# ON GAPPING GRAMMARS

by

Veronica Dahl and Harvey Abramson

Technical Report 84-2

April 1984

# On Gapping Grammars

*Veronica Dahl*
*&*
*Harvey Abramson*

Department of Computer Science
Simon Fraser University
Burnaby, B.C. Canada

-

Department of Computer Science
University of British Columbia
Vancouver, B.C. Canada
604-228-4907

*ABSTRACT*

A Gapping Grammar (GG) has rewriting rules of the form:

$$\alpha_1,\ gap(x_1),\ \alpha_2,\ gap(x_2),\ \ldots,\ \alpha_{n-1},\ gap(x_{n-1}),\ \alpha_n \rightarrow \beta$$

$$\alpha_i \in V_N \bigcup V_T$$

$$G = \{\ gap(x_1),\ gap(x_2),\ \ldots,\ gap(x_{n-1})\ \}$$

$$x_i \in V_T^*$$

$$\beta \in V_N^* \bigcup V_T^* \bigcup G^*$$

where $V_T$ and $V_N$ are the terminal and non-terminal vocabularies of the Gapping Grammar. Intuitively, a GG rule allows one to deal with unspecified strings of terminal symbols called *gaps*, represented by $x_1,x_2,\ldots,x_{n-1}$, in a given context of specified terminals and non-terminals, represented by $\alpha_1,\alpha_2,\ldots,\alpha_n$, and then to distribute them in the right hand side $\beta$ in any order. GG's are a generalization of Fernando Pereira's *Extraposition Grammars* where rules have the form (using our notation):

$$\alpha_1,\ gap(x_1),\ \alpha_2,\ gap(x_2),\ldots,gap(x_{n-1}),\alpha_n \rightarrow \beta,\ gap(x_1),\ gap(x_2),\ \ldots,\ gap(x_{n-1})$$

i.e., gaps are rewritten in their sequential order in the rightmost positions of the rewriting rule. In this paper we motivate GG's by presenting grammatical examples where XGs are not adequate and we describe and discuss alternative implementations of GGs in logic.

## 1. Introduction

A grammar is a finite way of specifying a language which may consist of an infinite number of "sentences". A logic grammar has rules that can be represented as Horn clauses. Such logic grammars can conveniently be implemented by the logic programming language Prolog: grammar rules are translated into Prolog rules which can then be executed for either recognition of sentences of the language specified, or (with some care) for generating sentences of the language specified.

Since the development of the first logic grammar formalism by A. Colmerauer in 1975 (Colmerauer,1975), and of the first sizeable application of logic grammars by V. Dahl in 1977

(Dahl,1977), several variants of logic grammars have been proposed, sometimes motivated by ease of implementation (Definite Clause Grammars, DCGs, [Pereira&Warren,1980]), sometimes by a need for more general rules with more expressive power (Extraposition Grammars, XGs, [Pereira,1981]), sometimes with a view towards a general treatment of some language processing problem such as coordination (Modifier Structure Grammars, MSGs, [Dahl&McCord,to appear]), or of automating some part of the grammar writing process, such as the automatic construction of parse trees and internal representations (MSGs, op.cit; Definite Clause Translation Grammars, DCTGs, [Abramson,1984]). Generality and expressive power seem to have been the main concerns underlying all these efforts.

In this paper we present another logic grammar formalism called Gapping Grammars, GGs, which we believe to be the most general to date. We examine three possible implementations, and discuss the adequacy of GGs for certain language processing problems that cannot be expressed as easily in any other formalism.

GG rules can be considered as meta-rules which represent a set (possibly infinite) of ordinary grammar rules. They permit one to indicate where intermediate, unspecified substrings can be skipped, left unanalysed during one part of the parse and possibly reordered by the rule's application for later analysis by other rules. For instance, the GG rule:

$$A,\ gap(X),\ B,\ gap(Y),\ C \rightarrow$$
$$gap(Y),\ C,\ B,\ gap(X)$$

can be applied successfully to either of the following strings:

$$A,\ E,\ F,\ B,\ D,\ C$$

with gaps $X = E\ F$ and $Y = D$, and

$$A,\ B,\ D,\ E,\ F,\ C$$

with gaps $X = //$ and $Y = D\ E\ F$. Application of the rule yields

$$D\ C\ B\ E\ F$$

and

$$D\ E\ F\ C\ B$$

respectively. We can therefore think of the above GG rule as a shorthand for, among others, the two rules:

$$A,\ E,\ F,\ B,\ D,\ C \rightarrow D,\ C,\ B,\ E,\ F$$
$$A,\ B,\ D,\ E,\ F,\ C \rightarrow D,\ E,\ F,\ C,\ B$$

The idea of gapping grammars, as well as of the compiler implementation scheme shown below in Section 3.1 was developed in 1981 by V. Dahl as a result of examining Fernando Pereira's work on Extraposition Grammars, and finding the formalism limited, mainly with respect to the problem of treating coordinated constructs. During a 1982 visit to the University of Kentucky by V. Dahl, these ideas were tested in joint work with Michael McCord, but were later suspended in favour of a more promising approach to the coordination problem (see [Dahl&McCord,to appear]). We (Dahl & Abramson) now resume this research, no longer with a view towards treating coordination, but because the formalism itself has some rather interesting aspects.

Gapping grammars are interesting in the first place because each meta-rule, somewhat like a restricted version of VanWijngarden's two-level grammars which were used in the definition of Algol 68 [VanWijngarden,1975], represents infinitely many specific rules: each gap can be satisfied by many strings of terminals; to specify each of these unstructured substrings might require infinitely many grammar rules in other formalisms. Gapping grammars therefore cover a wide variety of rewriting situations using very few rules.

Secondly, there seems to be some psychological basis to the idea of focusing on the next relevant substring during analysis and leaving an intermediate one suspended in the background of consciousness, to be brought back into the focus of attention later, possibly repositioned with other more closely related substrings. When parsing discontinuous constituents, for instance as in the course and colloquial sentences "Desmond knocked the girl with green eyes down" as opposed to "Desmond knocked the girl with green eyes up", the human hearer will probably suspend his attention from the intermediate string "the girl with green eyes" until the completing substring to "Desmond knocked", i.e., "down" or "up", is heard, repositioned, and comprehended within its interrupted context.

A third argument for sometimes not specifying which constituents should be intermediate between two substrings is the fact that there is some empirical linguistic evidence in support of the existence of categories intermediate between lexical and phrasal categories [Radford,1981]. While these aren't clearly captured as traditional categories in linguistic theory, it is possible to computationally account for them simply by perceiving and naming them as gaps.

## 2. Background, Motivation, and Definition of Gapping Grammars.

Logic grammars originated with A. Colmerauer's Prolog implementation of Metamorphosis Grammars as an alternative notation for logic programs. They consist of rewriting rules where the non-terminal symbols may have arguments, and rule application may therefore involve unification. For instance, a rule such as:

$$np(X) \rightarrow name(X)$$

can be applied to the strings $np(4)$ and $np(anne)$ yielding, respectively, $name(4)$ and $name(anne)$ but cannot be applied to either of the strings $np$ or $np(x,y)$. The left hand side of a normalized Metamorphosis Grammar rule must start with a non-terminal symbol, but may be followed by a sequence of terminals (terminal symbols are written between / and /), whereas the right hand side may contain any sequence of terminals and non-terminals, as in:

$$a, [b], [c] \rightarrow [b], a, [c]$$

(Unnormalized Metamorphosis Grammars may contain rules beginning with a terminal, followed possibly by other terminals and non-terminals; there is no loss of generality, however, in restricting oneself to normalized MGs. See [Colmerauer,1978].) Definite Clause Grammars, DCGs, are a simplification of MGs in that rules are allowed only a single non-terminal on the left hand side, as in:

$$verb\_phrase(X) \rightarrow verb(X,Y),\ object(Y)$$

Extraposition Grammars (XGs) allow the interspersing of gaps in the left hand side,and these are routinely rewritten in their sequential order at the rightmost end of the rule, as in:

$$rel\_marker,\ gap(X),\ trace \rightarrow$$
$$rel\_pronoun,\ gap(X)^1 \qquad (1)$$

In an XG rule, symbols on the left hand side following gaps represent left-extraposed elements (e.g., "trace" above marks the position out of which the "noun_phrase" category is being moved in the relativization process).

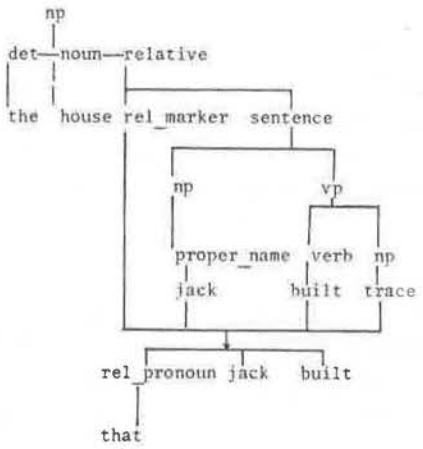
Let us briefly examine the step-by-step rewriting of a sentence with a relative clause to understand how the gapping rule above works. Our complete grammar is:

```
sentence -> np, vp
```

[1] We use our notation for consistency. Pereira's notation for *gap(X)* is written "..." in the left hand side and simply left implicit on the right.

```
np -> proper_name
np -> det, noun, relative
np -> trace

vp -> verb, np
vp -> verb

relative -> []
relative -> rel_marker, sentence

rel_marker,gap(X),trace ->
    rel_pronoun,gap(X)

det -> [the]

noun -> [house]

rel_pronoun -> [that]

proper_name -> [jack]

verb -> [built]
```

Applying these rules as graphed below, we analyse "the house that jack built" from *np:*




where the gap is "jack built". Notice that by adding appropriate symbol arguments to the rules, we can carry the antecedent's representation all the way to the constituent from which it was moved. Also notice that the same grammar, but with a larger lexicon, serves to analyse, for example, the sentence "the women who built the house", this time with an empty gap, and with the *trace* derived from the first *np* in the relative sentence.

Thus, XGs allow us to describe left-extraposition phenomena powerfully and concisely, and to arrange for the desired representations to be carried on to the positions from which something has been extraposed.

### 2.1. Motivation.

#### 2.1.1. Left extraposition with more than one gap.

While XGs have the expressive power just shown, the restriction on how gaps are rearranged poses some expressive constraints even within the framework of left-extraposition. Consider for instance the noun phrase:

the man with whose mother john left

We can consider this noun phrase as the result of left-extraposing two substrings from:

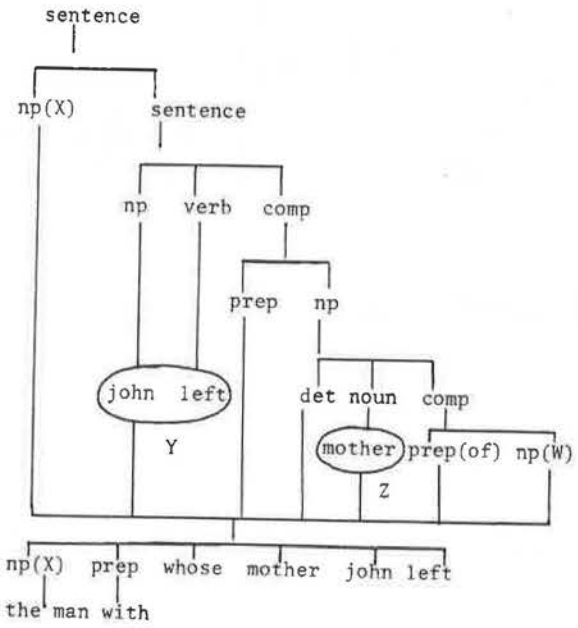
the man [ john left with [the] mother [of the man]]

where "of the man" is left-extraposed before "the", and subsumed with it into "whose", and the whole complement is extraposed to the left of "john left".

If we wanted to capture these movements in a single rule (which seems a practical way, since they are all related), we might express it through the somewhat simplistic but illustrative rule:

$$np(X),\ gap(Y),\ prep,\ det,\ gap(Z),\ prep(of),\ np(X)$$
$$\rightarrow np(X),\ prep,\ [whose],\ gap(Z),\ gap(Y)$$

where X stands for the internal representation that is built up from the noun phrase being analysed. A derivation graph for this example would look roughly like:




Notice that the gapping rule's application unifies the internal representation X for "the man" with the representation W of the rightmost complement. The result of one partial analysis thus spreads to cover all implicit occurrences of the same substring.

### 2.1.2. Equivalent, preferred gapping formulations.

Fernando Pereira gives the following XG for the language $\{a^n b^n c^n\}$:

#### 2.1.2.1. Grammar 1.

s -> as, bs, cs.

as -> [].
as, gap(X), xb -> [a], as, gap(X).

bs -> [].
bs, gap(X), xc -> xb, [b], bs, gap(X).

cs -> [].
cs -> xc, [c], cs.

Other formulations of grammars which use gaps are conceivable, however, and it should be up to the grammar writer to choose a formulation unconstrained by fixed reordering rules. The following GG, for example, describes the same language:

#### 2.1.2.2. Grammar 2.

```
s -> as, bs, cs.

as -> [].
as -> xa, [a], as.

bs -> [].

xa, gap(X), bs -> gap(X), [b], bs, xb.

cs -> [].

xb, gap(X), cs -> gap(X), [c], cs.
```

In Grammar 1, symbols such as *xb* can be considered as marks for *b*'s which are being left-extraposed. In Grammar 2, such marks can be seen as right-extraposed. Whereas in this particular example our choice may just be a matter of personal preference, there are cases in which movement is more naturally thought of in terms of right rather than left-extraposition (as in "The man is here that I told you about"). There also may be efficiency reasons to prefer a right-extraposing formulation: in the first implementation below of GGs, Grammar 2 above works faster than Grammar 1.

### 2.1.3. Interaction between different gapping rules.

Consider the language $\{a^n b^m c^n d^m\}$ which can be described by the following GG:

```
s -> as, bs, cs, ds.

as -> [].
as, gap(X), xc -> [a], as, gap(X).

bs -> [].
bs, gap(X), xd -> [b], bs, gap(X).
cs -> [].
cs -> xc, [c], cs.

ds -> [].
ds -> xd, [d], ds.
```

This is a perfectly good GG. XGs cannot, however, be used in this situation because of the XG constraint on the nesting of gaps: two gaps must either be independent, or one gap must lie entirely within the other.

## 3. Implementations of GGs.

### 3.1. A Compiler: beautiful but dumb.

Typically, logic grammars are translated into Prolog programs by augmenting each non-terminal symbol with two arguments: one argument $X$ which represents the input string yet to be parsed, and the other argument $Y$ which represents what is left of the input string after the rule being applied has consumed some of it. We then say that the string $X - Y$ (read as "$X$ minus $Y$) can be recognized as belonging to the category denoted by the non-terminal. Thus, a rule such as:

$$sentence \rightarrow name,\ verb.$$

is translated into a Prolog clause:

$$sentence(X_1,X_3) \leftarrow$$
$$name(X_1,X_2),\ verb(X_2,X_3) \qquad \text{(a)}$$

meaning roughly: "there is a sentence in the string $X_1 - X_3$ if there is an initial substring $X_1 - X_2$

that can be parsed as a *name* and is followed by a substring $X_2 - X_3$ that can be parsed as a *verb*".

Terminal symbols do not give rise to Prolog predicates, but become instead involved in the specification of the input and output strings being manipulated by the non-terminals. For instance, the rules

$$name \rightarrow [mary].$$

$$verb \rightarrow [laughs].$$

can be translated into the unit clauses:

$$name([mary|X],X) \leftarrow \quad (b)$$

$$verb([laughs|X],X) \leftarrow \quad (c)$$

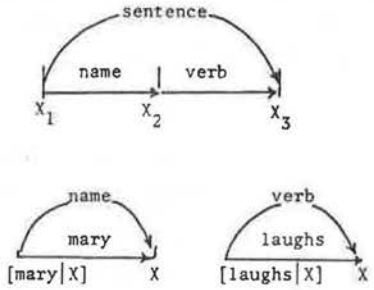
where (*b*) means roughly: "a *name* is recognized in any input string which begins with 'mary', yielding an output string which is the remainder of the input string after consuming 'mary'".

Thus, with respect to rules (*a*), (*b*), and (*c*), a query such as:

$$sentence([mary,laughs],[])$$

will unify $X_1$ with *[mary,laughs]* and $X_3$ with *[]*, proceed to consume a name from $X_1$, yielding $X_2 = [laughs]$, and then consuming a verb from $X_2$, yielding $X_3 = []$. The string $X_1 - X_3$, i.e., *[mary,laughs]* – *[]*, has been recognized as a sentence.

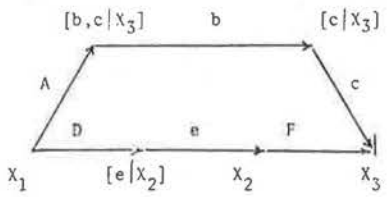
Let us now consider a graphical representation of this translation process, where non-terminals are viewed as labeled arcs connecting nodes representing phrase boundaries. Rules (*a*), (*b*), and (*c*) above can then be represented as follows:




Normalized MG rules accept a sequence of terminals after the single non-terminal head on the left hand side (since more than one non-terminal would result in a non-Horn clause). The translation of such a rule to Prolog may be represented graphically by adding more arcs to the upper part of the graph. The rule

$$A, [b], [c] \rightarrow D, [e], F$$

would translate as indicated by:



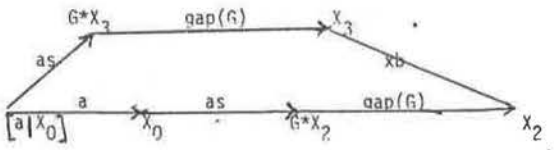

which is the Prolog clause:

$$A(X_1,[b,c|X_3]) \leftarrow D(X_1,[e|X_2]),\ F(X_2,X_3).$$

Let us now consider a rule with gaps and how it should be represented graphically:

$$as,\ gap(G),\ xb \rightarrow [a],\ as,\ gap(G)$$

We can think of a gap $G$ simply as a substring of the input that is skipped unanalyzed and appended elsewhere in the output string. Thus, if we denote the appending of $G$ to a string $x$ as $G*x$, we can represent this rule graphically by:




The symbol $gap(G)$, in fact, can be thought of as a version of *append*. When translating rules into clauses, $gap(G)$ becomes the predicate call $gap(G,X_1,X_0)$, which can be specified as the appending of $G$ to $X_0$ yielding $X_1$. In other words, the input string $X_1$ is non-deterministically divided into two substrings $G$ and $X_0$. The rule above can thus be expressed in Prolog as:

$$as([a|X_0],X) \leftarrow as(X_0,X_1),$$
$$gap(G,X_1,X_2),\ gap(G,X,X_3),\ xb(X_3,X_2)$$

or alternatively as:

$$as([a|X_0],X) \leftarrow as(X_0,X_1),\ append(G,X_{2,X_1}),$$
$$append(G,X_3,X),\ xb(X_3,X_2)$$

Notice that the remainder of the rule's left hand side becomes a part of the clauses's body, and we therefore remain within the Horn clause subset of first order logic. Notice too, that this translation scheme allows the left hand side of a rule to be nearly as unrestricted as the right hand side: though the head must be a non-terminal, any combination of terminals, non-terminals, and even Prolog calls can be expressed as context.

The compiler shown below produces the corresponding Prolog clauses from gapping grammar rules by first constructing two pseudo-rules. From a rule such as

$$A,\ B \rightarrow C$$

where $A$ is the non-terminal head symbol, and where $B$ is the remainder of the left hand side of the rule, and $C$ is the right hand side, it constructs clauses corresponding to the pseudo-rules:

$$c\_nonterm \rightarrow C$$
$$b\_nonterm \rightarrow B$$

where *c–nonterm* and *b_nonterm* are pseudo-non-terminals. In doing so, it also binds the output strings corresponding to both pseudo-non-terminals. In our example, the clauses generated are:

$$c\_nonterm([a|X_0],Z) \leftarrow as(X_0,X_1),\ gap(G,X_1,Z)$$
$$b\_nonterm(X,Z) \leftarrow gap(G,X,X_3),\ xb(X_3,Z)$$

Next it constructs the head of the desired clause by using and retrieving the input and output strings from the input strings of *c_nonterm* and *b_nonterm*. In our example this yields

$$as([a|X_0],X)$$

The desired clause's body is constructed by appending the two bodies of the pseudo-clauses

$$as([a|X_0],X) \leftarrow as(X_0,X_1),$$
$$gap(G,X_1,X_2),\ gap(G,X,X_3),xb(X_3,X_2)$$

The compiler's full listing is shown below. In addition to accepting purely syntactic gapping

grammar rules, it also accepts gapping grammar rules with a superadded *semantic* component to specify a translation. The general form of such a rule is:

$$A, B \rightarrow C <:> Sem$$

where *Sem* consists of one or more Horn clauses which specify how semantic attributes of the head symbol $A$ are computed in terms of semantic attributes of $C$ and possibly even of $B$. The Horn clauses in *Sem* govern traversal of the derivation or parse tree which is constructed automatically [Abramson,1984]. Gapping grammar rules which are purely syntactic have the trivial semantic unit clause *true* attached to them. The predicate *form_node* below creates the derivation tree for the head symbol $A$ by concatenating the trees for the pseudo-clauses corresponding to $B$ and $C$.

```
synal((A,B -> C<:>Sem),Clause) :- !,
 expand_term(
    (c_nonterm->C<:>Sem),CClause),
 expand_term((b_nonterm->B),BClause),
 clauseparts(CClause,CHead,CBody),
 clauseparts(BClause,BHead,BBody),
 CHead =.. [c_nonterm,CTree,X,Z],
 BHead =.. [b_nonterm,BTree,Y,Z],
 A     =.. [Pred|Args],
 form_node(CTree,BTree,Pred,ATree),
 concaten(Args,[ATree,X,Y],NewArgs),
 NewA  =.. [Pred|NewArgs],
 combine(CBody,BBody,Body),
 formclause(NewA,Body,Clause).

synal((A,B -> C),Clause) :- !,
   synal((A,B -> C<:>true),Clause).

clauseparts((Head :- Body),Head,Body) :- !.
clauseparts(Head,Head,true).

formclause(Head,true,Head) :- !.
formclause(Head,Body,(Head :- Body)).

combine(true,B,B) :- !.
combine(A,true,A) :- !.
combine(A,B,(A,B)).

form_node(node(_,N1,Sem),
        node(_,N2,_),
        Pred,node(Pred,N,Sem)) :-
   concaten(N1,N2,N).

gap([]) -> [].
gap([Word|List]) -> [Word], gap(List).

concaten([],X,X).
concaten([X|L],M,[X|N]) :- concaten(L,M,N).
```

The beauty of the compiler resides in its simplicity and conciseness. The compiler is dumb, however, in that the gap predicate successively consumes substrings of length 0, 1, 2,... with no further control than simple backtracking as to what should be in the gap. Thus, even on simple languages, such as $\{a^n b^n c^n\}$ with relatively low values of $n$, say $n = 5$, it is very slow. Some more information needs to be incorporated in the gap predicate, but this seems to involve dynamic information about the state of the computation, and such information is accessible only in some Prolog implementations. Another alternative which we are considering is to use concurrency in parsing; we sketch this idea below and are planning a future detailed article on the subject.

Although the ideas on compiling GGs are due to V. Dahl, credit is due to Michael McCord for the actual writing of the compiler in terms of pseudo-clauses.

### 3.2. Another Compiler: Efficient but not general.

In this section we introduce a class of Gapping Grammars which can be implemented in Prolog efficiently. This class consists of those Gapping Grammars in which each gapping rule is of the form:

$$\alpha,\ gap(X),\ /term/ \rightarrow \gamma,\ gap(X) \qquad (A)$$

That is, there is only one gap which is rewritten to the rightmost position of the right hand side, and on the left there is a single (pseudo-)terminal following the gap. This class of grammars includes a subclass of Pereira's Extraposition Grammars, but also, depending on the definition of the *gap* and *fill* predicates, may include grammars which cannot be handled by Extraposition Grammars, such as, for example, a grammar for the language $\{a^n b^m c^n d^m\}$, with $m,n \geq 0$.

This class may be viewed as a generalization of normalized Metamorphosis Grammars. A normalized Metamorphosis Grammar rule is of the form:

$$\alpha,\ \beta \rightarrow \gamma \qquad (B)$$

or

$$\alpha \rightarrow \gamma \qquad (C)$$

where

$$\alpha \in V_N$$

$$\beta \in V_T^*$$

and

$$\gamma \in V_T^* \bigcup V_N^*$$

The notation $gap(X),\ /term/$ therefore represents a large set of MG rules.

The implementation technique is based on message passing during parsing and rests on the following considerations. The terminal symbols which occur on the left hand side of XG rules and to the immediate right of a gap may be said to be pseudo-symbols in that they are generally not expected to occur in input strings to be parsed, but are generated during parsing to act as signals of some sort, and are absorbed later in the parse. Consider, for example, in the XG grammar for the language $\{a^n b^n c^n\}$ the rule:

$$as,\ gap(X),\ xb \rightarrow /a/,\ as,\ gap(X)$$

The $xb$ is generated to mark the end of the gap and to count an occurrence of an $/a/$. The $xb$ is then absorbed by a matching $/b/$ in the rule:

$$bs,\ gap(X),\ xc \rightarrow xb,\ /b/,\ bs$$

Similarly, in the XG for a small subset of English, the rule (1) in Section 2 generates *trace* to mark the point from which a noun phrase has been left-extraposed, and the rule

$$np \rightarrow trace$$

absorbs the *trace*. The introduction of such pseudo-symbols, moreover, produces a slight theoretical problem in that they may occur in some sentential forms of the grammar, but not in the terminal sentential forms.

Since there is only one gap in our restricted rules, and this gap is followed by a "terminal", we write instead of $A$ the following:

$$\alpha,\ gap(X),\ /term/ \rightarrow \gamma \qquad (D)$$

and read this: an $\alpha$, in the context of a gap which is terminated by a signal *term* may be

rewritten to a $\gamma$ followed by the gap. The gap is implicit on the right hand side of the rule. Thus our signal gapping grammar rules are of the form (B), (C) or (D). The sending of a signal which closes such a gap is indicated by the predicate

*fill(term)*

which generates (accepts) the empty string. Our version of the grammar for the language $\{a^n b^n c^n\}$ is as follows:

```
s->as,bs,cs.

as,gap(X),[xb]->[a],as.
as->[].

bs,gap(X),[xc]->fill(xb),[b],bs.
bs->[].

cs->fill(xc),[c],cs.
cs->[].
```

In implementing this form of GG we specialize the *synal* predicate as follows:

```
synal(((A,gap(Name),[Signal])->C<:>Sem),
   Clause) :- !,
 expand_term(
   (c_nonterm->(C,gap(Signal,Name))<:>Sem),
   CClause),
 clauseparts(CClause,CHead,CBody),
 CHead =.. [c_nonterm,G0,Gn,CTree,X,Z],
 A     =.. [Pred|Args],
 GTree = node(gap,[Signal,Name],true),
 form_node(CTree,GTree,Pred,ATree),
 append(Args,[G0,Gn,ATree,X,Y],NewArgs),
 NewA  =.. [Pred|NewArgs],
 combine(CBody,
        gap(Signal,Name,GTree,Y,Z),
        Body),
 formclause(NewA,Body,Clause).

synal(((A,gap(Name),[Signal]) -> C),
  Clause) :- !,
 synal(
  ((A,gap(Name),[Signal]) -> C<:>true),
  Clause).
```

In the goal *expand_term* the *Signal* is added to the named gap which is placed at the right end of the syntactic portion of the rule; since the only context of a rule is of the form *gap(Name), [Signal]*, we dispense with *BClause* and construct the clause for *A* directly; other changes in *form_node* involve the formation of a "tree" to record the contents of the gap as a difference list (see below). *synal* compiles, for example, the rule

*bs, gap(Name), [xc]* → *fill(xb), [b], bs.*

to:

```
bs(S0,
   S3,
   node(bs,
        [FillTree,
```

```
        b,
        BsTree,
        GapTree,
        node(gap,[xc,Name],true)],
       true),
  X,
  Y) :-
 fill(xb,S0,S1,FillTree,X,X1),
 c(X1,b,X2),
 bs(S1,S2,BsTree,X2,X3),
 gap(xc,Name,S2,S3,GapTree,X3,Z),
 gap(xc,Name,node(gap,[xc,Name],true),Y,Z).
```

The reader will notice that in addition to the pair of arguments for the "input" and "output" strings (*X*,*X*1,*X*2,*X*3,*Y*,*Z*), and the argument for the parse tree, there is another pair of arguments - the "input message stream" and the "output message stream" - which has been added to all the non-terminals except the rightmost instance of *gap*. These are *S*0, *S*1, *S*2, and *S*3, and are added to non-terminal symbols only by the predicate *translate_rule* (not shown here, but called by *expand_term*: see [Abramson,1984]) which processes non-gapping rules. Note that non-gapping rules are normalized metamorphosis grammar rules and are translated as outlined in Section 3.1. The ordinary non-terminals, such as *bs*, will neither add messages to the input stream nor delete messages from the input stream in order to produce a new output stream: the input stream will be passed to whatever is called, and a possibly new output stream will be formed as a result of the call. Messages are inserted by *gap* and removed by *fill*. Let us examine the definition of *gap* and *fill* to see how these streams are manipulated:

```
gap(Symbol,
    Gap,
    node(gap,[Symbol,Gap],true),
    StartGap,
    EndGap) :-
      Gap = StartGap - EndGap.

gap(Symbol,
    Gap,
    StackIn,
    [[Symbol,Gap]|StackIn],
    node(gap,[Symbol,Gap],true),
    StartGap,
    EndGap) :-
      Gap = StartGap - EndGap.

fill(Symbol,
     [[Symbol,Gap]|StackOut],
     StackOut,
     node(fill,[Symbol,Gap],true),
     EndGap,
     EndGap) :-
      Gap = StartGap - EndGap.
```

When *gap* is called with a pair of stream arguments, the start of a gap is known. *Gap* is instantiated to the difference list *StartGap* – *EndGap*, with *EndGap* uninstantiated. The pair *[Symbol,Gap]* is added to the input message stream to form a new output message stream. The *Symbol* is the signal which will indicate the end of a gap. When *gap* is called without the stream arguments, as in the last call to gap in the compiled version of *bs*, the context is merely being

checked (please refer to the discussion of *synal* in the previous section) and the input and output strings, *StartGap* and *EndGap*, respectively, verify the extent of the gap. *EndGap* will still be uninstantiated.

When *fill* is called, the end of a gap with the signal *Symbol* has been found. There must be a pair of the form */Symbol,Gap/* at the front of the input message stream. *EndGap* is instantiated at this point, and the pair is removed from the input message stream to yield a new output message stream. When *EndGap* is instantiated, the "trees" of the *gap* and *fill* predicates, which have been made to look like ordinary non-terminals, are also instantiated. The trees for both *gap* and *fill* contain a record of the signal *Symbol* and the gap itself as the difference list to which *Gap* is instantiated. The message streams act as a stacking mechanism for unfilled gaps. Note that *fill* accepts the empty string.

A string is parsed with this grammar by a call to:

```
s(Source) :-
    s([],[],Tree,Source,[]).
```

which indicates that *Source* is an *s*, with no input left, and that no messages are left in the streams, ie, the stack of messages, initially empty, is empty at the end of parsing. A parse tree *Tree* records the derivation. (See [Abramson,1984]).

With this definition of *gap* and *fill* we have a new implementation of a subset of XGs: it contains rules with only one gap followed by a terminal. The compiler for this subset, *synal* above, is somewhat simpler than the general processor of Pereira.

By changing the definition of *gap* and *fill*, however, we can process grammars which cannot be handled by XGs. Here is our signalling GG for the language $\{a^n b^m c^n d^m\}$:

```
s->as,bs,cs,ds.

as,gap(X),[xc]->[a],as.
as->[].

bs,gap(X),[xd]->[b],bs.
bs->[].

cs->fill(xc),[c],cs.
cs->[].

ds->fill(xd),[d],ds.
ds->[].
```

We redefine *gap* and *fill* so that the input and output message streams manipulate a pair of stacks, one to handle *xc* signals, and the other to handle *xd* signals. The gaps can now be dealt with independently of one another.

```
gap(Symbol,
    Gap,
    node(gap,[Symbol,Gap],true),
    StartGap,
    EndGap) :-
      Gap = StartGap - EndGap.

gap(xc,
    Gap,
    [StackC,StackD],
    [[[xc,Gap]|StackC],StackD],
    node(gap,[xc,Gap],true),
    StartGap,
    EndGap) :-
      Gap = StartGap - EndGap.
```

```
gap(xd,
    Gap,
    [StackC,StackD],
    [StackC,[[xd,Gap]|StackD]],
    node(gap,[xd,Gap],true),
    StartGap,
    EndGap) :-
      Gap = StartGap - EndGap.

fill(xc,
     [[[xc,Gap]|StackC],StackD],
     [StackC,StackD],
     node(fill,[xc,Gap],true),
     EndGap,
     EndGap) :-
      Gap = StartGap - EndGap.

fill(xd,
     [StackC,[[xd,Gap]|StackD]],
     [StackC,StackD],
     node(fill,[xd,Gap],true),
     EndGap,
     EndGap) :-
      Gap = StartGap - EndGap.
```

The general GG implementation is very powerful and inefficient; this implementation, although not general, is more efficient; and is, at the cost of some programming of the *gap* and *fill* predicates by the grammar writer, extendable to classes of grammars with independent gapping systems which cannot be handled by XGs. It is interesting that subclasses of GGs can be parameterized by data structures: one may think of trying to characterise the subclass of GGs with a queue (deque, tree) implementation of *gap* and *fill*, for example.

A complete listing of these Prolog implementations is available from H. Abramson or V. Dahl.

### 3.3. Towards a concurrent implementation of gapping grammars.

The beautiful but dumb compiler is inefficient because of the way it tries to establish what is contained in a gap. It simulates the non-deterministic breaking up of the input string into the contents of the gap and the unconsumed output string by trying one solution of *append(Gap,Output,Input)*, backtracking to the next solution if the first is not suitable, and so on. A concurrent implementation might, however, proceed as follows. For each solution of *append(Gap,Output,Input)* a copy of the process which represents the state of the parse so far is created. Each of these processes is a clone of the original process up to the call of *gap*. Each process continues, however, with a different solution to *append(Gap,Output,Input)*. Those processes which have not been given a solution which will permit the parse to continue will eventually die. Those processes which have been given a solution which allows the parse to complete will each be left suspended at the end with a derivation tree representing the successful parse. (Note that this notion of process is similar to the notion of process which is used in the Unix operating system.) For this strategy to work, it will be necessary to have a meta-logical predicate which gives access to the state of a Prolog computation. This strategy utilizes independent sequential Prolog processes - the parsing, except when handling a gap, proceeds by top-down, depth-first search with backtracking. An alternative strategy would be to develop an entirely concurrent implementation of grammars.

The authors plan to investigate whether Concurrent Prolog [Shapiro,1983], the distributed logic of [Monteiro,1982], or Epilog [Pereira,1982], [Porto,1982] could easily specify such implementations of Gapping Grammars.

## 4. Discussion, work in progress.

### 4.1. Advantages of gapping grammars.

GGs, although theoretically no more powerful than MGs - which have the computational power of a Turing machine - have more expressive power than MGs in that they permit the specification of rewriting transformations involving components of a string separated by arbitrary strings. The expressive power takes the form of conciseness: one does not have to give a rule or rules for the generation of the intervening string, but rather a single meta-rule involving gaps replaces a possibly infinite set of non-gapping rules.

One aspect of GG expressiveness has not yet been fully explored. GGs, like MGs and XGs, allow Prolog calls in the right hand side of a rule, but unlike them, GGs allow Prolog calls in the left hand side of a rule (refer to *synal* above to see why this is so). It is possible therefore to write GGs which can establish context checks dynamically during parsing.

The compiler for GGs - our second implementation - provides an alternative implementation of a restricted class of extraposition grammars, but also, depending on the definition of *gap* and *fill*, provides the grammar writer with a mechanism for writing rules which go beyond the nesting constraints of the XG formalism. Our example above shows how to deal with two independent gapping systems: the extension to the general case is obvious. Another possibility is to parameterize classes of grammars by the data structures used to implement the *gap* and *fill* predicates, for example, by queues instead of stacks, etc. Another extension lies in permitting the signal to be parameterized, i.e., instead of having rules of the form (*D*) with *term* a functor of zero arity, *term* might be a functor of positive arity. This would permit more sophisticated gap handling by the *gap* and *fill* predicates.

### 4.2. Limitations.

In some cases GGs may prove, however, to be too powerful. Consider, for instance, the following grammar which one naively might think suitable for checking that input strings are balanced with respect to ( and ):

```
left, gap(X), [')'] -> ['('], gap(X).

s -> left, [')'], gap(X), s.
s -> [].
```

With this grammar, strings such as $(a + ( b + c))$ and $((a + b) - (c * d)) / f$ are recognized as balanced, but also a string such as $(a + b$ is recognized as balanced. The reason for this error is that nothing in the grammar precludes the gaps from containing parentheses, so that unbalanced parentheses will be absorbed into gaps. The grammar can, of course, be modified so that only those strings which are balanced with respect to parentheses are accepted, but it seems appropriate for the grammar formalism to provide the user with a convenient means for constraining the gaps. It would be interesting to determine how much of an extension along these lines could be usefully provided without falling into the trap of describing the complement of a language.

Another approach to be investigated with respect to too general a notion of gaps is allowing strings *not* in the language to be generated, these strings to be subsequently filtered out by another process. Primitives for describing filters would then be necessary. In natural language applications, a mixture of both approaches may be needed. Both constraints and filters have already been proposed in Chomsky's Extended Standard Theory (see references in [Radford,1982]), and, it would be interesting to study ways of constraining and filtering GG rules in the light of this theory.

### 4.3. Work in progress.

We have only tentatively sketched a concurrent implementation of GGs. Details of this strategy have to be worked out and specified in Prolog, Concurrent Prolog [Shapiro,1983], the distributed logic of [Monteiro,1982], or Epilog [Pereira,1982], [Porto,1982]. Ideally, a parallel

architecture should support a concurrent GG system.

Another implementation of GGs which we are exploring is an interpreter which works with derivations directly rather than with Prolog calls of non-terminal procedures. By this method, we would for a rule such as

$$s \rightarrow as, bs, cs$$

rather than call *as*, then *bs*, then *cs*, maintain a list of goals which would represent a sentential form. The original list of goals *s* would be replaced by a list of goals *as*, *bs*, *cs*. Context sensitive rules would involve manipulation of the goals in the sentential form to see if some of them could appropriately derive the desired context.

These extensions, as well as the addition of constraints and filters to GGs, are the object of future research by the authors.

## 5. Acknowledgements.

This work was supported by the National Science and Engineering Research Council of Canada. Michael McCord's contribution, mentioned in Section 3.1, is gratefully acknowledged. A referee's suggestion that we expand on our comments about right extraposition will be treated in a later paper: length constraints prevent such expansion here.

## 6. References.

Abramson, H., Definite Clause Translation Grammars Proceedings IEEE Logic Programming Symposium, 6-9 February 1984, Atlantic City, New Joisey.

Colmerauer, A., Metamorphosis Grammars, in Natural Language Communication with Computers, Lecture Notes in Computer Science 63, Springer, 1978.

Dahl, V. Translating Spanish into logic through logic, American Journal of Computational Linguistics, vol. 13, pp. 149-164, 1981.

Dahl, V. & McCord, M. Treating Coordination in Logic Grammars, to appear in American Journal of Computational Linguistics.

Monteiro, L., A Horn clause-like logic for specifying concurrency, Proceedings of the First International Logic Programming Conference, pp. 1-8, 1982.

Pereira, F.C.N., Extraposition Grammars, American Journal of Computational Linguistics, vol. 7 no. 4, 1981, pp. 243-255.

Pereira, L.M., Logic control with logic, Proceedings of the First International Logic Programming Conference, pp. 9-18,1982

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

Porto, A. Epilog - a language for extended programming in logic, Proceedings of the First International Logic Programming Conference, pp. 31-37, 1982.

Radford, A., Transformational Syntax Cambridge University Press, 1981.

Shapiro, E.Y., A subset of Concurrent Prolog and its interpreter, ICOT Technical Report TR-003, 1983.

Van Wijngarden, A. et al., Revised report on the algorithmic language Algol 68, Acta Informatica, vol. 5, pp. 1-236, 1975.