Natural Language Engineering **2** (1): 1–13 © 1996 Cambridge University Press

Efficient generation of random sentences

MARK-JAN NEDERHOF†

University of Groningen, Faculty of Arts,
P.O. Box 716, 9700 AS Groningen, The Netherlands

(*Received 17 November 1995; Revised 20 May 1996*)

Abstract

We discuss the random generation of strings using the grammatical formalism AGFL. This formalism consists of context-free grammars extended with a parameter mechanism, where the parameters range over a finite domain. Our approach consists in static analysis of the combinations of parameter values with which derivations can be constructed. After this analysis, generation of sentences can be performed without backtracking.

1 Introduction

Writing a grammar for a natural language is a difficult task. A number of tools have been devised to assist the author of a grammar. For example, detection of *undergeneration* (i.e. the grammar generates not all those sentences that it should) can be accomplished by automatically parsing a large corpus, and then singling out those sentences for which there is no parse according to the current version of the grammar (Nederhof and Koster, 1992).

For the detection of the opposite of undergeneration, viz. *overgeneration*, corpus analysis is not useful. The detection of overgeneration may be achieved however by (mechanical) random generation of sentences.[1] The grammar writer can inspect these sentences in order to find constructs which are unjustly generated by his grammar, and then make the appropriate corrections.

Random generators of sentences and sentence fragments have been incorporated into a number of development environments for grammatical formalisms (Yngve, 1962a; Yngve, 1962b; Lecarme, 1977; Karttunen and Kay, 1985; Boguraev *et al.*, 1988), among which is our own system, the *Grammar Workbench* (Nederhof and Koster, 1992). See Hofstadter (1979, Chapter 8) for more playful use.

Apart from the detection of overgeneration, some other applications can be found. For example, some grammar developers use sentence generation for monitoring the

† Supported by the Dutch Organization for Scientific Research (NWO), under grant 305-00-802. Much of the ideas in this paper were developed at the Computer Science Department of the University of Nijmegen.

[1] With "random" we mean in particular that the sentences are *not* generated from some semantic value.

https://doi.org/10.1017/S1351324996001234 Published online by Cambridge University Press

consequences that alterations to a grammar have with respect to the generated language.[2] This is done as follows. A set of sentences is generated from the grammar, once before and once after the alteration. The grammaticality of the sentences from one version of the grammar is then checked with regard to the other. This may reveal wanted or unwanted differences in the languages generated by the two versions. This is an example of *progressive evaluation* (Balkan *et al*., 1995).

Random generation of sentences may not only be applied for testing grammars, but also for testing parsers (Purdom, 1972; Mellish, 1989) and NLP systems (Balkan *et al*., 1995). In the area of programming languages, randomly generated programs may be used to test compilers (Payne, 1978; Celentano *et al*., 1980; Bazzichi and Spadafora, 1982; Murali and Shyamasundar, 1983; Bird and Munoz, 1983).

Further use may be found in the area of natural language interfaces: a frequent complaint is that a system's linguistic capabilities are not obvious to the user (Androutsopoulos *et al*., 1995, Section 3). Perhaps, automatically generated sentences may be used by the developer of the system to compose an example list of sentences within the linguistic coverage. This list is then applied to instruct users in the possibilities of the system.

This paper deals with some technical aspects of the random generation of sentences and sentence fragments. We use a formalism called AGFL, which is a type of context-free grammar extended with arguments.

Our first realisations of the generator made use of naive top-down backtracking techniques, with a limit on the depth of derivations to ensure termination in the presence of recursion. However, extensive experimentation convinced us that the time complexities of such algorithms were unacceptably high for practical grammars, even with some techniques of smart backtracking.

Additional problems were caused by informal requirements that the generated sentences should not be too short, to allow interesting constructs to occur, but neither too long, to allow insight into the structure of the sentences.

We have solved the problem of efficiency by precomputing for each non-terminal those combinations of argument values with which derivations from that non-terminal are possible. Together with such a combination of values, we also compute the minimal depth of such derivations.

The actual generation algorithm does not use any backtracking, so it has a very low time complexity. The problem of generating sentences of convenient lengths has been solved by applying appropriate heuristics.

Because non-terminals may have very large numbers of arguments, an efficient representation for sets of tuples of argument values was required. (This representation can also be applied for *parsing*, which has been discussed in Nederhof and Sarbo (1993).)

The algorithm presented in this paper was implemented in the aforementioned Grammar Workbench, a robust and stable piece of software which has been in use for over three years as tool in the development of grammars, and as an aid in

[2] Personal communication with Nelleke Oostdijk.

https://doi.org/10.1017/S1351324996001234 Published online by Cambridge University Press

teaching formal language theory to students. Although it can be used at all stages of grammar development, the most frequent use is after completion of (parts of) a grammar, for checking various well-formedness conditions and computing other properties.

The structure of this paper is as follows. In section 2 we introduce our formalism AGFL. The analysis of AGFLs is discussed in section 3, and the actual random generation of sentences, which uses the results of this analysis, is discussed in section 4. An efficient representation for sets of tuples, or in other word, for subsets of Cartesian products, is discussed in section 5. Some possible variants and extensions of the basic analysis and generation algorithms are outlined in section 6.

2 Affix grammars over finite lattices

Affix Grammars over Finite Lattices (AGFLs) are a restricted form of affix grammars (Koster, 1971; Koster, 1991; Nederhof, 1996). An affix grammar can be seen as a context-free grammar of which the rules are extended with *affixes* (cf. parameters, attributes, or features) to express agreement between parts of the rule. The distinguishing property of AGFLs is that the domains of the affixes are given by a restricted form of context-free grammar, the *meta grammar*, in which each right-hand side consists of one terminal.

Formally, an AGFL G is a 7-tuple $(A_n, A_t, A_p, G_n, G_t, G_p, S)$ with the following properties. The disjoint sets A_n and A_t and the function A_p together form the *meta grammar* of G, where

- A_n is the finite set of *affix non-terminals*;
- A_t is the finite set of *affix terminals*;
- A_p is a function from elements in A_n to subsets of A_t. The fact that A_p maps some affix non-terminal A to some set of affix terminals $\{x_1, \ldots, x_m\}$ is written as

$$A :: x_1; \ldots; x_m.$$

  We call the set $\{x_1, \ldots, x_m\}$ the *domain* of A.

The elements from A_n occur in the rules from G_p, which we define shortly, in the so called *displays*. A display is a list of elements from A_n, separated by ',' and enclosed in brackets '(' and ')'. Displays consisting of zero affix non-terminals are omitted.

The set of lists of zero or more elements from some set D is denoted by D^*. Thus, the set of all displays is formally described as A_n^*.

For the second part of an AGFL G we have

- G_n is the finite set of *non-terminals*. Each non-terminal has a fixed arity, which is a non-negative integer;
- G_t is the finite set of *terminals* ($G_n \cap G_t = \emptyset$);
- G_p is the finite set of *rules* $\subseteq (G_n \times A_n^*) \times ((G_n \times A_n^*) \cup G_t)^*$. Rules are written in the form

$$N(\vec{d}) : N_1(\vec{d}_1), \ldots, N_m(\vec{d}_m).$$

https://doi.org/10.1017/S1351324996001234 Published online by Cambridge University Press

```
PER :: 1; 2; 3.

NUM :: sing; plur.

simple sentence: pers pron (PER, NUM),
                 to be (PER, NUM),
                 adjective.

pers pron (1, sing): "I".
pers pron (1, plur): "we".
pers pron (2, NUM ): "you".
pers pron (3, sing): "he".
pers pron (3, plur): "they".

to be (1,   sing): "am".
to be (PER, NUM ): "are", [PER = 2;
                           PER = 1 | 3, NUM = plur].
to be (3,   sing): "is".

adjective: "supercalifragilisticexpialidocious".
```

Fig. 1. An example AGFL (with thanks to Mary Poppins).

where $m \geq 0$, and N is a non-terminal, which is followed by a display ($\vec{d}$), and each member N_i, $1 \leq i \leq m$, is a terminal or non-terminal, which is followed by a display ($\vec{d}_i$). The number of affix non-terminals in a display should correspond with the arity of the symbol preceding it (we assume the arity of terminals to be zero).

- The start symbol $S \in G_n$ has arity 0.

Example 1

The small AGFL in figure 1 illustrates some of the definitions above. This example incorporates a number of shorthand constructions, which are explained below.

The affix terminal '`sing`' in the first alternative of '`pers pron`' constitutes a so called *affix expression*, which is to be seen as an anonymous affix non-terminal, the domain of which is the singleton set {`sing`}. In general, an affix expression consists of one or more affix terminals, separated by '|', and denotes an anonymous affix non-terminal, the domain of which is the set of all specified affix terminals. For example, the affix expression '`1 | 3`' denotes an affix non-terminal with domain {`1`, `3`}.

Another example of shorthand is the *guard* in the second alternative of '`to be`', which specifies under what conditions a form of '`to be`' is `"are"`. The comma is to be seen as 'and' and the semicolon as 'or'. The operator '`=`' denotes unification.

The full syntax of AGFLs is defined in Dekkers *et al.* (1992). □

We use the term *variable* to refer to an instance of an affix non-terminal in an instance of a rule in a derivation (during parsing or generating).

Because the domains are finite, all possible values that variables can be instantiated with may be computed simultaneously, which reduces the amount of

https://doi.org/10.1017/S1351324996001234 Published online by Cambridge University Press

non-determinism during parsing (and generating). Unification of variables now is no longer a test on equality of values, but causes an intersection of two sets of values, which should be non-empty for the unification to succeed. It should be noted that even some formal presentations of AGFLs consider variables to be set-valued.

For now, we will assume that variables are instantiated with single affix terminals. We define an *instantiation* σ of a rule to be a mapping from the affix non-terminals in the rule to affix terminals from the domains of the non-terminals. If $(\vec{d})$ is some display in a rule and σ is an instantiation of that rule, then $\sigma(\vec{d})$ is defined to be the tuple of affix terminals that the affix non-terminals in $(\vec{d})$ are mapped to by σ.

In section 5, we will return to the idea that variables may be instantiated with sets of affix terminals.

3 Analysis of AGFLs

For the generation of sentences to be possible using an algorithm which does not use backtracking, we need to know in advance whether a derivation is possible from some non-terminal with some affix values within some restriction on the depths of derivations. The extra restriction on the depths of derivations pertains to our previously mentioned desire to generate sentences which are not too long. It also ensures termination of the generation process.

This information for each non-terminal and each tuple of affix values can be obtained by bottom-up analysis of the grammar. The result is a set of tuples for each non-terminal, such that derivations with those tuples of affix terminals exist. The tuples are partitioned according to the minimal depth of the possible derivations.

For our algorithm, we need the following variables. For each non-terminal N we have the sets $D_i(N)$ of all tuples t such that with t a derivation of depth i has been found, but not of any depth $i' < i$; the maximum value for i that we need depends on the grammar. We also have the sets $D(N) = \bigcup_i D_i(N)$ of all tuples with which derivations of any depth have been found.

We compute the sets $D_i(N)$ for all non-terminals N, for $i = 1, 2, 3, \ldots$ in that order, based on the sets $D_{i'}(N')$, for other non-terminals N' and for $i' < i$, which have been computed earlier. We assume that initially $D_i(N) = \emptyset$ and $D(N) = \emptyset$ for all non-terminals N and indices i. For technical reasons we also have constant values $D_0(T) = \{()\}$ for all terminals T.

Figure 2 presents the algorithm in pseudo code, which is stated in a naive way to enhance the simplicity of our presentation. The algorithm we actually implemented is more complicated and more efficient. In particular, we do not manipulate individual tuples but sets of tuples (see section 5).

Note that we can correctly deal with rules with empty right-hand sides if we define $maximum() = 0$.

https://doi.org/10.1017/S1351324996001234 Published online by Cambridge University Press

$i := 0$;
repeat $i := i + 1$;
  for each *rule* $N(\vec{d}) : N_1(\vec{d}_1), \ldots, N_m(\vec{d}_m)$.
    and instantiation σ *of this rule*
    and tuples $t_1, \ldots, t_m$
    and indices $i_1, \ldots, i_m$
  such that *for* $1 \leq j \leq m$ *we have*
    $t_j \in D_{i_j}(N_j)$ *and*
    $t_j = \sigma(\vec{d}_j)$, *and*
    $maximum(i_1, \ldots, i_m) = i - 1$
  do $t := \sigma(\vec{d})$;
    if $t \notin D(N)$
    then $D(N) := D(N) \cup \{t\}$;
      $D_i(N) := D_i(N) \cup \{t\}$
    end
  end
until $D_i(N) = \emptyset$ *for all* N
end.

Fig. 2. The analysis algorithm.

Example 2
Consider the following AGFL.

```
X :: 1; 2.
Y :: 1; 2; 3.
Z :: 1; 2.
W :: 2; 3.

a(X, Z): b(X, Y), c(Y, Z).
a(X, Z): a(X, Z), d(Z).

b(X, X): "p".
c(W, Z): "q".
d(W): "r".
```

The algorithm computes

$$D_0(\texttt{"p"}) = D_0(\texttt{"q"}) = D_0(\texttt{"r"}) = \{()\}$$
$$D_1(\texttt{b}) = \{(1,1),(2,2)\}$$
$$D_1(\texttt{c}) = \{(2,1),(2,2),(3,1),(3,2)\}$$
$$D_1(\texttt{d}) = \{(2),(3)\}$$
$$D_2(\texttt{a}) = \{(2,1),(2,2)\}$$

The $D_i(N)$ for all other combinations of i and N remain $\emptyset$. □

4 Random generation of sentences

Relying on the sets $D_i(N)$ which have been computed by the algorithm from the previous section, we can generate sentences top-down without backtracking. We

https://doi.org/10.1017/S1351324996001234 Published online by Cambridge University Press

procedure $\mathcal{G}(N, t, k)$:
  if N *is a terminal*
  then return *the name of* N
  else for some *rule* $N(\vec{d}) : N_1(\vec{d_1}), \ldots, N_m(\vec{d_m})$.
    and instantiation σ *of this rule*
    and tuples $t_1, \ldots, t_m$
    and indices $i_1, \ldots, i_m$
    such that $\sigma(\vec{d}) = t$, *and*
    for $1 \leq j \leq m$ *we have*
    $t_j = \sigma(\vec{d_j})$ *and*
    $t_j \in D_{i_j}(N_j)$ *and*
    $i_j < k$
    do *choose some numbers* $k_1, \ldots, k_m$ *such that*
    $i_j \leq k_j < k$ *for* $1 \leq j \leq m$;
    return $\mathcal{G}(N_1, t_1, k_1) + \mathcal{G}(N_2, t_2, k_2) + \ldots + \mathcal{G}(N_m, t_m, k_m)$
    end
  end
endproc

Fig. 3. The generation algorithm.

usually begin at the start symbol S, but we can also begin at any other non-terminal N. In the case we begin at N, we take some tuple t from $D_i(N)$, for some i. We know that derivations within some maximum depth $k \geq i$ from N with t are possible.

We determine some value $k \geq i$ using some heuristics (the higher k is chosen the deeper the computed derivation may be). We then call a procedure $\mathcal{G}$ with arguments N, t and k. This procedure is to return a string generated by some derivation from N with t, of some depth $\leq k$. (For some applications the derivation itself may be returned.)

The procedure $\mathcal{G}$ is a recursive function, which selects some rule such that derivations from the members with appropriate affix values are possible, and of which the depths are within the limit dictated by the argument k. The procedure then calls itself recursively for each member in the chosen rule with an appropriate tuple of values, and with the limit on the depths of derivations reduced by (at least) one. The string formed by the concatenation of the strings yielded by the recursive calls is the result of the procedure. (We denote concatenation of strings by the operator $+$.)

Formally, $\mathcal{G}$ is defined by figure 3.

The non-determinism in this algorithm may be resolved using random generators, possibly guided by some heuristics.

Example 3

For the running example, we may call, e.g. $\mathcal{G}(\mathtt{a}, (2, 2), 6)$, since $(2, 2) \in D_2(\mathtt{a})$ and $6 \geq 2$. One of the sentences which may be generated by this call is `"pqr"`, as shown by the call-graph in figure 4.

Due to the non-determinism of the generation algorithm, other third components in calls of $\mathcal{G}$ might have been used. For example, instead of the call $\mathcal{G}(\mathtt{d}, (2), 4)$, we may have had any call $\mathcal{G}(\mathtt{d}, (2), k)$ for k less than 6 (the third component of the

https://doi.org/10.1017/S1351324996001234 Published online by Cambridge University Press

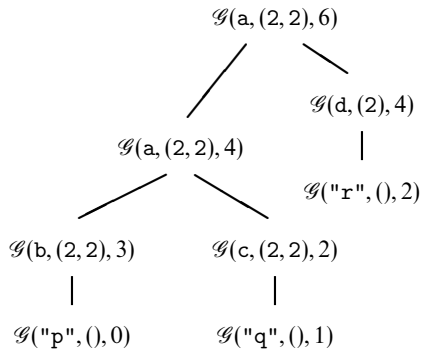


Fig. 4. One possible call-graph for generation of "pqr".

mother-node in the call-graph) but greater than or equal to 1 (the value i such that $(2) \in D_i(\mathtt{d})$).

Furthermore, other sentences, such as "pq" and "pqrr", would have been generated by $\mathscr{G}(\mathtt{a}, (2, 2), 6)$ if other rules had been chosen in recursive incarnations of $\mathscr{G}$. □

5 Efficient processing of sets of tuples

Typically, non-terminals in AGFLs for natural languages have large arities, in some cases more than eight. Furthermore, the domains sometimes contain more than fifty affix terminals. Consequently, processing and storage of all possible combinations of affix terminals is often not feasible.

Fortunately, the set of tuples of affix terminals with which derivations exist from a certain non-terminal is typically a Cartesian product of a number of domains. (For non-terminals where this does not hold, usually a mistake in the design of the grammar can be found.) Note that a Cartesian product of domains may be represented as a list of those domains, which requires $\mathcal{O}(m \times q)$ space although $\mathcal{O}(q^m)$ tuples may be represented, where m is the arity of the non-terminal and q is the size of the largest domain.

Also for the intermediate results in the analysis of section 3 and during random generation of sentences this representation may be used. In those cases, however, we sometimes need a union of a number of Cartesian products to represent a set of tuples. This amounts to the representation of a set of tuples of affix terminals by a set of tuples of sets of affix terminals.

For example, the set of tuples

$$\begin{aligned} \{\ (a,b,d) \quad &, \quad (a,b,e)\ , \\ (a,c,d) \quad &, \quad (a,c,e)\ , \\ (p,x,y) \quad &, \quad (q,x,y)\ \} \end{aligned}$$

can be represented by the set of tuples of sets

$$\{\ (\{a\},\{b,c\},\{d,e\}) \quad , \quad (\{p,q\},\{x\},\{y\})\ \}$$

https://doi.org/10.1017/S1351324996001234 Published online by Cambridge University Press

This form can be obtained by first taking the following set of tuples of singleton sets:

$$\begin{array}{lll} \{\ (\{a\},\{b\},\{d\}) & , & (\{a\},\{b\},\{e\})\ , \\ \ (\{a\},\{c\},\{d\}) & , & (\{a\},\{c\},\{e\})\ , \\ \ (\{p\},\{x\},\{y\}) & , & (\{q\},\{x\},\{y\})\ \} \end{array}$$

and by then merging pairs of tuples. E.g. $(\{a\},\{b\},\{d\})$ and $(\{a\},\{b\},\{e\})$ can be merged into $(\{a\},\{b\},\{d,e\})$. Repeated application of merging leads to the desired form.

The union of two sets represented in this form is complicated by the fact that two tuples may overlap. For example, if we want to compute the union of $\{(X,Y,Z)\}$ and $\{(P,Q,R)\}$, where X,Y,Z,P,Q,R denote sets of affix terminals, we may have that $X \cap P \neq \emptyset$ and $Y \cap Q \neq \emptyset$ and $Z \cap R \neq \emptyset$. A solution is to mould, for instance, $\{(P,Q,R)\}$ into a different form:[3]

$$\begin{array}{llll} \{\ (\ P-X\ , & Q & , \ R &)\ , \\ \ (\ P\cap X\ , & Q-Y\ , & R &)\ , \\ \ (\ P\cap X\ , & Q\cap Y\ , & R-Z &)\ , \\ \ (\ P\cap X\ , & Q\cap Y\ , & R\cap Z &)\ \} \end{array}$$

Possibly some of the tuples in this form may be discarded because one or more of their parts may be the empty set.

The union of the two sets is now represented by:

$$\begin{array}{llll} \{\ (\ X & , \ Y & , \ Z &)\ , \\ \ (\ P-X\ , & Q & , \ R &)\ , \\ \ (\ P\cap X\ , & Q-Y\ , & R &)\ , \\ \ (\ P\cap X\ , & Q\cap Y\ , & R-Z &)\ \} \end{array}$$

This example demonstrates that this representation can become fragmented considerably because of the union operation (this also holds for subtraction of sets of tuples, not shown here). To be exact, the union of k (non-overlapping) tuples of sets with m (non-overlapping) tuples of sets, respectively, may yield a set consisting of more than $k+m$ (again non-overlapping) tuples of sets, in general less than or equal to $k+m\times t^k$ tuples, where t is the width of the tuples.

To prevent quick degeneration of this representation to sets of tuples of singleton sets, it is necessary to regularly apply the merge operation, which reduces the number of tuples (possibly after deliberately fragmenting the representation further to allow more extensive merging).

We have found that the costs of merging tuples of sets do not outweight the high costs of operating on tuples of affix terminals. It required some experimenting, however, to determine how often the expensive merging operation had to be applied to improve instead of deteriorate the time complexity. In some cases, allowing temporary overlapping of tuples in a set proved to reduce the time-costs of the analysis.

[3] Note a similarity to lifting disjunction in feature descriptions (Eisele and Dörre, 1988).

https://doi.org/10.1017/S1351324996001234 Published online by Cambridge University Press

In the present implementation, overlapping is allowed during an iteration of the analysis algorithm, i.e. for one instance of counter i. At the end of each such iteration, the sets $D_i(N)$ and $D(N)$ are restructured by eliminating the overlapping of tuples and by applying the merging operation.

A representation of sets of tuples similar to ours is proposed in Zampuniéris and Le Charlier (1994). This representation, called *sharing trees*, may be more compact than ours, and has the advantage that a unique sharing tree exists for each set of tuples, which simplifies the test for equivalence. However, sharing trees were developed with a certain application in mind (analysis of synchronized automata), and it is not clear yet whether for our case sharing trees would behave any better than the above representation, using sets of tuples of sets. This is because the analysis of AGFLs requires different operations (such as subtraction), some of which are more complicated for sharing trees than for our representation.

6 Variants and extensions

The analysis from section 3 and the generation algorithm from section 4 are based on the idea of computing, for each non-terminal and each tuple, the minimal depth of which derivations exist. An obvious variant results from using other quantities besides the minimal depths of derivations. We could for instance also take the minimal lengths of generated sentences from non-terminals with certain tuples. Yet other quantities, such as the number of nodes in derivations, are possible as well. For some grammars such alternative quantities may give better results.

One may further consider the use of head information (van Noord, 1993) to influence the generation process from section 4. For example, one may choose k_j to be $k-1$ if $N_k(\vec{d}_k)$ is the head of the rule $N(\vec{d}) : N_1(\vec{d}_1), \ldots, N_m(\vec{d}_m)$., and i_j otherwise. This results in deep but narrow derivations, with the depth in the direction of the heads.

Such assignments to k_j may also play a role when sentences of specific forms are desired: the user may want to find sentences exhibiting a certain linguistic phenomenon. For this, certain subderivations may need to be deep, whereas other subderivations should be short and simple since they do not directly pertain to the investigated phenomenon (Balkan *et al.*, 1995). This idea may be realised by marking certain grammar rules that should be contained in the constructed derivations, or by allowing the user instead of the automatic generator to select the appropriate grammar rules at some critical moments (Boguraev *et al.*, 1988). After construction of a part of a derivation containing the required rules, where all k_j are chosen to be $k-1$, the remaining parts of the derivation are completed in the most simple way, by choosing all k_j to be i_j.

We have not considered the task of generating a set of sentences which is as small as possible, yet contains at least one occurrence of each rule. We refer to Purdom (1972), Celentano *et al.* (1980) and Bazzichi and Spadafora (1982) for such algorithms. We feel that these algorithms, devised for the testing of compilers for programming languages, are not very useful for the detection of overgeneration of natural language grammars, since overgeneration seems to be typically caused by

https://doi.org/10.1017/S1351324996001234 Published online by Cambridge University Press

unforeseen interaction of several rules, instead of by individual rules. The requirement that each rule occurs at least once is therefore not sufficient. However, the general idea of increasing the coverage of a limited set of generated sentences by avoiding frequent repetition of language constructions may lead to interesting variants of our algorithm.

Other possible refinements include extension of the grammar with probabilities (Payne, 1978; Murali and Shyamasundar, 1983; Bird and Munoz, 1983). Another extension is the application to unification grammars. For example, we have made a second implementation of the sentence generator for definite clause grammars (Pereira and Warren, 1980). Since the domains of the arguments are no longer finite, the analysis algorithm may not terminate if it is implemented as in figure 2. Therefore, we simplify the argument values according to certain criteria (Shieber, 1985). Because of this simplification of the analysis, the subsequent generation algorithm may fail on some occasions. Our experiments with HPSG-style grammars (Pollard and Sag, 1987) have shown, however, that failure occurs for about two out of three attempts to generate a sentence, which still allows significantly faster generation than when backtrack algorithms would be used.

A last remark concerns an optimization of the analysis with regard to the subsequent generation. Our algorithm from section 4 namely also allows generation from non-terminals other than from the start symbol S. Since from a certain non-terminal N not every other non-terminal M may be reachable, the execution of $\mathscr{G}(N,t,k)$, for some t and k may be performed without previous computation of $D(M)$ and $D_i(M)$, for many non-terminals M. In addition, the tuple t with which $\mathscr{G}$ is called may give rise to a further reduction of the search space, since not all tuples t' for a non-terminal M reachable from N may be relevant for derivations from N with t. Computing only the relevant tuples can, for example, be achieved by *magic transformations* (Debray and Ramakrishnan, 1994).

7 Conclusions

We have presented an algorithm which finds derivations without resorting to backtracking. This algorithm relies on an analysis of the grammar. The reason the algorithm always terminates is that a non-negative value is kept which decreases each time we descent the derivation under construction. The reason the algorithm cannot fail is that the results of the analysis provide enough information to successfully guide the algorithm towards a complete derivation.

The feasibility of the analysis depends on efficient processing of sets of tuples. Experience has shown that the analysis takes less time than finding a single derivation using a top-down backtrack algorithm.

Our technique allows much flexibility: many variants are possible, and the underspecified parts of the generation algorithm can be made more precise so as to generate sentences according to certain syntactic requirements.

https://doi.org/10.1017/S1351324996001234 Published online by Cambridge University Press

Acknowledgements

An anonymous referee provided helpful comments, which were gratefully used to improve this paper.

References

Androutsopoulos, I., Ritchie, G.D. and Thanisch, P. (1995) Natural language interfaces to databases – an introduction. *Natural Language Engineering* **1**(1):29–81.

Balkan, L., Arnold, D. and Fouvry, F. (1995) Test suites for evaluation in natural language engineering. *Proceedings of the Second Language Engineering Convention*, pp. 203–210, London.

Bazzichi, F. and Spadafora, I. (1982) An automatic generator for compiler testing. *IEEE Transactions on Software Engineering* **8**(4):343–353.

Bird, D. L. and Munoz, C. U. (1983) Automatic generation of random self-checking test cases. *IBM Systems Journal* **22**(3):229–245.

Boguraev, B., Carroll, J., Briscoe, T. and Grover, C. (1988) Software support for practical grammar development. *Proceedings of the 12th International Conference on Computational Linguistics*, **1** pp. 54–58, Budapest.

Celentano, A. *et al.* (1980) Compiler testing using a sentence generator. *Software—Practice and Experience* **10**:897–918.

Debray, S. and Ramakrishnan, R. (1994) Abstract interpretation of logic programs using magic transformations. *Journal of Logic Programming* **18**:149–176.

Dekkers, C., Koster, C. H. A., Nederhof, M.-J. and van Zwol, A. (1992) Manual for the Grammar WorkBench Version 1.5. Technical Report no. 92–14, University of Nijmegen, Department of Computer Science, July.

Eisele, A. and Dörre, J. (1988) Unification of disjunctive feature descriptions. *26th Annual Meeting of the Association for Computational Linguistics, Proceedings of the Conference*, pp. 286–294, Buffalo, NY.

Hofstadter, D. R. (1979) *Gödel, Escher, Bach: an Eternal Golden Braid*. Basic Books.

Karttunen, L. and Kay, M. (1985) Parsing in a free word order language. In D. R. Dowty, L. Karttunen and A. M. Zwicky (editors), *Natural Language Parsing: Psychological, computational, and theoretical perspectives*, pp. 279–306. Cambridge University Press.

Koster, C. H. A. (1971) Affix grammars. In J. E. L. Peck (editor), *ALGOL68 Implementation*, pp. 95–109. North Holland.

Koster, C. H. A. (1991) Affix grammars for natural languages. In *Attribute Grammars, Applications and Systems, International Summer School SAGA: Lecture Notes in Computer Science, 545*, pp. 358–373, Prague, Czechoslovakia. Springer-Verlag.

Lecarme, O. (1977) Usability and portability of a compiler writing system. In *Methods of Algorithmic Language Implementation: Lecture Notes in Computer Science, 47*, pp. 41–62. Springer-Verlag.

Mellish, C. S. (1989) Some chart-based techniques for parsing ill-formed input. *27th Annual Meeting of the Association for Computational Linguistics, Proceedings of the Conference*, pp. 102–109, Vancouver, British Columbia, Canada.

Murali, V. and Shyamasundar, R. K. (1983) A sentence generator for a compiler for PT, a Pascal subset. *Software—Practice and Experience* **13**:857–869.

Nederhof, M.-J. and Koster, K. (1992) A customized grammar workbench. In: J. Aarts, P. de Haan and N. Oostdijk (editors), *English Language Corpora: Design, Analysis and Exploitation*, Papers from the thirteenth International Conference on English Language Research on Computerized Corpora, pp. 163–179, Nijmegen. Rodopi.

Nederhof, M.-J. and Sarbo, J. J. (1993) Efficient decoration of parse forests. In: H. Trost (editor), *Feature Formalisms and Linguistic Ambiguity*, pp. 53–78. Ellis Horwood.

https://doi.org/10.1017/S1351324996001234 Published online by Cambridge University Press

Nederhof, M.-J. (1994) *Linguistic Parsing and Program Transformations*. PhD thesis, University of Nijmegen.

Nederhof, M.-J. (1996) On the borderline between finite and infinite argument domains. In: C. H. A. Koster and E. Oltmans (editors), *Proceedings of the First AGFL Workshop*, Technical Report CSI-R9604, Computing Science Institute, pp. 17–43, University of Nijmegen.

Payne, A. J. (1978) A formalised technique for expressing compiler exercisers. *SIGPLAN Notices* **13**(1):59–69.

Pereira, F. C. N. and Warren, D. H. D. (1980) Definite clause grammars for language analysis – a survey of the formalism and a comparison with the augmented transition networks. *Artificial Intelligence* **13**:231–278.

Pollard, C. and Sag, I. A. (1987) *Information-Based Syntax and Semantics*, vol. 1. Center for the Study of Language and Information, Leland Stanford Junior University.

Purdom, P. (1972) A sentence generator for testing parsers. *BIT* **12**:366–375.

Shieber, S. M. (1985) Using restriction to extend parsing algorithms for complex-feature-based formalisms. *23rd Annual Meeting of the Association for Computational Linguistics, Proceedings of the Conference*, pp. 145–152, Chicago, IL.

van Noord, G. (1993) *Reversibility in Natural Language Processing*. PhD thesis, University of Utrecht.

Yngve, V. H. (1962a) Computer programs for translation. *Scientific American* June, pp. 68–76.

Yngve, V. H. (1962b) Random generation of English sentences. *1961 International Conference on Machine Translation of Languages and Applied Language Analysis*, vol. 1, pp. 66–82, National Physical Laboratory, Teddington, Middlesex. HMSO.

Zampuniéris, D. and Le Charlier, B. (1994) A yet more efficient algorithm to compute the synchronized product. Research Report 94/5, Institute of Computer Science - FUNDP Namur, Belgium.

https://doi.org/10.1017/S1351324996001234 Published online by Cambridge University Press