UNRESTRICTED GAPPING GRAMMARS*

Fred Popowich

Natural Language Group
Laboratory for Computer and Communications Research
Computing Science Department
Simon Fraser University
Burnaby. B.C.. CANADA V5A 1S6

ABSTRACT

Since the introduction of *metamorphosis grammars* (MGs) 'Colmerauer. 1978). with their associated *type* 0-like grammar rules, there has been a desire to allow more general rule formats in logic grammars *Gaps,* which refer to strings of unspecified symbols, were added to the MG rule, resulting in *extraposition grammars* (XGs) (Pereira, 1981) and *gapping grammars* (GGs) (Dahl and Abramson, 1984). *Unrestricted gapping grammars,* which provide an even more general rule format, possess rules of the form "a - > B" where a and /3 many contain any number of terminal nonterminal or gap symbols in any order. FIGG. a Flexible Implementation of Gapping Grammars, is an implementation of a large subset of unrestricted GGs which allows either bottom-up or top-down parsing of sentences. This system provides more built in control facilities than previous logic grammar implementations, which allows the user to restrict the applicability of the rules, and to create grammar rules that will be executed more efficiently

1. INTRODUCTION

Gaps have been introduced into logic grammars, resulting in *extraposition grammars* (Pereira. 1981) and *gapping grammars* (Dahl and Abramson. 1984). to express a more general grammar rule that can be interpreted with "reasonable" efficiency by a computer. The rules of these grammars are of the form "n/. *a* > B". where *ni* is a nonterminal symbol called the *head,* and *a* and B may contain terminal symbols, nonterminal symbols, procedure calls, and gap symbols. Fxtraposition grammars are able to provide concise descriptions for left extraposition as found in sentences like *the mouse that the cat chased squeaked.* However, extraposition grammar rules are more restrictive than those of gapping grammars, since the gaps referenced on the left hand side of the rule must be repositioned in the same order at the end of the right hand side Also, the contents of multiple gaps must be nested (one gap totally contained within another), or non-mtersecting. With gapping grammars, concise descriptions of coordination, free word order, and right extraposition can be obtained (Dahl. 1984).

Unrestricted gapping grammars extend gapping grammars by the removal of the restriction that the left hand side ol all rules must start with a nonterminal symbol. Consequently, the unrestricted gapping grammar rules resemble "a —> 0". where a and B may contain terminals, nonterminals, gaps and procedure calls in any order. This type of rule facilitates easier description of unrestricted left movement of symbols

Unfortunately, the use of gaps can result in less efficient computer processing of the rules. Consequently, many applications of gapping grammars have not been explored except from a theoretical point of view. One method to circumvent this "efficiency problem" is to add procedural control to the otherwise declarative grammar rules. (The *cut* facility of Prolog (Clocksin and Mellish. 1981) is an example of this procedural intervention). FIGG. a *Flexible Implementation of Gapping Grammars,* is a programming language that incorporates procedural control to process many unrestricted gapping grammars with *tolerable response* time in an interactive environment. Using FIGG, the different forms of procedural control can be examined for their uses with unrestricted gapping grammars.

2- UNRESTRICTED GAPPING GRAMMARS

Unrestricted gapping grammars. like other logic grammars, use *logic terms* as grammar symbols. A *logic term* consists of a *functor,* which may possess zero or more *arguments.* Each *functor* possesses an *order,* which corresponds to the number of arguments, and is an element of some finite set F. The *arguments,* which are enclosed in parenthesis and separated by commas, may be *logic terms,* or *variables, H* [F] is used to refer to the set of logic terms that can be constructed from F. In this paper, elements of F will be represented by words starting with a lower case letter, or enclosed in single quotes. Words which start with an upper case letter or an *underscore "_".* will represent *variables.* A *list,* which is

This work was supported by the Natural Sciences and Engineering Research Council of Canada under Operating Grant no. A4309, Installation Grant no. SMI-74 and Postgraduate Scholarship #800.

"Gap symbols are used to reference sequences of unspecified symbols.

a logic term of the form '.'(α_1,'.'(α_2,...,'.'(α_n,nil)...)), is usually represented as [α_1,α_2, ...α_n]. Also, [*tl*] is a shorthand for '.'(*t*,*l*). During a derivation according to the grammar, variables may be *unified* with other logic terms (Clocksin and Mellish, 1981). Logic grammars also tend to possess facilities for handling *procedure calls* appearing within the grammar rules.

An unrestricted gapping grammar can be defined as a quintuple (V_N, V_T, Γ, Σ, P) where: V_N is the set of nonterminal symbols, $V_N \subset \vec{H}$ [F]; V_T is the set of terminal symbols, $V_T \subset \vec{H}$ [F], with $V_N \cap V_T = \phi$; Γ is the set of gap symbols, $\Gamma \subset \vec{H}$ [F], with $\Gamma \cap V = \phi$, where $V = V_N \cup V_T$; Σ is the set of starting symbols, with $\Sigma \subset V_N$, and P is the set of productions of the form

(2.1) α_0, gap(G_1), α_1, ... , gap(G_m), α_m --> β_0, gap(G'_1), β_1, ... gap(G'_n), β_n

with $m, n \geq 0$, $0 \leq i \leq m$, $0 \leq j \leq n$, $\alpha_i, \beta_j \in V^*$,*** and $gap(G_i)$, $gap(G'_j) \in \Gamma$. The rewrite relation, ==>, between elements of V^* may be defined as

(2.2) $\alpha_0\ \gamma_1\ \alpha_1 \ldots \gamma_m\ \alpha_m$ ==> $\beta_0\ \gamma'_1\ \beta_1 \ldots \gamma'_n\ \beta_n$

for a production (2.1) where $\gamma_i, \gamma'_j \in V^*$. Ignoring variable substitution, the language, L(G), associated with this grammar is

(2.3) $L(G) = \{\omega \in V_T^* \mid s ==>^* \omega \text{ for } s \in \Sigma\}$****

Unrestricted GGs can provide simpler description of some forms of left extraposition than was possible using GGs. To illustrate this point, let us examine a language, L_1, described in (Joshi, 1983). This language is obtained from $\{(ba)^n c^n \mid n \geq 1\}$ by "dislocating some *a*'s to the left." Using an unrestricted gapping grammar, this language can be described by the following productions.

(2.4) (a) s --> [b], a, s, [c].
(b) s --> [b], a, [c].
(c) gap(G), a --> [a], gap(G).

The rules are constructed to allow an *a* to be moved only once. An equivalent grammar using GG rules, that does not shift *b*'s to the right, requires the addition of at least one production.

(2.5) (a) s --> target, [b], target, a, s, [c].
(b) s --> target, [b], target, a, [c].
(c) target, gap(G), a --> [a], target, gap(G).
(d) target --> ϵ.

In (2.5), the nonterminal *target* represents a location where an *a* may be moved to, while epsilon, ϵ, corresponds to the empty string.

Along with easier description of left relocation of symbols, there is another phenomenon that follows from the removal of the nonterminal head restriction. The definition of rules resembling "ϵ --> β" is no longer prohibited. With this style of production, rule (2.4c) could be replaced by the following two rules.

(2.6) (a) target, gap(G), a --> [a], gap(G).
(b) ϵ --> target.

Any *targets* introduced somewhere to the left of an *a* by (2.6b), can be replaced by an *a* which is dislocated to the left according to (2.6a).

To facilitate further study of unrestricted gapping grammars, and to examine mechanisms for introducing procedural control to provide more efficiently *executable* productions, the FIGG programming language was developed. FIGG is a Prolog programme that implements a large subset of unrestricted gapping grammars.

3. FIGG

FIGG currently consists of a bottom-up shift-reduce parser and a top-down depth-first parser which can operate (independently) on a set of unrestricted GG rules. The system also provides built-in control operators which allow the user to create efficiently executable grammar rules. The top-down depth first backtrack parser incorporates these procedural control mechanisms in a parser which is based on one described in (Dahl and Abramson, 1984). It differs from its predecessor by allowing left recursion in its grammar rules, and by being more efficient, although not as general. Rules are still required to have a nonterminal as the head. The shift reduce parser used with FIGG is a variation of a context free shift reduce parser (Aho and Ullman, 1972) (Stabler, 1983) extended to allow non-context-free rules and gaps. To mirror the left to right processing of the top-down parser, the shift reduce parser processes a sentence from right to left. Details about the syntax and implementation of FIGG can be found in (Popowich, forthcoming).

The implementations of many previous logic grammars incorporated clumsy mechanisms for procedural control. Unless one resorted to arbitrary procedure calls, the only options available for such control were rule order, the introduction of marker symbols, or the *cut* operation. Increased control facilities provided in FIGG include control of gap processing, more sophisticated variations of *cut*, and restrictions on applicability of rules. For example, when the FIGG parser is processing a gap symbol, gap(G), it initially assumes an empty gap and then attempts to parse the next symbol. Through *backtracking*, the gap size is increased. However, a *decreasing gap*, gap(-G), is initially assumed to contain the rest of the sentence, and is decreased in size during backtracking. Also, although the *cut* behaves as in Prolog during top-down parsing, it behaves differently during bottom-up parsing since the top-down parser operates on terminal symbols like the definite clause grammar

***For any set *S*, $S^* = \cup_{i=0}^{\infty} S^i$.

****==>* is the reflexive transitive closure of ==>.

parser (Pereira and Warren. 1980) — while the shift reduce parser works with sentential forms. For bottom-up processing, each rule is converted into a single Prolog clause that modifies a *list* which corresponds to a *sentential form* When the right hand side of a rule matches part of a sentential form, the matched region can be replaced by the left hand side of the rule. Consequently, cuts and other control mechanisms — that appear in the right hand of a rule affect the left to right matching of the rule to a sentential form. Once the portion of a rule to the left of a cut has matched a sentential form, a subsequent failure in the match occurring to the right of the cut cannot force the match to the left of the cut to be reattempted. A cut found in the left hand side of a rule. *R*. will prevent any subsequent rule. R', from matching a region entirely to the right of the cut. That is. the application of $R^{>}$ to the sentential form resulting from the application of *R* must include at least one symbol to the left of the cut. If a rule. A', is entirely enclosed in a cut. *{R)*\, then the decision to apply A' to a sentential form cannot be revoked once the rule has been successfully applied.

4. USE OF PROCEDURAL CONTROL

Unrestricted gapping grammars, as implemented in FIGG. can be considered as a programming language, and can be used to provide parsers for languages, given a grammatical specification. Few studies have been done to examine the applicability of gapping grammars as a programming tool, since the earlier implementations were either inefficient or processed too small of a subset of these grammars. We will examine some ways to control the processing of unrestricted *GG* specifications to improve the efficiency of the parsing, and to restrict the language described. In this section, we shall use a selection of familiar formal languages to examine the use of the various control mechanisms. The use of F1GG with natural languages is examined in (Popowich. 1985) and (Popowich forthcoming).

Consider the context sensitive language $L_1 = \{a^m b^n c^m d^n \mid m,n \geq 0\}$. A set of productions (Dahl. 1984) of a grammar, G_1, that describes this language is provided in (4.1).

(4.1) (a) s --> as. bs. cs. ds.
(b) as. gap(G). cs --> [a]. as. gap(G). [c]. cs.
(c) bs. gap(G). ds --> [b]. bs. gap(G). [d]. ds.
(d) as. gap(G). cs --> [a]. gap(G). [c].
(e) bs. gap(G). ds --> [b]. gap(G). [d].

Behaviour of FIGG with this grammar and with, strings of increasing length is summarised in Table 4-1. Times are in CPU seconds for a SUN Workstation running C-Prolog (Pereira, 1984) under UNIX.***** The first number represents the time required for a successful parse. and the second number includes the time spent looking for all other possible parses. The results expose a severe parsing problem for G_1 with increasing sentence length. However, closer examination of (4.1) illustrates that the *gaps* should result in the *ith a* matching the *ith c*. and similarly for the *b*'s and *d*'s. If a decreasing gap is used in the productions. then the first successful gap followed by a *c* (or *d*. depending on the rule) will result in the correct matching. A cut can then prevent the other alternatives from being tried. Thus. G'_1 is obtained by modifying (4.1b-e) as shown in (4.2). resulting in much improved performance as illustrated in Table 4-1

(4.2) (b) as. gap(-.G). cs --> [a]. as. gap(-.G). [c]. !. cs.
(c) as. gap(-.G). cs --> [a]. gap(-.G). [c]. !.
(d) bs. gap(-.G). ds --> [b]. bs. gap(-.G). [d]. !. ds.
(e) bs. gap(-.G). ds --> [b]. gap(-.G). [d]. !.

While G_1 and G'_1 are processed by the top-down parser. G''_1 in Table 4-1 represents a grammar equivalent to G'_1 that is processed by the bottom-up parser. In this case. the bottom-up processing takes about two and a half times longer than top-down parsing

Table 4-1: Parse and total analysis times for $a^m b^m c^m d^m$

grammar	m=1	m=5	m=10	m=15	m=20	m=25	m=30
G_1	0.1	1.4	9.8	32.			
	0.2	6.3	85.	420.			
G'_1	0.1	0.5	1.6	3.2	5.4	8.4	11.8
	0.2	0.7	1.9	3.6	6.0	9.0	13.
G''_1	0.2	1.4	4.0	8.2	14.	22.	30.
	0.3	1.5	4.3	8.7	15.	22.	31.

Now consider the productions that describe the language $L_2 = \{a^n b^n c^n \mid n > 0\}$.

(4.3) (a) s --> [a]. bs. [c].
(b) s --> [a]. s. b. [c].
(c) bs. gap(G). b --> [b]. bs. gap(G).
(d) bs --> [b].

Unfortunately. a gapping grammar containing these productions would be ambiguous. The ambiguity can be removed through modification of (4.3b-c). as shown in (4.4). resulting in a new gapping grammar G'_2.

(4.4) (b) s --> [a]. s. b. c.
(c) bs. gap(-.G). b. c --> [b]. bs. gap(-.G). [c]. !.

An unambiguous unrestricted gapping grammar, G''_2, which has one less production and one less nonterminal symbol than G'_2, can also be provided for this language.

(4.5) (a) s --> [a]. [b]. [c].
(b) s --> [a]. s. b. c.
(c) ([b]. !. gap(-.G). b. c --> [b]. [b]. gap(-.G). [c])!.

The parse times for various sentences. where G'_2 and G''_2 are processed by the top-down and bottom-up parsers respectively. are summarised in Table 4-2. This time. the

*****UNIX is a trademark of Bell Laboratories.

bottom up parser is slower by a constant multiple of three However, its slowness is offset by the fact that it can process many more grammars than its top-down counterpart. An unrestricted gapping grammar can be used by the bottom-up parser as long as it does not result in *bottom-up cycles.* For example, any grammar containing the rule *"nt* --> €*. can not be used by this parser. Further development on this Prolog parser may improve its efficiency.

Table 4-2: Parse and total analysis times for $a^m b^m c^m$

grammar		n=1	n=5	n=10	n=15	n=20	n=25	n=30
G'_2	|	0.1	0.2	0.7	1.3	2.3	3.5	5.0
	|	0.1	0.3	0.9	1.7	2.7	4.0	5.7
G''_2	|	0.1	0.5	1.7	3.7	6.5	11.	15.
	|	0.1	0.7	2.2	4.5	7.9	12.	18.

Thus, the results illustrate that the introduction of some limited procedural control can be done simply with very beneficial results. Without its introduction, the processing time may be intolerable in some cases, it will not always be possible though, to introduce simple restrictions on gaps and parsing. The effect of a control mechanism is also very dependent on the grammar itself. As illustrated in the examples, the same operator — the cut — can result in more efficient parsing, or can restrict the language described by the grammar. The determination of which control to use. and how to use it. is the responsibility of the person who contructs the grammar. Obviously, more study of procedural control is required.

5. SUMMARY

Unrestricted gapping grammars provide more concise grammatical descriptions than previous logic grammar formalisms for many languages due to the more general rule formal allowed. However, with such a general rule formal, caution must be taken to insure that the grammar is restricted, by some form of control, to describe the required language Control facilities provided in F1GG permit refined control mechanisms while maintaining a high degree of descriptiveness in the grammar rules. These control facilities can be used either to restrict the language described by the grammar, or to obtain more efficient parsing The parsers of FIGG have successfully combined processing efficiency along with a large subset of unrestricted gapping grammars to produce a programming environment to study grammars and their languages.

ACKNOWLEDGEMENTS

I would like to thank Nick Cercone and the referees for their comments and suggestions. Facilities for this research were provided by the laboratory for Computer and Communications Research.

REFERENCES

Aho, A.V. and Ullman. J.D. The Theory of Parsing, Translation and Compiling. Englewood Cliffs. N.J Prentice Hall Inc.. 1972.

Clocksin. W.I. and Mellish. C.S. Programming in Prolog. Berlin-1 leidelberg-New York:Springer-Verlag, 1981.

Colmerauer. A. Metamorphosis Grammars. In L. Bole (Ed). Natural Language Communication with Computers, Springer Verlag. Berlin. 1978.

Dahl, V. More On Gapping Grammars. Proceedings of the International Conference on Fifth Generation Computer Systems. Institute for New Generation Computer Technology. Tokyo, 1984.

Dahl. V. and Abramson. 11 On Gapping Grammars. Proceedings of the Second International Joint Conference on Logic. University of Uppsala. Sweden. 1984.

Joshi, A.K Factoring Recursion and Dependencies: An Aspect of Tree Adjoining Grammars (TAGs) and a Comparison of Some Formal Properties of TAGs, GPSGs, PLGs and LPGs. pages 7-15. Proceedings of the 21th Annual Meeting of the Association for Computational Linguistics. June. 1983.

Pereira. F.C.N. Extra position Grammars. Ajnerican Journal of Computational Linguistics. 1981. 7(4). 243-256.

Pereira. F.C.N.(ed). C-Prolog User's Manual. Technical Report. SKI International. Menlo Park. California, 1984.

Pereira. F.C.N, and Warren. D.H.D. Definite Clause Grammars for Language Analysis. Artificial Intelligence. 1980. 13. 231-278.

Popowich. P. Unrestricted Gapping Grammars for ID/LP Grammars. Proceedings of Theoretical Approaches to Natural Language Understanding. Dalhousie University. Halifax Canada. 1985

Popowich. P. Effective Implementation and Application of Unrestricted Gapping Grammars. Master's thesis. Department of Computing Science. Simon Fraser University, forthcoming.

Stabler, E.P. (Jr) Deterministic and Bottom-Up Parsing in Prolog, pages 383-386. Proceedings of the American Association for Artificial Intelligence, August, 1983.