is that each deterministic pushdown automaton can be transformed into an equivalent one which is "loop-free". This requires a fairly deep discussion of individual pushing and popping moves. In essence, automata are changed in such a way that the stack contents after certain moves must reflect the *amount* of processed input. Using tabulation, the transformed pushdown automata can be simulated at all input positions in linear time. This result is part of a proof of the fact that languages in the regular closure of the deterministic languages can be recognized in linear time, by means of a two-level device, to be defined in Section 5. An "on-line" variant of this device is presented in Section 6.

Although the concept of parse tree is less immediate for the new kind of language description than for ordinary grammars, we are able to sketch an efficient transduction procedure yielding representations of the syntactic structure of given inputs (Section 7).

An application in pattern matching is described in Section 8.

2 NOTATION

A finite automaton $\mathcal{F}$ is a 5-tuple (S, Q, q_s, F, T), where S and Q are finite sets of input symbols and states, respectively; $q_s \in Q$ is the *initial* state, $F \subseteq Q$ is the set of *final* states; the transition relation T is a subset of $Q \times S \times Q$.

An input $b_1 \cdots b_m \in S^*$, is *recognized* by the finite automaton if there is a sequence of states $q_0, q_1, \ldots, q_m$ such that $q_0 = q_s$, $(q_{k-1}, b_k, q_k) \in T$ for $1 \leq k \leq m$, and $q_m \in F$. For a certain finite automaton $\mathcal{F}$, the set of all such strings w is called the language *accepted by* $\mathcal{F}$, denoted $L(\mathcal{F})$. The languages accepted by finite automata are called the *regular* languages.

In the following, we describe a type of pushdown automaton without internal states and with very simple kinds of transition. This is a departure from the standard literature

but considerably simplifies our definitions in the remainder of the paper. The generative capacity of this type of pushdown automaton is not affected with respect to any of the more traditional types.

Thus, we define a pushdown automaton (PDA) $\mathcal{A}$ to be a 5-tuple $(\Sigma, \Delta, X_{initial}, F, T)$, where Σ, Δ and T are finite sets of input symbols, stack symbols and transitions, respectively; $X_{initial} \in \Delta$ is the *initial* stack symbol, $F \subseteq \Delta$ is the set of *final* stack symbols.

We consider a fixed input string $a_1 \cdots a_n \in \Sigma^*$. A *configuration* of the automaton is a pair (δ, v) consisting of a stack $\delta \in \Delta^*$ and the remaining input v, which is a suffix of the original input string $a_1 \cdots a_n$.

The *initial* configuration is of the form $(X_{initial}, a_1 \cdots a_n)$, where the stack is formed by the initial stack symbol $X_{initial}$. A *final* configuration is of the form $(\delta X, \epsilon)$, where the element on top of the stack is some final stack symbol $X \in F$.

The transitions in T are of the form $X \overset{z}{\mapsto} XY$, where $z = \epsilon$ or $z = a$, or of the form $XY \overset{\epsilon}{\mapsto} Z$.

The application of such a transition $\delta_1 \overset{z}{\mapsto} \delta_2$ is described as follows. If the top-most symbols on the stack are δ_1, then these may be replaced by δ_2, provided either $z = \epsilon$, or $z = a$ and a is the first symbol of the remaining input. If $z = a$ then furthermore a is removed from the remaining input.

Formally, for a fixed PDA we define the binary relation $\vdash$ on configurations as the least relation satisfying $(\delta\delta_1, v) \vdash (\delta\delta_2, v)$ if there is a transition $\delta_1 \overset{\epsilon}{\mapsto} \delta_2$, and $(\delta\delta_1, av) \vdash (\delta\delta_2, v)$ if there is a transition $\delta_1 \overset{a}{\mapsto} \delta_2$.

In the case that we consider more than one PDA at the same time, we use symbols $\overset{z}{\mapsto}_{\mathcal{A}}$ and $\vdash_{\mathcal{A}}$ instead of $\overset{z}{\mapsto}$ and $\vdash$ if these refer to one particular PDA $\mathcal{A}$.

The recognition of a certain input v is obtained if starting from the initial configuration for that input we can reach a final configuration by repeated application of transitions, or, formally, if $(X_{initial}, v) \vdash^* (\delta X, \epsilon)$, with some $\delta \in \Delta^*$ and $X \in F$, where $\vdash^*$ denotes the reflexive and transitive closure of $\vdash$ (and $\vdash^+$ denotes the transitive closure of $\vdash$). For a certain PDA $\mathcal{A}$, the set of all such strings v which are recognized is called the language *accepted by* $\mathcal{A}$, denoted $L(\mathcal{A})$. A PDA is called *deterministic* if for all possible configurations at most one transition is applicable. The languages accepted by deterministic PDAs (DPDAs) are called *deterministic languages*.

We may restrict deterministic PDAs such that no transitions apply to final configurations, by imposing $X \notin F$ if there is a transition $X \overset{z}{\mapsto} XY$, and $Y \notin F$ if there is a transition $XY \overset{\epsilon}{\mapsto} Z$. We call such a DPDA *prefix-free*. The languages accepted by such deterministic PDAs are obviously *prefix-free*, which means that no string in the language is a prefix of any other string in the language. Conversely, any prefix-free deterministic language is accepted by some prefix-free DPDA, the proof being that in a deterministic DPDA, all transitions of the form $X \overset{z}{\mapsto} XY$, $X \in F$, and $XY \overset{\epsilon}{\mapsto} Z$, $Y \in F$, can be removed without consequence to the accepted language if this language is prefix-free.

In compiler design, the deterministic languages are better known as LR(k) languages, and the prefix-free deterministic languages as LR(0) languages [5].

A prefix-free DPDA is in normal form if, for all input v, $(X_{initial}, v) \vdash^* (\delta X, \epsilon)$, with $X \in F$, implies $\delta = \epsilon$, and furthermore F is a singleton $\{X_{final}\}$. Any prefix-free DPDA can be put into normal form. We define a *normal PDA* (NPDA) to be a prefix-free deterministic PDA in normal form.

We define a subrelation $\models^+$ of $\vdash^+$ as: $(\delta, vw) \models^+ (\delta\delta', w)$ if and only if $(\delta, vw) = (\delta, z_1 z_2 \cdots z_m w) \vdash (\delta\delta_1, z_2 \cdots z_m w) \vdash \ldots \vdash (\delta\delta_m, w) = (\delta\delta', w)$, for some $m \geq 1$, where $|\delta_k| > 0$ for all k, $1 \leq k \leq m$. Informally, we have $(\delta, vw) \models^+ (\delta\delta', w)$ if configuration $(\delta\delta', w)$ can be reached from (δ, vw) without the bottom-most part δ of the intermediate stacks being affected by any of the transitions; furthermore, at least one element is pushed on top of δ. Note that $(\delta_1 X, vw) \models^+ (\delta_1 X\delta', w)$ implies $(\delta_2 X, vw') \models^+ (\delta_2 X\delta', w')$ for any δ_2 and any w', since the transitions do not address the part of the stack below X, nor read the input following v.

3 META-DETERMINISTIC LANGUAGES

In this section we define a new sub-class of the context-free languages, which results from combining deterministic languages by the operations used to specify regular languages.

We first define the concept of *regular closure* of a class of languages.[3] Let $\mathcal{L}$ be a class of languages. The regular closure of $\mathcal{L}$, denoted $C(\mathcal{L})$, is defined as the smallest class of languages such that:

- $\emptyset \in C(\mathcal{L})$,
- if $l \in \mathcal{L}$ then $l \in C(\mathcal{L})$,
- if $l_1, l_2 \in C(\mathcal{L})$ then $l_1 l_2 \in C(\mathcal{L})$,
- if $l_1, l_2 \in C(\mathcal{L})$ then $l_1 \cup l_2 \in C(\mathcal{L})$, and
- if $l \in C(\mathcal{L})$ then $l^* \in C(\mathcal{L})$.

Note that a language in $C(\mathcal{L})$ may be described by a regular expression over symbols representing languages in $\mathcal{L}$.

Let $\mathcal{D}$ denote the class of deterministic languages. Then the class of *meta-deterministic* languages is defined to be its regular closure, $C(\mathcal{D})$. This class is obviously a subset of the class of context-free languages, since the class of context-free languages is closed under concatenation, union and Kleene star, and it is a *proper* subset, since, for example, the context-free language $\{ww^R \mid w \in \{a, b\}^*\}$ is not in $C(\mathcal{D})$. (w^R denotes the mirror image of w.)

Finite automata constitute a computational representation for regular languages; DPDAs constitute a computational representation for deterministic languages. By combining these two mechanisms we obtain the meta-deterministic automata, which constitute a computational representation for the meta-deterministic languages.

Formally, a meta-deterministic automaton $\mathcal{M}$ is a triple $(\mathcal{F}, A, \mu)$, where $\mathcal{F} = (S, Q, q_s, F, T)$ is a finite automaton, A is a finite set of deterministic PDAs with identical alphabets Σ, and μ is a mapping from S to A.

The language accepted by such a device is composed of languages accepted by the DPDAs in A according to the transitions of the finite automaton $\mathcal{F}$. Formally, a string v is *recognized by* automaton $\mathcal{M}$ if there is some string $b_1 \cdots b_m \in S^*$, a sequence of PDAs $\mathcal{A}_1, \mathcal{A}_2, \ldots, \mathcal{A}_m \in A$, and a sequence of strings $v_1, \ldots, v_m \in \Sigma^*$ such that

- $b_1 \cdots b_m \in L(\mathcal{F})$,
- $\mathcal{A}_k = \mu(b_k)$, for $1 \leq k \leq m$,
- $v_k \in L(\mathcal{A}_k)$, for $1 \leq k \leq m$, and
- $v = v_1 \cdots v_m$.

[3] This notion was called *rational* closure in [2].

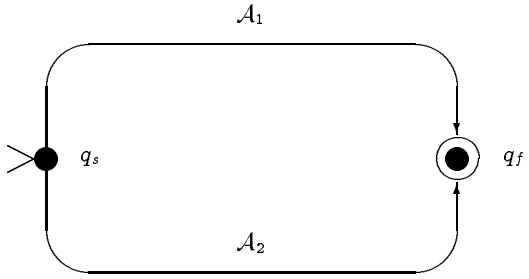

Figure 1. A meta-deterministic automaton

The set of all strings recognized by automaton $\mathcal{M}$ is called the language *accepted by* $\mathcal{M}$, denoted $L(\mathcal{M})$.

Example 1 As a simple example of a language accepted by a meta-deterministic automaton, consider $L = L_1 \cup L_2$, where $L_1 = \{a^m b^n c^n \mid n, m \in \{0, 1, \ldots\}\}$ and $L_2 = \{a^m b^m c^n \mid n, m \in \{0, 1, \ldots\}\}$. It is well-established that L is not a deterministic language [5, Example 10.1]. However, it is the union of two languages L_1 and L_2, which are by themselves deterministic. Therefore, L is accepted by a meta-deterministic automaton $\mathcal{M}$ which uses two DPDAs $\mathcal{A}_1$ and $\mathcal{A}_2$, accepting L_1 and L_2, respectively.

We may for example define $\mathcal{M}$ as $(\mathcal{F}, \{\mathcal{A}_1, \mathcal{A}_2\}, \mu)$ with $\mathcal{F} = (S, Q, q_s, F, T)$, where

- $S = \{b_1, b_2\}$,
- $Q = \{q_s, q_f\}$,
- $F = \{q_f\}$,
- $T = \{(q_s, b_1, q_f),\ (q_s, b_2, q_f)\}$, and
- $\mu(b_1) = \mathcal{A}_1$ and $\mu(b_2) = \mathcal{A}_2$.

A graphical representation for $\mathcal{M}$ is given in Figure 1. States $q \in Q$ are represented by vertices labelled by q, triples $(q, b, p) \in T$ by arrows from q to p labelled by $\mu(b)$.

That the meta-deterministic automata precisely accept the meta-deterministic languages is reflected by the following equation.

$$C(\mathcal{D}) = \{L(\mathcal{M}) \mid \mathcal{M} \text{ is a meta-deterministic automaton}\}$$

This equation straightforwardly follows from the equivalence of finite automata and regular expressions, and the equivalence of deterministic pushdown automata and deterministic languages.

Let $\mathcal{N}$ denote the class of prefix-free deterministic languages. In the same vein, we have

$$C(\mathcal{N}) = \{L(\mathcal{M}) \mid \mathcal{M} = (\mathcal{F}, A, \mu) \text{ is a meta-deterministic automaton where } A \text{ is a set of normal PDAs}\}$$

In the sequel, we set out to investigate a number of properties of languages in $C(\mathcal{D})$, represented by their meta-deterministic automata (i.e. their corresponding recognition devices). The DPDAs in an arbitrary such device cause some technical difficulties which may be avoided if we restrict ourselves to meta-deterministic automata which use only normal PDAs, as opposed to arbitrary deterministic PDAs. Fortunately, this restriction does not reduce the class of languages that can be described, or in other words, $C(\mathcal{N}) = C(\mathcal{D})$. We prove this equality below.

Since $C(\mathcal{N}) \subseteq C(\mathcal{D})$ is vacuously true, it is sufficient to argue that $\mathcal{D} \subseteq C(\mathcal{N})$, from which $C(\mathcal{D}) \subseteq C(C(\mathcal{N})) = C(\mathcal{N})$ follows using the closure properties of C, in particular monotonicity and idempotence.

We prove that $\mathcal{D} \subseteq C(\mathcal{N})$ by showing how for each DPDA $\mathcal{A}$ a meta-deterministic automaton $\rho(\mathcal{A}) = (\mathcal{F}, A, \mu)$ may be constructed such that A consists only of prefix-free deterministic PDAs, and $L(\rho(\mathcal{A})) = L(\mathcal{A})$. This construction is given by:

Construction 1 Let $\mathcal{A} = (\Sigma, \Delta, X_{initial}, F_{\mathcal{A}}, T_{\mathcal{A}})$ be a deterministic PDA. Construct the meta-deterministic automaton $\rho(\mathcal{A}) = (\mathcal{F}, A, \mu)$, with $\mathcal{F} = (S, Q, q_s, F_{\mathcal{F}}, T_{\mathcal{F}})$, where

- $S = \{b_{X,Y} \mid X, Y \in \Delta\} \cup \{c_{X,Y} \mid X, Y \in \Delta\}$,
- $Q = \Delta$,
- $q_s = X_{initial}$,
- $F_{\mathcal{F}} = F_{\mathcal{A}}$,
- $T_{\mathcal{F}} = \{(X, b_{X,Y}, Y) \mid X, Y \in \Delta\} \cup \{(X, c_{X,Y}, Y) \mid X, Y \in \Delta\}$.

The set A consists of (prefix-free deterministic) PDAs $\mathcal{B}_{X,Y}$ and $\mathcal{C}_{X,Y}$, for all $X, Y \in \Delta$, defined as follows.

Each $\mathcal{B}_{X,Y}$ is defined to be $(\Sigma, \{X^{in}, Y^{out}\}, X^{in}, \{Y^{out}\}, T)$, where X^{in} and Y^{out} are fresh symbols, and where the transitions in T are

$$X^{in} \quad \overset{z}{\mapsto}_{\mathcal{B}_{X,Y}} \quad X^{in} Y^{out} \quad \text{for all } X \overset{z}{\mapsto}_{\mathcal{A}} XY, \text{ some } z$$

Each $\mathcal{C}_{X,Y}$ is defined to be $(\Sigma, \Delta \cup \{X^{in}, Y^{out}\}, X^{in}, \{Y^{out}\}, T)$, where X^{in} and Y^{out} are fresh symbols, and where the transitions in T are those in $T_{\mathcal{A}}$ plus the extra transitions

$$\begin{array}{lll} X^{in} & \overset{z}{\mapsto}_{\mathcal{C}_{X,Y}} & X^{in} Z \quad \text{for all } X \overset{z}{\mapsto}_{\mathcal{A}} XZ, \text{ some } z \text{ and } Z \\ X^{in} Z & \overset{\epsilon}{\mapsto}_{\mathcal{C}_{X,Y}} & Y^{out} \quad \text{for all } XZ \overset{\epsilon}{\mapsto}_{\mathcal{A}} Y, \text{ some } Z \end{array}$$

The function μ maps the symbols $b_{X,Y}$ to automata $\mathcal{B}_{X,Y}$ and the symbols $c_{X,Y}$ to automata $\mathcal{C}_{X,Y}$.

Each automaton $\mathcal{B}_{X,Y}$ mimics a single transition of $\mathcal{A}$ of the form $X \overset{z}{\mapsto}_{\mathcal{A}} XY$. Formally, $\mathcal{B}_{X,Y}$ recognizes a string z if and only if $(X, z) \vdash_{\mathcal{A}} (XY, \epsilon)$

Each automaton $\mathcal{C}_{X,Y}$ mimics a computation of $\mathcal{A}$ that replaces stack element X by stack element Y. Formally, $\mathcal{C}_{X,Y}$ recognizes a string v if and only if $(X, v) \models^{+}_{\mathcal{A}} (XZ, \epsilon) \vdash_{\mathcal{A}} (Y, \epsilon)$, for some $Z \in \Delta$.

We conclude

Theorem 1 $C(\mathcal{N}) = C(\mathcal{D})$

This theorem can be paraphrased as "The class of LR(k) languages is contained in the regular closure of the class of LR(0) languages".

Example 2 We demonstrate Construction 1 by means of an example. Consider the language $L_{Pal} = \{wcw^R \mid w \in \{a, b\}^*\}$, where w^R denotes the mirror image of string w. This language consists of palindromes in which a symbol c occurs as the center of each palindrome.

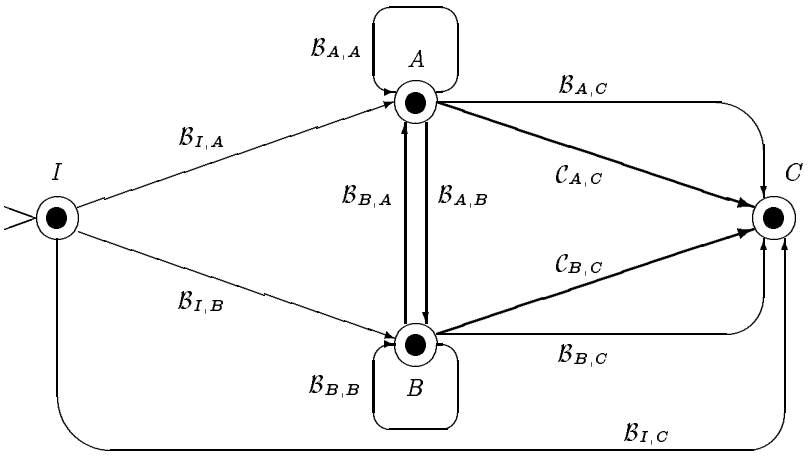

Figure 2. Meta-deterministic automaton $\rho(\mathcal{A}_{PrePal})$

Now consider the language $L_{PrePal} = \{v \mid \exists w[vw \in L_{Pal}]\}$, consisting of all prefixes of palindromes. This language, which is obviously not prefix-free, is accepted by the PDA $\mathcal{A}_{PrePal} = (\Sigma, \Delta, I, F, T)$, with $\Sigma = \{a, b, c\}$, $\Delta = \{I, A, B, C, \overline{A}, \overline{\overline{A}}, \overline{B}, \overline{\overline{B}}\}$, $F = \{I, A, B, C\}$, and T consists of the following transitions:

$$
\begin{array}{lcll}
X & \overset{a}{\mapsto} & XA & \text{for } X \in \{I, A, B\} \\
X & \overset{b}{\mapsto} & XB & \text{for } X \in \{I, A, B\} \\
X & \overset{c}{\mapsto} & XC & \text{for } X \in \{I, A, B\} \\
C & \overset{a}{\mapsto} & C\overline{A} & \\
C\overline{A} & \overset{\epsilon}{\mapsto} & \overline{\overline{A}} & \\
A\overline{\overline{A}} & \overset{\epsilon}{\mapsto} & C & \\
C & \overset{b}{\mapsto} & C\overline{B} & \\
C\overline{B} & \overset{\epsilon}{\mapsto} & \overline{\overline{B}} & \\
B\overline{\overline{B}} & \overset{\epsilon}{\mapsto} & C &
\end{array}
$$

The automaton operates by pushing each a or b it reads onto the stack in the form of A or B, until it reads c, and then the symbols read are matched against the occurrences of A and B on the stack. Note that F is $\{I, A, B, C\}$, which means that a recognized string may be the prefix of a palindrome instead of being a palindrome itself.

The upper level of the meta-deterministic automaton $\rho(\mathcal{A}_{PrePal})$ is shown in Figure 2. (Automata accepting the empty language have been omitted from this representation, as well as vertices which after this omission do not occur on any path from I to any other final state.)

The automaton $\mathcal{B}_{A,B}$ accepts the language $\{b\}$, since the only pushing transition of $\mathcal{A}_{PrePal}$ which places B on top of A reads b. As another example of a lower level automaton, automaton $\mathcal{C}_{A,C}$ accepts the language $\{wa \mid w \in L_{Pal}\}$, since $(A, v) \models^+_{\mathcal{A}} (AZ, \epsilon) \vdash_{\mathcal{A}} (C, \epsilon)$, some Z, only holds for v of the form wa, with $w \in L_{Pal}$; for example $(A, bcba) \vdash_{\mathcal{A}} (AB, cba) \vdash_{\mathcal{A}} (ABC, ba) \vdash_{\mathcal{A}} (ABC\overline{B}, a) \vdash_{\mathcal{A}} (AB\overline{\overline{B}}, a) \vdash_{\mathcal{A}} (AC, a) \vdash_{\mathcal{A}} (AC\overline{A}, \epsilon) \vdash_{\mathcal{A}} (A\overline{\overline{A}}, \epsilon) \vdash_{\mathcal{A}} (C, \epsilon)$.

4 RECOGNIZING FRAGMENTS OF A STRING

In this section we investigate the following problem. Given an input string $a_1 \cdots a_n$ and an NPDA $\mathcal{A}$, find all pairs of input positions (j, i) such that substring $a_{j+1} \cdots a_i$ is recognized by $\mathcal{A}$; or in other words, such that $(X_{initial}, a_{j+1} \cdots a_i) \vdash^* (X_{final}, \epsilon)$. It will be shown that this problem can be solved in linear time.

For technical reasons we have to assume that the stack always consists of at least two elements. This is accomplished by assuming that a fresh stack symbol $\perp$ occurs below the bottom of the actual stack, and by assuming that the actual initial configuration is created by an imaginary extra step $(\perp, v) \vdash (\perp X_{initial}, v)$.

The original problem stated above is now generalized to finding all 4-tuples (X, j, Y, i), with $X, Y \in \Delta$ and $0 \leq j \leq i \leq n$, such that $(X, a_{j+1} \cdots a_i) \models^+ (XY, \epsilon)$. In words, this condition states that if a stack has an element labelled X on top then the pushdown automaton can, by reading the input between j and i and without ever popping X, obtain a stack with one more element, labelled Y, which is on top of X. Such 4-tuples are henceforth called *items*.

The items are computed by a dynamic programming algorithm based on work from [1, 7, 4, 8].

It can be proved [1, 7] that Algorithm 1 in Figure 3 eventually adds an item (X, j, Y, i) to $\mathcal{U}$ if and only if $(X, a_{j+1} \cdots a_i) \models^+ (XY, \epsilon)$. Specifically, $(\perp, j, X_{final}, i) \in \mathcal{U}$ is equivalent to $(\perp, a_{j+1} \cdots a_i) \vdash (\perp X_{initial}, a_{j+1} \cdots a_i) \vdash^* (\perp X_{final}, \epsilon)$. Therefore, the existence of such an item $(\perp, j, X_{final}, i) \in \mathcal{U}$, or equivalently, the existence of $(j, i) \in \mathcal{V}$, indicates that substring $a_{j+1} \cdots a_i$ is recognized by $\mathcal{A}$, which solves the original problem stated at the beginning of this section.

If no restrictions apply, the number of 4-tuples computed in $\mathcal{U}$ can be quadratic in the length of the input. The central observation is this: It is possible that items $(X, j, Y, i) \in \mathcal{U}$ are added for several (possibly linearly many) i, with fixed X, j and Y. This may happen if $(\perp, a_h \cdots a_j \cdots a_{i_m}) \vdash^* (\delta X, a_{j+1} \cdots a_{i_m}) \models^+ (\delta XY, a_{i_1+1} \cdots a_{i_m})$ and $(Y, a_{i_1+1} \cdots a_{i_m}) \vdash^+ (Y, a_{i_2+1} \cdots a_{i_m}) \vdash^+ \ldots \vdash^+ (Y, a_{i_{m-1}+1} \cdots a_{i_m}) \vdash^+ (Y, \epsilon)$,

Algorithm 1 Consider an NPDA and an input string $a_1 \cdots a_n$.

1. Let the set $\mathcal{U}$ be $\{(\bot, i, X_{initial}, i) \mid 0 \leq i \leq n\}$.
2. Perform one of the following two steps as long as one of them is applicable.

push

1. Choose a pair, not considered before, consisting of a transition $X \overset{z}{\mapsto} XY$ and an input position j, such that $z = \epsilon \vee z = a_{j+1}$.
2. If $z = \epsilon$ then let $i = j$, else let $i = j + 1$.
3. Add item (X, j, Y, i) to $\mathcal{U}$.

pop

1. Choose a triple, not considered before, consisting of a transition $XY \overset{\epsilon}{\mapsto} Z$ and items $(W, h, X, j), (X, j, Y, i) \in \mathcal{U}$.
2. Add item (W, h, Z, i) to $\mathcal{U}$.

3. Finally, define the set $\mathcal{V}$ to be $\{(j, i) \mid (\bot, j, X_{final}, i) \in \mathcal{U}\}$.

Figure 3. Dynamic programming algorithm

which leads to m items (X, j, Y, i_1), ..., (X, j, Y, i_m). Such a situation can in the most trivial case be caused by a pair of transitions $X \overset{z}{\mapsto} XY$ and $XY \overset{\epsilon}{\mapsto} X$; the general case is more complex however.

On the other hand, whenever it can be established that for all X, j and Y there is at most one i with (X, j, Y, i) being constructed, then the number of entries computed in $\mathcal{U}$ is linear in the length of the input string, and we get a linear time bound by the reasoning presented at the end of this section.

The following definition identifies the intermediate objective for obtaining a linear complexity. We define a PDA to be *loop-free* if $(X, v) \vdash^+ (X, \epsilon)$ does not hold for any X and v. The intuition is that reading some input must be reflected by a change in the stack.

Our solution to linear-time recognition for automata which are not loop-free is the following: We define a language-preserving transformation from one NPDA to another which is loop-free. Intuitively, this is done by pushing extra elements $\overline{X}$ on the stack so that we have $(X, v) \vdash^+ (\overline{X}X, \epsilon)$ instead of $(X, v) \vdash^+ (X, \epsilon)$, where $\overline{X}$ is a special stack symbol to be defined shortly.

As a first step we remark that for a normal PDA we can divide the stack symbols into two sets $PUSH$ and POP, defined by

$$\begin{aligned} PUSH &= \{X \mid \text{there is a transition } X \overset{z}{\mapsto} XY\} \\ POP &= \{Y \mid \text{there is a transition } XY \overset{\epsilon}{\mapsto} Z\} \cup \{X_{final}\} \end{aligned}$$

It is straightforward to see that determinism of the PDA requires that $PUSH$ and POP are disjoint. We may further assume that each stack symbol belongs to either $PUSH$ or POP, provided we assume that the PDA is *reduced*, meaning that there are no transitions or stack symbols which are useless for obtaining the final configuration from an initial configuration.

Construction 2 Consider an NPDA $\mathcal{A} = (\Sigma, \Delta, X_{initial}, \{X_{final}\}, T)$ of which the set of stack symbols Δ is partitioned into $PUSH$ and POP, as explained above. From this NPDA a new PDA $\tau(\mathcal{A}) = (\Sigma, \Delta', X'_{initial}, \{X'_{final}\}, T')$ is constructed, $X'_{initial}$ and X'_{final} being fresh symbols, where $\Delta' = \Delta \cup \{X'_{initial}, X'_{final}\} \cup \{\overline{X} \mid X \in PUSH\}$, $\overline{X}$ being fresh symbols, and the transitions in T' are given by

$$\begin{array}{llll} XY & \overset{\epsilon}{\mapsto}_{\tau(\mathcal{A})} & Z & \text{for } XY \overset{\epsilon}{\mapsto}_{\mathcal{A}} Z \text{ with } Z \in POP \\ XY & \overset{\epsilon}{\mapsto}_{\tau(\mathcal{A})} & \overline{Z} & \text{for } XY \overset{\epsilon}{\mapsto}_{\mathcal{A}} Z \text{ with } Z \in PUSH \\ \overline{X} & \overset{\epsilon}{\mapsto}_{\tau(\mathcal{A})} & \overline{X}X & \text{for } X \in PUSH \\ \overline{X}Y & \overset{\epsilon}{\mapsto}_{\tau(\mathcal{A})} & Y & \text{for } X \in PUSH, Y \in POP \\ X & \overset{z}{\mapsto}_{\tau(\mathcal{A})} & XY & \text{for } X \overset{z}{\mapsto}_{\mathcal{A}} XY \end{array}$$

and the two transitions $X'_{initial} \overset{\epsilon}{\mapsto}_{\tau(\mathcal{A})} X'_{initial}X_{initial}$ and $X'_{initial}X_{final} \overset{\epsilon}{\mapsto}_{\tau(\mathcal{A})} X'_{final}$.

Example 3 We demonstrate this construction by means of an example.

Consider the NPDA $\mathcal{A} = (\{a, b\}, \{X, Y, Z, P\}, X, \{P\}, T)$, where T contains the transitions given in the left half of Figure 4. It is clear that $\mathcal{A}$ is not loop-free: we have $(X, a) \vdash (XY, \epsilon) \vdash (X, \epsilon)$. If the input $a_1 \cdots a_n$ to Algorithm 1 is a^n, then $(\bot, a_{j+1} \cdots a_i) \models^+ (\bot X, \epsilon)$ and therefore $(\bot, j, X, i) \in \mathcal{U}$, for $0 \leq j \leq i \leq n$. This explains why the time complexity is quadratic.

We divide the stack symbols into $PUSH = \{X\}$ and $POP = \{Y, Z, P\}$. Of the transformed automaton $\tau(\mathcal{A}) = (\{a, b\}, \{X, Y, Z, P, X', P', \overline{X}\}, X', \{P'\}, T')$, the transitions are given in the right half of Figure 4.

The recognition of *aab* by $\mathcal{A}$ and $\tau(\mathcal{A})$ is compared in Figure 5.

Without proof we state that if $\mathcal{A}$ is an NPDA, then $\tau(\mathcal{A})$ is a loop-free NPDA that accepts the same language as $\mathcal{A}$.

Because of this property of construction τ, we can state the following without loss of generality for NPDAs:

Theorem 2 *For a loop-free NPDA, Algorithm 1 has linear time demand, measured in the length of the input.*

$\mathcal{A}$			$\tau(\mathcal{A})$		
			X'	$\overset{\epsilon}{\mapsto}$	$X'X$
X	$\overset{a}{\mapsto}$	XY	X	$\overset{a}{\mapsto}$	XY
XY	$\overset{\epsilon}{\mapsto}$	X	XY	$\overset{\epsilon}{\mapsto}$	$\overline{X}$
			$\overline{X}$	$\overset{\epsilon}{\mapsto}$	$\overline{X}X$
X	$\overset{b}{\mapsto}$	XZ	X	$\overset{b}{\mapsto}$	XZ
XZ	$\overset{\epsilon}{\mapsto}$	P	XZ	$\overset{\epsilon}{\mapsto}$	P
			$\overline{X}P$	$\overset{\epsilon}{\mapsto}$	P (Some other transitions of this form have been omitted, because they are useless.)
			$X'P$	$\overset{\epsilon}{\mapsto}$	P'

Figure 4. The transformation τ applied to a NPDA $\mathcal{A}$

$\mathcal{A}$		$\tau(\mathcal{A})$	
stack	input	stack	input
X	aab	X'	aab
		$X'X$	aab
XY	ab	$X'XY$	ab
X	ab	$X'\overline{X}$	ab
		$X'\overline{X}X$	ab
XY	b	$X'\overline{X}XY$	b
X	b	$X'\overline{X}\,\overline{X}$	b
		$X'\overline{X}\,\overline{X}X$	b
XZ		$X'\overline{X}\,\overline{X}XZ$	
P		$X'\overline{X}\,\overline{X}P$	
		$X'\overline{X}P$	
		$X'P$	
		P'	

Figure 5. The sequences of configurations recognizing aab, using $\mathcal{A}$ and $\tau(\mathcal{A})$

5 META-DETERMINISTIC RECOGNITION

With the results from the previous section we can prove that the recognition problem for meta-deterministic languages can be solved in linear time, by giving a tabular algorithm simulating meta-deterministic automata.

Consider a meta-deterministic automaton $\mathcal{M} = (\mathcal{F}, A, \mu)$. Because of Theorem 1 we may assume without loss of generality that the DPDAs in A are all normal PDAs. For deciding whether some input string $a_1 \cdots a_n$ is recognized by $\mathcal{M}$ we first determine which substrings of the input are recognized by which NPDAs in A. Then, we traverse the finite automaton, identifying the input symbols of $\mathcal{F}$ with automata which recognize consecutive substrings of the input string. In order to obtain linear time complexity, we again use tabulation, this time by means of pairs (q, i), which indicate that state q has been reached at input position i.

The complete algorithm is given in Figure 6.

We now get the main theorem of this paper.

Theorem 3 *Recognition can be performed in linear time for all meta-deterministic languages.*

6 ON-LINE SIMULATION

The nature of Algorithm 2 as simulation of meta-deterministic automata is such that it could be called an *off-line* algorithm. A case in point is that it simulates steps of PDAs at certain input positions where this can never be useful for recognition of the input if the preceding input were taken into account. By processing the input strictly from left to right and by computing the table elements in a demand-driven way, an *on-line* algorithm is obtained, which leads to fewer table elements, although the *order* of the time complexity is not reduced.

Algorithm 2 Consider a meta-deterministic automaton $\mathcal{M} = (\mathcal{F}, A, \mu)$, where $\mathcal{F} = (S, Q, q_s, F, T)$ and A is a finite set of NPDAs, and consider an input string $a_1 \cdots a_n$.

1. Construct the tables $\mathcal{V}_\mathcal{A}$ as the sets $\mathcal{V}$ in Algorithm 1, for the respective $\mathcal{A} \in A$ and input $a_1 \cdots a_n$.
2. Let the set $\mathcal{W}$ be $\{(q_s, 0)\}$. Perform the following as long as it is applicable.

A. Choose a quadruple not considered before, consisting of

- a pair $(q, j) \in \mathcal{W}$,
- a PDA $\mathcal{A} \in A$,
- a pair $(j, i) \in \mathcal{V}_\mathcal{A}$, and
- a state $p \in Q$,

such that $(q, b, p) \in T$ for some b with $\mu(b) = \mathcal{A}$.
B. Add (p, i) to $\mathcal{W}$.

3. Recognize the input when $(q, n) \in \mathcal{W}$, for some $q \in F$.

Figure 6. Meta-dynamic programming algorithm

The realisation of this on-line algorithm consists of two steps: first we adapt the pushing step so that the PDAs by themselves are simulated on-line, and second, we merge Algorithm 1 and Algorithm 2 such that they cooperate by passing control back and forth concerning (1) where a PDA should start to try to recognize a subsequent substring according to the finite automaton, and (2) at what input position a PDA has succeeded in recognizing a substring. Conceptually, the finite automaton and the PDAs operate in a routine-subroutine relation.

The on-line algorithm is given in Figure 7. The popping and pushing steps, simulating the PDA steps, operate much as in Algorithm 1. An important difference is that the pushing step no longer operates irrespective of preceding input: it only simulates a push on some stack element X, if it has been established with regard to previously processed input that such an element may indeed appear on top of the stack.

A second difference is that the PDA steps are simulated by starting at input positions computed by the "down" step, which adds $(\bot, i, X_{initial}, i)$ to $\mathcal{U}_\mathcal{A}$ only if recognition of a substring recognized by $\mathcal{A}$ is needed from position i in order to enable a transition to a next state in the finite automaton.

The "up" step constitutes a shift of control back to the finite automaton after some PDA has succeeded in recognizing a substring.

A device which recognizes some language by reading input strings from left to right is said to satisfy the *correct-prefix property* if it cannot read past the first incorrect symbol in an incorrect input string. A different way of expressing this is that if it has succeeded in processing a prefix w of some input string wv, then w is a prefix of some input string wv' which can be recognized.

A consequence of the on-line property of Algorithm 3 is that it satisfies the correct-prefix property, provided that both the finite automaton $\mathcal{F}$ and the PDAs in A satisfy the correct-prefix property.

7 PRODUCING PARSE TREES

We have shown that meta-deterministic recognition can be done efficiently. The next step is to investigate how the recognition algorithms can be extended to be parsing algorithms.

The approach to tabular context-free parsing in [7, 4] is to start with pushdown transducers. A pushdown transducer can be seen as a PDA of which the transitions produce certain *output symbols* when they are applied. The *output string*, which is a list of all output symbols which are produced while successfully recognizing an input, is then seen as a representation of the parse.

If the pushdown transducers are to be realized using a tabular algorithm such as Algorithm 1 then we may apply the following to compute all output strings without deteriorating the time complexity of the recognition algorithm. The idea is that a context-free grammar, the *output grammar*, is constructed as a side-effect of recognition. For each item (X, j, Y, i) added to the table, the grammar contains a nonterminal $A_{(X,j,Y,i)}$. This nonterminal is to generate all lists of output symbols which the pushdown transducer produces while computing $(X, a_{j+1} \cdots a_i) \models^+ (XY, \epsilon)$. The rules of the output grammar are created when items are computed from others. For example, if we compute an item (W, h, Z, i) from two items $(W, h, X, j), (X, j, Y, i) \in \mathcal{U}$, using a popping transition $XY \overset{a}{\mapsto} Z$ which produces output symbol a, then the output grammar is extended with rule $A_{(W,h,Z,i)} \rightarrow A_{(W,h,X,j)}\ A_{(X,j,Y,i)}\ a$.

The start symbol of the output grammar is $A_{(\bot,0,X_{final},n)}$, for recognition of the complete input. For Algorithm 1 however, which recognizes fragments of the input, we have several output grammars, of which the start symbols are of the form $A_{(\bot,j,X_{final},i)}$. The sets of rules of these grammars may overlap.

The languages generated by output grammars consist of all output strings which may be produced by the pushdown transducer while successfully recognizing the corresponding substrings. In the case of deterministic PDAs, these are of course singleton languages.

In a straightforward way this method may be extended to off-line simulation of a meta-deterministic automaton $\mathcal{M} = (\mathcal{F}, A, \mu)$, where A is now a set of push-down transducers:

1. We create subgrammars for v and the respective automata in A separately, following the ideas above.
2. We merge all grammar rules constructed for the different

Algorithm 3 Consider a meta-deterministic automaton $\mathcal{M} = (\mathcal{F}, A, \mu)$, with $\mathcal{F} = (S, Q, q_s, F, T)$ and A is a finite set of NPDAs, and consider an input string $a_1 \cdots a_n$.

1. Let the set $\mathcal{W}$ be $\{(q_s, 0)\}$.
2. Let the sets $\mathcal{U}_\mathcal{A}$ be $\emptyset$ for all $\mathcal{A} \in A$.
3. Perform one of the following four steps as long as one of them is applicable.

down

1. Choose a pair, not considered before, consisting of
   - a pair $(q, i) \in \mathcal{W}$ and
   - a PDA $\mathcal{A} \in A$,

   such that $(q, b, p) \in T$ for some b with $\mu(b) = \mathcal{A}$ and some p.
2. Add $(\perp, i, X_{initial}, i)$ to $\mathcal{U}_\mathcal{A}$.

push

1. For some PDA $\mathcal{A} \in A$, choose a pair, not considered before, consisting of a transition $X \overset{z}{\mapsto}_\mathcal{A} XY$ and an input position j, such that there is an item $(W, h, X, j) \in \mathcal{U}_\mathcal{A}$, for some W and h, and such that $z = \epsilon \vee z = a_{j+1}$.
2. If $z = \epsilon$ then let $i = j$, else let $i = j + 1$.
3. Add item (X, j, Y, i) to $\mathcal{U}_\mathcal{A}$.

pop

1. For some PDA $\mathcal{A} \in A$, choose a triple, not considered before, consisting of a transition $XY \overset{\epsilon}{\mapsto}_\mathcal{A} Z$ and items $(W, h, X, j), (X, j, Y, i) \in \mathcal{U}_\mathcal{A}$.
2. Add item (W, h, Z, i) to $\mathcal{U}_\mathcal{A}$.

up

1. Choose a quadruple, not considered before, consisting of
   - a pair $(q, j) \in \mathcal{W}$,
   - a PDA $\mathcal{A} \in A$,
   - an item $(\perp, j, X_{final}, i) \in \mathcal{U}_\mathcal{A}$, and
   - a state $p \in Q$,

   such that $(q, b, p) \in T$ for some b with $\mu(b) = \mathcal{A}$.
2. Add (p, i) to $\mathcal{W}$.

4. The input is recognized when $(q, n) \in \mathcal{W}$, for some $q \in F$.

Figure 7. On-line meta-dynamic programming algorithm

automata $\mathcal{A} \in A$. We assume the sets of stack symbols from the respective automata are pairwise disjoint, in order to avoid name clashes.
3. For each automaton $\mathcal{A} \in A$ we add rules $A_{(\mathcal{A},j,i)} \rightarrow A_{(\perp,j,X_{final},i)}$, if $A_{(\perp,j,X_{final},i)}$ is a nonterminal found while constructing $\mathcal{U}_\mathcal{A}$.
4. While constructing table $\mathcal{W}$ the output grammar may be extended with a rule $A_{(p,i)} \rightarrow A_{(q,j)}\ A_{(\mathcal{A},j,i)}$, when a pair (p, i) is derived from a pair $(q, j) \in \mathcal{W}$ and a pair $(j, i) \in \mathcal{V}_\mathcal{A}$.
5. We extend the output grammar with all rules of the form $S \rightarrow (q, n)$, where $q \in F$. S is the start symbol of the grammar.

(For on-line processing similar considerations apply.)

In this way, we may produce a context-free grammar reflecting the structure of the input string, without deteriorating the time complexity of the recognition algorithm.

8 GENERALIZED PATTERN MATCHING

In [6] the following problem is treated. Given are a finite set of input symbols Σ, an input string $a_1 \cdots a_n \in \Sigma^*$ and a pattern $b_1 \cdots b_m \in \Sigma^*$. To be decided is whether $a_1 \cdots a_n = vb_1 \cdots b_m w$, some $v, w \in \Sigma^*$, or in words, whether $b_1 \cdots b_m$ is a substring of $a_1 \cdots a_n$.

This problem can also be stated as follows. To be decided is whether $a_1 \cdots a_n$ is a member of the language $\Sigma^*\{b_1 \cdots b_m\}\Sigma^*$. This language is described as a regular expression over deterministic languages, i.e. Σ and $\{b_1 \cdots b_m\}$, and therefore this language is meta-deterministic. Consequently, the algorithms in this paper apply.

The time demand can then be shown to be $\mathcal{O}(n \cdot m)$, which is, of course, $\mathcal{O}(n)$ if n is taken as sole parameter. This is in contrast to the algorithm in [6], which provides a complexity of $\mathcal{O}(n+m)$. This seems a stronger result if time complexity is the only matter of consideration. From a broader perspective however, one finds that our approach allows a larger class of problems to be solved.

For example, the substring problem can be generalized as follows. Given are a finite set of input symbols Σ, an input string $a_1 \cdots a_n \in \Sigma^*$ and a deterministic language $L \subseteq \Sigma^*$. To be decided is whether $a_1 \cdots a_n = uvw$, some $u, w \in \Sigma^*$ and $v \in L$, or in words, whether some substring of $a_1 \cdots a_n$ is in L. As before, the problem can be translated into a membership problem of some string in a meta-deterministic language, and therefore our approach allows this problem to be solved in $\mathcal{O}(n)$ time.

9 CONCLUSIONS

We have introduced a new subclass of the context-free languages, the meta-deterministic languages, which include the deterministic languages properly. We have given recognition algorithms for this class, and have shown that they have a linear time complexity. In effect, we have extended a well-known class of languages that can be recognized in linear time, viz. the deterministic languages, to a much broader class. Our results are non-trivial since this class contains inherently ambiguous languages. It is still an open problem whether a constructive definition exists for *all* context-free languages which can be recognized in linear time.

ACKNOWLEDGEMENTS

The second author has had fruitful discussions with Joop Leo about linear-time recognizability of subclasses of context-free languages.

REFERENCES

[1] A.V. Aho, J.E. Hopcroft, and J.D. Ullman, 'Time and tape complexity of pushdown automaton languages', *Information and Control*, **13**, 186–206, (1968).

[2] J. Berstel, *Transductions and Context-Free Languages*, B.G. Teubner, Stuttgart, 1979.

[3] E. Bertsch, 'An asymptotically optimal algorithm for non-correcting $LL(1)$ error recovery', Bericht Nr. 176, Fakultät für Mathematik, Ruhr-Universität Bochum, (April 1994).

[4] S. Billot and B. Lang, 'The structure of shared forests in ambiguous parsing', in *27th Annual Meeting of the Association for Computational Linguistics, Proceedings of the Conference*, pp. 143–151, Vancouver, British Columbia, Canada, (June 1989).

[5] J.E. Hopcroft and J.D. Ullman, *Introduction to Automata Theory, Languages, and Computation*, Addison-Wesley, 1979.

[6] D.E. Knuth, J.H. Morris, Jr., and V.R. Pratt, 'Fast pattern matching in strings', *SIAM Journal on Computing*, **6**(2), 323–350, (1977).

[7] B. Lang, 'Deterministic techniques for efficient non-deterministic parsers', in *Automata, Languages and Programming, 2nd Colloquium*, volume 14 of *Lecture Notes in Computer Science*, pp. 255–269, Saarbrücken, (1974). Springer-Verlag.

[8] M.J. Nederhof, *Linguistic Parsing and Program Transformations*, Ph.D. dissertation, University of Nijmegen, 1994.

[9] M.J. Nederhof and E. Bertsch, 'Linear-time suffix parsing for deterministic languages', *Journal of the ACM*, **43**, (1996). To appear.