REGULATED ARRAY GRAMMARS
OF FINITE INDEX
Part II: Syntactic pattern recognition

Henning FERNAU
Markus HOLZER
Wilhelm-Schickard-Institut für Informatik,
Universität Tübingen,
Sand 13, D-72076 Tübingen, Germany
fernau@informatik.uni-tuebingen.de
holzer@informatik.uni-tuebingen.de

Rudolf FREUND
Institut für Computersprachen,
Technische Universität Wien,
Resselg. 3, A-1040 Wien, Austria
rudi@logic.at

Abstract. We introduce special $k$-head finite array automata, which characterize the array languages generated by specific variants of regulated $n$-dimensional context-free array grammars of finite index we introduced in the first part of this paper. As a practical application we show how these analyzing devices in the two-dimensional case can be used in the field of syntactic character recognition.

1 Bridging the gap between theory and practice

The first part of this paper laid some theoretical foundations concerning regulated array grammars of finite index (with various regulation mechanisms) and cooperating/distributed array grammar systems with prescribed teams. (We will not repeat the definitions of these mechanisms in this second part.) We concentrated on questions concerning the equivalence of different definitions regarding their descriptive capacity.

In this second part of our paper, we will focus on practical, especially algorithmic aspects of the problem of recognizing handwritten characters by means of certain array grammars, roughly describing a recognition system which is currently under development; prototype versions running on personal computers can be obtained by contacting the second author. One of the advantages of such syntactic methods for character recognition is the fact that they describe certain typical features of the characters instead of comparing characters bit by bit with a reference character. This may lead to recognition algorithms which are less font-sensitive, see [8, page 274].

On the other hand, recently the authors have considered bounded parallelism within array grammars [I-7] on practical grounds. This feature can nicely be formulated in terms of cooperating/distributed array grammar systems with prescribed teams as introduced in part I of our paper. In practical applications of these theoretical models as for character recognition the number of active areas processed in parallel is quite limited. Hence, our theoretical models of regulated array grammars of finite index are fitting quite well for being applied in the field of syntactic character recognition.
Part II of our current paper is organized as follows: In the next section we address the important stages on the process of syntactic character recognition. In the third section we discuss some problems arising when going from theory to reality, e.g., when going to implement a tool based on the theoretical model of \( n \)-dimensional array grammars with prescribed teams of finite index. In the fourth section we elaborate how array grammars with prescribed teams can be interpreted as analyzing mechanisms, which then leads us to the definition of \( n \)-dimensional \( k \)-head finite array automata. The implementation of a prototype for the syntactic analysis of hand-written (upper-case) characters based on two-dimensional \( k \)-head finite array automata is discussed in the fifth section. A short summary concludes the paper. The bibliography is just a supplement of the one given in the end of part I of the paper.

2 Aspects of syntactic character recognition

In this section we give a short overview of the stages in syntactic character recognition and of the data used in our practical experiments.

2.1 Data acquisition

Hand-written characters were acquired from hundreds of persons on specific forms and then scanned in order to obtain a digital pixel image. A reference to the database obtained in that way can be found in [3].

2.2 Preprocessing

Before we can use the scanned characters for a syntactic analysis, some preprocessing steps are necessary in order to obtain suitable data. We should like to mention that the two crucial steps of the preprocessing procedure, i.e., noise elimination and thinning to unitary skeletons, can be carried out within the theoretical framework of parallel array grammars, which was already exhibited in some details in [I-11].

Normalisation and noise elimination The scanned characters first are normalized to fill out a \( 320 \times 400 \) grid in order to get comparable patterns. Then noisy pixels are eliminated. After noise elimination, the resulting arrays on the \( 320 \times 400 \) grid are mapped on a \( 20 \times 25 \) grid.

Thinning The arrays on the \( 20 \times 25 \) grid now are subjected to a thinning algorithm which finally yields unitary skeletons of the digitized characters. In the literature a lot of such thinning algorithms can be found, which reduce the thickness of the lines constituting a character to one, e.g., see [15].
2.3 Syntactic analysis

The unitary skeleton of a character obtained after the thinning procedure (the last step of preprocessing) now is the input for an off-line tool analyzing this pattern according to a specific syntactic model. In our approach, regulated array grammars of finite index as discussed in part I of our paper or $k$-head finite array automata as introduced later in this paper build up the syntactic model.

![Fig. 1. An ideal character “H”.](image)

For example, the cluster of ideal letters “H” of arbitrary size (an instance is shown in Figure 1) can be described by the following two-dimensional array grammar with prescribed teams of index 4:

$$G = (n, \{S, L, R, D_L, U_L, D_R, U_R\}, \{a\}, \#, (P, R, \emptyset), \{(0, 0), S\});;$$

$$P = \{ S \# \rightarrow LR, \# L \rightarrow La, R \# \rightarrow a R, \}$$

$$L \rightarrow U_L a, R \rightarrow a U_R \quad D_L \rightarrow \# a U_L \# \rightarrow U_L \# \rightarrow \# U_R, \# \rightarrow \# a, D_R \rightarrow \# a, U_L \rightarrow \# a, U_R \rightarrow a, D_L \rightarrow a, D_R \rightarrow a \}$$
Fig. 2. On the border between “H” and “A”.

\[ R = \{ \langle S \# \rightarrow L R \rangle, \langle \# \rightarrow L a, R \# \rightarrow a R \rangle, \]
\[ \langle \# \rightarrow U_L a, R \rightarrow a \rangle, \]
\[ \langle U_L \rightarrow a, U_R \rightarrow a \rangle, \langle U_L \rightarrow U_R \rangle \}. \]

A typical derivation in \( G \) is the following one:

\[ S \Rightarrow_G L R \Rightarrow_G L a a R \Rightarrow_G U_L a a a D_L \Rightarrow_G a a a \]
\[ U_L \rightarrow a, U_R \rightarrow a, U_L \rightarrow U_R \]
\[ a a a a \rightarrow_G a a a a \rightarrow_G a a a a \]
\[ a a a a \rightarrow_G a a a a \rightarrow_G a a a a \]
\[ a a a a \rightarrow_G a a a a \rightarrow_G a a a a \]
\[ D_L a a a a \rightarrow_G D_L a a a a \rightarrow_G D_L a a a a \]
The main problems occurring in realistic patterns are the deviations of the lines forming a character and possible gaps in these lines (see Figure 2). For example, in order to cover deviations of the horizontal line, together with the array production $R\# \rightarrow aR$ we also have to consider the array productions 
\[
\# \rightarrow R \quad R \rightarrow a \quad \# \rightarrow \quad R.
\]

One of the most important features of an efficient tool is to use suitable error measures which allow us to obtain reasonable clusters for the different letters in the alphabet. In fact, sometimes the border line between two different letters is quite “fluent”. For example, the array represented by the filled circles will still be recognized as an “H” by a lot of people, whereas when adding the two pixels indicated by the non-filled circles nearly all people will agree in recognizing this array as an “A”, because the upper endings of the vertical lines now are close enough to each other; yet the question remains how to determine exact values for the distance of these endings as well as for the deviations of the vertical lines in order to separate the cluster of arrays representing the symbol “A” from the cluster of arrays representing the symbol “H”.

The main features of a given character that may contribute to an error measure are the deviations from the lines building up an ideal letter and the remaining pixels not covered by the syntactic analysis. Yet also more elaborated features as the distances of end points of lines (compare the discussion above concerning the letters “A” and “H”) may increase the error and thus help to distinguish between two clusters representing different letters.

3 Theory and reality

As shown in [1-8], the embedding of any one-dimensional recursively enumerable array language in the two-dimensional space can be generated by a two-dimensional #-context-free array grammar; the proof even shows that already for strictly context-free two-dimensional array grammars the fixed and the general membership problems become undecidable. Even for regular array languages and grammars, these problems are NP-complete as stated in [1-22]. Yet these “limit features” need not have such a deadly importance on such a restricted domain as a $20 \times 25$ grid we use in our implementations of a syntactical character recognition model based on regulated array grammars.

In reality, a more powerful theoretical model may yield a much faster and therefore much more efficient tool for character recognition as long as the increase in the theoretical complexity reduces the parsing complexity, especially by reducing the number of non-deterministic choices of rules during the parsing procedure. Hence, already in [4] the theoretical mechanism of graph controlled (analyzing) array grammars was chosen as the theoretical basis of the character recognition tool proposed there. In fact, controlling the dynamic program flow by graphs is very useful; therefore, a tool based on graph controlled array grammars (as, e.g., described in [4]) is much more efficient than a tool based on array grammars without regulation (as proposed in [16]). Moreover, as characters can
be seen as being composed of a very few lines only, a small number of active areas analyzing these lines, which often even have some interdependency relations like equal lengths, is another promising approach we already proposed in [I-6].

Hence, characters of even arbitrary size can be characterized by languages in $PT^{(k)}(2-cf)$ for rather small $k$; an example for an array grammar with prescribed teams of index four representing the cluster of symbols “H” of arbitrary size was already exhibited in the preceding section. From part I of our paper we know that $PT^{(1)}(2-cf) = L(2-reg)$, which family of array languages already has a hard enough membership problem as stated above, but in fact analyzing array grammars with prescribed teams of context-free array productions of finite index as proposed in [I-6] allow even deterministic parsing of specific characters as we will elaborate in the next section.

4 k-head finite array automata

In the string case, multihead finite automata belong to the oldest subjects of study, see [11]. As regards multi-dimensional automata, we refer the reader to [1, 6] and [I-19]. Our aim is to define array automata in such a way that they characterize families of array language defined by regulated (strictly) context-free array grammars of finite index introduced in part I of the paper.

On an intuitive level, context-freeness in the string case means that a characterizing automaton model has to be essentially a one-way model. This restricts the movements of the input heads such that each head can read the same information only once, since it scans the input word from left to right.

Array grammars do not process only symbol information (as in the string case) but also position information (and hence direction information), so that we cannot hope for a purely one-way automaton analogue. Instead, in the (strictly) context-free case (but not in the #-context-free case!) we have the restriction that a position which is looked up once in the derivation process will eventually carry some terminal symbol. From this, we can deduce that a reasonable automaton model should obey the restriction that the same information is read only once. This formulation resembles the characterization of the one-way-property in the string case very much, but there is an important difference: while in the string case, $k$ heads may scan the same symbol $k$ times (each of them can read the same information once), in the array case for our purposes we require that the whole automaton may scan a certain position only once. (Let us mention that two-dimensional automata which cannot visit one point twice are also called “worms” in [2].) In passing, this excludes a head sensing ability: it is not possible for two heads to assume the same position at the same time. Moreover, in our model we include the possibility that an automaton head may split in a certain, local way, and that it may be totally removed if it is not necessary any more. Observe that this “recycling feature” of reading heads fits very well in the idea that the $k$ heads are essentially $k$ pointer within the array (especially, from the point of view of implementation of the formalism), but we do not allow arbitrary pointer calculations. Instead, we stick to local head movements.
Alternatively, there is the possibility to define really parallel Turing machines or finite automata working on multi-dimensional input tapes. Here, we refer the reader to [5, 18], but we do not use this approach here.

Instead of elaborating new sophisticated definitions, we use our knowledge from the first part of our paper in order to define a suitable description of $k$-head finite array automata. In the model of array grammars with prescribed teams working with the finite index restriction we have already incorporated the idea of a limited number of active positions in an underlying array. In [I-6], the idea of analyzing array grammars with prescribed teams has already been discussed in such a way that a derivation step with a selected team is only possible if the following conditions hold:

1. By applying the team all the non-terminal symbols appearing in the current array are derived in parallel.
2. The shape of the current array after this derivation step is part of the shape of the originally given array and, moreover, at each position where we already find a terminal symbol in this array, this terminal symbol must coincide with the corresponding symbol at this position in the originally given array.

Condition 1 is just the *finite index restriction* introduced in part I of the paper. Condition 2 in this form is only reasonable for the case of (strictly) context-free array productions; if we also allow $\#$-context-free array productions we have to use the following weaker condition:

2'. At each position where we already find a terminal symbol in the underlying array, this terminal symbol must coincide with the corresponding symbol at this position in the originally given array.

This weaker condition means that the non-terminal symbols of the current array may also occupy positions that are only occupied by the blank symbol in the originally given array.

Taking over this idea of an analyzing array grammar as described above, we can give a formal definition of a $k$-head finite automaton in the following way:

An *$n$-dimensional $k$-head finite array automaton of type $X$, $X \in \{n$-$\#$-cf, $n$-cf, $n$-scf, $n$-cf1, $n$-scf1 | $n \geq 1\}$* is a construct

$$(n, V_N, V_T, \#, (P, R, F), \{v_0, S\}),$$

which is an $n$-dimensional array grammar with prescribed teams $G$ of finite index $k$, such that

1. each team in $R$ contains at most $k$ array productions of type $X$;
2. $G$ is in the normal form established in Lemma 5 of part I;
3. for any array production $p \in P$, $p$ is of the form

$$p = (W, \{(\Omega_n, A)\} \cup \{(v, \#) \mid v \in W \setminus \{\Omega_n\}\}, \{(v, X_v) \mid v \in W\})$$

with $\{v \in W \mid X_v \in V_T\} \subseteq \{\Omega_v\}$. 
The work of the automaton $M$ on a given array from $V_T^n$ is defined as follows: $M$ works on objects from $\{ (a,a), (a,X) \mid a \in V_T, X \in V_N \cup \{ \# \} \}^n$, where the first component contains the given array and the second component contains the array generated so far by $G$.

The current state of the automaton $M$ is represented by the set of non-terminal symbols $Y$ occurring in the pairs $(a,Y)$ of the current array.

The derivation relation for $M$, $\Rightarrow_M$, on the second component corresponds with the derivation relation $\Rightarrow_G$, with the additional restriction that every terminal symbol generated in the second component must be equal to the terminal symbol in the first component. A parsing derivation of $M$ is called accepting, if finally all non-blank positions are terminal, i.e., occupied by symbols of the form $(a,a), a \in V_T$. The array language accepted by $M$ therefore is defined by

$$L(M) = \{ A \mid A \in V_T^n, \{(v,(A(v),\#)) \mid v \in \text{shape}(A)\} \Rightarrow_M^* \{(v, (A(v), A(v))) \mid v \in \text{shape}(A)\} \}$$

Observe that due to our definitions, a head of the automaton, which is represented by one of the at most $k$ (different) variables appearing in the current array, reads out the symbol in the first component just when leaving this position with putting there the exactly same terminal symbol into the second component. As the terminal symbol at a specific position is already uniquely determined by the first position we could also put only a specific marker symbol into the second components just to mark these positions as non-reachable by any head of the automaton any more.

As it is quite obvious from the definitions given above, the families of array languages accepted by $k$-head automata of type $X$ with or without ac exactly coincide with the corresponding families of array languages $PT_{ac}^{\lceil k \rceil}(X)$ and $PT^{\lceil k \rceil}(X)$, respectively, where $X \in \{ n\text{-}#\text{-}cf, n\text{-}cf, n\text{-}scf, n\text{-}cf_1, n\text{-}scf_1 \mid n \geq 1 \}$ and $k \geq 1$. Hence all the theoretical results obtained in part I for the generating devices considered there directly carry over to the parsing mechanism of $k$-head finite automata as defined above. In the string case, similar devices were considered in [9].

One might argue that in this construction of an $n$-dimensional $k$-head finite automaton given above the basic concept of states, which usually constitutes an important feature of an automaton model, only appears in a weak variant, i.e., as the set of all subsets $V$ of $V_N$ with $\text{card}(V) \leq k$. Yet as we can derive from the theoretical results proved in part I, using a graph control structure will not increase the power of the model; hence it seems reasonable to keep the formal definitions on the chosen level and to discuss possible extensions in an informal way only. For example, as exhibited in [4], using a graph control structure is a powerful means for reducing the non-determinism in tools for syntactic character recognition based on array grammar models. Therefore, it is reasonable to add this control mechanism to the model of $k$-head finite automata when implementing this theoretical approach in a tool as described in [3], yet we shall not go into formal details here.

Furthermore, observe that the heads occurring in our model can be viewed as agents (workers) which are sent to their working places in order to perform
their work (a derivation step); so, the idea of multi-agent systems (which was one of the basic motivations for introducing cooperating/distributed grammar systems, cf. [I-1]) emerges quite naturally in our automaton setting.

As illustration of our definition, we review the example for the array grammar with prescribed teams of index four described in section 2 and show the analysis of the pattern whose generation was given there:

\[
\begin{array}{c}
(a, \#) \quad (a, \#) \\
(a, \#) \quad (a, \#) \\
(a, \#) \quad (a, S) (a, \#) (a, \#) \quad \Rightarrow M \\
(a, \#) \quad (a, \#) \\
(a, \#) \quad (a, \#) \\
(a, \#) \quad (a, \#) \\
(a, \#) \quad (a, \#) \\
(a, \#) \quad (a, \#) \\
(a, \#) \quad (a, \#) \\
(a, \#) \quad (a, \#) \\
(a, \#) \quad (a, \#) \\
(a, \#) \quad (a, \#) \\
(a, L) (a, R) (a, \#) \quad \Rightarrow M \\
(a, \#) \quad (a, \#) \\
(a, \#) \quad (a, \#) \\
(a, \#) \quad (a, \#) \\
(a, \#) \quad (a, \#) \\
(a, \#) \quad (a, \#) \\
(a, \#) \quad (a, \#) \\
(a, \#) \quad (a, \#) \\
(a, \#) \quad (a, \#) \\
(a, \#) \quad (a, \#) \\
(a, \#) \quad (a, \#) \\
(a, \#) \quad (a, \#) \\
(a, \#) \quad (a, \#) \\
(a, a) (a, a) (a, a) \quad \Rightarrow M \\
(a, D_L) \quad (a, D_R) \\
(a, \#) \quad (a, \#) \\
(a, \#) \quad (a, \#) \\
(a, \#) \quad (a, \#) \\
(a, \#) \quad (a, \#) \\
(a, U_L) \quad (a, U_R) \\
(a, a) \quad (a, a) \\
(a, a) \quad (a, a) \quad \Rightarrow M \\
(a, a) \quad (a, a) \\
(a, D_L) \quad (a, D_R) \\
(a, \#) \quad (a, \#) \\
(a, \#) \quad (a, \#)
\end{array}
\]
An important feature of this parsing sequence depicted above is the determinism of the given derivation, i.e., for each array in $L(M)$ there is exactly one parsing derivation. For arrays not in $L(M)$, the crucial moment is the change from the horizontal line to the vertical lines. Yet in this special case of $M$, the possibility of a non-deterministic choice in the underlying pattern immediately implies that this pattern cannot belong to $L(M)$. Yet for practical implementations where we also want to recognize non-ideal patterns in a decent way, this is one of the most important problems we have to deal with.

Let us remark that according to Lemma I-7 one-head finite array automata characterize regular array languages, confer to [I-2]. Rosenfeld in [12] compares several alternative definitions of finite-state picture languages and shows that they do not characterize regular array languages. Moreover, it is known that non-deterministic finite one-head automata (which are allowed to read the same input more than once and can move in four directions, i.e., they are the natural two-dimensional equivalent to classical two-way automata) recognize a language class which is strictly contained in the so-called recognizable picture languages, which in turn can be seen as the generalization of algebraic characterizations of regular string languages, cf. [I-17] and especially [7, Cor. 3.2]. A characterization of strictly context-free array languages via an automaton model with a rather tedious defition was given in [10]. The interrelation with so-called pushdown automata on arrays [14] seems to be open.

We do not want to conceal one theoretical drawback of our automaton model somewhat hidden in the acceptance condition: an array pattern must be parsed completely in order to get accepted. This is a condition which comes from outside the model. Nakamura managed to include such a test within his automaton model for context-free array languages in [10]. From a practical point of view, this drawback is not so important, since unvisited points may be found quite efficiently in a post-processing phase. Moreover, superfluous pixels not covered by the syntactic analysis may occur anyway when dealing with “real” characters. Finally, for type $n$-$\#$-cf such a test is not possible at all.
5 A prototype implementation

In this section we describe some interesting observations made during the prototype implementation [3] of the model of $k$-head finite array automata for syntactic character recognition. In fact, the tool also incorporates a graph control structure in order to reduce the non-determinism arising from non-ideal patterns due to deviations of lines and gaps within the lines. The $ac$ mode in the graph control structure also allows us to consume the pixels along a line exhaustively, which often is of advantage because remaining pixels increase the error. The type of the $k$-head finite automata, i.e., the type of the array productions, is chosen as $2-cf$. Due to possible gaps in realistic characters, we cannot restrict ourselves to norm 1, i.e., to $2-cf_1$, yet on the other hand we can avoid to have to use rules of type $2-\#-cf$. From a theoretical point of view, rules of type $2-scf$ might be sufficient (for ideal patterns, even rules of type $2-scf_1$ are sufficient, e.g., see the array grammar with prescribed teams of index four for the set of arrays representing the symbol “H”). Yet as already mentioned in the previous section, situations like at crossing points of lines in realistic patterns cause possible non-deterministic choices how to proceed. In order to make such decisions easier (and more deterministic, i.e., in this way reducing the need for back-tracking), we allow a larger neighbourhood for looking ahead, which also includes the possibility to check some of these positions for not yet having been reached by other heads of the automaton.

In order to obtain suitable criteria for look-ahead neighbourhood patterns and other features introduced for improving the efficiency of the tool, even some heuristic investigations were carried out to optimize the efficiency and the recognition rate of the tool.

Observe that it would be quite easy to incorporate such look-ahead features formally in the automaton model introduced in the preceding section. Obviously, the automaton model would then resemble very much the LL parsers well-known from string language compilers, since basically a top-down parse through the grammar is done by the automaton. In the string case, context-free graph-controlled LL parsers (where it is required that always the left-most symbol is rewritten) are known to characterize the deterministic context-free languages [13], which might be characterized via bottom-up parsers without regulations alternatively. So, this quite practically motivated class of recognizers gives rise to the interesting theoretical question what kind of arrays are recognized by such devices with look-ahead.

6 Conclusions

The theoretical models of regulated array grammars of finite index have turned out to constitute suitable mechanisms for syntactic pattern recognition, e.g., for the recognition of hand-written upper-case characters. Combinations of these mechanisms with other approaches as neural networks should allow the development of an even more efficient tool with very high recognition rate.
Finally, we should like to mention that the use of (regulated) array grammars (with finite index) is not restricted to the recognition of characters; these mechanisms may even be used to characterize three-dimensional objects (see [17]). Hence, in the field of syntactic pattern recognition a lot of applications to be considered in the theoretical framework presented in the current paper remain for future research projects.

Acknowledgements. The work of the first author was supported by Deutsche Forschungsgemeinschaft grant DFG La 618/3-2.

References