ON THE RELATIONSHIP BETWEEN INDEXED GRAMMARS AND LOGIC PROGRAMS

EBERHARD BERTSCH

▷ This article provides detailed constructions demonstrating that the class of indexed grammars introduced as a simple extension of context-free grammars has essentially the same expressive power as the class of logic programs with unary predicates and functions and exactly one variable symbol.

Some additional considerations are concerned with parsing procedures. ◁

1. INTRODUCTION

In a comprehensive survey of extant research results [6], Deransart and Maluszynski relate grammars and logic programs. Their main idea is that the declarative and procedural readings of a logic program can be complemented by a grammatical reading, where clauses are considered to be rewrite rules of a grammar. They try to show that this point of view facilitates the transfer of expertise from logic programming to other research focusing on or employing grammars and vice versa.

Some examples of such transfer are extensively discussed in that article. In particular, the well-known class of Van-Wijngaarden grammars can be restricted to be comparable to definite clause programs. On the other hand, an immediate extension of logic programs leads to definite clause grammars. Furthermore, there are various connections between appropriately modified notions of attribute grammars and logic programs.

As an extension of earlier work by the same authors [5], the relationship between attribute grammars and logic programs is studied in terms of their respective capabilities to generate valid attributed trees and proof trees.

Related results based on a different notational framework are presented in [10]. A theoretical treatment of the expressive power of logic programs is contained in

Address correspondence to Eberhard Bertsch, Lehrstuhl Praktische Informatik, Fakultät für Mathematik, Ruhr-Universität, D-44780 Bochum, Germany. E-mail: Eberhard.Bertsch@ruba.rz.ruhr-uni-bochum.dbp.de.

Received May 1992; accepted May 1993.

[11]. Various publications in this area have also dealt with parsing questions, [14] being an important early example. Many further references can be found in the book [1] by Abramson and Dahl.

The present paper describes an interesting correspondence for definite logic programs, all of whose predicate symbols have arity 1, all of whose function symbols have arity 1, and all of whose clauses contain exactly 1 variable name, say X. Such programs will be called "unary programs."

Rather surprisingly, a grammar class with the generative power thus specified turns out to have been defined and studied over two decades ago. We are referring to A.V. Aho's "indexed grammars" [2], which were introduced as simple extensions of context-free grammars (Chomsky-type-2) to provide some of the facilities otherwise obtained by means of the more complex context-sensitive grammars (Chomsky-type-1) [9].

A class of automata which recognize exactly the class of indexed languages was introduced by Aho in a subsequent article [3]. For the sake of comparison, we describe the salient features of that class informally in a separate section.

The structure of this article is roughly as follows. We review the definition of indexed grammars and discuss a simple example. The notion of unary logic program is made explicit. Then we present a construction which associates a logic program with each indexed grammar in such a way that goals stand in an essentially one-to-one relationship with sentential forms. The correctness proof of this construction is outlined with the necessary amount of detail.

The converse direction of simulation is established by another explicit construction which is presented and discussed in the subsequent section. Thereby, goals occurring in computations of unary logic programs are coded in terms of sentential forms. A one-to-one relationship may be seen to hold if one distinguishes between sets of proper and auxiliary nonterminal symbols of the grammar so constructed.

Both this distinction and the more basic one between terminal and nonterminal symbols contribute to a noticeable lack of elegance in the formulation of the main results. The section which follows takes account of these difficulties and provides a partial solution by means of a (technically motivated) normal form.

The automata-theoretical counterpart of indexed grammars is then described and compared to the present discussion in rather informal terms.

The final section gives an outline of a concrete parsing technique for indexed languages based on the constructions presented before. Possibilities and limitations of the approach are discussed.

2. DEFINITION OF INDEXED GRAMMARS

An *indexed grammar* G is a 5-tuple $G = (N, T, F, R, S)$ where N, T, F are alphabets whose elements are called *nonterminals, terminals,* and *indices* (or *flags*), respectively.

A finite set R_f of *index rules* $A \to w$ $(A \in N,\ w \in (T \cup N)^*)$ is associated to each index $f \in F$. R is a finite set of rewriting rules

$$A \to x_1 z_1 \ldots x_m z_m,$$

where $A \in N$, $x_i \in N \cup T$, $z_i \in F^*$, and $z_i = \epsilon$ if $x_i \in T$ $(i \leq m)$. $S \in N$ is called the *initial* nonterminal.

Let $M = NF^*$ and $K = (M \cup T)^*$ as standard notations throughout this paper. We write $x \rightarrow_G y$ for x, $y \in K$ if either

(i) $x = uAwv$, $u, v \in K$, $A \in N$, $w \in F^*$, $A \rightarrow x_1 z_1 \ldots x_m z_m \in R$,
$y = ux_1 z'_1 \ldots x_m z'_m v$ with $z'_i = \epsilon$ if $x_i \in T$ and $z'_i = z_i w$ for $x_i \in N$ $(i \leq m)$

or

(ii) $x = uAfwv$, $u, v \in K$, $A \in N$, $w \in F^*$, $f \in F$, $A \rightarrow x_1 \ldots x_m \in R_f$,
$y = ux_1 z'_1 \ldots x_m z'_m v$ with $z'_i = \epsilon$ if $x_i \in T$ and $z'_i = w$ for $x_i \in N$ $(i \leq m)$.

The reflexive, transitive closure $\Rightarrow_G$ of $\rightarrow_G$ combined with the notion of terminal symbol gives us the *language* of a grammar

$$L(G) := \{w \in T^* | S \Rightarrow_G w\}.$$

To discuss intermediate stages of indexed derivations, we shall make use of the set of sentential forms

$$SeF(G) := \{W \in K | S \Rightarrow_G W\}$$

and the set of terminal-free sentential forms

$$Sent(G) := SeF(G) \cap M^*.$$

Furthermore, for subsets N' of N, it is useful to have the following subsets of $Sent(G)$.

$$Sent(G, N') := \{W \in (N'F^*)^* | S \Rightarrow_G W\}.$$

As an informal explanation of the derivation process in indexed grammars, please note that an index rule is applicable for rewriting a particular nonterminal if the leftmost flag (f) associated to that nonterminal designates (labels) the set containing the index rule (denoted as R_f). Thus, rules which generate flags provide a simple way of stacking control information for later derivation steps.

In [2], various interesting properties of indexed languages are demonstrated. [3] gives an automata-theoretic characterization.

An efficient parsing procedure for a subclass characterized in terms of unambiguity conditions is presented in [4].

A standard example of a noncontext-free language will serve to illustrate the core idea: The language $L = \{a^n b^n a^n | n \geq 1\}$ is generated by the indexed grammar

$$G = (\{S, T, U\}, \{a, b\}, \{f, g\}, \{S \rightarrow aTg, T \rightarrow aTf, T \rightarrow U\}, S)$$

with index rule sets

$$R_f = \{U \rightarrow bUa\}, \quad R_g = \{U \rightarrow ba\}.$$

The flags f serve as a counter of unindexed derivation steps. When T has been rewritten as U, fs can be taken off again. The additional flag symbol g precludes premature termination.

3. UNARY LOGIC PROGRAMS

Throughout this paper, we use the notations and definitions of [12] without explicit reference. Unary (1-ary) terms, atoms, clauses, and programs are defined in the following straightforward way.

A variable X is a *unary term.*

If t is a unary term and f is a unary function symbol, then $f(t)$ is a *unary term.*

We also say that f, t, and the terms and function symbols occurring in t *occur* in $f(t)$.

If t is a unary term and p is a unary predicate symbol, then $p(t)$ is a *unary atom.*

We also say that p, t, and the terms and function symbols occurring in t *occur* in $p(t)$.

For unary atoms $A, B_1, \ldots, B_n$,

$$A :- B_1, \ldots, B_n$$

is a *unary clause* if the same variable, say X, occurs in all atoms. (The body $B_1, \ldots, B_n$ may be empty.)

A *unary program* is a finite set of unary clauses.

For a given program P, $Func(P)$ and $Pred(P)$ denote the set of function symbols and predicate symbols, respectively, which occur in the atoms of its clauses.

To discuss goals, subgoals, and SLD-derivations, we shall use a unique constant term c. Please note that c is *not* a component of unary programs. It will occur only in goals which are processed by applying unary clauses. Given sets *Func* and *Pred* containing unary function and predicate symbols, respectively, every ground atom has the form

$$a(f_1(\cdots f_n(c)\cdots))$$

for $f_1, \ldots, f_n \in Func$, $a \in Pred$ (including $a(c)$ as a special case), and every ground goal has the form

$$:- P_1, \ldots, P_m$$

where $P_1, \ldots, P_m$ are ground atoms (including the empty goal as a special case).

Let us call the set of ground goals based on *Func* and *Pred* $Ground(Func, Pred)$ or simply *Ground* if the choice of *Func* and *Pred* is evident.

Given a unary program P and a definite goal Gl, we define the set $Deriv(P, Gl)$ to contain all and only those goals Gl' which are obtained by an SLD-derivation of $P \cup \{Gl\}$ (cf. [12, p. 41]) consisting of the sequence of goals $Gl = Gl_0, Gl_1, \ldots, Gl_n = Gl' (0 \leq n)$.

The notion of definite program in [12] does not require that each predicate symbol occur in the head atom of some clause. In our framework of associating logic programs with grammars, predicate symbols not occurring in any head atom may be thought to correspond to nonterminal symbols which can only be rewritten by strings containing terminal symbols. As a practical justification, such predicates may be implemented by a forward move on input combined with checking for a particular lexical item. This line of thought is taken up in more detail in the concluding section.

4. A BRIEF SURVEY OF THE MAIN RESULTS

This paragraph provides an informal view of the correspondence between logic programs and indexed grammars. In a derivation by means of an indexed grammar, the basic rewriting process is that of context-free productions. Additionally, as in various other kinds of enhanced context-free grammars, there is a method by which rewriting may be restricted or controlled. For indexed grammars and languages, this restriction is achieved by flag lists which are associated to individual nonterminal symbols. If their length were bounded, the expressive power of indexed languages could readily be shown to be equivalent to that of context-free languages. Flag lists may become longer or shorter in individual derivation steps. The derivation process is thus used in a twofold way: for ordinary string derivation and for addition or subtraction of flags. A significant feature is that entire lists are passed on in several copies whenever a nonterminal symbol equipped with flags is expanded into more than one new nonterminal. There is an immediate similarity between this type of rewriting and the application of clauses in logic programs if we compare flag lists to nested expressions made out of unary (one-ary) function symbols. For unary terms to have a ground substitution, there must also be at least one constant (zero-ary) term available. What corresponds to rewrite rules? Since flag lists are passed on—either changed or unchanged—they must be modeled by some kind of parameter. The simplest case of parameter passing is that of an individual variable symbol occurring identically on the left-hand and right-hand sides of program clauses. We restrict the class of logic programs to permit exactly that kind of parameter transmission. Now, rules $A \rightarrow BC$ correspond to clauses $A(X)$:– $B(X), C(X)$. Rules $A \rightarrow Bf$ which produce flags are modeled by $A(X)$:– $B(f(X))$. Rules $A \rightarrow B$ in flag sets used for erasing f become $A(f(X))$:– $B(X)$. In the construction required to prove Proposition 1, we show that this type of simulation covers all indexed derivations. The result is clearly much stronger than usual automata-theoretical theorems, in which the identity of classes of generated or recognized sets of terminal strings is claimed. Here we show the existence of bijective mappings between sets of *intermediate* stages in derivations. The structure of the respective generation procedures is thus shown to be (almost) exactly the same. The essence of Proposition 2 and its corollary is that the converse holds as well. The effects of all unary, one-parameter clauses can be simulated by indexed rewriting. There are some technical incompatibilities which turn out to be of minor importance. Others are less straightforward—most notably the special role of terminal symbols in indexed grammars. Partial solutions to this problem are offered in a later section.

5. SIMULATING INDEXED GRAMMARS BY LOGIC PROGRAMS

In this section, we describe a simple construction which yields a unique definite logic program $P(G)$ for a given indexed grammar G. We claim that $P(G)$ simulates the generative capability of G.

Proposition 1. Let $G = (N, T, F, R, S)$ be an indexed grammar. Then there exists a unary program $P(G)$ and a bijective mapping

$$\mu\colon M^* \rightarrow Ground(Func(P(G)), Pred(P(G)))$$

such that

$$w_1 \ldots w_n \in Sent(G) \qquad (w_i \in M, 1 \leq i \leq n)$$

if and only if

$$\mu(w_1 \ldots w_n) \in Deriv(P(G), \mu(S)).$$

PROOF. To avoid notational overloading, we distinguish between a *bar*-mapping $\bar{\mu}$ defined on an alphabet of symbols and flags, a *hat*-mapping $\hat{\mu}$ defined on strings consisting of single nonterminals and arbitrarily many flags, and a mapping μ defined on concatenations of such strings.

We associate a unique new predicate symbol $\bar{\mu}(n)$ to every nonterminal $n \in N$, and a unique new function symbol $\bar{\mu}(k)$ to every flag $k \in F$. We call the set of all such predicate symbols *Pred*, and the set of all such function symbols *Func*.

Please note that we are leaving *terminal* symbols out of consideration. The results about smooth grammars in the special section on terminal symbols indicate the importance of this restriction. A brief discussion of the treatment of terminals in a parsing framework is given in the final part of the paper.

We construct a bijective mapping

$$\mu: M^* \to Ground(Func, Pred)$$

by first letting

$$\hat{\mu}(s_i) = \bar{\mu}(\alpha)(\bar{\mu}(k_1)(\ldots(\bar{\mu}(k_l)(c))\ldots))$$

for $s_i = \alpha k_1 \ldots k_l \in M$ and fixed constant c and extending $\hat{\mu}$ to μ by $\mu(s_1 \ldots s_n) =$ $:-\ \hat{\mu}(s_1), \ldots, \hat{\mu}(s_n)$ where $s_i \in M$ $(1 \leq i \leq n)$.

As a special case, we let $\mu(\epsilon)$ be the empty goal (:–).

The decomposition of words in M^* is obviously well defined because each s_i starts with a single nonterminal.

We need to consider two kinds of rules. Let us first turn to rewriting rules

$$r = (A \to x_1 z_1 \ldots x_m z_m) \in R \qquad \text{where } x_i \notin T.$$

(Rules producing terminal symbols do not need to be considered here because they lead outside of M^*.)

Each z_i is composed of individual flag symbols $z_{1,1}, \ldots, z_{i,n_i}$ or is the empty string. Then we construct the clause

$$p(r) = \bar{\mu}(A)(X) :- p_r^{x_1 z_1}, \ldots, p_r^{x_m z_m}$$

where $p_r^{x_i z_i}$ $(1 \leq i \leq m)$ stands for

$$\bar{\mu}(x_i)(X) \qquad \text{if } z_i = \epsilon$$

or else

$$\bar{\mu}(x_i)\big(\bar{\mu}(z_{i,1})\big(\bar{\mu}(z_{i,2})(\ldots\bar{\mu}(z_{i,n_i})(X)\ldots)\big)\big).$$

Empty right sides of rules r do not require special treatment.

Thus, each nonempty flag sequence is coded by a term containing unary function symbols. The variable X occurs in each such term at the innermost level of nesting. On the other hand, index rules

$$r_f = (A \to w_1 \ldots w_n) \in R_f \qquad \text{where } f \in F, w_i \in N\ (1 \leq i \leq n)$$

lead to the construction of clauses $p(r_f)$ of the form

$$\bar{\mu}(A)(\bar{\mu}(f)(X)) :- \bar{\mu}(w_1)(X), \ldots, \bar{\mu}(w_n)(X).$$

Here, the flag symbol which is used up in applications of the rule r_f appears as a single function symbol in the term on the left-hand side of the corresponding clause.

We form $P(G)$ by assembling all clauses $p(r)$ and $p(r_f)$ so constructed.

To prove the proposition, we have to discuss individual steps of the derivation. The key to the correctness of our construction is that flag sequences are coded as terms made up of 1-ary function symbols.

As a basis of the induction, note that $\mu(S)$ is identical to the initial goal and derived in 0 steps.

The induction hypothesis is that the if-and-only-if assertion of the proposition holds under the condition that the number of derivation steps involved in either the indexed grammar or the logic program is no more than a fixed finite number k.

Then we want to show that a subsequent derivation step in the indexed grammar leads to a string whose image is also obtained by resolution from the image of the original string in the logic program. Conversely, we demonstrate that a resolution step starting with the image of a particular string can only lead to a goal which is the image of a string derived by one application of some rule of the indexed grammar from the previous string.

The induction on the number of derivation steps contains a case distinction depending on whether flags are added or taken off.

Suppose a string $x = uAwv$ $(u, v \in M^*,\ A \in N,\ w \in F^*)$ has been derived in no more than k steps and is rewritten by a rule $r = (A \rightarrow x_1 z_1 \ldots x_m z_m)$. The result is (by definition of the rewriting process)

$$y = u x_1 z'_1 \ldots x_m z'_m v$$

where $z'_i = z_i w$ $(1 \leq i \leq m)$.

In $\mu(x)$, we consider the subgoal $\hat{\mu}(Aw)$. From the induction hypothesis, we get that for $w = w_1 \ldots w_n$ $(w_i \in F,\ 1 \leq i \leq n)$

$$\hat{\mu}(Aw) = \bar{\mu}(A)(\bar{\mu}(w_1)(\ldots \bar{\mu}(w_n)(c)\ldots))$$

is a subgoal of $\mu(x)$.

Performing a resolution step with the clause $p(r)$, the most general unifier of

$$\bar{\mu}(A)(X) \text{ and } \bar{\mu}(A)(\bar{\mu}(w_1)(\ldots \bar{\mu}(w_n)(c)\ldots))$$

is given by substituting the term $\bar{\mu}(w_1)(\ldots \bar{\mu}(w_n)(c)\ldots)$ occurring in the latter atom for X.

It follows that the atoms in the body of $p(r)$ become new subgoals after substituting the same term for their respective occurrences of X. This gives us subgoals

$$\bar{\mu}(x_i)\Big(\bar{\mu}(z_{i,1})\Big(\bar{\mu}(z_{i,2})\Big(\ldots \bar{\mu}(z_{i,n_i})\big(\bar{\mu}(w_1)(\ldots \bar{\mu}(w_n)(c)\ldots)\big)\ldots\Big)\Big)\Big)$$

for $1 \leq i \leq m$.

Applying our mapping μ, the sequence of subgoals $\hat{\mu}(x_i z'_i)$ $(i = 1, \ldots, m)$ results from adding the function symbols which correspond to the flags of z_i to the function symbols already present in $\hat{\mu}(Aw)$ for all terms. The predicate symbols are necessarily the images of x_i under $\overline{\mu}$.

This is exactly what we have just obtained by one resolution step.

On the other hand, if some $p(r)$ constructed from a rule $r = (A \rightarrow x_1 z_1 \ldots x_m z_m)$ in R is applicable to a subgoal sg whose predicate symbol is necessarily $\overline{\mu}(A)$, this fact is due to the possibility of rewriting A in a string $x = uAwv$ with $\hat{\mu}(Aw) = sg$ and getting $y = ux_1 z'_1 \ldots x_m z'_m v$ such that the images of all nonterminals and flags in r are predicate and function symbols occurring in $p(r)$ in the particular order prescribed by its construction from r. It follows that the images of all flags in z_i $(1 \leq i \leq m)$ are added to the images of the flags in w for the ith new subgoal.

Each $x_i z'_i$ $(1 \leq i \leq m)$ satisfies $z'_i = z_i w$, and $\hat{\mu}(x_i z_i w)$ contains a term made up of function symbols which are images of the flags in z_i and w in that order. The sequence of subgoals resulting from the resolution step is therefore identical to the image under μ of the string y produced by r. This settles the first case.

The applicability of flag rules presupposes the existence of a particular flag. Now, if $x = uAfwv$ is reached in k or fewer steps and $r_f = (A \rightarrow w_1 \ldots w_n) \in R_f$ is used to derive a string y, the sequence $\mu(x)$ contains $\overline{\mu}(A)(\overline{\mu}(f)(\ldots(c)\ldots))$ by the induction hypothesis, and the rule $p(r_f)$ is indeed applicable. The argument proceeds as in the previous case by keeping in mind that no flags are added. Thus, the subgoal $sg = \overline{\mu}(A)(\overline{\mu}(f)(\ldots(c)\ldots))$ is replaced by a sequence of subgoals whose predicate symbols are exactly the images of the nonterminals w_i on the right side of r_f in the same order and whose terms are obtained from the term occurring in sg by dropping the first function symbol $\overline{\mu}(f)$ and its opening and closing parentheses. Conversely, if $p(r_f)$ is applicable at all in some goal gl by virtue of our construction, then the corresponding string which is mapped on gl by μ contains the flag f as the leftmost symbol of some flag string by the induction hypothesis, and r_f is applicable there. r_f produces a string, which in turn is mapped on the goal resulting from the application of $p(r_f)$ in the way just shown.

This completes the case distinction.

6. SIMULATION OF UNARY LOGIC PROGRAMS BY INDEXED GRAMMARS

The basic idea of this section is that SLD-derivation of goals by means of unary logic programs can also be performed in an indexed grammar setting. In conjunciton with the previous result, the specific way in which rules and flags of indexed grammars can be used is thus shown to be essentially equivalent to resolution in the context of logic programming for an easily characterized class of predicates.

Proposition 2. Let P be a unary logic program. Single out a predicate symbol π occurring in the head atom of some clause.

Then there is an indexed grammar $G = (N, T, F, R, S)$, a subset N' of N, and a bijective mapping β: Ground $\rightarrow (N'F^*)^*$ such that

$$g \in Deriv(P, :- \pi(c)) \qquad \text{if and only if } \beta(g) \in Sent(G, N').$$

PROOF. Similarly to the procedure used in showing Proposition 1, we introduce mappings on an alphabet, on the set of atomic subgoals, and on the set of goals, built successively on top of each other.

We define $\bar{\beta}$ to map *Pred*(P) bijectively into a set N' of symbols and to map *Func*(P) bijectively into a set F of flags. Furthermore, $\hat{\beta}$ mapping atoms on strings in $N'F^*$ is constructed by letting

$$\hat{\beta}(p_i) = \bar{\beta}(s)\bar{\beta}(t_1)\ldots\bar{\beta}(t_n)$$

for atoms of the form $p_i = s(t_1(t_2(\ldots(t_n(c))\ldots)))$ such that $t_1, t_2, \ldots, t_n \in Func$, and finally, we let

$$\beta(:-p_1,\ldots,p_n) = \hat{\beta}(p_1)\ldots\hat{\beta}(p_n)$$

where p_i are atoms $(1 \le i \le n)$.

As a special case, the empty goal maps on the empty string.

This gives us a mapping: β: *Ground* $\rightarrow (N'F^*)^*$.

Each string in $(N'F^*)^*$ is uniquely decomposable into individual symbols and flags. For each such item, there is a unique predicate or function symbol mapped on it by $\bar{\beta}$. We construct the sets of rules and flag rules of an indexed grammar $G(P)$ as follows: Let a unary program P be given. Suppose clause *cl* in P has the following form:

$$p_0(f_{0,1}(\cdots f_{0,z}(X)\cdots))$$
$$:-p_1\big(f_{1,1}(\cdots f_{1,n_1}(X)\cdots)\big),\ldots,p_m\big(f_{m,1}(\cdots f_{m,n_m}(X)\cdots)\big).$$

Define in the set of flag rules $R_{\bar{\beta}(f_{0,1})}$ a new rule

$$\bar{\beta}(p_0) \rightarrow p_1^{cl},$$

in the sets of flag rules $R_{\bar{\beta}(f_{0,i})}$ $(2 \le i \le z)$ new rules

$$p_{i-1}^{cl} \rightarrow p_i^{cl}$$

and finally, in the set of rules R a new rule

$$p_z^{cl} \rightarrow \bar{\beta}(p_1)\bar{\beta}(f_{1,1})\ldots\bar{\beta}(f_{1,n_1})\ldots\bar{\beta}(p_m)\bar{\beta}(f_{m,1})\ldots\bar{\beta}(f_{m,n_m}).$$

The special case that the left-hand side of *cl* contains no function symbol $(p_0(X))$ is conveniently dealt with by defining no new flag rules and taking $\bar{\beta}(p_0)$ instead of p_z^{cl} in the only new rule of R.

Please note that we have introduced additional, auxiliary nonterminals which are not images of predicate symbols. $p_1^{cl}, p_2^{cl}, \ldots, p_z^{cl}$ are taken as new elements of the alphabet N.

The set of terminal symbols of the grammar may be any (possibly the empty) alphabet. The start symbol S is chosen as $\bar{\beta}(\pi)$.

We begin with the induction by stating that after zero steps of grammatical derivation, we have $S = \beta(:-\ \pi(c))$ as the only sentential form of G, and after zero steps of SLD-resolution, we have $:-\ \pi(c)$ as the only goal.

We define $Deriv_k(P, Gl)$ as the set of goals reached in no more than k resolution steps from Gl.

An analogous set $Sent_k(G(P), N')$ is defined to be the set of all sentential forms $W \in (N'F^*)^*$ that can be reached from S by $S = x_0 \rightarrow_G x_1 \ldots \rightarrow_G x_m = W\ (m \ge 1)$ such that no more than $k \le m$ elements of the sequence $x_1 \ldots x_m$ are contained in $(N'F^*)^*$.

In this way, intermediate stages containing nonterminals not in N' are skipped in the step count.

As an induction hypothesis, we now assume that the proposition holds for $Deriv_k(P :- \pi(c))$ and $Sent_k(G(P), N')$.

Then for a given goal

$$Gl = \left(:- P_1\left(f_{1,1}\left(\dots f_{1,n_1}(c)\dots\right)\right),\dots, P_k\left(f_{k,1}\left(\dots f_{k,n_k}(c)\dots\right)\right)\right)$$

reached from the initial goal after k or fewer resolution steps, the sentential form $\beta(Gl)$ is indeed derivable from S in the indexed grammar $G(P)$ and is an element of $Sent_k(G(P), N')$.

Suppose that the next resolution step uses $P_i(f_{i,1}(\dots f_{i,n_i}(c)\dots))$ and that, furthermore, $P_i(f_{i,1}(\dots f_{i,n_i}(c)\dots))$ unifies with the head of a rule

$$\begin{aligned} &P_i\left(h_{0,1}\left(\dots h_{0,q_0}(X)\right)\right)\\ &\qquad :- P_{i1}\left(h_{1,1}\left(\dots h_{1,q_1}(X)\dots\right)\right),\dots, P_{im}\left(h_{m,1}\left(\dots h_{m,q_m}(X)\dots\right)\right).\end{aligned}$$

For this to be possible, $P_i(f_{i,1}(\dots f_{i,n_i}(c)\dots))$ must have the sequence of function symbols $h_{0,1},\dots,h_{0,q_0}$ as an initial subsequence of length q_0 of its own sequence of function symbols $f_{i,1},\dots,f_{i,n_i}$.

Then we know that the resulting goal will have the form

$$\begin{aligned} Gl' = {} :- {} & P_1\left(f_{1,1}\left(\dots f_{1,n_1}(c)\dots\right)\right),\dots,\\ & P_{i-1}\left(f_{i-1,1}\left(\dots f_{i-1,n_{i-1}}(c)\dots\right)\right),\\ & P_{i1}\left(h_{1,1}\left(\dots h_{1,q_1}\left(f_{i,q_0+1}\left(\dots f_{i,n_i}(c)\dots\right)\right)\dots\right)\right),\dots,\\ & P_{im}\left(h_{m,1}\left(\dots h_{m,q_m}\left(f_{i,q_0+1}\left(\dots f_{i,n_i}(c)\dots\right)\right)\dots\right)\right),\\ & P_{i+1}\left(f_{i+1,1}\left(\dots f_{i+1,n_{i+1}}(c)\dots\right)\right),\dots, P_k\left(f_{k,1}\left(\dots f_{k,n_k}(c)\dots\right)\right).\end{aligned}$$

Now, by construction, the flag rules in $G(P)$ have been defined in such a way that for $z = q_0$, the sets $R_{\bar{\beta}(f_{i,1})},\dots,R_{\bar{\beta}(f_{i,z})}$ contain rules taking off the indices $h_{0,1},\dots,h_{0,z}$ and leading to an auxiliary nonterminal p_z^{cl} specific to that rule.

Furthermore, the rule

$$p_z^{cl} \to \bar{\beta}(P_{i1})\bar{\beta}(h_{1,1})\dots\bar{\beta}(h_{1,q_1}),\dots,\bar{\beta}(P_{im})\bar{\beta}(h_{m,1})\dots\bar{\beta}(h_{m,q_m})$$

adds index symbols at its jth nonterminal in agreement with the individual functioı symbols $h_{j,1},\dots,h_{j,q_j}$ occurring in $P_{ij}(h_{j,1}(\dots h_{j,q_j}(f_{i,z+1}(\dots f_{i,n_i}(c)\dots))\dots))$ $(1 \leq$
$\leq m)$.

Therefore, $\beta(Gl')$ is indeed a sentential form of $G(P)$. Since the intermediat stages leading to this string contained auxiliary symbols not in N', the step count i increased by 1 in both the grammar and the program.

On the other hand, there is no way for a derivation in $G(P)$ to proceed, excep by using rules constructed as just stated. In particular, an index sequence can onl be shortened by $\bar{\beta}(f)$ if the rule of P which leads to the construction of such a fla rule has the function symbol f in its head atom at the appropriate position.

To make this argument more exact, consider a sequence of derivation steps in $G(P)$ leading from a string x, all of whose nonterminals are images under $\bar{\beta}$ of predicate symbols, to another string with that property. In general, there will be intermediate stages whose number depends on the number of flag symbols that must be taken off. Note that the sentential forms containing auxiliary symbols are *not* images of goals. All rules and flag rules applied in this part of the derivation were constructed from a single clause cl in P.

Based on our knowledge that the proposition holds if the number of clause applications is no more than a fixed number k, let x be the image of a goal reached in k or fewer steps.

Putting aside the left and right contexts, $A_0 f_1 f_2 f_n$ is rewritten as $A_1 f_2 \ldots f_n, \ldots, A_{i-1} f_i \ldots f_n$ (for nonterminals $A_0, A_1, \ldots,$ and flags $f_1, f_2, \ldots$), and finally, as $y = x_1 z_1' \ldots x_m z_m'$ where z_j' $(1 \leq j \leq m)$ are concatenated from the flags given by the rewriting rule and the flags $f_i \ldots f_n$ left over from before. By the construction of β, this sequence of rewriting steps simulates a single application of cl. Since the induction hypothesis gives us that $A_0 f_1 f_2 \ldots f_n$ is $\beta(g)$ for some subgoal g, y is $\beta(g')$ for the sequence of subgoals g' resulting from g if cl is applied in P.

We have performed one resolution step and a sequence of derivation steps in $G(P)$, one of which contributes to the count in $Sent_k(G(P), N')$.

The arbitrary choice of a particular predicate symbol in the formulation of Proposition 2 plays only the minor role of determining a start symbol in the corresponding grammar. If we are willing to deal with strings generated from other symbols, the following version of the result may be preferred.

Corollary. Let P be a unary logic program. Then there is an indexed grammar $G = (N, T, F, R, S)$, and an injective mapping ζ: Ground $\to M^*$ such that for any predicate symbol π, any constant c, and any goal g

$$g \in Deriv(P, :- \pi(c)) \qquad \text{if and only if } \zeta(:- \pi(c)) \Rightarrow_G \zeta(g).$$

REMARK. The only difference in the construction is that instead of choosing an arbitrary predicate symbol which is then mapped on the start symbol, we proceed by first defining the mapping, and later on picking an arbitrary nonterminal as the start symbol of the grammar. The notion of sentential form is thus replaced by the more general transitive closure of the rewrite relation on pairs of strings. By using injectivity instead of bijectivity, the specifics of our subset technique are hidden.

7. TREATMENT OF TERMINAL SYMBOLS AND RELATED ISSUES

In defining the set of sentential forms which are then related to logic program goals by a one-to-one mapping, we have excluded all strings containing terminal symbols from consideration. There are two reasons for doing so. The first and most fundamental reason is that the essential distinction between terminals and nonterminals does not have any obvious analog in terms of predicate symbols used in logic programs. Strings consisting entirely of terminal symbols are incapable of being rewritten by the application of grammatical rules. This marks them as final products of a derivational process. On the logic programming side, goals incapable of being rewritten (i.e., resolved) signal a failed attempt to derive the empty goal, which is in turn the only successfully derived final product. To introduce a distinction similar to the one between terminals and nonterminals would require an artificial and otherwise unmotivated separation between two kinds of atoms of a logic program.

Apart from this aspect, a less fundamental but challenging problem arises from the different ways in which flag symbols are consumed in indexed grammar derivations. To see this, consider the rule set consisting of two rewrite rules

$$S \to Sf \quad \text{and} \quad S \to a,$$

where S may be assumed to be the start symbol, f the only available flag, and a the only terminal symbol. By repeated application of $S \to Sf$, we get sentential forms

$$Sff\ldots f$$

for arbitrarily long sequences of flags. Using the construction presented in the previous section, the goal corresponding to this by one-to-one mapping has the form

$$:\!-\,p_S(f(\ldots f(c)\ldots)).$$

Now, the decisive point is that such sentential forms can be replaced by the terminal symbol a in a single step because, according to the definition of indexed rewriting, flag strings are only passed on to occurrences of nonterminals. In unary logic programs, on the other hand, the terms resulting from an individual resolution step can only differ by a bounded number of (added or subtracted) function symbols from the term contained in the subgoal chosen for resolution. This property is quite straightforward because all terms in program clauses contain the (same) variable X at their innermost level of nesting. X unifies with all but a bounded-sized part of the term chosen and is reproduced in the resulting terms. A simulation of the grammatical derivation step considered here would therefore require an arbitrarily long sequence of resolution steps, and it is obvious that there can be no *bijective* mapping having these intermediate stages of resolution as images.

The purpose of this section is to show that by using a certain normalized version of indexed grammar, bijective mappings between goals and sentential forms are again possible.

We introduce the notion of "smooth" indexed grammars. Call an indexed grammar *smooth* if the start symbol S never occurs on the right side of any rule or flag rule and occurs on the left side of only one rule which has the form

$$S \to T\bot$$

where T is a nonterminal and $\bot$ is a flag not generated by any other rule, and if the set of flag rules (index rules) associated to the flag $\bot$ consists of rules of the form

$$A \to a$$

where A is a nonterminal, a is a terminal, and there are no other rules or flag rules with terminal symbols on their right-hand sides.

We can now claim the following.

Proposition 3. Every indexed language is generated by a smooth indexed grammar.

Furthermore, for smooth indexed grammars, an analog of Proposition 1 holds even if sentential forms containing terminals are included in the domain of the mapping.

Proposition 4. Let $G = (N, T, F, R, S)$ be a smooth indexed grammar. Then there exists a unary program $P(G)$ and a bijective mapping

$$\mu\colon K \to Ground(Func(P(G)), Pred(P(G)))$$

such that

$$w_1 \ldots w_n \in SeF(G) \qquad (w_i \in M \cup T, 1 \le i \le n)$$

if and only if

$$\mu(w_1 \ldots w_n) \in Deriv(P(G), \mu(S)).$$

The motivation of the smoothness construction is that rewriting rules with terminal symbols must somehow be forced to be applicable only after the flag sequences which they might otherwise delete in a single step have been cancelled flag by flag. This is achieved by placing a unique bottom symbol at the start of every flag sequence. Terminal replacements are restricted to the context of a reemergent bottom flag. To get to that point, nonterminal place-holders for the terminals must be capable of cancelling all other flags in individual auxiliary derivation steps.

To show Proposition 3, let

$$G = (N, T, F, R, S)$$

be an indexed grammar. Choose a new start symbol S' and construct a rule

$$S' \to S \perp$$

where $\perp$ is a new flag symbol. For every terminal t of G, provide a new unique nonterminal N_t. Replace every occurrence of t in some rule or index rule by N_t. Additionally, let $N_t \to N_t$ be an element of the index rule set R_f for all flags $f \neq \perp$ and all new symbols N_t, and let $N_t \to t$ be an element of the index rule set $R_\perp$ for all new symbols N_t.

As a consequence, for every derivation step

$$\alpha N \phi \beta \Rightarrow \alpha t \beta$$

where $\alpha, \beta \in K$, N a nonterminal, $\phi \in F^*$, the modified grammar permits a derivation step

$$\alpha N \phi \beta \Rightarrow \alpha N_t \phi \beta.$$

It should be noted that the final flag in ϕ is always $\perp$ by construction of the starting rule. The $N_t \to N_t$ rules permit a sequence

$$\alpha N_t \phi \beta \Rightarrow \cdots \Rightarrow \alpha N_t \perp \beta.$$

As $R_\perp$ contains $N_t \to t$, $\alpha N_t \perp \beta \Rightarrow \alpha t \beta$ holds. Alternative ways of deriving strings containing terminals have been excluded by the construction. On the other hand, every terminal word of the language is generated in this way.

To complete our discussion, we sketch the way in which a one-to-one mapping between derivation steps and resolution steps is obtained. This gives us Proposition 4.

As stated, derivation steps producing terminal symbols have the general form

$$\alpha N_t \perp \beta \Rightarrow \alpha t \beta.$$

Clearly, we can have clauses in logic programs of the type

$$p_{N_t}(f_\perp(X)) :- p_t(X).$$

Considering a smooth indexed grammar constructed as shown, we establish a bijection which associates, for instance, the terminal symbol t with a predicate symbol p_t. All derivation steps now have the property of consuming at most one flag. It is thus easy to see that the bijection can be extended to all sentential forms.

Let us now turn our attention to the nonterminal asymmetry.

While the set of terminal-free sentential forms and the set of goals constructed to simulate them stand in a one-to-one relationship in the proof of our Proposition 1, the analogous relationship induced by the construction in the proof of our Proposition 2 is weaker.

A one-to-one relationship holds between the set of goals and a specially characterized subset of the set of sentential forms.

The reason for this difference can be stated quite easily: By definition, indexed grammars cancel flag symbols one at a time. Clauses which contain terms consisting of two or more function symbols on their left-hand sides must therefore be simulated by several consecutively applied flag rules.

This situation in turn excludes the possibility of having a bijective mapping with regard to the entire set of sentential forms. While the difference appears somewhat superficial, it must obviously be respected unless we want to change the original concept of an indexed grammar to suit our purposes.

8. AN INFORMAL PRESENTATION OF NESTED STACK AUTOMATA

Shortly after introducing the class of indexed grammars, Aho gave an automata-theoretic characterization of the corresponding language class in terms of what he called *nested stack automata* (*nsa*) [3]. A discussion of the possible formal relationships between nested stack automata and logic programs is clearly beyond the scope of this article. In fact, the definition of nsa in [3] alone requires about two and a half pages of formalism. Nevertheless, it may be helpful to provide a general survey of the simulation in [3] as a particular kind of low-level implementation of indexed grammars. In contrast, definite clause programs obviously exemplify a very high level of representation. As an enhanced facility by comparison to those of the well-known pushdown automata, stack automata are capable of accessing any symbol previously written on the stack. This access is done by means of a pointer which can travel up and down the pushdown list one symbol at a time. However, replacements occur only at the top of the stack. A nested stack is a yet more powerful memory structure obtained by permitting stacks to be embedded or nested within stacks to arbitrary depths. Each such substack is delimited by top-of-stack and bottom-of-stack markers. There are four ways in which a configuration may change:

1. A new stack with a finite number of symbols may be formed between the presently active (i.e., pointed to) symbol and the symbol below it.

2. Empty stacks may be removed.
3. If the active symbol is at the top of some stack, it may be replaced by a finite length string of symbols, possibly by the empty string.
4. The pointer may be moved both ways with one restriction: It cannot be moved out of an existing stack at the top symbol. Moving out at the bottom is permitted.

The usual distinctions between one-way and two-way reading of input symbols and between deterministic and nondeterministic state changes give us four concrete classes of nsa. Aho's main result in [3] states that L is an indexed language if and only if L is accepted by some one-way nondeterministic nested stack automaton. Further results deal with closure properties of the classes of languages generated by variants of nsa. In the present context, we are interested in how an appropriate nsa recognizes the words generated by a given indexed grammar. We sketch the construction of the moves corresponding to individual rules of the grammar. For details, the reader is encouraged to consult the original publication.

At a stack top, changes of the stack corresponding to individual rewriting rules are possible. In this way, flag symbols, as well as terminal and nonterminal symbols, are generated in the same order as in the indexed grammar. Terminal symbols occurring at a stack top and matching the next symbol on the input tape are consumed and the input pointer is simultaneously moved on. Alternatively, when reading any particular nonterminal symbol A on the stack, the automaton may go into a unique state q_A associated to that symbol, erase A, and move to the right while in that state. When the symbol being read is a flag symbol f and there exists a flag rule $A \rightarrow B$ for flag f, the flag symbol is NOT erased, but the state q_A is changed into the neutral state q and a new nested stack containing the initial string B is created next to the flag symbol. Furthermore, empty stacks can be erased, and the automaton may move to the left in its neutral state. Finally, flag symbols at the top of stacks may be erased. This corresponds to the consumption of flags by terminal symbols in indexed grammars. The important point is that the entire derivation from a nonterminal rewritten by means of a flag rule is performed in a separate nested stack placed next to the flag. The flag itself survives that partial derivation, and is again available when another nonterminal must be rewritten in the same way.

In contrast to the derivation of sentential forms of an indexed grammar, we never get more than one copy of a particular flag instance generated by means of a rewriting rule. Instead, the same flag instance is used for all replacements. In global terms, this represents an extreme case of structure sharing. It is implemented by (nondeterministically) moving back and forth between symbols to be replaced and flag instances enabling their replacement. The development of a more efficient version of this procedure remains as a research problem.

Due to the close simulation of unary logic programs by indexed grammars presented above, the construction of an equivalent nsa for such programs is quite straightforward. There appears to be no reasonable alternative to first applying the procedure given in the proof of our Proposition 2 and subsequently transforming the indexed grammar as stated in [3]. A direct route might provide notational simplifications, but no advantages in principle.

By considering Aho's other main theorem, an nsa could be implemented in terms of unary logic programs. In that context, there would be no resultant separation of various rule sets, the analogous distinction being whether function symbols occur on the left-hand or right-hand side of a clause.

9. DISCUSSION OF PARSING QUESTIONS AND CONCLUDING REMARKS

The relationships which exist between the set of derivations in some indexed grammar G and the set of goals obtained by applying clauses of an appropriate logic program to a unique initial goal have been presented and discussed so far without consideration of the grammar-specific role of terminal symbols as distinguished from nonterminal symbols.

It has been felt that modifications of the concept of logic program aiming at some kind of analog of this distinction would be artificial. Nevertheless, it appears to be an interesting problem to what extent the class of indexed *languages* is affected by the results of this article.

While the focus of this paper has been on logic programs as such, rather than on any of the extant PROLOG-based grammar formalisms [1] which are in turn compiled or interpreted to provide parsers for their languages, it is worth noting that $P(G)$ of Proposition 1 can also be easily enhanced to serve as a parser for $L(G)$. In that sense, indexed grammars appear to be natural candidates for treatment in a logic programming framework.

The technique is, in fact, fairly standard.

Given an indexed grammar G, the first step would be the replacement of terminal symbols t in all rules by uniquely associated new nonterminals $N(t)$. Let us call the resulting grammar G'. Then the construction yielding a unary program $P(G')$ is used. Quite obviously, the predicate symbols $\overline{\mu}(N(t))$ do not occur in the head of any clause. Without wanting to get into technical details at this point, the subsequent development essentially requires two additional arguments for each predicate, giving us ternary (3-ary) instead of unary predicate symbols. Generally,

$$A(t_0) :- B_1(t_1), \ldots, B_n(t_n)$$

is transformed into

$$A(t_0, L_0, L) :- B_1(t_1, L_0, L_1), \ldots, B_n(t_n, L_{n-1}, L).$$

In the case of empty bodies, L is taken instead of L_0. This provides for the case of ϵ-productions. Furthermore, for all terminals t, we need a clause of the form

$$\overline{\mu}(N(t)) :- check(L_0, t, L_1)$$

where goals $check(L_0, t, L_1)$ are defined to succeed if the first argument is a list, the second is its head, and the third is its tail. *check* is usually available as a built-in predicate (called "C", for example) in PROLOG systems. Instead of $:- \overline{\mu}(S)(c)$ as the initial goal, we now have

$$:- \overline{\mu}(S)(c, TerminalString, [\ ])$$

provided that *TerminalString* is instantiated to a list of terminals.

To demonstrate this procedure, we take the example grammar shown before. The terminal symbols a and b are replaced by nonterminals A and B in all rules and flag rules. With function symbols f and g and predicate symbols S, T, U, A, B, the set of clauses $P(G')$ may be written as

$$S(X) :- A(X), T(g(X)). \quad T(X) :- A(X), T(f(X)). \quad T(X) :- U(X).$$
$$U(f(X)) :- B(X), U(X), A(X). \quad U(g(X)) :- B(X), A(X).$$

The above augmentation yields $S(X, L_0, L) :- A(X, L_0, L_1)$, $T(g(X), L_1, L)$, and likewise for the other clauses of $P(G')$.

$A(X, L_0, L_1) :- \mathit{check}(L_0, a, L_1)$ and $B(X, L_0, L_1) :- \mathit{check}(L_0, b, L_1)$ are added to deal with terminal symbols. Please note that the argument X is superfluous in these two rules. It is there because of the uniform treatment of all predicates in the construction of Proposition 1.

The construction which leads to the simulation of unary logic programs P by indexed grammars has the property that SLD-refutations in P correspond to the derivation of ϵ in G. As stated before, the existence of a refutation is thus reduced to a parsing problem.

It is admittedly hard to assess the practical usefulness of the techniques presented here. As an immediate observation, what we do have is a rare case of easy implementability for a well-known type of controlled context-free grammar. In [4], we showed that a simply stated unambiguity condition provides fast parsing for indexed grammars. It is to be expected that testable sufficient conditions for unambiguity will lead to optimized implementation of corresponding parsers. Beyond that, fast processing—even in terms of orders of magnitude—remains a difficult research problem. In their article on parsing and deduction [14], F.C.N. Pereira and D.H.D. Warren discuss a mixed top-down bottom-up (elsewhere called yo-yo [8]) strategy that follows Earley's [7] context-free parsing concept. In spite of the inherent efficiency of that procedure, its application to definite clauses appears already too general to transfer favorable time bounds. In fact, they feel compelled to state that "it is not at all obvious that grammar formalisms based on unification can be parsed within reasonable bounds of time and space." Nevertheless, they are able to present interesting estimates for the complexity of individual deduction steps and checks.

The extensive comments and suggestions of three reviewers are gratefully acknowledged. The corollary to Proposition 2 was, in fact, first formulated by one of them. Special thanks are due to the area editor, Veronica Dahl, for substantial assistance in preparing the revised version.

REFERENCES

1. Abramson, H. and Dahl, V., *Logic Grammars*, Springer, 1989.
2. Aho, A. V., Indexed Grammars—An Extension of Context-Free Grammars, *J. Assoc. Comput. Mach.* 15:647–671 (1968).
3. Aho, A. V., Nested Stack Automata, *J. Assoc. Comput. Mach.* 16:383–406 (1969).
4. Bertsch, E., Two Thoughts on Fast Recognition of Indexed Languages, *Information and Control* 29:381–384 (1975).
5. Deransart, P. and Maluszynski, J., Relating Logic Programs and Attribute Grammars, *J. Logic Programming* 1:119–225 (1985).
6. Deransart, P. and Maluszynski, J., What Kind of Grammars are Logic Programs?, in: Saint-Dizier, P. and Szpakowicz, S. (eds.), *Logic and Logic Grammars for Language Processing*, Ellis Horwood, 1990, pp. 29–55.

7. Earley, J., An Efficient Context-Free Parsing Algorithm, *Comm. ACM* 13:94–102 (1970).
8. Fisher, A. J., A "Yo-Yo" Parsing Algorithm for a Large Class of Van Wijngaarden Grammars, *Acta Informatica* 29:461–472 (1992).
9. Hopcroft, J. E. and Ullman, J. D., *Introduction to Automata Theory, Languages and Computation*, Addison-Wesley, 1979.
10. Isakowitz, T., Can We Transform Logic Programs into Attribute Grammars, *Informatique Theorique et Applications* 25:499–543 (1991).
11. Kolaitis, P. G., The Expressive Power of Stratified Logic Programs, *Information and Computation* 90:50–66 (1991).
12. Lloyd, J. W., *Foundations of Logic Programming*, Springer, 1987.
13. Pereira, F. and Warren, D. H. D., Definite Clause Grammar for Language Analysis, *Artificial Intelligence* 13:231–278 (1980).
14. Pereira, F. and Warren, D. H. D., Parsing as Deduction, in *Proceedings of the 1983 Conference of the Association for Computational Linguistics*, pp. 137–144.
15. Warren, D. S., Memoing for Logic Programs, *Comm. ACM* 35:93–111 (1992).