CANADIAN THESES

THÈSES CANADIENNES

NOTICE

The quality of this microfiche is heavily dependent upon the quality of the original thesis submitted for microfilming. Every effort has been made to ensure the highest quality of reproduction possible.

If pages are missing, contact the university which granted the degree.

Some pages may have indistinct print especially if the original pages were typed with a poor typewriter ribbon or if the university sent us an inferior photocopy.

Previously copyrighted materials (journal articles, published tests, etc.) are not filmed.

Reproduction in full or in part of this film is governed by the Canadian Copyright Act, R.S.C. 1970, c. C-30.

AVIS

La qualité de cette microfiche dépend grandement de la qualité de la thèse soumise au microfilmage. Nous avons tout fait pour assurer une qualité supérieure de reproduction.

S'il manque des pages, veuillez communiquer avec l'université qui a conféré le grade.

La qualité d'impression de certaines pages peut laisser à désirer, surtout si les pages originales ont été dactylographiées à l'aide d'un ruban usé ou si l'université nous a fait parvenir une photocopie de qualité inférieure.

Les documents qui font déjà l'objet d'un droit d'auteur (articles de revue, examens publiés, etc.) ne sont pas microfilmés.

La reproduction, même partielle, de ce microfilm est soumise à la Loi canadienne sur le droit d'auteur, SRC 1970, c. C-30.

THIS DISSERTATION
HAS BEEN MICROFILMED
EXACTLY AS RECEIVED

LA THÈSE A ÉTÉ
MICROFILMÉE TELLE QUE
NOUS L'AVONS REÇUE

NL-339 (r 86/06)

Canada

Unrestricted Gapping Grammars: Theory, Implementations, and Applications

by

Fred P. Popowich

BSc, University of Alberta, 1982

THESIS SUBMITTED IN PARTIAL FULFILLMENT OF
THE REQUIREMENTS FOR THE DEGREE OF
MASTER OF SCIENCE
in the Department
of
Computing Science

SIMON FRASER UNIVERSITY
July 1985

ISBN 0-315-30849-4

Approval

Name: Fred P. Popowich

Degree: Master of Science

Title of Thesis: Unrestricted Gapping Grammars: Theory, Implementations, and Applications

Examining Committee:

Chairperson: Hassan Reghbati

Nick Cercone
Senior Supervisor

Veronica Dahl
Senior Supervisor

Robert Hadley

Harvey Abramson
External Examiner
Associate Professor
Department of Computer Science
University of British Columbia

July 10, 85
Date Approved

PARTIAL COPYRIGHT LICENSE

I hereby grant to Simon Fraser University the right to lend my thesis, project or extended essay (the title of which is shown below) to users of the Simon Fraser University Library, and to make partial or single copies only for such users or in response to a request from the library of any other university, or other educational institution, on its own behalf or for one of its users. I further agree that permission for multiple copying of this work for scholarly purposes may be granted by me or the Dean of Graduate Studies. It is understood that copying or publication of this work for financial gain shall not be allowed without my written permission.

Title of Thesis/Project/Extended Essay

UNRESTRICTED GAPPING GRAMMARS:

THEORY, IMPLEMENTATIONS, AND APPLICATIONS

Author:

(signature)

Fred POPOWICH

(name)

July 11, 1985

(date)

Abstract

Since Colmerauer's introduction of *metamorphosis grammars* (MGs), with their associated *type 0*-like grammar rules, there has been a desire to allow more general rule formats in logic grammars. *Gap symbols* were added to the MG rule by F. Pereira, resulting in *extraposition grammars* (XGs). *Gaps*, which are referenced by gap symbols, are sequences of zero or more unspecified symbols which may be present anywhere in a sentence or in a *sentential form*. However, XGs imposed restrictions on the position of gap symbols and on the contents of gaps. With the introduction of *gapping grammars* (GGs) by Dahl, these restrictions were removed but the rule was still required to possess a nonterminal symbol as the first symbol on the left hand side. This restriction is removed with the introduction of *unrestricted gapping grammars*. FIGG, a Flexible Implementation of Gapping Grammars, represents an implementation of a large subset of unrestricted GGs which allows either bottom-up or top-down parsing of sentences. The system provides more built-in control facilities than previous logic grammar implementations. This makes it easier for the user to create efficiently executable grammar rules and restrict the applicability of certain rules. FIGG can be used to examine the usefulness of unrestricted GGs for describing phenomena of natural languages such as free word order, and partially free word/constituent order. It can also be used as a programming language to implement natural language systems which are based on grammars (or metagrammars) that utilise the *gap* concept, such as Gazdar's *generalised phrase structure grammars*.

Acknowledgements

I would like to thank Nick Cercone for his comments and suggestions. His assistance with the organisation of this work, and his suggestions relating to literary style were also invaluable.

Much of this thesis was inspired by research done by Veronica Dahl. Her comments and questions were extremely helpful.

I would also like to extend my gratitude to the referees who reviewed (Popowich, 1985a) and (Popowich, 1985b), since these papers were based on an early version of this work. Their suggestions and comments proved to be helpful in the revision of sections 5.1 and 5.2.

The Laboratory for Computer and Communications Research at Simon Fraser University supplied the facilities for the research described within this thesis, and for the text processing of this manuscript. Ed Bryant was especially helpful in providing technical support.

Finally, I would like to thank the Natural Sciences and Engineering Research Council (NSERC) of Canada for support under Postgraduate Scholarship #800 during this research. This work was also supported under NSERC Operating Grant no. A4309, and Installation Grant no. SMI-74.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

It is often desirable to capture generalisations about the syntactic structure of a language with concise, high level grammar productions. These high level descriptions can often decrease the number of grammar rules required, and may, in practice, result in more efficient parsing (Berwick and Weinberg, 1982). One method to provide more general grammars is to introduce *gap symbols*, which refer to sequences of unspecified symbols called *gaps*, into the grammar rules. *Unrestricted gapping grammars* incorporate gaps within the grammar. The motivation for the development of unrestricted gapping grammars was derived from the development of the SAUMER system (Popowich, 1985c), and from the study of *gapping grammars* (GGs) (Dahl and Abramson, 1984).

1.1. Motivation

SAUMER allows specifications of natural language grammars, which consist of rules and metarules, to be used to provide a semantic interpretation of an input sentence. The two level grammar is based on the format used within *generalised phrase structure grammars* (GPSGs) (Gazdar, 1981). From a set of *base rules*, the *metarules* will generate *derived rules*. Through repeated operation of the metarules over the base and derived rules, a complete set of rules for a grammar can be generated. Consider the set of context-free rules shown in (1.1) which describe some active and passive verb phrases.

(1.1) (a) vp_{act} --> v, np_{acc}, pp_{to}. eg. *gives the ball to Mary*
(b) vp_{act} --> v, np_{dat}, np_{acc}. eg. *gives Mary the ball*
(c) vp_{act} --> v, np_{acc}. eg. *throws the ball*
(d) vp_{pas} --> aux, v, pp_{to}. eg. *is given to Mary*
(e) vp_{pas} --> aux, v, np_{acc}. eg. *is given the ball*
(f) vp_{pas} --> aux, v. eg. *is thrown*

The same set of rules can also be represented by the first three rules (1.1a-c) alone, along with the metarule

(1.2) vp_{act} --> v, np, X ==> vp_{pas} --> aux, v, X.

Any rule that matches the left hand side (pattern) of the metarule, will cause the generation of a new rule as specified by the right hand side (template). The *X* in (1.2) is a *string variable* (Thompson, 1982) that can match zero or more grammar symbols. There are two approaches for the the use of (1.2) in conjunction with (1.1a-c). The *compiled* approach, which is used by SAUMER, uses the metarule to generate (1.1d-f), and then uses base rules along with the derived rules during parsing. The *interpreted* approach would involve the use of the base rules and (1.2) during the parse without the generation of (1.1d-f). Although this would require parsing according to a non-context-free specification, it might be more efficient than the compiled approach in some cases (Berwick and Weinberg, 1982). Perhaps the conversion of the two level grammar into some other grammar, could result in a concise specification for interpreted processing.

One property of a SAUMER grammar, which was observed in grammars of the original GPSG theory, is that a large number of rules are needed to describe structures whose constituents may appear in many arrangements. Consider the English sentence

(1.3) The ball was thrown in the park to Fido by Marvin.

No matter what order the three prepositional phrases following the verb are placed in,

a syntactically valid sentence results. The verb phrase in (1.3) can be described by the context-free rule

(1.4) verb_phrase --> verb, prep_phrase$_{loc}$, prep_phrase$_{dat}$, prep_phrase$_{acc}$

where the abbreviations *loc*, *dat* and *acc*, correspond to the locative, dative, and accusative arguments of the verb respectively. To maintain this type of structure and describe the other possible orders of the prepositional phrases would require five additional context-free rules. If context-sensitive rules were allowed, the same result could be achieved with the introduction of rules for switching the order of these phrases. All six context-free rules can be described by a single *immediate dominance* rule, along with a set of *linear precedence* relations (Gazdar and Pullum, 1982). An *immediate dominance* (ID) rule resembles a context-free rule, but it specifies only that the symbol on the left hand side of the rule immediately dominates (is the parent of) the symbols of the right side. The order of the right hand side symbols is restricted by the *linear precedence* (LP) relations. A linear precedence relation, $\beta_i < \beta_j$ is a transitive relation between two symbols of the grammar, β_i and β_j, that states which symbol must precede the other if they both appear in the right hand side of a context-free rule. For our example, the ID rule

(1.5) verb_phrase --> verb prep_phrase$_{loc}$ prep_phrase$_{dat}$ prep_phrase$_{acc}$

can describe the structure of the context-free rule, while the LP relations

(1.6) (a) verb < prep_phrase$_{loc}$
(b) verb < prep_phrase$_{dat}$
(b) verb < prep_phrase$_{acc}$

will require the verb to precede any prepositional phrase. The current generalised phrase structure grammar theory (Gazdar and Pullum, 1982) uses this formalism to describe the context-free rules of the grammar. Once again, there are two ways of using these grammars to parse sentences. The conversion of the ID/LP rules into their corresponding context-free rules before parsing will be referred to as the

compiled approach to ID/LP rule processing. There is also interest in the *interpreted* approach (Shieber, 1982) (Evans and Gazdar, 1984) which entails parsing according to the ID/LP grammar, instead of using the context-free grammar. Perhaps the ID/LP grammar can be used within some grammatical formalism that will allow interpreted processing.

The rules of gapping grammars allow explicit reference to gaps between constituents. Consequently, rules like

(1.7) noun-phrase(obj), and, gap(G), noun-phrase(obj)
--> [and], gap(G), noun-phrase(obj)

can account for linguistic phenomena such as the deletion of the object in sentences like

(1.8) John saw *the train and Mary heard the train.*

Application of (1.7) to (1.8) will result in the following sentence.

(1.9) John saw *and Mary heard the train.*

Gapping grammar rules can also describe other phenomena like extraposition of constituents, and totally free word order (Dahl, 1984).

Since the string variables of GPSG metarules can be thought of as gaps symbols, and since the ID/LP grammars describe *free word order* subject to certain restrictions (the LP relations), it was decided to use gapping grammars to investigate the interpreted processing of metarules and ID/LP specifications. Unfortunately, the implementations of gapping grammars (Dahl and Abramson, 1984) were inadequate, and the theory itself was slightly too restrictive, which lead to the formulation of unrestricted gapping grammars and the development of a system to process them.

1.2. Gaps in Grammars

The notion of gaps is not new to formal grammar study, linguistics, or programming languages. It should not, however, be confused with a linguistic definition of gap which refers to the *trace* left by a moved constituent. Transformational grammars (Radford, 1981) introduce a *trace* symbol, which is also called a gap, that occupies the position of a moved or deleted constituent. For instance, the WH-movement transformation, (responsible for *wh*-questions), can be applied to the question

(1.10) John will give *which book* to Mary?

resulting in the following question containing a trace, _.

(1.11) *Which book* will John give _ to Mary?

The *trace* differs from the *gap* discussed in this paper since it acts as a grammar symbol which may be present in the sentence, rather than a meta-symbol which refers to grammar symbols.

Scattered context grammars (Greibach and Hopcroft, 1969) use gaps in the description of the derivations allowed by the grammar. The grammar rules, however, do not explicitly use gap symbols. Productions are of the form

(1.12) $(A_1,....,A_n) \text{ --> } (\omega_1,....,\omega_n)$

where the A's represent nonterminal symbols, and the ω's correspond to sequences of terminals and nonterminals. Gaps are introduced by the rewrite relation

(1.13) $(x_1,A_1,....,x_n,A_n,x_{n+1}) \text{ ==> } (x_1,\omega_1,....,x_n,\omega_n,x_{n+1})$

associated with each rule. The *x*'s represent gaps of any number of terminals or nonterminals present in a *sentential form*. A *sentential form* is defined recursively as a string consisting of the start symbol of the grammar, and any string, ω, that can be obtained from a sentential form, ψ, by the application of a grammar production,

ψ ==> ω. These grammars describe a set of languages which are a subset of context-sensitive languages.

The transformations of *transformational grammars* (TGs) (Radford, 1981) can be viewed as rules containing gaps. For example, a transformation of NP-movement which allows any noun phrase to be moved into an empty NP position (subject to certain restrictions) could be represented by the rules shown below.

(1.14) (a) NP, X, NP(ϵ) --> NP(ϵ), X, NP
(b) NP(ϵ), X, NP --> NP, X, NP(ϵ)

The gap is referenced by *X*, with *NP*(ϵ) denoting any empty NP node. (1.14a) describes movement to the right, while (1.14b) is required for movement to the left.

As was mentioned earlier, *generalised phrase structure grammars* (Gazdar, 1981) (Gazdar and Pullum, 1982), which, like TGs, are used in linguistic studies, utilise the gap concept within the metarules of the grammar. The original GPSG framework contains a context-free rule base and set of metarules which generated new context-free rules from the existing rules. Consequently, a string variable in a metarule references an unspecified region (or gap) within a context-free rule. The actual metarule used to generate passive verb phrases resembles

(1.15) VP --> V NP X ==> VP --> V_{pas} X (PP_{by})

where *X* is a gap symbol.

Certain programming languages, which are based on pattern matching, also use gaps in their specifications. In particular, SNOBOL (Griswold, Poage and Polonsky, 1971) possesses functions like *SPAN*, *BREAK* and *REM* for this purpose. The SNOBOL code

(1.16) Rule 'VP' '-->' 'V_{trans}'.v 'NP' REM.x :S(Matched)

⋮

Matched OUTPUT = '$VP_{passive}$' '-->' v x 'PP_{by}'

roughly corresponds to the metarule cited in (1.15). If the *Rule* matches the pattern, the new passive rule is printed.

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

(1.17) *nt*, α --> β

where *nt* is a nonterminal symbol called the *head*, and α and β may contain terminal symbols, nonterminal symbols, procedure calls, and gap symbols. Extraposition grammar rules also have restrictions on the position of gap symbols within the rules, and on the contents of the gaps. Gaps may be relocated by the application of the rule. Additional details about logic grammars, and logic grammars with gaps, will be presented in chapter two.

Unrestricted gapping grammars extend gapping grammars in one important aspect. The adjective *unrestricted* refers to the removal of the requirement that the left hand side of all rules must start with a nonterminal symbol. Consequently, the unrestricted gapping grammar rules resemble

(1.18) α --> β

where α and β may once again contain terminals, nonterminals, gaps and procedure calls in any order. The formal introduction of unrestricted gapping grammars appears in chapter three.

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 introduced in chapter four as a programming language that incorporates procedural control to provide an implementation of unrestricted gapping grammars. It is assumed that the reader is familiar with Prolog in the discussion of the FIGG implementation in chapter four.

With the aid of this system, one can further examine the uses for unrestricted gapping grammars, and can examine the use of procedural control to obtain more efficient computer processing of the grammar rules (chapter five). The use of unrestricted gapping grammars in the specification of natural language phenomena appears to be the most interesting. Dahl has already advocated their use for unbounded relocation of sentence components and thus for describing *free word order* in natural languages (Dahl, 1984). Until now however, little work has been done with respect to their use for partially free word/constituent order. These grammars can also be used to describe the metarule component of generalised phrase structure grammars. Thus FIGG might be used as a programming language for implementing systems based on these theories about natural language grammars. However, before considering these applications in greater detail, it is appropriate to examine the history of logic grammars leading to the formulation of unrestricted gapping grammars and the FIGG implementation.

Chapter 2

Logic Grammars

The use of logic in natural language analysis has for a long time been a subject of study for linguists, logicians, and computer scientists (McCawley, 1981). However, its use was usually restricted to the domain of knowledge representation. With the introduction of *logic grammars* by Colmerauer (Colmerauer, 1978), logic programming entered the domain of natural language grammars. Logic grammars differ from conventional formal grammars since they possess *logic terms* as grammar symbols. Consequently, derivations according to a grammar may involve *unification* rather than mere replacement of grammar symbols. Through the insertion of arbitrary predicates into a grammar rule as *procedure calls*, and through the use of *logic variables*, a logic grammar can often provide a more concise description for a language than is possible using conventional formal grammars (Pereira, 1981) (Dahl, 1984).

The *logic terms* which act as the symbols of a logic grammar consist of a *functor*, along with 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**. **H[F]**, which is also referred to as the *Herbrand Universe*, represents the set of logic terms without variables. 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

denote *variables*. A *list*, which is a logic term of the form '.'(α_1,'.'(α_2,....'.'(α_n,nil)...)), is usually represented as [α_1,α_2, ...,α_n]. Also, [*t*|*l*] 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. Procedure calls are often used to restrict of rule applicability, to introduce semantic components into the syntax of a language, and to obtain more efficient parsing.

Although all logic grammars have the same computational power (recursive enumerable), restrictions on the grammar rule format can make certain languages more difficult to describe under one formalism than under another. We shall examine several logic grammars in order of increasing generality of rule format and observe the change in the number and type of rules required to describe the sample context-sensitive language

(2.1) $L_1 = \{a^n b^n c^n \mid n > 0\}$

2.1. Definite Clause Grammars

Definite clause grammars (Pereira and Warren, 1980) possess rules that closely resemble those of a context-free grammar in structure. Each production has a single nonterminal on the left hand side of the production arrow, -->, with terminals, nonterminals and procedure calls forming the right hand side. By convention, terminal symbols are represented as *lists*, while procedure calls are enclosed in braces, {}. Although these grammars resemble context-free grammars, they derive their recursive enumerable power from the addition of arguments to the symbols, and obviously, from the arbitrary procedures that can be embedded within the rules.

Based on (Colmerauer, 1978) and (Pereira and Warren, 1980), a more formal, but

not rigorous, definition which excludes procedure calls can be provided. A definite clause grammar, G, is a quadruple (V_N, V_T, Σ, P) where V_N is the set of nonterminal symbols, $V_N \subset \mathbf{H}[F]$; V_T is the set of terminal symbols, $V_T \subset \mathbf{H}[F]$,[1] with $V_N \cap V_T = \phi$; Σ is the set of starting symbols, with $\Sigma \subset V_N$; and P is the set of productions of the form:

(2.2) $nt \text{ --> } \beta_1, \beta_2, \ldots, \beta_n$

with $nt \in V_N$, $n \geq 0$, and $\beta_i \in V$ for $1 \leq i \leq n$, where $V = V_N \cup V_T$. Ignoring variable substitution, the language, L(G), associated with this grammar is defined by

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

In subsequent definitions, variable substitution will also be ignored. Details regarding it can be found in (Colmerauer, 1978). S^* represents the Kleene closure of a set S, with $S^* = \cup_{i=0}^{\infty} S^i$. $\rightarrow^*$ is the reflexive transitive closure of --->.

One can provide a grammar for the language L_1 by permitting each nonterminal to have arguments which serve as counters. A set of DCG productions that correspond to this language is given in (2.4).

(2.4)
(a) start --> x(zero).
(b) x(A) --> [a], x(s(A)).
(c) x(Stack) --> y(b, Stack), y(c, Stack)
(d) y(T, s(Stack)) --> [T], y(T, Stack).
(e) y(T, s(zero)) --> [T].

Rule (b) is used to produce/parse an arbitrary number of *a*'s with *A* of *x(A)* representing the number of *a*'s that have been generated/parsed so far. Counting is done by the *successor* function, *s*, with *zero*, *s(zero)*, and *s(s(zero))* representing 0, 1 and 2 respectively. Once *n* *a*'s have been generated/parsed, rule (c) is used to request processing of *n* *b*'s and *c*'s by rules (d) and (e). The first argument of the

---

[1] Usually, the Herbrand universe is used in place of $\mathbf{H}[F]$ since variables do not generally appear in terminal symbols.

nonterminal *y(T,S)* states whether *b*'s or *c*'s are being processed, while the second counts the number of characters that have yet to be generated/parsed. Figure 2-1 shows the parse tree for the sentence $a^3b^3c^3$ using these rules.

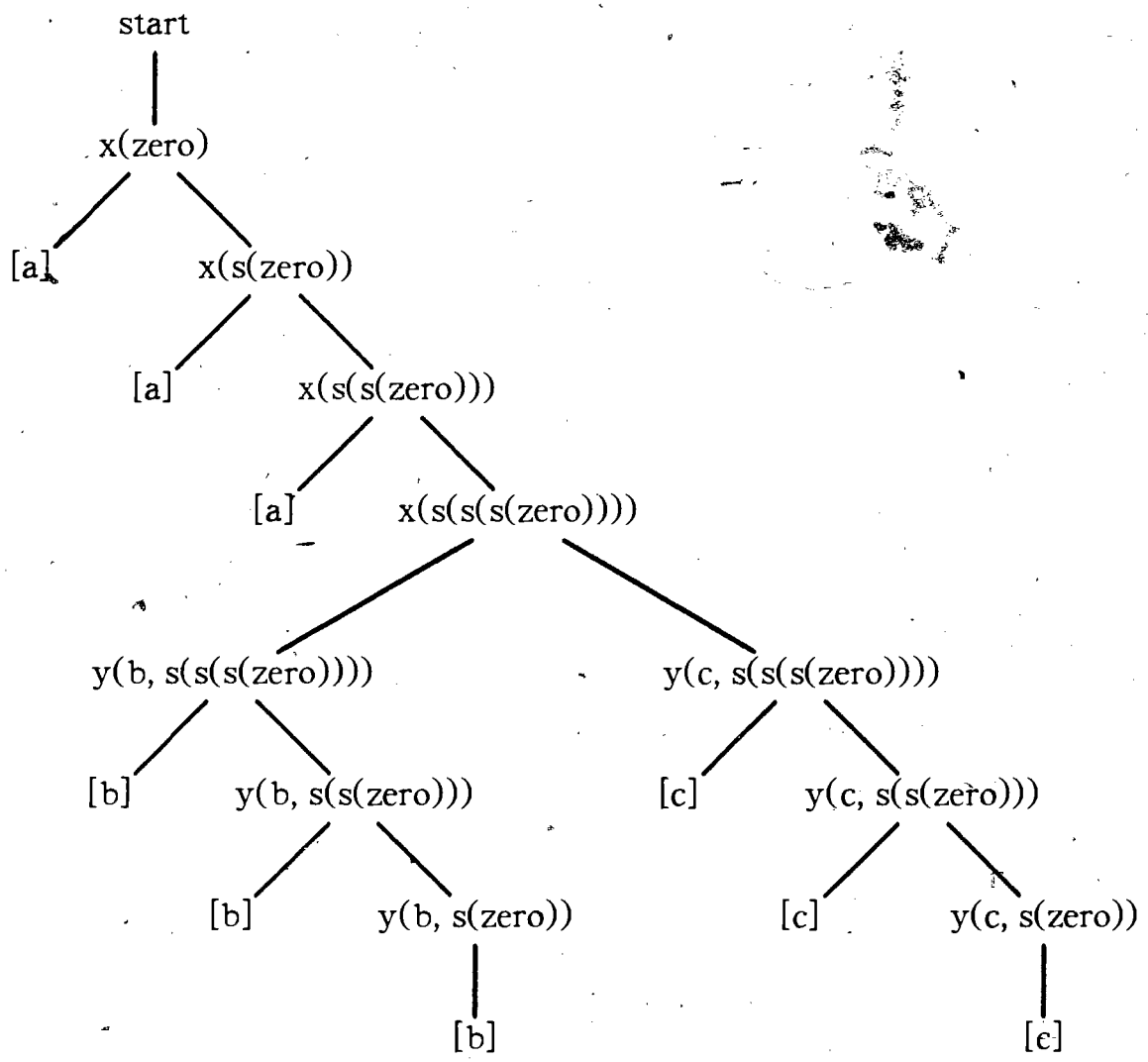

Figure 2-1: Parse Tree for $a^3b^3c^3$ using a Definite Clause Grammar

A top-down depth-first parser for DCG rules can be implemented easily in Prolog by converting each rule into a *definite clause* (Pereira and Warren, 1980). A *definite clause* P :- $Q_1, Q_2, \ldots, Q_n$ can be translated as *P is true if Q_1 and Q_2 and ... and Q_n are all true*. Each nonterminal

(2.5) $f(\alpha_1,\alpha_2, ..., \alpha_n)$

is converted into a similar logic term with two additional arguments

(2.6) $f(\alpha_1,\alpha_2, ..., \alpha_n,X_1,X_2)$

The additional arguments, which are called the incoming and outgoing phrases, correspond to the phrase to be parsed, and the remainder of the phrase after $f(\alpha_1,\alpha_2, ..., \alpha_n)$ has been parsed. Phrases are represented as lists. Any terminal symbol, *[term]*, is replaced by the *connect* clause

(2.7) $c(X_1, Word, X_2)$[2]

which will remove *Word* from the front of X_1, and leave X_2. The definition of the connect clause is shown below.

(2.8) c([Word|X], Word, X)

So, (2.4b) could be translated into the Prolog clause

(2.9) x(A, _1, _3) :- c(_1, a, _2), x(s(A), _2, _3).

2.2. Metamorphosis Grammars

Metamorphosis grammars (Colmerauer, 1978) were the first of the logic grammars. The rule format resembles that of the type-0 grammars of the Chomsky hierarchy. Both the left and right hand sides of a rule can contain any combination of terminal and nonterminal symbols.

One can define a metamorphosis grammar, G, in a manner similar to a definite clause grammar as a quadruple (V_N, V_T, Σ, P) where V_N, V_T, and Σ have their usual interpretations, and P is the set of productions of the form

---

[2]It is not actually necessary to have a clause for each terminal symbol. Instead, terminal symbols can be incorporated into the arguments of the nonterminal symbols (Clocksin and Mellish, 1981). The addition of the extra term results in easier readability and it corresponds to the translation produced by C-Prolog (Pereira, 1984).

(2.10) $\alpha_1, \alpha_2, \ldots, \alpha_m \dashrightarrow \beta_1, \beta_2, \ldots, \beta_n$

with $m > 0$, $n \geq 0$, $\alpha_i \in V$ for $1 \leq i \leq m$, and $\beta_j \in V$ for $1 \leq j \leq n$. Once again, the language, L(G), associated with this grammar is defined by

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

Since DCGs are a subset of MGs, we could use the productions stated in (2.4) as MG rules for the language L_1. However, we can eliminate the need for arguments on the nonterminals by taking advantage of the more general rule format allowed by MGs. The following productions, which are adapted from (Aho and Ullman, 1972), can be used to describe L_1.

(2.12) (a) s --> [a], [b], [c].
(b) s --> [a], s, b, [c].
(c) [c], b --> b, [c].
(d) [b], b --> [b], [b].

The first two productions are used to generate/parse equal quantities of *a*'s, *b*'s and *c*'s. Rule (c) is used to shift the *b*'s to the left, until they reach their final position as determined by rule (d). A derivation of $a^3b^3c^3$ according to this grammar is shown in Figure 2-2.

Colmerauer also introduced the notion of *normalised metamorphosis grammars*, which are a restricted form of MGs that can be converted into Prolog clauses in a straightforward manner (Colmerauer, 1978). MGs in *normal form* have productions of the form

(2.13) $\alpha_1, \alpha_2, \ldots, \alpha_m \dashrightarrow \beta_1, \beta_2, \ldots, \beta_n$

with $m > 0$, $n \geq 0$, $\alpha_1 \in V_N$, $\alpha_i \in V_T$ for $2 \leq i \leq m$, and $\beta_j \in V$ for $1 \leq j \leq n$. The symbol α_1 will be referred to as the *head* of the rule. Any MG may be converted into normal form by following a simple procedure which may introduce extra terminal and nonterminal symbols. The procedure, which is described in (Colmerauer, 1978), is summarised below.

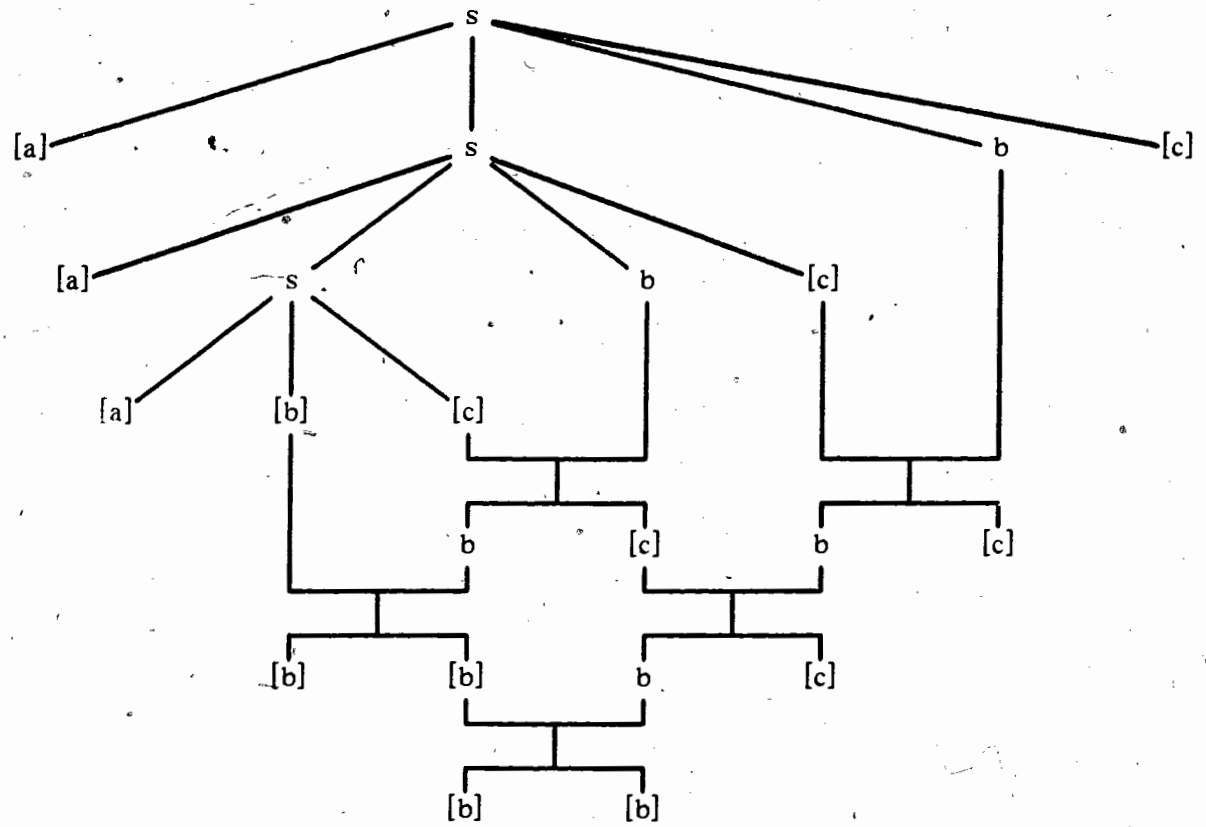

Figure 2-2: Derivation of $a^3b^3c^3$ using a Metamorphosis Grammar

- If α_1 is a terminal symbol, replace it by the new nonterminal "$nt(\alpha_1)$" and add the production "$nt(\alpha_1)$ --> α_1" to the grammar.
- If α_i (for $i \geq 2$) is a nonterminal, replace it by the new terminal "$te(\alpha_i)$", and add the production "α_i --> $te(\alpha_i)$" to the grammar.
- Finally, if β_j is a terminal symbol and if β_j is the *head* of some other production, then replace it by "$nt(\beta_j)$" and add the rule "$nt(\beta_j)$ --> β_j" to the grammar.

Normalised MGs may be transformed into Prolog clauses by a process similar to the one performed on DCGs in order to obtain a top-down depth-first parser. Using the notation introduced in the previous section, the rule (2.13) would result in the clause

(2.14) $\alpha_1(X, [\alpha_2, \ldots, \alpha_m|Y]) :- \beta_1(X,X_1), \beta_2(X_1,X_2), \ldots, \beta_n(X_{n-1},X_n)$

The notation used in (Colmerauer, 1978) differs slightly. According to (2.14), given the goal of parsing an α_1, first try to parse all of the β_i's. If this parse succeeds, insert the terminals; α_i's, at the beginning of the outgoing phrase argument to be subsequently parsed by some outstanding *goal*.

2.3. Extraposition Grammars

Extraposition grammars (XGs) (Pereira, 1981) introduced the gap concept into the logic grammar domain. During a derivation, an XG rule allows one to reference gaps in the left hand side of the rule, and reposition them (in the same order) to the right of all the constituents referenced in the right hand side of the rule. The contents of the gaps are also restricted to be nested (one totally contained in another), or non-intersecting. An XG rule can be considered as a 'rule schema, representing many MG rules in a single rule, or it may be viewed as a context-sensitive rule, where the context need not be adjacent to the symbol being rewritten.

An extraposition grammar may be defined as a quintuple (V_N, V_T, γ, Σ, P) where: V_N, V_T, and Σ have their usual interpretations, γ is the gap symbol, with $\gamma \notin V$; and P is the set of productions of the form

(2.15) $nt, \alpha_0, \gamma, \alpha_1, \ldots, \gamma, \alpha_m \dashrightarrow \beta_0, \beta_1, \ldots, \beta_n$

with $nt \in V_N$, $m, n \geq 0$, $0 \leq i \leq m$, $0 \leq j \leq n$, and $\alpha_i, \beta_j \in V^*$. Now let $V' = V \cup \{<,>\}$. The function f: $V' \rightarrow V \cup \mathbf{fail}$ is defined as

(2.16)
(a) $f(x\omega) = xf(\omega)$ for $x \in V$, $\omega \in V'^*$
(b) $f(x) = x$ for $x \in V$
(c) $f(<\omega>) = f(\omega)$ for $\omega \in V'^*$
(d) $f(\omega) = \mathbf{fail}$ otherwise

Define the rewrite relation, ==>, between elements of V'^* as

(2.17) $\gamma_0\ nt\ \alpha_0\ \gamma_1\ \alpha_1\ \ldots\ \gamma_m\ \alpha_m\ \gamma_{m+1}$
==> $\gamma_0\ \beta_0\ \beta_1\ \ldots\ \beta_n <<\ldots<\gamma_1>\gamma_2>\ldots\gamma_m>\gamma_{m+1}$

for a production (2.15) if $f(\gamma_i) \in V^*$ forall $1 \leqslant i \leqslant m$, where $\gamma_i \in V'^*$. The language described by grammar G can be described as

(2.18) $L(G) = \{f(\omega) \mid s ==>^* \omega \text{ for } s \in \Sigma, \omega \in (V_T \cup \{<,>\})^*\}$

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

The extraposition grammar for L_1 shown in (2.19) does not require *shifting* rules like (2.12c).

(2.19) (a) s --> [a], bs, [c].
(b) s --> [a], s, b, [c].
(c) bs ... b --> [b], bs.
(d) bs .., b --> [b], [b].

After rules (a) and (b) have generated/parsed the required number of *a*'s, *b*'s, and *c*'s, rule (c) is used to bring a distant *b* to its final location. Rule (d) is used to relocate the final *b*. The logic term "..." is used as the gap symbol, γ. A derivation of $a^3b^3c^3$ according to (2.19) appears in Figure 2-3. Each ellipse that appears in this figure represents the contents of a gap. According to the nesting constraint on gaps, once symbols are encased in an ellipse, they lose their individual identities. So subsequent rule applications must reference the entire ellipse.

The implementation of extraposition grammars, which is described in detail in (Pereira, 1981), translates the grammar rules into definite clauses for execution by Prolog. Whereas DCGs required the addition of two arguments to each nonterminal, XGs need a total of four extra arguments. As with DCGs, two arguments are for the *incoming* and *outgoing* phrase, while the other two are used in a similar manner for the *extraposition list*. While Colmerauer's MG implementation required the grammars to be normalised by the user, it is done automatically in Pereira's system. When processing a rule, like (2.15), with a gap, all of the left hand side of the production, with the exception of *nt*, is inserted at the front of the extraposition list.

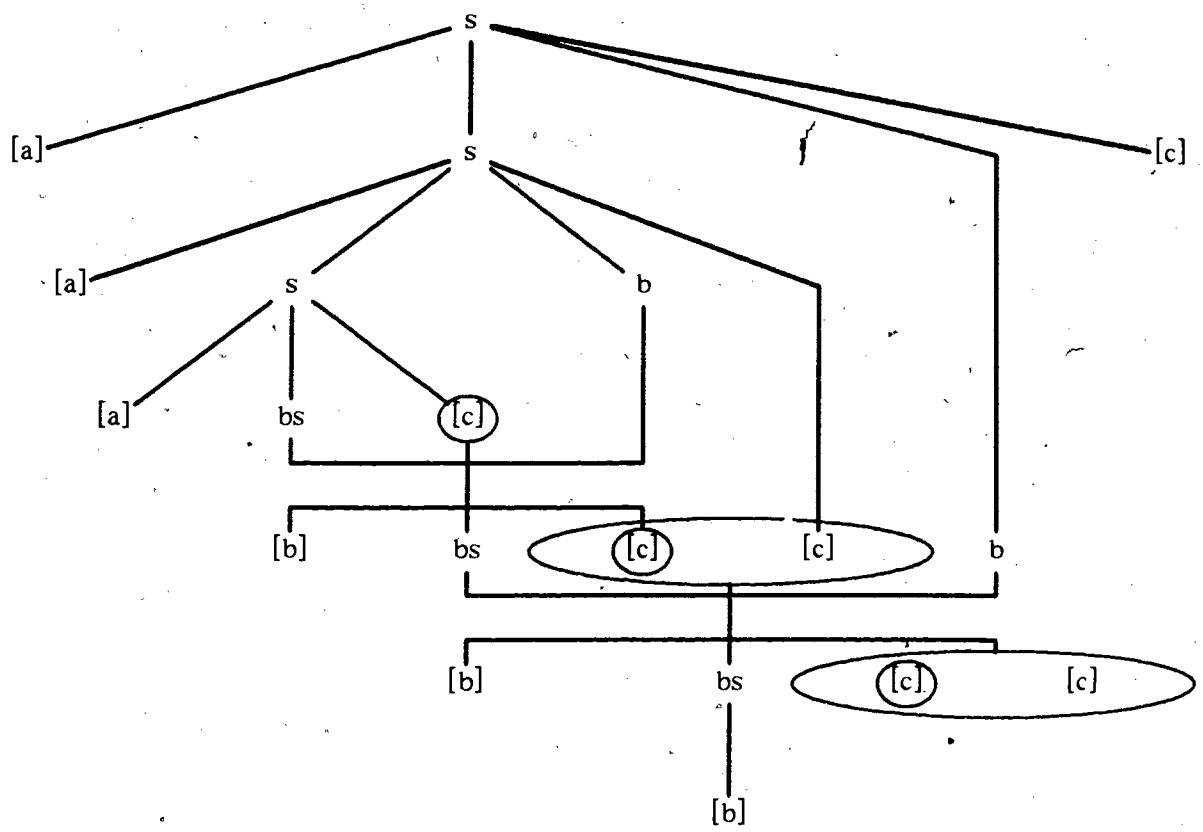

Figure 2-3: Derivation of $a^3b^3c^3$ using an Extraposition Grammar

This differs from the MG rule processing, where these symbols were put into the outgoing phrase list. Nonterminals and terminals may, then, under certain conditions, be "read off" from this list, instead of from the input phrase.

2.4. Gapping Grammars

Gapping grammars (GGs), as introduced by (Dahl and Abramson, 1984), extended the extraposition grammar rule by the removal of the restrictions on the locations of the gaps in the rule, and by the elimination of the nesting constraint imposed on the contents of a gap. Consequently, the GG rule has the general form

(2.20) *nt*, α --> β

where *nt* is a nonterminal symbol (the *head*), and α and β may contain terminal

symbols, nonterminal symbols, procedure calls, and gap symbols which are traditionally of the form $gap(G_i)$.

A more formal definition of a gapping grammar, G, which is based on a definition appearing in (Dahl, 1984), defines it as a quintuple $(V_N, V_T, \Gamma, \Sigma, P)$ where: V_N, V_T, and Σ have their usual interpretations, Γ is the set of gap symbols, with $\Gamma \cap V = \phi$; and P is the set of productions of the form

(2.21) $nt, \alpha_0, gap(G_1), \alpha_1, \ldots, gap(G_m), \alpha_m$
$\text{--}> \beta_0, gap(G'_1), \beta_1, \ldots, gap(G'_n), \beta_n$

with $nt \in V_N$, $m, n \geq 0$, $0 \leq i \leq m$, $0 \leq j \leq n$, $\alpha_i, \beta_j \in V^*$, and $gap(G_i), gap(G'_i) \in \Gamma$. The rewrite relation between sentential forms, which are elements of V^*, may be defined as

(2.22) $nt\ \alpha_0\ \gamma_1\ \alpha_1 \ldots \gamma_m\ \alpha_m \Longrightarrow \beta_0\ \gamma'_1\ \beta_1 \ldots \gamma'_n\ \beta_n$

for a production (2.21) where $\gamma_i, \gamma'_j \in V^*$. The language described by grammar G can be described as

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

A valid gapping grammar for L_1 which is equivalent to the XG grammar (2.19) is

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

However, due to the absence of a nesting restriction on the contents of a gap, this is an *ambiguous* grammar. One derivation for $a^3b^3c^3$, which differs from the one presented in Figure 2-3, is illustrated in Figure 2-4. Once again, an ellipse corresponds to the contents of a gap. Notice that individual symbols can be removed from an ellipse. So the capacity for more general rules may have unwanted side effects, like the creation of ambiguity in a grammar.

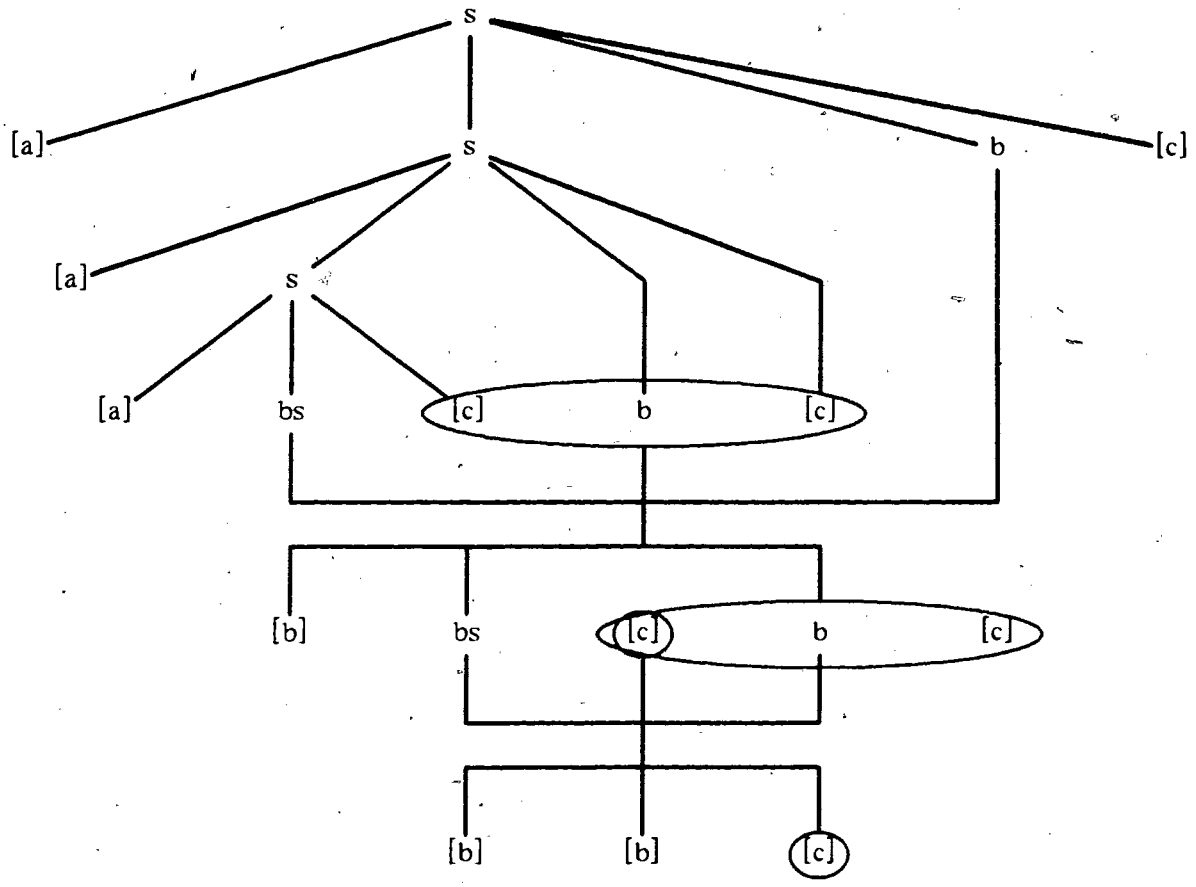

Figure 2-4: Derivation of $a^3b^3c^3$ using a Gapping Grammar

The Prolog implementations of gapping grammars presented in (Dahl and Abramson, 1984) generally follow the methods used in the implementation of the other logic grammars described in this chapter. The rules are converted into definite clauses, with each nonterminal acquiring two extra arguments for manipulation of the phrase to be parsed. The gap predicate, *gap(G)*, is converted to a clause of the form

(2.25) gap(G, X_1, X_2)

which is defined in Prolog as

(2.26) gap([], X, X).
gap([Word|Rest], [Word|X_1], X_2) :- gap(Rest, X_1, X_2).

Consequently, *gap(G)* will first return an empty string, and upon failure will return

strings of increasing length. Since the gap predicate is simply a variation of concatenation, (2.25) is equivalent to

(2.27) append(G, X_2, X_1)

which appends X_2 to G and returns X_1 as the result.

There are actually two gapping grammar implementations presented in (Dahl and Abramson, 1984). The first implementation, which shall be referred to as GG1, is extremely concise, but the Prolog clauses it produces result in inefficient parsing. To simplify the discussion that follows, we will refer to a GG rule like (2.21) where α_i, $\beta_i \in V$. Such a rule would be converted into the Prolog clause

(2.28) *nt*(X,Y) :-
(a) $\beta_0(X,X_0)$, gap(G'_1,X_0,X_1), $\beta_1(X_1,X_2)$, ... gap(G'_n,X_{2n-2},X_{2n-1}), $\beta_n(X_{2n-1},Z)$,
(b) $\alpha_0(Y,Y_0)$, gap(G_1,Y_0,Y_1), $\alpha_1(Y_1,Y_2)$, ... gap(G_m,Y_{2m-2},Y_{2m-1}), $\alpha_m(Y_{2m-1},Z)$.

where *term*(X_1,X_2) would be replaced with *c*(X_1,*term*,X_2) for terminal symbols. During the top-down parse of *nt*, in a manner similar to the MG parsing, the input phrase is first checked for a string satisfying (2.28a). Upon finding it, the parser then places a new string, generated according to (2.28b), at the front of the outgoing phrase. The creation of this new string can be inefficient since the variable Y in (2.28) is usually free when *nt*(*X*,*Y*) is invoked as a goal. Excessive backtracking is thus required when executing (2.28b) to find a Y that will leave Z. In fact, this can lead to catastrophic results if some α_i equal to *nt* appears in (2.28b); since the second call to *nt* will have both X and Y as free variables!

The second implementation, GG2, described in (Dahl and Abramson, 1984), permits only a subset of the rules described by extraposition grammars. Not surprisingly, this is a more efficient implementation than GG1. The rules accepted by this implementation are of the form

(2.29) *nt*, gap(G), [*term*] --> β, gap(G)

By modifying the definition of the gap predicate though, it is possible to permit intersecting gaps, and thus obtain a more concise description for some languages than is possible with XGs. The implementation is based on a message passing scheme and the assumption that the *term* of (2.29) is a marker which is introduced for control reasons, to be *absorbed* by the appearance of *fill* in another rule. Details of this implementation can be found in (Dahl and Abramson, 1984).

It is possible to generalise the gapping grammar definition one step further, and introduce a similar formalism that does not possess the restriction of a rule requiring an initial nonterminal. This new logic grammar formalism will form the basis of the FIGG system.

Chapter 3

Unrestricted Gapping Grammars

The requirement of a nonterminal *head* in the production rule of logic grammar formalisms appears to be a product of its need by the top-down depth-first parsers described in the previous chapter. The introduction of *unrestricted gapping grammars* removes this restriction and provides a more general rule format which includes all MG and GG rules. An unrestricted GG rule will consequently permit a terminal symbol or even a gap as the first symbol on the left hand side of the rule.

An unrestricted gapping grammar is a quintuple (V_N, V_T, Γ, Σ, P) where V_N, V_T, Γ, and Σ represent the same sets described in the gapping grammar definition, and P is the set of productions of the form:

(3.1) $\alpha_0, gap(G_1), \alpha_1, \ldots, gap(G_m), \alpha_m$
$\quad --> \beta_0, gap(G'_1), \beta_1, \ldots, gap(G'_n), \beta_n$

with $m, n \geq 0$, $0 \leq i \leq m$, $0 \leq j \leq n$, $\alpha_i, \beta_j \in V^*$, and $gap(G_i), gap(G'_j) \in \Gamma$. Rules where m or n are non-zero are called *gapping* rules, since they contain at least one gap. The rewrite relation between elements of V^* may be defined as

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

for a production (3.1) where $\gamma_i, \gamma'_j \in V^*$. Once again, the language described by grammar G is

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

With the removal of the *nonterminal head* restriction associated with gapping

grammars, unrestricted GGs can be used to describe some forms of left extraposition more simply. To illustrate this point, let us examine the language L'_2 described in (Joshi, 1983). This language is obtained from

(3.4) $L_2 = \{(ba)^n c^n \mid n \geq 1\}$

by "dislocating some *a*'s to the left." Using an unrestricted GG, this language can be described by the following productions.

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

Rules (a) and (b) correspond to L_2, the basis of the grammar, while (c) is used to *dislocate* an *a*. (3.5c) can also be used to leave an *a* in its current location if the gap is empty. The productions are designed to allow an *a* to be moved only once. To provide an equivalent grammar using GG rules, without shifting *b*'s to the right, would require the replacement of (3.5c), and minor modifications to the first two rules. One possible set of productions is illustrated below.

(3.6) (a) s --> b, a, s, [c].
(b) s --> b, a, [c].
(c) b, gap(G), a --> [a], b, gap(G).
(d) a, gap(G), a --> [a], a, gap(G).
(e) a --> [a].
(f) b --> [b].

Since an *a* can be moved to the left of a *b*, or to the left of another *a*, or can remain where it is, rules (c) and (d) are required. (3.6e-f) are needed since (3.6c-d) cannot have a terminal, like [a] or [b], as a head symbol. If one introduces the nonterminal symbol, *target*, into the grammar, the following gapping grammar rules can be used to describe L'_2.

(3.7) (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 (3.7), the nonterminal *target* represents a location where an *a* may be moved to,

while epsilon, ϵ, corresponds to the empty string. Rule (3.7c) has the same use as (3.5c). Nonetheless, this gapping grammar requires one additional production, and one additional nonterminal than the unrestricted GG described in (3.5).

Along with easier description of unbounded left relocation of symbols, there is another phenomenon that follows from the removal of the nonterminal head restriction. The definition of an unrestricted GG does not prohibit rules resembling "ϵ --> β". This restriction was either implicit or explicit in previous logic grammar formalisms. The use of this type of production may be unclear, however it can be used to grammatically characterise a certain phenomenon found in some spoken languages, specifically the introduction of *words* (syllables) like *umm* and *ahh* into phrases.[3] Consider the following sentence which could be spoken by an absent minded person on Christmas Day, while trying to recall *who gave whom which presents*.

(3.8) Ahh, I gave umm, John, umm, a shirt.

A grammar that generates this sentence could include the productions "ϵ --> [umm]", and "ϵ --> [ahh]". This style of production could also be used to introduce nonterminal symbols (markers), like *target*, into arbitrary locations.[4] With this in mind, the productions illustrated in (3.7) could be restated as shown in (3.9).

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

Any *target*'s introduced somewhere to the left of an *a* by (3.9d), can be replaced by an *a* which is dislocated to the left according to (3.9c).

---

[3]The semantic properties of such productions are beyond the scope of this paper.

[4]These productions are reminiscent of how *markers* are introduced within Markov algorithms (Korfhage, 1966).

One other observation about the formal definition of unrestricted GGs is the absence of the restriction "$\{gap(G_i)\} = \{gap(G'_j)\}$". There is no requirement for the same gap to appear on both sides of the production. GGs did not require this restriction either, but this was not discussed in the previous literature. The effect of this property is that an arbitrary number of unspecified terminals and nonterminals could be generated or absorbed in the parsing or generation process. Consider the case where "$m < n$" in the definition of an unrestricted GG. One possible use for productions of this form, which possess an extra gap symbol on the right hand side, may be for nonsense sentences. Imagine the following phrase being uttered.

(3.10) He was so drunk last night, he said "coloured sleep pink elephants."

The quote of the drunk man could be expressed by the rule "sentence --> gap(G)". A case where the left hand side of the rule contains an extra gap, $m > n$, is illustrated in rule (e) of a grammar for L'_3.

(3.11) (a) s --> x, y, [b], a, s, [c].
(b) s --> x, y, [b], a, [c].
(c) ϵ --> target.
(d) target, gap(G), a --> [a], gap(G).
(e) x, gap(G), y --> [x], [y].

L'_3 is obtained from L_3

(3.12) $L_3 = \{(xyba)^n c^n \mid n \geq 1\}$

by dislocating some of the *a*'s to the left, but the *a*'s are not allowed to be moved between an *x* and a *y*. (3.11e) can be used to remove any *target*'s that are inserted between *x* and *y*. However, for this grammar to generate only L'_3, it would be necessary to insert some control mechanism to ensure that (3.11e) is used only after all necessary applications of rules (3.11a-d) have been performed, and to prevent the gap of (3.11e) from containing other *x*'s and *y*'s. One other example illustrates one gap on each side of the rule where the two gaps are not identical.

(3.13) gap(G1) --> gap(G2), {quote(G1,G2)}.

The predicate *quote* adds a quote symbol to each symbol of G1, returning the quoted symbols as G2. Consequently, this rule will rewrite sentential forms like $x\ y\ z$ as $x'\ y'\ z'$. A rule similar to (3.13) is described in section 5.2. Other uses for such rules without the same gap appearing on both sides of the rule is a subject for further investigation.

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

Chapter 4

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. Due to the general form of the unrestricted GG rule, FIGG can also parse sentences using the rules of formalisms like extraposition grammars (the nesting constraint must be added), metamorphosis grammars, context-sensitive grammars and context-free grammars.

The implementations of logic grammars presented in chapter two illustrated 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 *dominators*, which are used to restrict the rules which can be applied to the symbols introduced by another rule. Different forms of the Prolog *cut* facility are available. While the ordinary *cut* (Clocksin and Mellish, 1981) prevents backtracking into goals before the cut in the current clause, the *local cut* prevents backtracking within a specified region of the current clause. More details on these forms of procedural control, along with a description of the syntax of FIGG, can be found in the section 4.1.

The top-down depth-first backtrack parser incorporates these procedural control

mechanisms in a parser that is based on the GG1 parser described in chapter two. It also differs from its predecessor by allowing left recursion in its grammar rules. Rules are still required to have a nonterminal as the head, so it is really only a gapping grammar processor. It is more efficient than GG1, but not quite as general. Specifically, the parse illustrated in 4-1, which uses the set of productions specified in (4.1), will not be found. This problem has been christened the *nested head problem*.

(4.1) (a) s --> x, y, z.
(b) x, gap(G), z --> [x], gap(G), z.
(c) y, z --> [y].

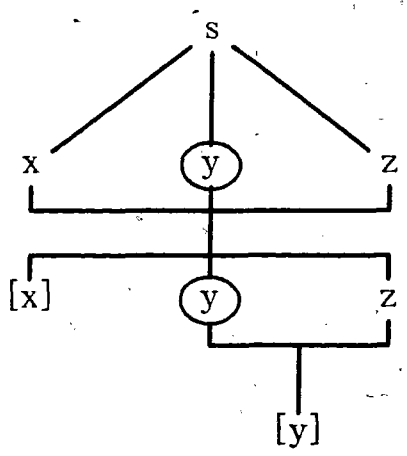

Figure 4-1: Nested Head Problem

To parse the sentence *xy*, (4.1c) must be applied to the *y* that is nested inside the gap of (4.1b). With the current implementation, *xy* could be parsed by rewriting (4.1) in a form similar to

(4.2) (a) s --> begin, x, y, z.
(b) begin, x, gap(G), z --> begin, [x], gap(G), z.
(c) begin, gap(G), y, z --> gap(G), [y].

The *nested head problem* refers to the inability of the top down parser to parse structures where the head symbol of one structure is contained within a gap. The head symbol of a structure is the head of the rule that corresponds to the structure. This limitation is due to the parser's depth-first goal-directed control strategy.

The shift reduce parser differs from the top-down parser since it operates in a bottom-up fashion from the input sentence. According to (Aho and Ullman, 1972), a shift reduce parsing algorithm consists of a *shift reduce* function and a *reduce* function. Using a left to right input scan, the *shift reduce* function will either *shift* the current input symbol onto the stack, call the *reduce* function, *succeed*, or *fail*, based on examination of the input and the stack. Using the same criteria, the *reduce* function can replace the top n elements of the stack $(\beta_1\beta_2...\beta_n)$ by the symbol α if the rule α --> $\beta_1\beta_2...\beta_n$ is present.

The shift reduce parser used with FIGG is a variation of this parser, extended to allow non-context-free rules and gaps. A major difference includes the capacity for a *reduction* to place more than one symbol on the stack. The reduce function may also perform other reductions while placing these multiple symbols on the stack. Also, the input is scanned from right to left to mirror the top-down processing. This allows some control structures to be interpreted the same way by both parsers. Implementation details are discussed in section 4.2.

The bottom-up parser does not suffer from the same restrictions as the top-down parser. As with many bottom-up parsers, however, ϵ-productions, which are of the form α --> ϵ, can *not* be used by the parser. Here, α represents any combination of terminals and nonterminals, and ϵ corresponds to the empty string. *Bottom-up cycles* may also cause problems during parsing. A bottom-up cycle is present if a derivation of the form $\alpha\omega\beta$ ==>* ω is permitted by the grammar. Notice that the ϵ-production is a special case of this restriction.

4.1. The Syntax of FIGG

These two parsers form the basis for FIGG, which is written in Prolog (Clocksin and Mellish, 1981). Since there are many similarities in the syntax of FIGG and Prolog, some knowledge of Prolog would be beneficial for understanding the syntax of FIGG, but it is not compulsory. One can describe FIGG in terms of the format required for specifying the grammar, the control mechanisms provided, and the system commands available.

4.1.1. The Grammar

Currently, there can only be one grammar at a time in the system. As is traditional in logic grammar syntax, terminal symbols are stated as lists. Gap symbols are represented as logic terms of the form *gap(G)*, where the gap is referenced by a variable, *G*, unique to the rule. Any other logic terms denote nonterminal symbols. Grammars processed by the bottom-up parser are allowed to have logic variables as grammar symbols. These variables can unify with any nonterminal symbol during derivations. The empty string is represented by the empty list, [ ]. Productions are of the form

(4.3) *RuleName* : *Rule*.

where *Rule* is an unrestricted GG rule, and *RuleName* is a logic term representing the name of the rule. The rule name is optional. The colon separating the rule and its name appears if and only if there is a rule name. All rules, along with any other commands, must be terminated with a period. The start symbol, $\sigma(\sigma_1,\sigma_2, \ldots ,\sigma_n)$, of the grammar is specified as

(4.4) start_symbol $\sigma(\sigma_1,\sigma_2, \ldots ,\sigma_n)$ / *Success*.

Success specifies, in Prolog, what to do when an input string is successfully parsed according to the grammar. If "/ *Success*" is omitted, then a parse found message will

be displayed for each successful parse. FIGG also allows the user to specify entire classes of rules via rule schemata. The structure of a schema is

(4.5) forall *Var* in $[\alpha_1, \alpha_2, \ldots \alpha_n]$, *Body*

where *Body* is executed once for *Var* equal to each α_i for all $1 \leq i \leq n$. *Body* may contain a grammar rule, another schema, or any Prolog code.

4.1.2. Control

Perhaps the most primitive form of control is rule order, since rules are examined sequentially for their applicability. The other control mechanisms provided in FIGG include those of most other logic grammars implementations, (the *cut*, and arbitrary procedure insertion), along with more sophisticated variations of *cut*, control on the size of the gap, and restrictions on applicability of rules (through dominators). When using control mechanisms, it should be noted that the right hand side of the rule is executed before the left hand side. The top-down parser, however, executes the head of the rule first, then processes the right hand side and the rest of the left hand side of the rule. Each side is processed from left to right.

Procedures

Arbitrary procedures can be inserted into the right and left sides of a rule by enclosing the procedural predicates in braces, {}. The procedures may use variables referenced in the terminal symbols, the nonterminals, the gaps, and in other procedures.

Gaps

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. If the parse fails, it will eventually backtrack to this gap and assume a gap containing one more symbol. In

this manner, the gap size keeps increasing until a successful parse is found, or until all possibilities have been tried. A gap symbol that is processed in this manner is called an *increasing* gap. FIGG allows the user to override this default to obtain *decreasing* gaps, which are initially assumed to contain the rest of the sentence, and are decreased in size during backtracking. A decreasing gap is specified as *gap(-,G)*, with both *gap(G)* and *gap(+,G)* interpreted as an increasing gap.

Until now, the contents of the gaps have been unrestricted. This variety of gap is known as an *essential* gap.[5] There are also *restricted* gaps. The symbols contained in the gap are restricted to be elements of a specified set. This set can be described by a list of valid members, or by a list of elements that are not in the set. So, *gap(/a,b,c/,G)* specifies that the gap, *G*, can contain only *a*'s, *b*'s, and *c*'s, while *gap(~/x,y,z/,G)* prevents the gap from containing an *x*, *y*, or *z*. More details about gap processing can be found in section (4.2).

Dominators

Dominators are used to specify which *rule(s)* may introduce a symbol that appears on the left hand side of another rule. The notion is derived from the concept of one symbol *immediately dominating* (being the parent of) another in a parse tree. Currently, dominators can only be used in conjunction with the bottom-up parser. For a symbol, *sym*, that appears on the left hand side of a rule, *sym^dom* specifies that *sym* must be introduced by the rule named *dom*. A dominator can not be used with the empty string symbol, [ ]. Behaviour of a dominator can be illustrated using the following FIGG grammar.

---

[5] This term is adapted from the notion of *essential variables* (Shieber et. al., 1983) in metarules.

(4.6) (a) start_symbol s.
(b) 1: s --> [x], y.
(c) 2: s --> y.
(d) 3: y --> [b].
(e) 4: y^1 --> [a].

The language recognised by this grammar is $\{xa, xb, b\}$. It is the dominator on (4.6e) that permits this rule's use only after rule 1 is used. Consequently, the string a is not included in the language as illustrated in Figure 4-2. Dominators can also be specified for the symbols of the input sentence. This is achieved with the command

(4.7) sentence^*dom*

where *dom* is once again the dominator.

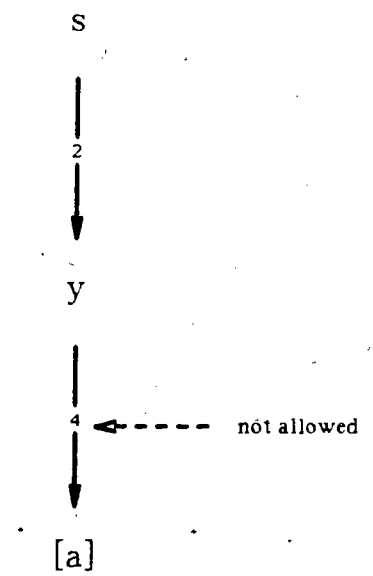

Figure 4-2: Behaviour of Dominator in a Parse Tree

Cuts

The behaviour of the cut symbol varies, depending on whether it is being processed by the top-down or bottom-up parser. The parsers differ in that the top-down parser manipulates lists of terminal symbols, while the shift reduce parser works with sentential forms.

When the top-down parser is invoked, the conventional cut of Prolog, !, prevents backtracking to *goals* to the left of it in the current clause. Examine the following grammar rule.

(4.8) s --> w, x, !, y, z

After a *w* and an *x* have been successfully parsed, the cut is encountered. Then if *y* is successfully parsed but *z* fails, a new parse will be tried for *y*. Subsequent failure of *y* will not cause new parses for *w* or *x* to be attempted. Application of this rule will fail, and the cut will prevent other rules possessing *s* as a head from being tried in lieu of (4.8).

The effect of the local cut, (...)!, is to prevent backtracking within a specified region.

(4.9) s --> w, (x, y)!, z

For the rule illustrated in (4.9), the local cut prevents backtracking into *x* and *y* once they have succeeded. So, if all of *w*, *x*, and *y* are successfully parsed according to (4.9), failure to parse *z* will cause a new parse for *w* to be attempted. If this new parse is found, then *x*, *y* and *z* are tried. The local cut can be useful in conjunction with the gap predicate as illustrated in (5.3).

During bottom-up parsing, the right hand side of a rule is matched against a sentential form. A successful match results in the replacement of the matched region of the sentential form 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, R, is entirely enclosed in a cut, $(R)!$, then the decision to apply R to a sentential form cannot be revoked once the rule has been successfully applied.

Left Recursion

Left recursive GG rules, like

(4.10) nt, α --> nt, β

which could not be processed by the logic grammar implementations discussed in chapter two, can be processed by the FIGG implementation. This means that rules resembling

```
(4.11)  noun_phrase(NP,nom), gap(G), noun_phrase(NP,acc)
        -->  noun_phrase(NP,nom), gap(G), pronoun(reflex,NP,acc)
```

might be used to process natural language sentences like *John wants to shoot himself*. In (4.11), the arguments of the noun phrase and the reflexive pronoun represent the parse tree, and the case. Additional arguments for concepts such as person, number and gender could also be included.

While rules like (4.11) can be directly processed by the bottom-up parser, there are two restrictions on left recursive rules which are required by the top-down parser. First, there must be a nonterminal, β_k, in β (4.10) which does not unify with nt, that can be used to *break* the recursion. The first such *nonrecursive nonterminal* is used automatically by the system to break the recursion. However, in cases where there is mutual recursion between symbols, or when some other symbol is desired to break the recursion (for efficiency reasons), the nonrecursive nonterminal can be explicitly stated as shown below.

(4.12) nt, α --> β_k \ nt, β

For example, consider a rule responsible for converting (4.13a) into the sentence (4.13b).

(4.13) (a) John wants *John throws the ball.*
(b) John wants *to throw the ball.*

Ignoring number agreement and verb agreement, this can be described by the following rule, where *to* is used as the nonrecursive nonterminal.

(4.14) np(NP,nom), v(want), np(NP,nom), v(V)
--> to \ np(NP,nom), v(want), to, v(V).

The second restriction requires that when a nonrecursive nonterminal from a rule R appears in the left side of any rule, R'; then unless it is the head it must also appear on the right side of R'. Moreover, both instances must be appended with @ followed by a variable unique to the rule. The need for this restriction is described in the discussion of the top-down parser in the next section. According to these restrictions, (4.11) could be rewritten as

(4.15) noun_phrase(NP,nom), verb(V)@V1, gap(G), noun_phrase(NP,acc)
--> noun_phrase(NP,nom), verb(V)@V1, gap(G), pronoun(reflex,NP,acc)

with *verb(V)* added to the rule and used as the nonrecursive nonterminal.

4.1.3. System Commands

FIGG commands are translated into Prolog for execution. If a command is entered that is not a FIGG command, it will be passed to the Prolog processor for execution. Consequently, most Prolog commands are also allowed by the system. A sample terminal session that illustrates the use of some commands is provided in Appendix A.

Commands can be entered interactively, or they can be read from a series of files by entering

(4.16) [$file_1$, $file_2$, ... , $file_n$].

As with Prolog, these files may themselves consult other files. The file name *user* is reserved to represent the keyboard and terminal. Files that contain Prolog code can also be processed by the system, by using

(4.17) -prolog [$file_1$, $file_2$, ... , $file_n$].

If the Prolog file's name is preceded by a minus sign, -, the file will be *reconsulted*. That is, the existing definitions for all predicates defined in the file will be eliminated and replaced by the new definitions. Any Prolog file may be reconsulted, even those used by the *lexicon*.

Currently, there can only be one grammar at a time in the system. To remove all rules associated with a grammar from the database, the **clear** command need only be entered. This command also removes any specified start symbols, and clears all system flags to their default values.

A lexicon, which is written in Prolog, can be entered with the command

(4.18) lexicon [$file_1$, $file_2$, ... , $file_n$].

The lexicon should supply a definition for the predicate *lookup(String,Word)*. Given a string corresponding to a word of the input sentence, *lookup* should return an atom representing the word found in the lexicon, and should fail otherwise. So, if *String="loves"*, then *lookup(String,Word)* would result in *Word=loves* if *loves* were in the lexicon. Entering the *lexicon* command with no arguments will disable any user defined lexicon.

The parser can be called with

(4.19) parse, input $file_i$, output $file_o$

where the input and output files, $file_i$ and $file_o$ respectively, are optional. Sentences,

which are terminated with periods, will be read and parsed until an end of file, CTRL-D, is encountered. When sentences are being read for parsing, the FIGG command prefix, >, is replaced by the prompt, ?. When *parse mode* is entered, the system will state whether the system will attempt to find all parses, or if it will look for only a single parse. Parsing can be suspended, to call the FIGG command interpreter, by entering ">" in place of a sentence. This facilitates the changing of system flags (such as number of parses). The prefix is changed to "?>" at this time to remind the user that parsing has been suspended. Parsing is continued until an end of file is reached. If the same output file is referenced in subsequent parse commands, the output will be appended to the end of the file, preventing the previous output from being overwritten. An output file can be closed by entering

(4.20) close *file*.

The + and - commands are used to set and clear the system flag whose name follows the command. The *oneparse* flag specifies if one parse or all parses will be attempted. Initially, this flag is not set. So to request *single parse mode*, the command **+oneparse** must be entered. To return to *all parses mode*, **-oneparse** can be entered. When the *display* flag is set, the Prolog translation of any rules processed by the system will be displayed. No rules are added to the database when this flag is set. This allows the user to examine the translation of the grammar without modifying the database. Entering **-display** will return the system to *generate mode*. By default, the bottom-up parser is used in the system. If the top-down parser is to be used, the *topdown* flag must be set *before* the grammar is read into the system.

(4.21) +topdown.

The bottom-up parser is called by entering **-topdown**. This parser flag should not be changed when there is an active grammar in the system. The **flags** command will display the status of all flags.

If the execution of the FIGG processor is aborted, due to an interrupt or an error, it may be re-invoked by entering the predicate **figg** from Prolog.

4.2. Implementation of FIGG

FIGG is written in C-Prolog (Pereira, 1984), and runs in a UNIX[6] environment on a VAX 750 and on a Motorola 68000 based SUN Workstation. The source code for the system can be found in Appendix D. To describe the implementation, it is better to restate the form of a unrestricted gapping grammar rule as

(4.22) $\alpha_0, \text{gap}(G_1), \alpha_1, \ldots, \text{gap}(G_m), \alpha_m$
$\quad \text{--> } \beta_0, \text{gap}(G'_1), \beta_1, \ldots, \text{gap}(G'_n), \beta_n$

with $m, n \geq 0$, $0 \leq i \leq m$, $0 \leq j \leq n$, $\alpha_i, \beta_j \in V_N \cup V_T \cup \{\epsilon\}$, and $\text{gap}(G_i), \text{gap}(G'_j) \in \Gamma \cup \{\epsilon\}$. (4.22) is equivalent to (3.1). We can now examine how unrestricted GG rules are processed by both the top-down and bottom-up parsers.

4.2.1. The Top-Down Parser

The top-down parser used in FIGG is based on one of the GG implementations, GG1, proposed in (Dahl and Abramson, 1984). For a nonrecursive rule such as (4.22), processing proceeds in much the same manner as in GG1. The major difference is that the translation into Prolog will result in each nonterminal, $\alpha_{nt} \in \{\alpha_i\}$, being replaced by a *pseudo-terminal*, $[\text{te}(\alpha_{nt})]$ (Colmerauer, 1978). In gapping grammar notation, this can be expressed as

(4.23) $\alpha_0, \text{gap}(G_1), [\text{te}(\alpha_1)], \ldots, \text{gap}(G_m), [\text{te}(\alpha_m)]$
$\quad \text{--> } \beta_0, \text{gap}(G'_1), \beta_1, \ldots, \text{gap}(G'_n), \beta_n$

assuming all α_i to be nonterminals. This is part of the normalisation process, which was described in chapter two (Colmerauer, 1978). Each pseudo-terminal is related to its corresponding nonterminal through a normalisation rule resembling

---

[6]Trademark of Bell Labs

(4.24) $\alpha_{nt}(a_1,a_2,\ldots,a_n)$ --> [te($\alpha_{nt}(a_1,a_2,\ldots,a_n)$)]

Pseudo-terminals and normalisation rules are generated automatically by the system. The translation of (4.23) into Prolog proceeds in a manner similar to gapping grammar processing, resulting in the clause

(4.25) $\alpha_0(X,Y)$:-
(a) $\beta_0(X,X_0)$, gap(G'_1,X_0,X_1), $\beta_1(X_1,X_2)$, ..., gap(G'_n,X_{2n-2},X_{2n-1}), $\beta_n(X_{2n-1},Z)$,
(b) gap(G_1,Y_0,Y_1), c(Y_1,te(α_1),Y_2), ..., gap(G_m,Y_{2m-2},Y_{2m-1}), c(Y_{2m-1},te(α_m),Z).

In (4.25) we assume that none of the rule symbols equals ϵ. If some symbol equals ϵ, replace its translation by "$X = X'$" where X and X' are the incoming and outgoing phrase arguments. If any of the α_i's or β_j's are terminals, they are translated as $c(X_{2i-1},\alpha_i,X_{2i})$ and $c(X_{2j-1},\beta_j,X_{2j})$ respectively. The productions specified in (2.24), which are restated below,

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

can result in clauses similar to

(4.27) (a) s([],X,Y) :- c(X,a,X0), bs(_,X0,X1), c(X1,c,Y).
(b) s([],X,Y) :- c(X,a,X0), s(_,X0,X1), b(_,X1,X2), c(X2,c,Y).
(c) bs([],X,Y) :- c(X,b,X0), bs(_,X0,X1), gap(G,X1,Z), gap(G,Y,Y0), c(Y0,te([],b),Z).
(d) bs([],X,Y) :- c(X,b,X0), c(X0,b,X1), gap(G,X1,Z), gap(G,Y,Y0), c(Y0,te([],b),Z).
(e) b(_1,X,Y) :- c(X,te(_1,b),Y).

The first argument of the translation of nonterminal symbols (the *recursive argument*) is used for processing left recursive rules. (4.27e) is a normalisation rule. Figure 4-3 illustrates the depth first goal tree for the top-down parse of the sentence *aaabbbccc* shown in Figure 2-4. The superscript associated with each nonterminal node corresponds to the rule used from (4.27). Since the recursive argument is not required to parse this sentence, it is omitted in Figure 4-3.

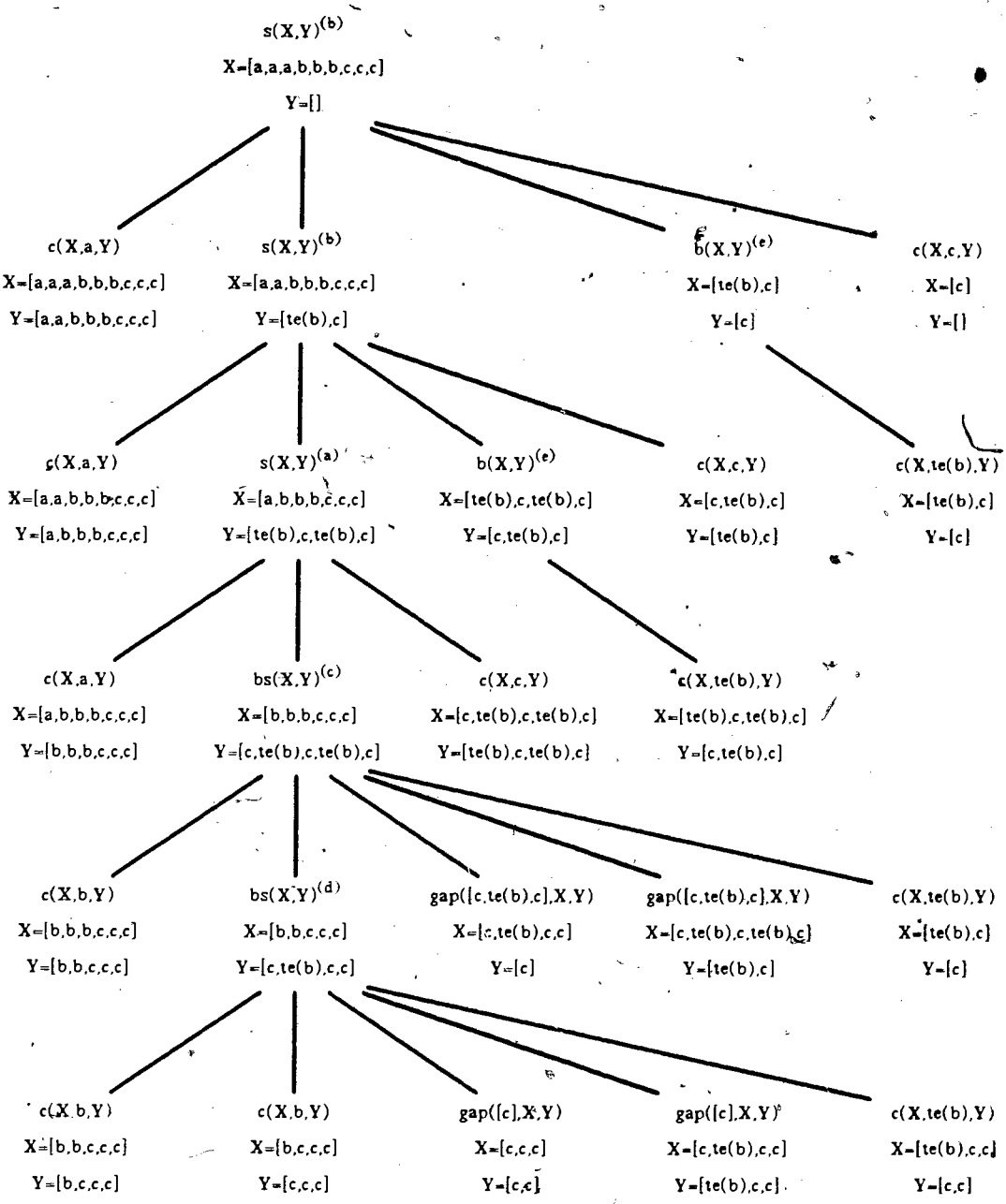

Figure 4-3: Goal Tree for Top-Down Parse of $a^3b^3c^3$

Cuts and procedures that appear in the rules are inserted directly into the translation. For example, the rule

(4.28) s --> x, !, y, {foo}.

would be translated as

(4.29) s([],X,Y) :- x(_,X,X0), !, y(_,X0,Y), foo.

The local cuts are implemented using the *call1* predicate. A call to this predicate will execute its single argument, and once this call succeeds, backtracking into the argument will not be allowed.

(4.30) call1(X) :- call(X), ! ; fail.

To process the various gap symbols, backtracking is used to obtain the various gap sizes. Since *increasing* gaps are initially assumed to contain no symbols, and are subsequently increased in size, they can be implemented according to the following rules.

(4.31) gap(+,[]) --> [].
gap(+,[Word|G]) --> [Word], gap(+,G).

For a *decreasing* gap, the order of the rules is reversed.

(4.32) gap(-,[Word|G]) --> [Word], gap(-,G).
gap(-,[]) --> [].

The restricted gaps require an addition check to see if the *Word* in the gap is an element of the specified set. When a list of valid gap elements, */X|Y/*, is provided, the rule used for the gap predicate is shown below.

(4.33) gap([X|Y],[]) --> [].
gap([X|Y],[Word|G]) --> [Word], {element(Word,[X|Y])}, gap([X|Y],G).

Similarly, if an exclusion list is given, the definition is modified by the insertion of the *not* operator.

(4.34) gap(~[X|Y],[]) --> [].
gap(~[X|Y],[Word|G]) --> [Word], {not element(Word,[X|Y])}, gap(~[X|Y],G).

The gap predicates could also be defined more efficiently in terms of Prolog clauses, instead of grammar rules, but the definitions would be less clear.

Left recursive rules, resembling (4.22) where $\alpha_0 = nt = \beta_0$.

(4.35) nt, gap(G_1), α_1, ... gap(G_m), α_m
--> nt, gap(G'_1), β_1, ... gap(G'_n), β_n

are *interpreted* rather than converted into similar rules that are not left recursive. A simple approach taken in (Popowich, 1985c) involves skipping over the region corresponding to nt to look for the next symbol, then examining the gap for a valid nt. Extending this for (4.35) would result in something similar to

(4.36) nt(X,Y) :- gap(G,X,X_0), β_k(X_0,X'), nt(G,[]), $tr(\beta)$, $tr(\alpha)$

where β_k is the nonrecursive nonterminal, $tr(\beta)$ is the translation of the right side of (4.35) with β_k omitted and X' used as the argument for the initial incoming phrase, and $tr(\alpha)$ is the translation of the left side of (4.35) with the head omitted. (4.36) will not work however, since nt will be processed by the predicate nt(G,[]) without the benefit of the context — that is, the rest of the string.

To process (4.35) the following translation is used

(4.37) nt([], X, Y) :-
(a) gap(Gap,[R|N],NewB_k,X,X_{2k-1}), β_k(B_k,X_{2k-1},X_{2k}),
(b) numgen(N), concaten(Gap,[te(NewB_k,β_k)|X_{2k}],X'),
(c) nt(B_0,X',X'_0), gap(G'_1,X_0,X_1), ...
(d) nonemptylist(B'_k), β_k(B'_k,X'_{2k-1},X'_{2k}), element([R|N],B'_k), ...
(e) gap(G'_n,X_{2n-2},X_{2n-1}), β_n(B_n,X_{2n-1},Z),
(f) gap(G_1,Y_0,Y_1), c(Y_1,te([],α_1),Y_2), ... gap(G_m,Y_{2m-2},Y_{2m-1}), c(Y_{2m-1},te([],α_m),Z).

The string is first checked (4.37a) for a substring satisfying the *nonrecursive nonterminal*, β_k, to determine the applicability of the rule. Recall that, by default, the nonrecursive nonterminal is the first β_i that does not unify with α_0. If such a substring is found, it is then replaced by a pseudo-terminal which is marked with a number that corresponds to the rule (4.37b). This prevents the same string from being used to break the recursion in subsequent applications of the same rule. Then

the modified string is processed according to the rule (4.37c-f), while forcing β_k to use the normalising rule to match with the pseudo-terminal marked by the current application of the rule (4.37d). The 5-ary gap predicate in (4.37a) *breaks the recursion* by ensuring that the current rule, R, will not succeed more than once for the same gap. To keep track of which rules have used which symbols as nonrecursive nonterminals, a *recursive argument*, which appears as the first argument, is automatically generated for every nonterminal. It contains a list of pairs, where each pair is of the form "[*R*|*N*]" with *R* and *N* integers unique to each rule, and each application of a rule respectively. The appearance of *@Var* following a nonterminal causes *Var* to be used as the recursive argument. Thus when *@Var* appears on both sides of a rule the associated nonterminals will possess the same recursive argument. So, the restriction mentioned in section 4.1.2 ensures that the recursive argument associated with a symbol is not *forgotten* when a rule is applied. The translation of (4.15) is shown in (4.38)

```
(4.38)      noun_phrase([],NP,nom,X,Y) :-
(a)             gap(Gap,[1|N],NewRA,X,X1), verb(_,V,X1,X2),
(b)             numgen(N), concaten(G,[te(NewRA,verb(V))|X2],NewX),
(c)             noun_phrase(_,NP,nom,NewX,NewX1),
(d)             nonemptylist(V1), verb(V1,V,NewX1,NewX2), element([1|N],V1),
(e)             gap(G,NewX2,X3), pronoun(_,reflex,NP,acc,X3,Z),
(f)             c(Y,te(V1,verb(V)),Y1), gap(G,Y1,Y2),
                c(Y2,te([],noun_phrase(NP,acc)),Z).
```

Now let us examine the use of (4.38) for parsing the nominative noun phrase and the anaphor in the sentence *John mixes a drink for himself*.

We start with the goal

(4.39) noun_phrase(_, Tree, [John,mixes,a,drink,for,himself], Rest)

where *Tree* will be our parse tree for the noun phrase and *Rest* will be the remainder of the sentence after the noun phrase has been found. (4.40) traces the execution of (4.38) for the goal (4.39). In (4.40a), the indented lists before and after

each satisfied goal correspond to their values for *X* and *Y* respectively. The indented lists of (4.40b) represent the value of *Rest* before and after the execution of each clause.

(4.40) (a)

```
[John, mixes, a, drink, for, himself]
gap([John], [1|N], [[1|N]], X, Y)
        [mixes, a, drink, for, himself]
verb([], verb(mixes), X, Y)
        [a, drink, for, himself]
numgen(2)
concaten([John], [te([[1|2]],verb(verb(mixes)))|X], Y).
        [John, te([[1|2]],verb(verb(mixes))), a, drink, for, himself]
noun_phrase([], np(noun(proper,John)), nom, X, Y)
        [te([[1|2]],verb(verb(mixes))), a, drink, for, himself]
nonemptylist([_a|_b])
verb([[1|2]|[]], verb(mixes), X, Y)      /* _a=[1|2] and _b=[] */
element([1|2], [[1|2]])
        [a, drink, for, himself]
gap([a,drink,for], X, Y)
        [himself]
pronoun([], reflex, np(noun(proper,John)), acc, X, Y)
        []
```

(b)

```
        Rest
c(Rest, te([[1|2]],verb(verb(mixes))), Y1)
        [te([[1|2]],verb(verb(mixes))) | Y1]
gap([a,drink,for], Y1, Y2)
        [te([[1|2]],verb(verb(mixes))), a, drink, for | Y2]
c(Y2, te([],noun_phrase(np(noun(proper,John)),acc)), []).
        [te([[1|2]],verb(verb(mixes))), a, drink, for,
            te([],noun_phrase(np(noun(proper,John)),acc))]
```

Recall that there was also a restriction imposed on a nonrecursive nonterminal that prohibited it from appearing on only the left side of a rule, unless it was the head of the rule. To see why this restriction is necessary, consider the following rule which could be used in conjunction with (4.15) to parse the sentence *A drink is mixed by John for himself*.

(4.41)
```
noun_phrase(NPn,nom), verb(V), noun_phrase(NPa,acc) -->
noun_phrase(NPa,nom), v(v(be),aux), v(V,pstprt), [by], noun_phrase(NPn,acc).
```

To obtain the sentence, (4.41) must be applied after (4.15). However, (4.15) will fail since the nonrecursive nonterminal, *verb(V)*, requires *mixes* to be present in the sentence. *Mixes* would be introduced later by the application of (4.41).

4.2.2. The Bottom-Up Parser

The bottom-up shift reduce parser used with FIGG is based on the predicates *sr_parse(Input,Stack,NewStack)* and *reduce(Stack,NewStack)*. The arguments of *sr_parse* correspond to the input phrase, *Input*, the initial stack, *Stack*, and the stack after the parse has been attempted, *NewStack*. *Sr_parse* will shift one input symbol at a time onto the stack, and will let the *reduce* predicate perform zero or more reductions on the stack symbols. Recall that the input is processed from right to left. The actual Prolog definition of this predicate is shown in (4.42).

(4.42)
```
sr_parse([Word|Rest], Stack, NewerStack) :-
    sr_parse(Rest, Stack, NewStack), reduce([Word|NewStack], NewerStack).

sr_parse([], Stack, Stack).
```

The decision to perform a reduction is determined by the Prolog control structure according to the order of the clauses. Reductions are attempted in an order that corresponds to the order of the grammar rules. Initially, if a reduction is possible, it will be performed. Backtracking to this decision will cause it to be revoked. The parse of the entire sentence succeeds if the start symbol is the only symbol left on the stack. For a sentence, *Sentence*, a parse is requested by the top-level goal

(4.43)
```
sr_parse(Sentence,[],[s])
```

where *s* is the start symbol.

If this parser only had to process context-free rules of the form

(4.44) α --> β_1 β_2 ... β_n

then each reduce predicate could resemble

(4.45) reduce([β_1,β_2,....,β_n|X], NewStack) :- reduce([α|X],NewStack).

This would replace the top n symbols of the stack that correspond to the right hand side of (4.44) with α, and then attempt further reductions on the stack. The predicate *reduce(X,X)* is used when no reduction takes place. Since FIGG must also process context-sensitive rules, more than one symbol must be added to the stack. One might therefore be tempted to translate

(4.46) $\alpha_1\ \alpha_2\ \ldots\ \alpha_m$ --> $\beta_1\ \beta_2\ \ldots\ \beta_n$

as follows.

(4.47) reduce([β_1,β_2,....,β_n|X], NewStack) :- reduce([α_1,α_2,....,α_m|X], NewStack).

But, this would block any reduction of $\alpha_i\alpha_{i+1}...\alpha_m$, where $i \neq 1$, during the next reduction application. Therefore, reductions after the addition of each α_i to the stack must be allowed. To achieve this, the following translation of (4.46) is used.

(4.48) reduce([β_1,β_2,....,β_n|X],NewStack) :- sr_parse([α_1,α_2,....,α_m],X,NewStack).

The *sr_parse* predicate will add the new stack symbols, one by one, and allow reductions to take place. To simplify the translation procedure context-free rules are also translated in this manner, although this results in less efficient parsing.

Until now, the translation of the gap symbols and procedural control that may appear in the rules has been ignored. Gaps and control that appear on the right hand side will affect the *pattern matching* of the stack symbols. Placement of gaps and control in the left side of the rule will influence the *symbol generation* of the new stack symbols. The pattern matching process occurs first as specified by the left to right processing of the right hand side of the rule. Afterwards, the symbol generation is done, also in a left to right fashion. We shall now examine how gap symbols, cuts, procedures, and dominators are incorporated in the translation of the rules into *reduce* predicates.

To incorporate gap symbols into these translations, the same definition of the *gap* predicate as presented in the description of the top-down parser can be used. The gap symbols that appear on the right hand side will contain stack symbols that can be inserted into the new stack by the gaps on the left side of rule. Consider the following unrestricted gapping grammar for L_1.

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

According to the translation procedure presented so far, this would result in the generation of the clauses listed below.

```
(4.50) (a) reduce([[a],[b],[c]|X],NewStack) :- sr_parse([s],X,NewStack).
       (b) reduce([[a],s,b,c|X],NewStack) :- sr_parse([s],X,NewStack).
       (c) reduce([[b],[b]|X0],NewStack) :- gap(-,G,X0,[[c]|X]), gap(-,G,Y0,[b,c]),
                 sr_parse([[b]|Y0],X,NewStack).
```

Figure 4-4 shows part of the goal tree for the parse of the sentence $a^3b^3c^3$ using the clauses provided in (4.42) and (4.50). In this tree, *reduce* and *sr_parse* are abbreviated as *rd* and *sr* respectively. This goal tree corresponds to the parse shown in Figure 2-4.

Procedures, which are included in braces, can affect pattern matching and symbol generation. They are inserted directly into the definition of *reduce* for the rule. However, with the addition of procedures, it is necessary to ensure that the correct order of pattern matching, procedure execution, and symbol generation occurs. Consider the following rule which is similar to (4.49c), but does not permit an empty gap.

(4.51) [b], gap(-,G), b, c --> [b], [b], gap(-,G), {G\==[]}, [c].

During the pattern match, the procedure call should be executed before the attempt to match the */c/*. This is done in the following translation.

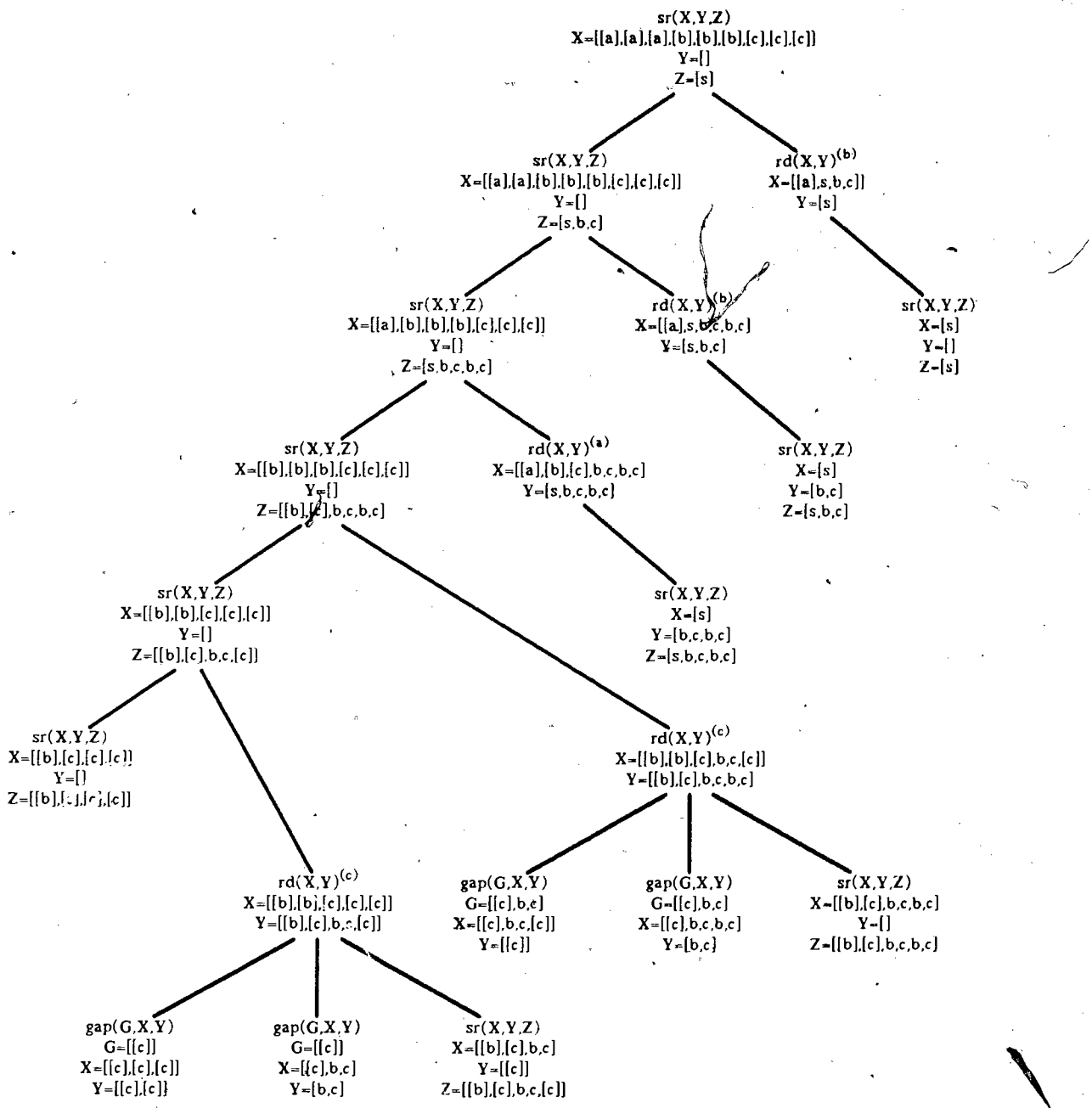

Figure 4-4: Goal Tree for Shift-Reduce Parse of $a^3b^3c^3$

(4.52) reduce([[b],[b]|X0],NewStack) :- gap(-,G,X0,X1), G\==[], X1=[[c]|X], gap(-,G,Y0,[b,c]), sr_parse([[b]|Y0],X,NewStack).

This translation and (4.50c) are very similar except for the appearance of the procedure call and an extra variable, X1, in (4.52). The extra variable is required to defer the match of the *[c]* until after the execution of "G\==[]".

When a cut appears on the right hand side of a rule, it is interpreted in exactly the same manner as a procedure call. In fact, enclosing the cut in braces will result in an identical translation as a *bare* cut. The cut prevents backtracking into regions before the cut during the pattern matching phase. The left hand side may contain a single cut, but this cut behaves differently than the conventional cut. In the translation, the symbols to the left of the cut are placed into the *Input* argument of *sr_reduce*, with the symbols to the right of the cut placed directly onto the stack. Thus, a rule with a cut at the right end of the left hand side is equivalent to a similar rule without the cut. This use of the cut prevents any reduction attempts until after the first symbol to the left of the cut has been placed onto the stack. Consider the following rule which results from modification of (4.49c).

(4.53) [b], !, gap(-,G), b, c --> [b], !, [b], gap(-,G), [c].

The translation method for cuts would result in a clause similar to

(4.54) reduce([[b]|X0],NewStack) :- !, X0=[[b]|X1], gap(-,G,X1,[[c]|X]), gap(-,G,Y,[b,c|X]), sr_parse([[b]],Y,NewStack).

Local cuts are once again implemented with the *call1* predicate. So, the translation of

(4.55) [b], !, gap(-,G), b, c --> [b], !, [b], (gap(-,G), [c])!.

would result in the following Prolog code.

(4.56) reduce([[b]|X0],NewStack) :- !, X0=[[b]|X1], call1(gap(-,G,X0,[[c]|X])), gap(-,G,Y,[b,c|X]), sr_parse([[b]],Y,NewStack).

There is a special case of the local cut in which the entire rule is enclosed in the cut.

(4.57) ([b], !, gap(-,G), b, c --> [b], [b], gap(-,G), [c])!.

While cuts within a rule affect the pattern matching and symbol generation, this local cut prevents backtracking once the rule has successfully reduced the stack.

(4.58) reduce([[b],[b]|X0],NewStack) :- gap(-,G,X0,[[c]|X]), gap(-,G,Y,[b,c|X]), sr_parse([[b]],Y,NewStack), !.

This also prevents any subreductions done by *sr_parse* in (4.58) from being reattempted.

The dominators work in conjunction with the rule name. For each grammar symbol, *x*, the stack symbol is actually of the form *x^Name*. When a dominator is specified for a rule symbol, it becomes the *Name* of the stack symbol. Each symbol of the right side of the rule, *y*, is translated into *y^RuleName* where *RuleName* is the name of the rule. If the rule does not have a name, the system gives it a unique name. Thus the specification of a dominator during the symbol generation stage, will result in a stack symbol that will match only the specified rule during subsequent pattern matching stages. Since dominators can also be specified for gaps, a redefinition of the gap predicate is required. The *gapD* predicate is similar to *gap* except that it possesses an extra argument (the first argument) which is used for the rulename. This argument is the *Name* of each symbol contained in the gap. The name *'$undef'* is reserved to signify that no dominator was specified for a gap which is on the left hand side of the rule. So, consider the following *efficient* grammar.

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

Since no rule names are specified, the system will generate them. Assuming that (4.59a-c) are named 1, 2, and 3 respectively, *reduce* clauses similar to the following would be generated.

(4.60) (a) reduce([[a]^1,[b]^1,[c]^1|X],NewStack) :- sr_parse([s^D1],X,NewStack).
(b) reduce([[a]^2,s^2,b^2,c^2|X],NewStack) :- sr_parse([s^D1],X,NewStack).
(c) reduce([[b]^3,[b]^3|X0],NewStack) :- gapD(3,-,G,X0,[[c^3]|X]),
gap('$undef',-,G,Y,[b^D1,c^D2|X]), sr_parse([[b^D3]],Y,NewStack), !.

The actual translation produced by the system differs slightly but is equivalent to

(4.60). This discrepancy arises from the desire for a concise translation procedure, which can be found in Appendix D. The actual translation of (4.59) produced by FIGG is shown in Figure 4-5.

```
reduce([[a]^$name1,[b]^$name1,[c]^$name1|_36],_29) :-
    _39=_36,
    sr_parse([s^_37],_39,_29),
    true.
reduce([[a]^$name2,s^$name2,b^$name2,c^$name2|_36],_29) :-
    _39=_36,
    sr_parse([s^_37],_39,_29),
    true.
reduce([[b]^$name3,[b]^$name3|_36],_29) :-
    gapD($name3,-,_43,_36,[[c]^$name3|_37]),
    _38='[]',
    gapD($undef,-,_43,_44,[b^_39,c^_40|_37]),
    sr_parse([[b]^_41|_38],_44,_29),
    !.
```

Figure 4-5: FIGG translation of an Unrestricted GG Rule

Chapter 5

Applications of Unrestricted Gapping Grammars

Unrestricted gapping grammars, as implemented in FIGG, can be considered as a programming language, and can be used to parse sentences according to 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. As a programming language, the *recursive enumerable* power of unrestricted gapping grammars is a benefit, not a hindrance. We will first examine the use of procedural control with unrestricted gapping grammars. Then, FIGG will be used to examine the use of unrestricted gapping grammars in describing free and partially free word order languages, and in implementing the metarules of generalised phrase structure grammars.

5.1. Use of Procedural Control

Procedural control can be introduced into the productions of the grammar to restrict the language described, or to improve parsing efficiency. In this section, we shall use a selection of familiar formal languages to examine the use of the various control mechanisms.

Since initial results suggest that the determination of what goes into the gap to be

one of the major problems (the gap determination problem, (Dahl, 1984)), procedural control can help determine the contents of a gap. 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 (5.1).

(5.1) (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 5-1. 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 (5.1) 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 (5.1b-e) as shown in (5.2), resulting in much improved performance as illustrated in Table 5-1.

(5.2) (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 5-1 represents a grammar equivalent to G_1' that is processed by the bottom-up parser. The productions of G_1'' are provided in (5.3).

(5.3) (a) s --> as, bs, cs, ds.
(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])!.

In this case, the bottom-up processing takes about two and a half times longer than top-down parsing.

Table 5-1: 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	12.
	|	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 $L_2 = \{a^n b^n c^n \mid n > 0\}$.

(5.4) (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 grammer containing these productions would be ambiguous. It should be noted that these productions, would not form an ambiguous extraposition grammar. The ambiguity can be removed through modification of (5.4b-c), as shown in (5.5), resulting in a new gapping grammar G'_2.

(5.5) (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.

(5.6) (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 5-2. This time, the bottom-up parser is slower by a constant multiple of three. Further development on this Prolog parser may improve its efficiency.

Table 5-2: Parse and total analysis times for $a^m b^m c^m$

grammar	m=1	m=5	m=10	m=15	m=20	m=25	m=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.

5.2. Description of Non-Fixed Word Order

One of the benefits that has been cited for gapping grammars is the ease with which they can describe languages with free word order (Dahl, 1984). MGs, augmented with a capacity for *floating terminals*, have also been used in an attempt to grammatically capture partially free word order (Bien et.al., 1980). *Floating terminals*, as opposed to the conventional, *anchored* terminal symbols, are introduced into the MG rule. They denote symbols that may be found in a sentence anywhere to the right of the position specified in the rule. However, arbitrary relocation of nonterminal symbols in not permitted. While this method successfully describes totally free word order, it can not adequately capture restrictions on symbol relocation of a partially free word order language in a concise form. We shall examine the use of FIGG for processing totally free word order and partially free word order languages by providing unrestricted gapping grammars that describe the languages.

5.2.1. Free Word Order

Dahl proposed the use of GGs for describing totally free word order (Dahl, 1984). As with *floating terminals*, the unrestricted relocation of a symbol is obtained by shifting symbols to the right.

(5.7) *sym*, gap(G) --> gap(G), *sym*.

A symbol, *sym*, is shifted to the left by shifting of all symbols between *sym* and its new location to the right. Since GG rules can not start with a gap symbol, direct specification of *left shifting* is not possible. However, it could be described using an unrestricted gapping grammar rule of the form

(5.8) gap(G), *sym* --> *sym*, gap(G).

Latin verse is an example of a language that allows an arbitrary rearrangement of words. Since FIGG has been shown to have trouble parsing sentences of increasing length, the grammar shown in (5.9), which is an extension of one found in (Dahl, 1984), was examined to determine if FIGG encountered a similar problem with it. This grammar does not include restrictions on person/number agreement. The last four rules of (5.9) give the language its free word order property. Table 5-3 summarises the results obtained using the top-down parser on the sentences shown in (5.10).

(5.9)

```
start_symbol sentence(Tree) / (write('Parse Tree:'), writetree(Tree))

sentence(s(NP,VP))                --> noun_phrase(NP,nom), verb_phrase(VP).

verb_phrase(vp(ABL,DAT,ACC,V)) --> prep_phrase(ABL), noun_phrase(DAT,dat),
     noun_phrase(ACC,acc), verb(V,acc_dat).
verb_phrase(vp(DAT,ACC,V))       --> noun_phrase(DAT,dat),
     noun_phrase(ACC,acc), verb(V,acc_dat).
verb_phrase(vp(ACC,V))           --> noun_phrase(ACC,acc), verb(V,trans).
verb_phrase(vp(V))               --> verb(V,intrans).

noun_phrase(np(N),Case)          --> noun(N,Case,Gender).
noun_phrase(np(N,ADJS),Case)     --> noun(N,Case,Gender),
     adjectives(ADJS,Case,Gender).

prep_phrase(pp(PREP,ABL))        --> prep(PREP), noun_phrase(ABL,abl).

adjectives(ADJ,Case,Gender)      --> adjective(ADJ,Case,Gender).
adjectives(adjs(ADJ,ADJS),Case,Gender) --> adjective(ADJ,Case,Gender),
     adjectives(ADJS,Case,Gender).

noun(n(Word),Case,Gender), gap(G)        --> gap(G), [Word],
     {dict(noun(Case,Gender),Word)}.

adjective(adj(Word),Case,Gende ), gap(G) --> gap(G), [Word],
     {dict(adjective(Case,Gender),Word)}.

verb(v(Word),Type), gap(G) --> gap(G), [Word], {dict(verb(Type),Word)}.

prep(prep(Word)), gap(G)    --> gap(G), [Word], {dict(prep,Word)}.
```

(5.10) (a) vir est.
(b) puella puerum amat.
(c) puella bona puerum amat.
(d) puella bona puerum parvum amat.
(e) puer bonus puellae parvae florem dat.
(f) puer bonus puellae parvae florem rubrum dat.
(g) vir bonus vetus puellae parvae florem rubrum dat.
(h) vir bonus in agro puellae florem rubrum dat.
(i) vir bonus in agro puellae parvae florem rubrum dat.
(j) vir bonus in agro rubro puellae parvae florem rubrum dat.

Table 5-3: Summary of results for parsing according to the latin grammar

sentence	length (words)	parse time (sec)	total time (sec)	parse time reverse (sec)	total time reverse (sec)
(a)	2	0.2	0.5	0.3	0.5
(b)	3	0.3	0.7	0.3	0.7
(c)	4	0.8	1.2	0.9	1.2
(d)	5	1.2	1.8	1.3	1.8
(e)	6	2.1	3.2	2.2	3.1
(f)	7	3.1	4.7	3.5	4.8
(g)	8	8.0 2.3	12.	8.4 2.3	12.
(h)	8	3.5	6.3	4.0	6.4
(i)	9	5.9	9.9	6.4	9.9
(j)	10	9.6	14.	10.	14.

The results illustrate satisfactory behaviour with increasing sentence length. The exponential growth is still present, but not to the degree observed with the rules illustrated in (5.1). One may even be tempted to improve the efficiency by placing a cut, !, at the end of the last four clauses to prevent backtracking. But, it is worth noting that when a sentence contains two adjectives which modify the same noun, the introduction of the cut would prevent both parses from being found. (5.10g), which

is shown below along with its translation, is an example of a sentence with this property.

(5.11) (a) vir bonus vetus puellae parvae florem rubrum dat.
(b) the good old man gives the small girl the red flower.

5.2.2. Partially Free Word/Constituent Order

While totally free word order may be very easy to describe with GGs, most natural languages possess some restrictions on the location of the phrasal constituents. The *immediate dominance/ linear precedence* format for grammar rules, which is used with generalised phrase structure grammars (Gazdar and Pullum, 1982), allows a concise description of a potentially very large set of context-free rules that describe the allowed constituent order.

Recall that an *immediate dominance* (ID) rule resembles a context-free rule, but it specifies only that the symbol on the left hand side of the rule immediately dominates (is the parent of) the symbols of the right side. The order of the right hand side symbols is not restricted by this rule. Instead, it is restricted by the *linear precedence* (LP) relations. A linear precedence relation, $\beta_i < \beta_j$, is a transitive relation between two symbols of the grammar, β_i and β_j, that states which symbol must precede the other if they both appear in the right hand side of a context-free rule. Not all context-free grammars can be described using the ID/LP format. For example, if the symbol *sym* precedes another symbol sym_1 in one context-free rule, the reverse order can not appear in any context-free rule.

Although ID/LP relations are intended to represent a collection of context-free rules, (which will be referred to as *base rules*), they can be interpreted by a collection of unrestricted gapping grammar rules that contain procedural control. The method for

converting an ID/LP specification into a unrestricted GG will be called the *ID/LP - UGG conversion procedure*. Let us first consider how to convert a single ID rule

(5.12) $nt \dashrightarrow \beta_1\ \beta_2\ \ldots\ \beta_n$

into a FIGG specification according to the LP restrictions.

We will assume that each β_i is unique. An ID rule where each β_i is unique will be called *nonambiguous*. Let $\mathbf{B} = \cup_{i=1}^{n}\{\beta_i\}$. If $\beta_i = \beta_j$ and $i < j$, then this *ambiguous* ID rule can be converted to a nonambiguous ID rule by the following three steps.

- Replace β_j by a new symbol β'_j which is unique to the rule.
- Introduce the linear precedence relation $\beta_i < \beta'_j$.
- Add the rule $\beta'_j \dashrightarrow \beta_j$ to the grammar.

From the nonambiguous ID rule, construct one context-free rule that does not violate the LP restrictions. (This is trivial if we assume that the order of the symbols on the right hand side of the ID rule does not violate the LP restrictions.)

(5.13) $nt \dashrightarrow \beta_1, \beta_2, \ldots \beta_n$.

Given this context-free rule, we can obtain all other permutations of the β_i's that are allowed by the LP restrictions by introducing a gapping rule similar to

(5.14) $\text{gap}(G), \beta_i \dashrightarrow \beta_i, \text{gap}(G)$

along with the following restrictions.

(5.15) (a) Each symbol, β_i, can be relocated at most one time per derivation.

(b) For each symbol, g, contained in the gap, G, the LP restriction $g < \beta_i$ must not be present.

The first restriction ensures that the procedure will terminate, while the second prevents violation of the LP restrictions.

Let us examine one method for incorporating the first restriction into the grammar.

Given an ID rule (5.12), create a context-free rule that does not violate any LP relations where each symbol, β_i, on the right hand side of (5.12) will be replaced by the nonterminal β_i'.

(5.16) nt --> β_1', β_2', . . . β_n'

Let $\mathbf{B'} = \cup_{i=1}^{n}\{\beta_i'\}$. The quote means that the symbol can be moved, whereas unquoted symbols cannot be moved. To incorporate the restrictions of (5.15), (5.14) can be rewritten as

(5.17) gap(G), β_i' --> β_i, gap(G), {no_LPs(G,β_i)}.

Restriction (5.15b) is incorporated in (5.17) by the predicate *no_LPs(x,y)*, where x is a list ($x = [x_1 x_2 ... x_n]$). This predicate succeeds when none of the LP relations $x_1 < y$, $x_2 < y$, ... $x_n < y$ are present. This rule can be thought of as a rule schema for the many context-sensitive rules that would be required to shift the symbol to the left. What would take several rule applications of context-sensitive rules can be achieved by the application of a single unrestricted GG rule. For an ID rule **R**, (5.12), the set $\mathbf{P_R}$ will refer to the two productions, (5.16) and (5.17), required to describe the set of base rules, $\mathbf{R_R}$, constructable from **R** and the LP relations.

Given a base rule, $\mathbf{R}_{Ri}$, a unique *leftmost derivation* can be obtained, using $\mathbf{P_R}$, which is equivalent to the application of $\mathbf{R}_{Ri}$. The definition of *leftmost derivation*, which is based on one found in (Hopcroft and Ullman, 1979), is that a rule is applied to the leftmost nonterminal of a sentential form. With respect to (5.17), this means that the leftmost quoted symbol is shifted according to the rule. By definition, the grammar will be unambiguous if and only if the leftmost derivation for each sentential form permitted by the grammar is unique. For example, all possible leftmost derivations of the unrestricted GG based on the ID rule

(5.18) s --> a b c

with no LP restrictions are provided in the *permutation tree* illustrated in Figure 5-1.

A *permutation tree* for an ID rule (5.12) possesses sentential forms as nodes with "$\beta'_1\beta'_2...\beta'_n$" as the root node. Nodes which contain no quoted symbols are terminal nodes. The root node will be considered a *level 0* node, while a descendant of a node will have a level number which is one greater than that of its parent. A node, ψ, has a descendant, ω, iff ψ ==> ω using rule (5.17).

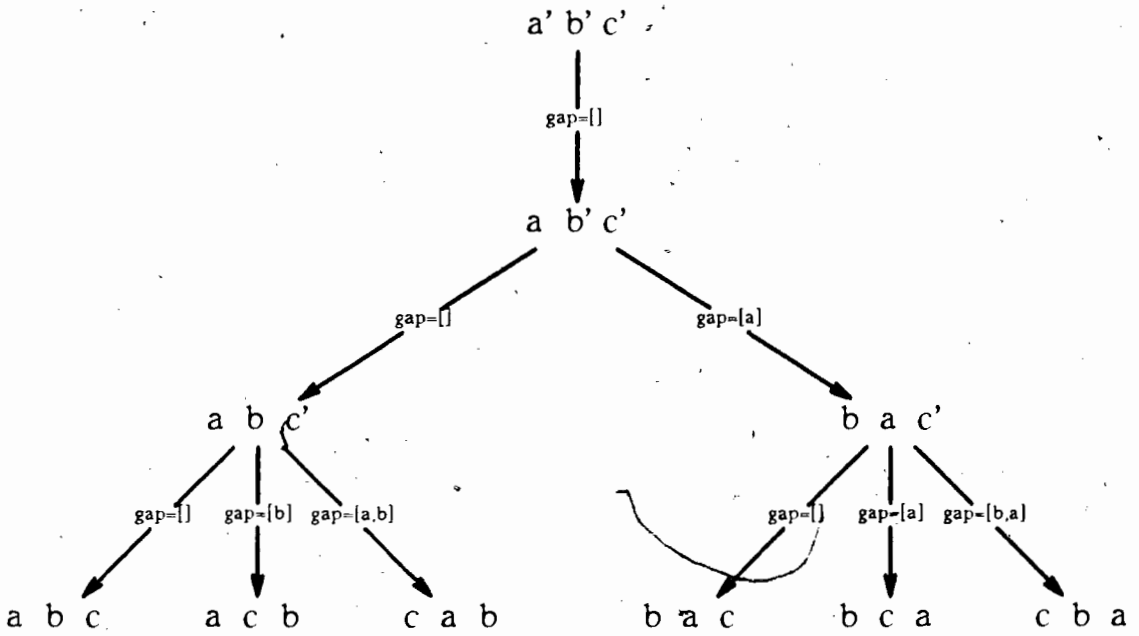

Figure 5-1: Permutation Tree

Lemma 1: All sentential forms of a permutation tree are of the form $\chi_i\chi'_i$, with $|\chi_i| = i$ and $\chi'_i = \beta'_{i+1}...\beta'_n$ where i is the level of the node, $\beta'_1\beta'_2...\beta'_n$ is the root node, with $\chi_i \in B^*$ and $\chi'_i \in B'^*$.[7]

Proof: The root node, $\beta'_1\beta'_2...\beta'_n$, which is on level zero is of the required form since $\chi_0 = \epsilon$ and $\chi'_0 = \beta'_1\beta'_2...\beta'_n$. ($\epsilon$ denotes the empty string). Assume that a sentential form of level i, $\upsilon_i\omega_i\beta'_{i+1}\omega'_i$ (where $\upsilon_i,\omega_i \in B^*$, $\omega'_i \in B'^*$, and $|\upsilon_i\omega_i| = i$) is of this form, and $\beta'_{i+1}\omega'_i = \beta'_{i+1}\beta'_{i+2}...\beta'_n$. Application of (5.17) to

[7] $|x|$ represents the length of the string x.

this sentential form yields the new sentential form $\nu_i\beta_{i+1}\omega_i\omega'_i$, which will be on level $i+1$. $|\nu_i\beta_{i+1}\omega_i| = i+1$, and $\omega'_i = \beta'_{i+2}...\beta'_n$. By induction, we can conclude that this is a property of all nodes.

Corollary 1: No sentential form can appear on two different levels of a permutation tree.

Corollary 2: For any node, ω, of a permutation tree corresponding to a rule, **R**, $|\omega| = n$, where n is the number of symbols on the right hand side of **R**.

Lemma 2: For any node, $\beta_1\beta_2...\beta_i\beta_{i+1}...\beta_n$, of a permutation tree, where $\beta_j \in \mathbf{B}$ when $1 \leqslant j \leqslant i$ and $\beta_j \in \mathbf{B}'$ when $i+1 \leqslant j \leqslant n$, each β_j is *unquoted unique*. That is, $f(\beta_j) = f(\beta_k)$ *if and only if* $j = k$, where the function f: $\mathbf{B} \cup \mathbf{B}' \rightarrow \mathbf{B}$ is defined as follows.

(5.19) $f(x') = x$ if $x' \in \mathbf{B}'$
$f(x) = x$ otherwise.

Proof: From our assumption that the ID rule is *nonambiguous*, the unquoted uniqueness property holds for the root of the permutation tree. By straight forward induction on the level of the node, it can be shown to be true for all nodes of the tree.

Theorem 1: The unrestricted gapping grammar $\mathbf{G_R} = (V_N, V_T, \Gamma, \Sigma, P)$ associated with **R**, where $V_N = \mathbf{B}' \cup \{nt\}$, $V_T = \mathbf{B}$, $\Gamma = \{\text{gap}(G)\}$, $S = \{nt\}$, and $P = \mathbf{P_R}$ is not ambiguous.

Proof: Assume that there are two leftmost derivations for some sentential form. Then, $\psi_1 ==> \omega$ and $\psi_2 ==> \omega$, where $\psi_1 \neq \psi_2$. If $\psi_1 = nt$, then ω must equal $\beta'_1\beta'_2...\beta'_n$, which would in turn imply $\psi_2 = nt = \psi_1$, resulting in a contradiction. Similarly, a contradiction also arises if $\psi_2 = nt$. For the other

cases, let the level of ω be i. From corollary 1 and from the definition of level, we know that both ψ_1 and ψ_2 are of level i-1. From lemma 1, let $\psi_1 = \upsilon_1\omega_1\beta'_i\omega'$ and $\psi_2 = \upsilon_2\omega_2\beta'_i\omega'$ where $\upsilon_1,\upsilon_2,\omega_1,\omega_2 \in B^*$, and $\omega' \in B'^*$. Application of (5.17) to ψ_1 and ψ_2 yields

(5.20) $\omega = \upsilon_1\beta_i\omega_1\omega' = \upsilon_2\beta_i\omega_2\omega'$

Since each β_i is unquoted unique (lemma 2), this implies $\upsilon_1 = \upsilon_2$ and $\omega_1 = \omega_2$. Consequently, $\psi_1 = \psi_2$, which contradicts our assumption. Therefore $\mathbf{G_R}$ is not ambiguous.

Lemma 3: When no LP relations are present, the jth level of the permutation tree will contain $j!$ nodes, where $0 \leq j \leq n$.

Proof: The root of the tree, is the only node of level zero by definition. Assume that there are $i!$ nodes on level i. From lemma 1, each of these nodes will resemble $\upsilon_i\omega_i\beta'_{i+1}\omega'$, where $\upsilon_i,\omega_i \in B^*$, $\omega' \in B'^*$, and $|\upsilon_i\omega_i| = i$. Application of rule (5.17) to this sentential form will result in i+1 descendants corresponding to the i+1 different possibilities for the contents of the gap, ω_i. Since each node of level i will have i+1 descendants, this implies that there will be $(i+1)!$ nodes on level i+1. By induction, any level j of the tree contains $j!$ nodes.

Lemma 4: When no LP relations are present, $nt \Rightarrow_{\mathbf{G_R}}^* \omega$ *if and only if* the base rule nt --> ω is in $\mathbf{R_R}$, where $\omega \in \mathbf{B}^*$.

Proof: When no LP relations are present, there are $n!$ unique context-free rules in $\mathbf{R_R}$ corresponding to the $n!$ different permutations of the n symbols, **B**, on the right hand side of the ID rule, **R**. From lemma 2 and corollary 2, we can conclude that the terminal nodes of the permutation tree are also permutations of the elements of **B**. Also, from lemma 1, they will be on

level n of the permutation tree. Since there is a unique leftmost derivation (theorem 1) for each terminal node, all $n!$ terminal nodes (lemma 3) will be distinct permutations of the elements of **B**. So if $nt \dashrightarrow \omega$ is in $\mathbf{R_R}$, the derivation

(5.21) $nt \Longrightarrow \beta'_1\beta'_2...\beta'_n \Longrightarrow^* \omega$

will be allowed in $\mathbf{G_R}$. If $nt \underset{\mathbf{G_R}}{\Longrightarrow}^* \omega$, then since ω is a permutation of the symbols of **B**, it will correspond to the valid base rule $nt \dashrightarrow \omega$.

Lemma 5: The presence of the *no_LPs* predicate in (5.17) disallows those derivations, and only those derivations according to $\mathbf{G_R}$ that would result in a sentential form that is not allowed by the LP relations. (If a symbol, β_i, precedes another symbol, β_j, in a sentential form, the sentential form violates an LP relation if the restriction $f(\beta_j) < f(\beta_i)$ is present. The function f is defined in (5.19)).

Proof: The initial sentential form of the permutation tree does not violate any LP restrictions. Given a sentential form, $\upsilon\omega\beta'_i\omega'$, that does not violate any LP restrictions, assume that the application of (5.17) results in $\upsilon\beta_i\omega\omega'$, which does violate an LP restriction, where $\upsilon,\omega \in B^*$, $\omega' \in B'^*$. But the only symbols whose relative positions have changed are the β_i and the symbols in ω. However, the procedural restriction of (5.17) prevents these symbols from violating the LP restrictions. Therefore $\upsilon\beta_i\omega\omega'$ does not violate any restrictions. From this contradiction, we know that rule application can not violate any LP relations. So the presence of an LP relation disallows invalid sentential forms.

Now, let $\upsilon\omega_1\beta_k\omega_2\beta'_i\omega'$, be a valid sentential form, and let $\beta_k < \beta_i$ be an LP relation ($\upsilon,\omega_1\omega_2 \in B^*$, and $\omega' \in B'^*$). Application of (5.17) could not result

in $v\beta_i\omega_1\beta_k\omega_2\omega'$ since this would violate the LP relation. A sentential form which does not violate this LP restriction not is derivable from $v\beta_i\omega_1\beta_k\omega_2\omega'$ since it would require shifting β_k to the left. However, β_k can not be moved by (5.17) since it is not quoted.

Theorem 2: There is a derivation $nt \Rightarrow_{G_R}^{*} \omega$ *if and only if* the base rule nt --> ω is allowed by the ID rule **R** and the LP relations, where $\omega \in \mathbf{B}^{*}$.

Proof: This result follows directly from lemma 4 and lemma 5.

To adapt the ID/LP conversion procedure for use with FIGG, (5.17) is replaced by the following bottom-up FIGG rule.

(5.22) gap(G), X' --> X, {unquoted(X)}, !, gap(G), {no_LPs(G,X)}.

The reverse quote, ', is used in place of the quote, ', due to restrictions associated with the implementation. Since variables are allowed in bottom-up FIGG rules, we do not need a gapping rule for each symbol, *X*. The *cut*, !, is used to obtain the leftmost derivation, while the predicate *unquoted* is required to prevent a cycle during bottom-up parsing.

Until now, only the processing of single ID rules has been examined. To process all the ID rules according to the ID/LP-UGG conversion procedure we must combine the productions from each grammar, $\mathbf{G}_{\mathbf{R}_i}$, that corresponds to an ID rule, $\mathbf{R}_i$. The $\mathbf{G}_{\mathbf{R}_i}$'s will be called *subgrammars*. Also, we must restrict (5.17) to shift a symbol only over a region that corresponds to the sentential form from the permutation tree of the subgrammar. This *scope restriction* prevents interference with the *sentential forms* of other subgrammars. We will also include an *independence restriction* stating that once a production of a subgrammar is applied to a sentential form, only productions from that subgrammar can be used unless all symbols of the sentential form are

symbols from the ID/LP grammar. In other words, the permutation of the symbols of the right hand side of an ID rule must be completed before the processing of the next ID rule is started.

For a base grammar $\mathbf{G_B} = (V_T, V_N, \Sigma, P)$, we can provide an equivalent unrestricted gapping grammar $\mathbf{G'} = (V_T, V'_N, \Gamma, \Sigma, P')$, obtained from the ID/LP description specification according to the ID/LP-UGG conversion procedure. Let $P' = \cup_i \mathbf{P}_{\mathbf{R}_i}$, and $V'_N = V_N \cup \{x' | x \in V_N\}$. The means for including the scope restriction and independence restriction will be presented after the following theorems.

Theorem 3: Given an ID/LP specification which corresponds to a base grammar, $\mathbf{G_B}$, and an unrestricted gapping grammar, $\mathbf{G'}$, obtained according to the ID/LP-UGG conversion procedure, then for any string $\omega \in (V_N \cup V_T)^*$ $\sigma \overset{*}{\Rightarrow}_{\mathbf{G_B}} \omega$ *if and only if* $\sigma \overset{*}{\Rightarrow}_{\mathbf{G'}} \omega$, where $\sigma \in \Sigma$.

Proof: Given a sequence of m rule applications which derive ω using $\mathbf{G_B}$.

(5.23) $\sigma \overset{\mathbf{R}_1}{\Rightarrow} \omega_1 \overset{\mathbf{R}_2}{\Rightarrow} \ldots \overset{\mathbf{R}_m}{\Rightarrow} \omega$

where $\omega_i \in (V'_N \cup V_T)^*$ and $\mathbf{R}_i \in P$, the following derivation is possible in $\mathbf{G'}$ using the subgrammars, $\mathbf{G}_{\mathbf{R}_i}$ which correspond to each rule application in (5.23) (theorem 2).

(5.24) $\sigma \overset{*}{\Rightarrow}_{\mathbf{G}_{\mathbf{R}_1}} \omega_1 \overset{*}{\Rightarrow}_{\mathbf{G}_{\mathbf{R}_2}} \ldots \overset{*}{\Rightarrow}_{\mathbf{G}_{\mathbf{R}_m}} \omega$

Given a derivation according to $\mathbf{G'}$ that derives ω, the independence restriction requires any such derivation to resemble (5.24). For $\omega_i = v_1 x v_2$, where $v_1, v_2 \in (V_N \cup V_T)^*$ and $x \in V_N \cup V_T$, the scope restriction requires $\omega_{i+1} = v_1 \chi v_2$, where $\chi \in (V_N \cup V_T)^*$. According to theorem 2, the rule $x \dashrightarrow \chi$ is in the base grammar. From this, we can conclude that the derivation (5.23) is allowed by $\mathbf{G_B}$.

Theorem 4: **G** is ambiguous *if and only if* **G'** is ambiguous.

Proof: The *only if* case is trivial. To prove the *if* case, let us re-examine (5.24). Theorem 1 states that each $\mathbf{G}_{\mathbf{R}_i}$ is nonambiguous. Therefore, if **G'** is ambiguous then it must be due to two derivations with a different order of subgrammar application.

(5.25) $\sigma \overset{*}{\Rightarrow}_{\mathbf{G}_{\mathbf{R}_1}} \omega_1 \overset{*}{\Rightarrow}_{\mathbf{G}_{\mathbf{R}_2}} \ldots \overset{*}{\Rightarrow}_{\mathbf{G}_{\mathbf{R}_m}} \omega$

$\sigma \overset{*}{\Rightarrow}_{\mathbf{G}_{\mathbf{R}'_1}} \omega'_1 \overset{*}{\Rightarrow}_{\mathbf{G}_{\mathbf{R}'_2}} \ldots \overset{*}{\Rightarrow}_{\mathbf{G}_{\mathbf{R}'_m}} \omega$

This would imply that the following two derivations would be allowed by $\mathbf{G}_{\mathbf{B}}$.

(5.26) $\sigma \Rightarrow_{\mathbf{R}_1} \omega_1 \Rightarrow_{\mathbf{R}_2} \ldots \Rightarrow_{\mathbf{R}_m} \omega$

$\sigma \Rightarrow_{\mathbf{R}'_1} \omega_1 \Rightarrow_{\mathbf{R}'_2} \ldots \Rightarrow_{\mathbf{R}'_m} \omega$

To incorporate the scope restriction we can introduce nonterminal symbols (markers), *mk*, to delimit the symbols on the right hand side of ID rule. The addition of the linear precedence relation *mk* < *X* for all symbols *X* will prevent any symbols from being repositioned outside of these delimiters by (5.22). To ensure that the processing of one base rule is completed before another base rule is attempted, (5.22) can be modified as follows.

(5.27) , gap(G), !, X' --> X'', !, gap(G), {no_LPs(G,X)}.

Now, after a quoted symbol has been moved it will be double quoted, and it will not be allowed to move again. To complete the processing of the base rule, the symbols, which must all be double quoted, that appear between the markers will have the double quotes removed by the following rule.

(5.28) mk, gap(-,G1), mk --> gap(-,G), {double_quote(G,G1)} .

This rule will also remove the markers. Consider the following ID/LP grammar.

(5.29)
```
(a)  s --> np  vp
(b)  vp --> v  pp(with)  pp(in)
(c)  np --> ['John']
(d)  np --> ['Mary']
(e)  np --> ['London']
(f)  v --> [lives]
(g)  pp(with) --> [with]  np
(h)  pp(in)  --> [in]  np
(i)  np < vp.
(j)  v < pp(_).
(k)  [Word] < X.  /* for any category X */
```

This grammar can be translated into the following unrestricted gapping grammar, G_3.

(5.30)
```
(a)  r(1):  s  --> mk, np', vp', mk, {id_rule}.
(b)  r(2):  vp --> mk, v', pp(with)', pp(in)', mk, {id_rule}.
(c)  r(3):  np --> mk, ['John']', mk, {id_rule}.
(d)  r(4):  np --> mk, ['Mary']', mk, {id_rule}.
(e)  r(5):  np --> mk, ['London']', mk, {id_rule}.
(f)  r(6):  v  --> mk, [lives]', mk, {id_rule}.
(g)  r(7):  pp(with) --> mk, [with]', np', mk, {id_rule}.
(h)  r(8):  pp(in) --> mk, [in]', np', mk, {id_rule}.
(i)  lp(np,vp).
(j)  lp(v,pp(_)).
(k)  lp([Word],X) :- notlist(X).
(l)  lp(mk,_).
(m)  gap(G), !, X' --> X'', !, gap(G), {no_LPs(G,X)}.
(n)  mk^r(_), gap(-,G1), mk^r(_)  -->  {inter_rule}, gap(-,G),
        {G\==[], double_quote(G,G1)}.
```

Since this grammar must be processed by a bottom-up parser, some additional control was added to prevent cycles and obtain more efficient parsing. The procedures *id_rule* and *inter_rule* ensure that (5.30n) can not be used again until a corresponding *id_rule* is used.

(5.31)
```
inter_rule :- not inrule, !, ( assert(inrule) ; abolish(inrule,0), fail ).
id_rule :- abolish(inrule,0) ; assert(inrule) , fail.
```

The dominators, ^r(*X*), of (5.30n) also improve parsing efficiency by requiring both markers to be symbols from the right hand side of a rule named r(*X*), for some *X*. All quotes are removed from the symbols before linear precedence relations are checked. For any two symbols, β_i and β_j, the clause $lp(\beta_i,\beta_j)$ is present if the relation $\beta_i<\beta_j$ holds.

For each ID rule one unrestricted GG rule is required, and for each LP relation a single clause is required. Note that the use of logic variables in LP relations allows LP schemata, representing classes of relations, to be specified as seen in (5.301). Only two extra rules, (5.30m) and (5.30n), and one new LP schema, (5.301), are required in the conversion process. So the conversion is linear with respect to the number of rules. This conversion also approximately triples the number of grammar symbols. The parse of the sentence *John lives in London with Mary* which uses G_3 is shown in Figure 5-2.

The parse was obtained in 3.3 seconds on a Motorola 68000 based SUN Workstation running C-Prolog in a UNIX environment. Total processing time, which includes time spent looking for other parses, was 103 seconds. No other parses were found. The *no_LPs* predicate was modified, to improve efficiency, so that it would *fail* if it found two symbols (between markers) that were not both from the same ID rule. As it stands, once the parser finds a gap that results in the violation of an LP relation, it still tries larger gaps to see if they might not violate this restriction. Therefore, the *gap* predicate used in (5.30m) can be modified to incorporate the *no_LPs* test. Also, ID rules that have a single symbol on the right hand side do not require any *permutation* of this lone symbol. With these modifications, the parse time was only 1.7 seconds, and the total processing time was reduced to 67 seconds.

The ProGram system (Evans and Gazdar, 1984) also uses ID/LP specifications without converting them into their corresponding context-free grammars. It preprocesses ID/LP grammars into a *normal form* which is then used for parsing. This system, however, was designed as a *grammar development system*, so efficiency was not a major consideration. Using a grammar similar to (5.29), the ProGram grammar development system (Evans and Gazdar, 1984) required 16 seconds to parse the same sentence with a total processing time of 49 seconds.

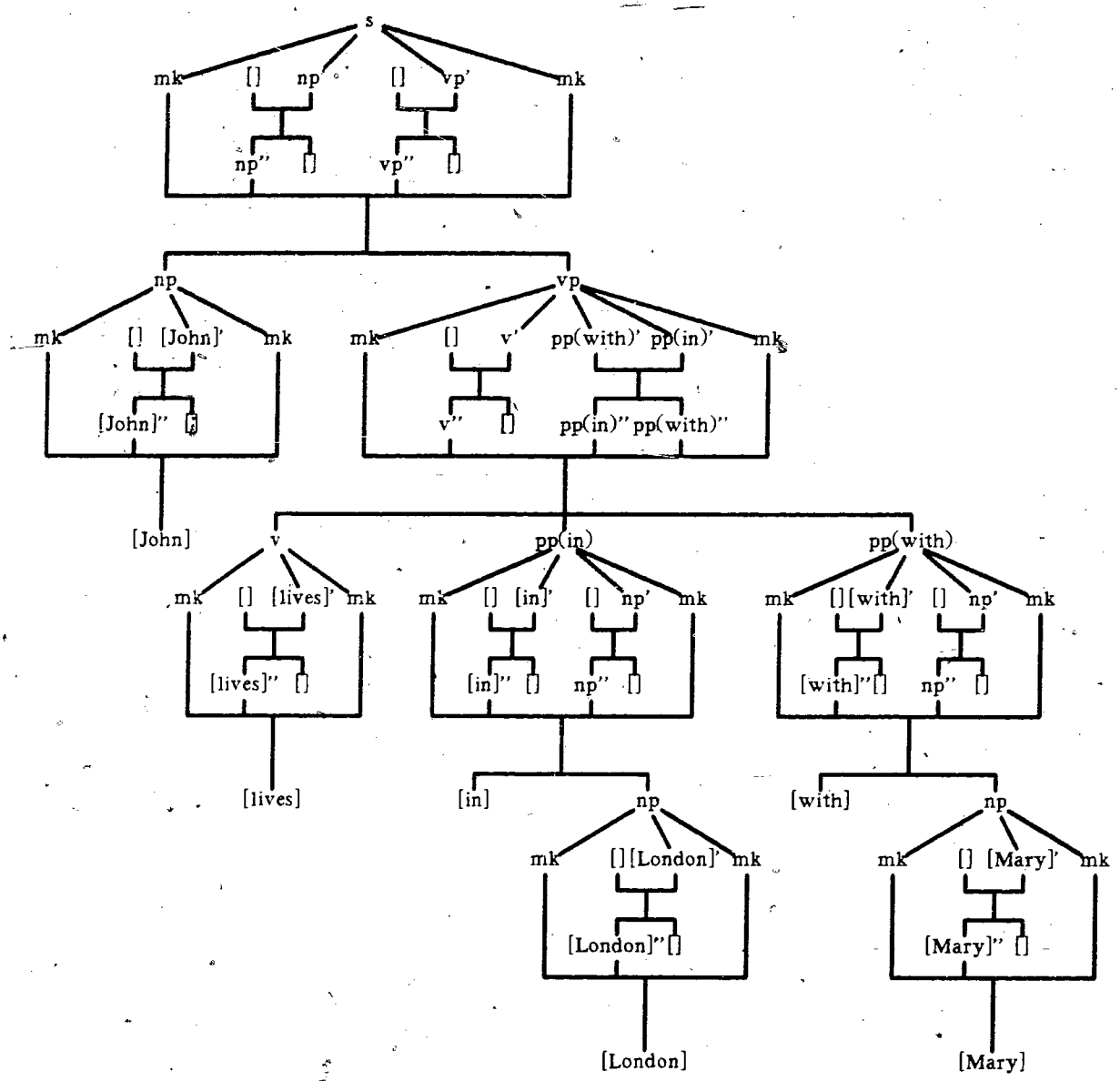

Figure 5-2: Parse using an unrestricted GG based on an ID/LP description

For a more thorough comparison of these two systems, consider the following ID/LP grammar which is adapted from (42) of (Gazdar and Pullum, 1982).

(5.32)

s --> np vp.	np < vp
vp --> v.	p < np
vp --> v np.	v < s
vp --> v s.	
vp --> v np pp.	n --> [n].
vp --> v np s.	v --> [v].
np --> n.	p --> [p].
pp --> p np.	

ProGram and FIGG were compared on the following collection of valid sentences of varying lengths.

(5.33)

(a) n v.	(g) n v n p n.	(m) n v n v n v.
(b) n v n.	(h) n v p n n.	(n) n v n n n v.
(c) n n v.	(i) n n v p n.	(o) n v n n v n.
(d) n v n v.	(j) n n p n v.	(p) n v n v n n.
(e) n v n v n.	(k) n p n v n.	(q) n n v n v n.
(f) n v n n v.	(l) n p n n v.	(r) n n v n n v.

Note that sentences (5.33e-f) and (5.33o) are ambiguous. The GPSG used by ProGram and the unrestricted gapping grammar used by FIGG are provided in Appendix B. Modification of (5.32) by increasing or decreasing the number of LP relations did not greatly affect the parse times of either system. Table 5-4 summarises the average time for finding the first parse, *1st*, and the average total processing time, *total*, for each system according to sentence length, *m*.

Table 5-4: Comparison of FIGG and ProGram using an ID/LP grammar.

		m=2	m=3	m=4	m=5	m=6
FIGG	1st	0.3	3.7	2.1	40.	470.
	total	1.9	14.	90.	330.	2700.
ProGram	1st	3.1	16.	12.	42.	200.
	total	12.	35.	110.	160.	980.

Firstly, one should notice that neither system is efficient at processing ID/LP grammars. They both exhibit exponential growth of processing time. Although the

FIGG system is markedly superior to ProGram in obtaining the first parse of shorter sentences, its growth of CPU time as a function of sentence length is greater than ProGram's. To illustrate why FIGG does not efficiently execute the unrestricted gapping grammar, consider the application of (5.30n) to some sentential form. The bottom-up parser will enclose arbitrary substrings within the markers, and it may take a great deal of processing before the system determines that the contents of gap does not correspond to the right hand side of some ID rule. When viewed from the top down, it is impossible to obtain a sentential form in which the contents of the markers does not correspond to a permutation of the right hand side of a rule. One may be tempted to introduce more procedural knowledge to rule out these sentential forms earlier, but addition of this large amount of *compiled* knowledge is contrary to our desire for an *interpreted* approach. A possible solution would require the development of a unrestricted gapping grammar parser that can parse the grammar produced by the ID/LP-UGG conversion procedure in a top-down manner. Currently, such a parser does not exist. Another solution might entail rewriting the shifting rule, (5.27), as

(5.34) X', gap(G), --> gap(G), X'', {no_LPs1(X,G)}.

for use by the current top-down parser. Instead of shifting symbols to the left, (5.34) would shift a symbol to the right as long as the LP relations were not violated, as determined by *no_LPs1*. Unfortunately, this would require a *rightmost* derivation, which the parser would be unable to obtain due to the *nested head problem* — which results from its depth-first parsing strategy.

5.3. Implementation of GPSG Metarules

Unrestricted gapping grammars may be used to describe the grammars and metagrammars of linguistic theories which possess the *gap* concept. This should not be confused with a claim that unrestricted gapping grammars constitute a linguistic theory. They merely provide the medium for expressing the theory. Although FIGG may not supply the most efficient implementation, it can be used to process generalised phrase structure grammars.

As was mentioned in chapter one, there are two approaches to metarule processing for use in a parser. The *compiled* approach uses the metarules to generate all possible context-free rules before the actual parsing begins. When using this method, care must be taken to ensure that the generation process terminates (Shieber et. al., 1983). The *interpreted* approach avoids generating all possible rules by using the metarule during parsing. However, the use of metarules in this manner may result in a grammar with more than context-free power (Shieber et. al., 1983) (Gazdar and Pullum, 1982). Unrestricted gapping grammars can be used to describe an *interpreted* approach for processing metarules that operate on context-free rules.[8]

The notation used in the GPSG references differs from the conventions used in this paper. The context-free rules of GPSG shall be presented here in the familiar notation

(5.35) $nt \dashrightarrow \beta_1, \beta_2, \ldots, \beta_n$

where nt is a nonterminal, and β_i is a terminal or nonterminal for $1 \leq i \leq n$.

---

[8] According to the description of generalised phrase structure grammars provided in (Gazdar, 1981), the metarules of the grammar operate on the context-free rules of the grammar to produce more context-free rules. The revised theory (Gazdar and Pullum, 1982) uses an ID/LP grammar to describe the context-free rule base, with the metarules operating on the ID relations to produce new ID rules.

Regular expressions, optional components, and other such abbreviatory devices which may appear in the right hand side of GPSG rules will be ignored in this discussion. Also, features and semantics will be ignored for the moment, with an outline of how they might be incorporated presented later. Metarules are of the form

(5.36) $nt \text{ --> } \beta_1, \beta_2, \ldots, \beta_m \text{ ==> } nt' \text{ --> } \beta_1', \beta_2', \ldots, \beta_n'$

where a context-free rule on the left side of the metarule arrow, ==>, is called the pattern, and the rule on the right hand side is the template. In (5.36), any β_i $(1 \leqslant i \leqslant m)$ or β_j' $(1 \leqslant j \leqslant n)$ can be a gap symbol (string variable). A gap contained in the pattern can correspond to a region of zero or more symbols in a rule that matches the pattern. The metarule may also contain variables. A variable that appears in the pattern may match any symbol. If a gap or a variable appears on one side of the metarule, it must also appear on the other side.

In order to describe how the metarules can be translated into unrestricted GGs, let us examine the following metarule.

(5.37) $nt \text{ --> } \beta \text{ ==> } nt_1 \text{ --> } \beta_1$

In (5.37), nt and nt_1 can be nonterminal symbols or variables. β and β_1 may contain terminals, nonterminals, variables, and gaps. Now consider the top down processing of the *virtual rule*

(5.38) $nt_1' \text{ --> } \beta_1'$

that results from the successful application of (5.37) to some *base rule*, $\mathbf{R}_b$.[9]

(5.39) $nt' \text{ --> } \beta'$

To obtain a derivation equivalent to the application of the virtual rule, we start with nt_1' and apply the context-free rule, $\mathbf{R}_{b_1}$,

---

[9]Notice that if neither nt nor nt_1 are variables, then $nt = nt'$ and $nt_1 = nt_1'$. Similarly, if no gaps or variables are present in β and β_1, then $\beta = \beta'$ and $\beta_1 = \beta_1'$.

(5.40) $nt_1 \text{ --> } <_1\ nt\ >_1$

which is derived from the metarule, (5.37). The nonterminals "$<_1$" and "$>_1$" of (5.40) are *scope markers* unique to this derivation. After application of (5.40) to nt'_1, $\mathbf{R}_b$ can be applied to nt' to obtain β'. If β' matches β in the pattern of the metarule, then we are allowed to replace it with the sequence described by β_1 in the template. To achieve this operation, the following unrestricted gapping rule, $\mathbf{R}_{r_1}$, is introduced.

(5.41) $<_1\ \beta\ >_1 \text{ --> } \beta_1$

This rule also removes the scope markers which ensure that β matches all of β'. The sentential form that results from the application of (5.41) will correspond to β'_1, which is the right hand side of the virtual rule. Procedural control must be added to ensure that the contents of scope markers, before the application of (5.41), is derived from the application of a single rule to the contents of these markers after the application of (5.40). If this procedural control were not included, the method would process metarules of the form

(5.42) $nt \text{ -->}^* \beta \text{ ==> } nt_1 \text{ -->}^* \beta_1$

This entire process is illustrated in Figure 5-3. The nodes of this graph are sentential forms. The dashed arrow corresponds to the application of the virtual rule, while the solid arrows represent the path actually taken to process the rule.

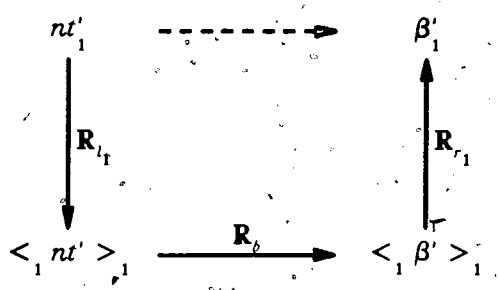

Figure 5-3: Processing of GPSG Metarules

This method can also be used to handle multiple application of metarules. Figure

5-4 illustrates the use of n, not necessarily distinct, metarules. Instead of having a base rule operate within the scope markers, $<_n$ and $>_n$, a virtual rule is used instead.

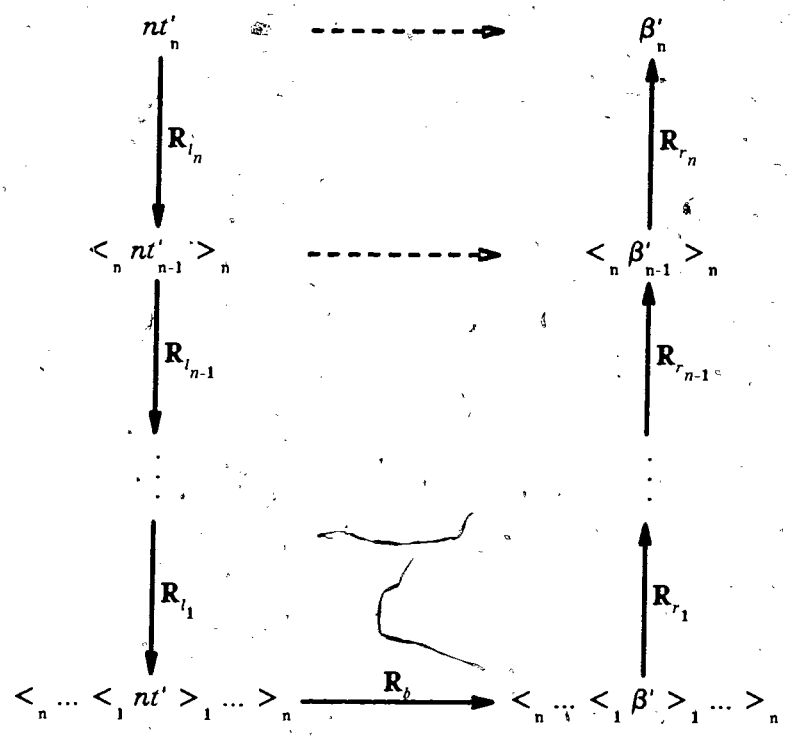

Figure 5-4: Processing of Multiple GPSG Metarules

A consequence of this method of metarule processing is that the resulting language is not necessarily context-free. Restrictions like finite closure (Thompson, 1982), could be imposed upon this process, by the addition of some procedures, to prevent this side-effect. Under finite closure, each metarule can only be applied one in the derivation of a rule.

To illustrate the use of this process, consider the following base grammar.

(5.43)
```
s:   s(decl) --> np(subj), vp(Voice).
np1: np(Case) --> det, noun(Case,-pn).
np2: np(Case) --> noun(Case,+pn).
vp:  vp(active) --> v(trans,active), np(obj), pp(to).
pp1: pp(by) --> [by], np(obj).
pp2: pp(to) --> [to], np(obj).
```

To permit passive and inverted sentences, the following two metarules can be used.

(5.44) (a) vp_{active} --> v_{trans} np_{obj} X ==> $vp_{passive}$ --> aux $v_{pastpart}$ X pp_{by}

(b) vp --> aux X ==> s_{inv} --> aux np_{subj} X

These metarules can be incorporated into the FIGG specification for the grammar by adding the following four rules.

(5.45) (a)
```
vp(passive) —> mk(X), vp(active), mk(X).

{numgen(X)}, mk(X), v(trans,active)^D, np(obj)^D, gap(G)^D, mk(X)
—> aux(be), v(trans,pstprt), gap(G), pp(by).
```

(b)
```
s(inv)  —> mk(X), vp(Voice), mk(X).

{numgen(X)}, mk(X), aux(Type)^D, gap(G)^D, mk(X)
—> aux(Type), np(subj), gap(G).
```

numgen(X) generates a unique number associated with each application of the rule. The scope markers, represented as *mk(X)*, use this unique number to allow multiple applications of the same rule. Notice that the dominators, ^D, in (5.45) will provide the necessary control on the contents of scope markers to ensure that only one rule application has been done. Figure 5-5 illustrates the derivation required to parse the sentence *Is the ball given to John by Mary*. The addition of extra arguments to the grammar symbols and markers can result in the construction of the actual context-free parse tree. This parse was obtained in 1.1 seconds. But the total processing time of 43 seconds is not at all encouraging considering the small size of the grammar. Once again, the system can be made more efficient with procedural control.

First notice that the use of markers to ensure the application of a single base rule is not foolproof. This is illustrated in the contrived grammar in (5.46).

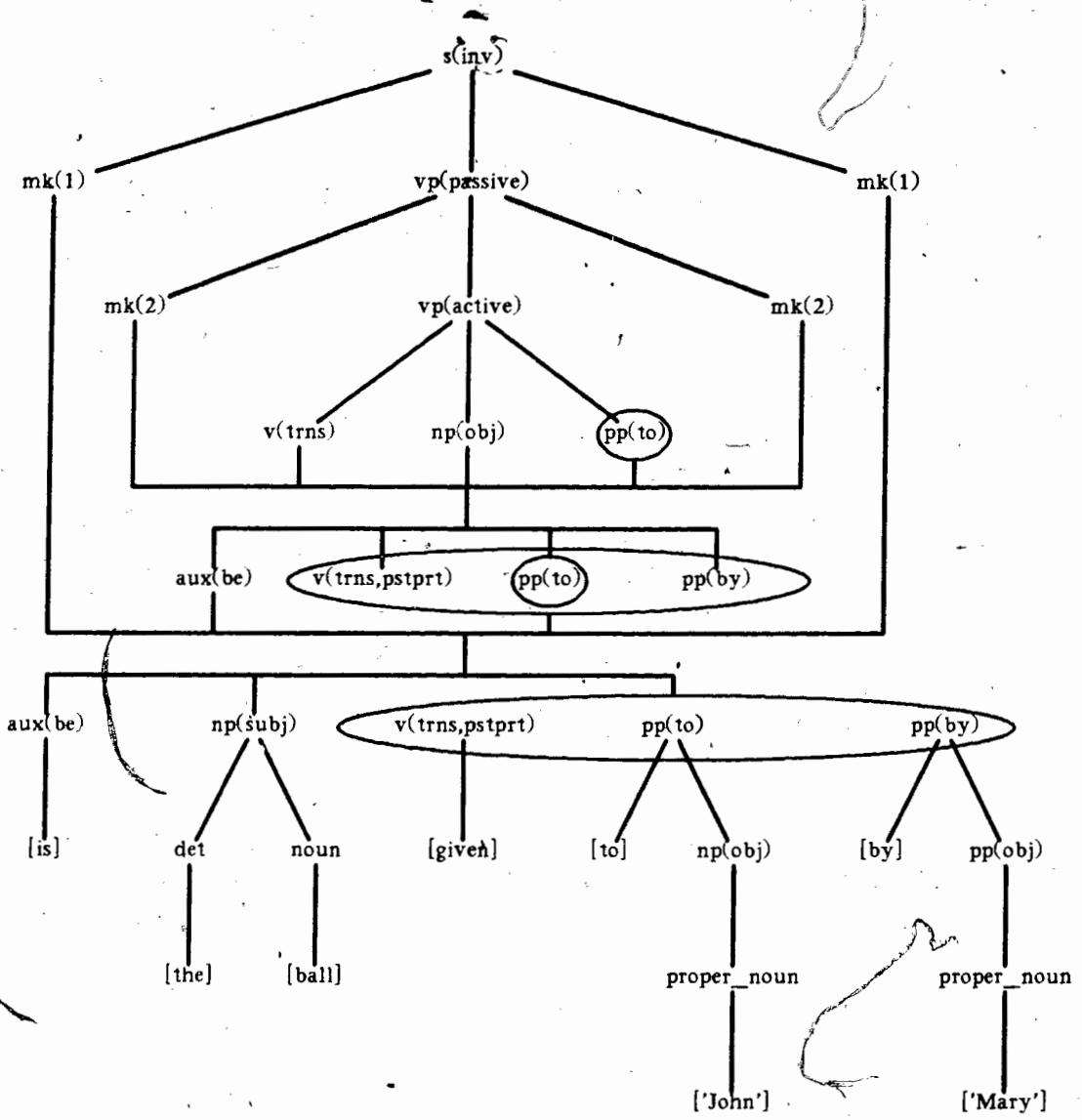

Figure 5-5: Derivation graph using an interpreted GPSG metarule

(5.46) (a) s --> a.
(b) a --> b.
(c) b --> [c].
(d) s --> b ==> s --> [d].

The metarule, (5.46d), should not generate any new context-free rules, but the unrestricted GG (5.47) which corresponds to this grammar will recognise *d* as a valid sentence.

(5.47)
```
(a)  s --> a.
(b)  a --> b.
(c)  b --> [c].
(d)  s --> mk(X), s, mk(X).
(e)  {numgen(X)}, mk(X), b^Dom, mk(X) --> [d].
```

Also, notice that once we process one of the $\mathbf{R}_{r_i}$ by the bottom-up parser, the diagram from Figure 5-4 illustrates that the next rule must be $\mathbf{R}_{r_{i-1}}$ or a base rule. Then, after a single base rule has been processed, the next n rules must be one of the $\mathbf{R}_{l_i}$, where n corresponds to the number of $\mathbf{R}_{r_i}$ used. This restriction can be incorporated by introducing the procedure *lhs* into all the $\mathbf{R}_{l_i}$'s, *rhs* into the $\mathbf{R}_{l_i}$'s, and *rule* into all the base rules. These predicates, which are defined in (5.48), maintain a stack for the number of *rhs*'s and *lhs*'s executed.

(5.48)
```
virtual([],[]).
rhs  :- virtual([],_), !, (virtual_push(right) ; virtual_pop(right), fail).
lhs  :- virtual([_|_],[]), !, (virtual_pop(left) ; virtual_push(left), fail).
rule :- virtual([],[]), !.
rule :- virtual([],_), !, (virtual_swap(left)  ; virtual_swap(right), fail).
```

The predicates *virtual_push*, *virtual_pop*, and *virtual_swap* modify the arguments of the *virtual(Left,Right)* predicate that is stored in the database. *Right* is a list whose length corresponds to the number of times *rhs* has been executed. Execution of *rule* moves this list into *Left*. Subsequent execution of *lhs* will decrement the number stored in *Left*. Notice that the definitions in (5.48) provide the required restrictions on the order of execution of all $\mathbf{R}_{r_i}$'s, $\mathbf{R}_{l_i}$'s, and base rules.

Until now, we have used *essential* gaps within our GPSG metarule translations. Consequently, the parser has allowed the gap to contain any nonterminals or terminals. This is actually too general for our needs. The formal definition of metarules used by Thompson in his GPSG metarule parser (Thompson, 1982) includes a *range* for his *string variables* (which correspond to our gaps). This range can be specified with the use of the *restricted* gap predicate. The gapping rules of (5.45) can be replaced by the following.

(5.49) (a)
```
{numgen(X)}, mk(X), v(trans,active)^D, np(obj)^D, gap(G)^D, mk(X)
    --> aux(be), v(trans,pstprt), gap([np(_),pp(_),vpinf],G), pp(by).
```

(b)
```
{numgen(X)}, mk(X), aux(Type)^D, gap(G)^D, mk(X)
    --> aux(Type), np(subj), gap([vp(_),v(_,_),np(_),pp(_),vpinf],G).
```

In (5.49a), the gap may contain *np*'s, *pp*'s, and *vpinf*'s. Since the sentence inversion metarule may be applied to a rule created by the passivisation metarule, its gap may contain any the previously mentioned categories in addition to *vp*'s and *v*'s.

The efficiency can also be improved by restructuring the grammar. It is possible to construct the grammar so that rules created by the passivisation metarule do not have to be operated on by the sentence inversion metarule. The two metarules of (5.44) can be restated as follows.

(5.50) (a) vp_{active} --> v_{trans} np_{obj} X ==> $vp_{pastpart}$ --> $v_{pastpart}$ X pp_{by}

(b) vp --> aux vp_1 ==> s_{inv} --> aux s_1

The verb phrase described by the template of (5.50a) must be used in conjunction with the rule

(5.51) $vp_{passive}$ --> aux_{be} $vp_{pastpart}$

to obtain passive verb phrases. (5.50b) generates inverted sentences by changing vp_1 into a sentence, s_1, and transferring all of the verb phrase features to this new symbol.

With these procedural control mechanisms and grammar modifications, a larger grammar was developed to examine the behaviour of FIGG. This grammar, which appears in Appendix C, uses a *foot* argument on the nonterminal symbols to capture the slash categories of GPSG. It can describe a variety of active and passive sentences, including inverted sentences, questions, topicalised sentences, and sentences containing relative clauses. Since ϵ-productions (eg. *nt* --> ϵ) cannot be processed by the bottom-up parser, the grammar was modified to remove these rules by the

addition of extra rules. Although this removal was done manually, it can be done automatically by standard ϵ-production removal techniques (Hopcroft and Ullman, 1979). This system was tested on a series of eight sentences, shown below.

(5.52) (a) John takes cmpt101
(b) does John take cmpt101
(c) John wants to take cmpt101
(d) does John want to take cmpt101
(e) John is loved by Mary
(f) does John want to be loved by Mary
(g) is the ball given to John by Mary
(h) does John want to see the house fly

The parse of sentences (5.52b,d,e,h) requires one virtual rule, while the parse of (5.52f,g) requires two virtual rules. All of the sentences are unambiguous, except (5.52h) which has two different readings. The parse times obtained with FIGG are shown in Table 5-5. They are compared with those obtained by ProGram with a similar grammar, which is also shown in Appendix C, and with the results from SAUMER which uses the grammar of the Automated Academic Advisor (Popowich, 1985d). Once again, for a given sentence and system, the times provided correspond to the time for the first parse and the total processing time (in seconds). Also, recall that the ProGram system uses ID/LP grammars, which contributes to its inefficient parsing. Words that have multiple lexical entries, such as *fly*, also lead to inefficient parsing by the ProGram parser. For this reason, the inverted auxiliary *is* was not included in its lexicon, which explains why the system could not parse (5.52g).

The results from this table show a very disappointing performance by FIGG on this larger grammar. Part of the reason for this performance is the shear inefficiency of the parser. $FIGG_b$ in Table 5-5 summarises the performance of FIGG using the same grammar *without* the metarules. When the metarules are added, the efficiency drop is considerable, but it is not as bad as suggested by the examination of the *grammar with metarule* results alone. Also, the procedural control predicates *lhs*, *rhs*, and *rule*

Table 5-5: Comparison of FIGG, SAUMER, and ProGram using GPSG

	(a)	(b)	(c)	(d)	(e)	(f)	(g)	(h)
FIGG	0.5	1.4	2.4	7.0	13.	200.	23.	340.
	4.0	13.	40.	130.	87.	1500.	1000.	2200.
SAUMER	1.0	3.6	2.0	5.4	4.4	9.7	12.	12.
	6.5	6.0	11.	10.	11.	15.	23.	27.
ProGram	22.	29.	36.	44.	38.	61.	—	200.
	100.	150.	420.	500.	900.	3700.	8400.	5900.
$FIGG_b$	0.5	—	2.3	—	—	—	—	—
	3.7	8.9	38.	89.	6.1	110.	110.	1500.

result in an extra overhead associated with each rule application. The restrictions on the gaps improve the performance dramatically however. Without these restrictions, the amount of time required to parse even a very short sentence would be prohibitive. Another factor that was not a problem with our grammar but could be in other applications is ambiguity introduced into the grammar by the translation of the metagrammar into an unrestricted GG. If a *derived rule* can be generated two ways by the metagrammar, FIGG will find a parse for each way the *virtual* rule can be processed. More study of the introduction of ambiguity by the translation process is required.

This discussion has concentrated on the translation of the rules and metarules into unrestricted GGs but has neglected many of the other components of GPSG grammars. Information, such as values for features or semantic interpretations, can be passed from one side of a virtual rule to the other by means of extra arguments on the nonterminal symbols and markers. For example, if the context-free parse tree is desired for a derivation, it can be contained in an extra argument of the grammar symbols. In the following rule, the first argument of a symbol is a list that

describes the tree which possesses the symbol as its root.[10]

(5.53) vp([vp(active),V,NP,PP],active) --> v(V,trans,active), np(NP,obj), pp(PP,to).

Now, the two rules associated with the passive metarule (5.50a) could be of the following form.

(5.54) (a) `vp([vp(pstprt)|List],pstprt) --> mk(X,List), vp(_,active), mk(X).`

(b)
```
{numgen(X)}, mk(X,List), v(_,trans,active)^D, np(_,obj)^D,
   gap(_,+,G)^D, mk(X)
--> v(V,trans,pstprt), gap(Tree,+,G), pp(PP,by),
   {join([V,Tree,PP],List)}.
```

The information is passed from the right hand side of the virtual rule to the left by means of the *List* argument of the first marker. To capture the *structured categories* of GPSGs, an approach like that taken in ProGram could be used. The grammar symbols could be replaced by the structures similar to

(5.55) node([cat,[bar,2],[head,[major,+n,-v]]])

This structure corresponds to a noun phrase in conventional GPSG grammars. Other features could be included in this single list, or added into extra arguments of the *node*. Various procedures would have to be supplied to manipulate the features, assign defaults, and rule out various feature combinations.

Although the method provided for converting GPSG grammars into unrestricted GGs is very straightforward, the resulting grammars can not currently be used efficiently by FIGG. A top-down parser may provide more efficient processing of these grammars, but the current top-down parser can not process these grammars due to the *nested head problem* — which was discussed in chapter four. Upon examination of the derivation tree in Figure 5-5, one notices that the *v(trns,pstprt)*, *pp(to)*, and *pp(by)* are *nested* in a gap and are each a *head* symbol of a rule.

---

[10] [s,[np,[n,John]],[vp,[v,runs]]] denotes a parse tree for *John runs*.

5.4. Summary

Unrestricted gapping grammars provide more concise grammatical descriptions than previous logic grammar formalisms for many languages due to the more general rule format allowed. However, with such a general rule format, caution must be taken to ensure that the grammar is restricted, by some form of control, to describe the required language. Control facilities provided in FIGG permit refined control mechanisms without detracting from the high degree of descriptiveness present 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 bottom-up parser of FIGG can process grammars obtained by ID/LP-UGG conversion procedure, and can also process those grammars obtained by the translation of GPSG-like rules and metarules into unrestricted GGs. Although this parser is far from efficient, it can process these grammars while the current top-down parser can not. The ID/LP-UGG conversion procedure is a *simple* (linear) method for translating ID/LP relations into a directly executable logic grammar which does not introduce ambiguity into the grammar. Unlike the approaches taken in (Shieber, 1982, Evans and Gazdar, 1984), the ID/LP-UGG conversion procedure is not an algorithm for direct processing of ID/LP grammars, but rather a method for converting these grammars into another grammar that can be directly executed. Consequently, the efficiency of ID/LP grammar processing is affected by the efficiency of the unrestricted gapping grammar interpreter, (as well as by the structure of the grammar). Similarly, the simple procedure for translating GPSGs into unrestricted gapping grammars is not in itself an algorithm for direct processing of metarules. While this approach to metarule processing is not feasible to use with the current implementation, future parsers of unrestricted gapping grammars should produce more reasonable results. The development of a less restrictive top-down parser would remove the need for much of

the procedural control that was required for the bottom-up parsing of grammars produced by the ID/LP-UGG conversion procedure, and could lead to more efficient *interpreted* processing of GPSG metarules, and ID/LP grammars.

Chapter 6
Conclusions

Unrestricted gapping grammars, an extension of the gapping grammar formalism, promote high-level descriptions for grammatical specifications of languages. They include grammars such as XGs, MGs, and the conventional grammars of the Chomsky hierarchy. Unrestricted GGs facilitate easier description of unbounded left symbol movement subject to constraints inserted directly into the grammar rule. This type of movement is useful to describe ID/LP grammars, which permit relocation of constituents subject to constraints.[11] In this thesis, a simple procedure for conversion of an ID/LP grammar into a unrestricted GG has been introduced. GPSG metarules, which can contain gap symbols, can be expressed in terms of a gapping grammar as illustrated in this work. But the conversion of a complete GPSG into an equivalent unrestricted GG requires further study. Other linguistic theories which use the gap concept — such as transformational grammars — might also be described in terms of GGs or unrestricted GGs. (Saint-Dizier, forthcoming) is currently examining the use of unrestricted GG for the description of some forms of movement allowed in transformational grammars. The FIGG implementation of unrestricted GGs should prove to be a useful tool for exploring these relationships.

Results obtained with FIGG illustrate that procedural control can successfully be

[11] Extraposition does not adequately describe this movement since it suggests the unbounded movement of a single constituent.

used to restrict the language described by the grammar, and to obtain more efficient rule processing. However, the definition of unrestricted GGs does not predefine a set of control mechanisms. FIGG introduces a selection of mechanisms to control the parse, but these features are implementation dependent. Some of the control operators, such as the cut, behave differently under the different parsers, while other forms of control, such as gap control, behave essentially the same. The control mechanisms supplied in FIGG are sufficient for the applications presented in this work, and will probably be useful for future applications. Greater control on rule selection is one feature that could be beneficial in many applications. This could eliminate the need for the procedures added during the ID/LP-UGG conversion procedure (*inter_rule* and *id_rule*), and for those required by the FIGG representation of a GPSG (*rhs*, *lhs* and *rule*). Rules could be divided into *classes* according to their names. Then a *metagrammar* could control rule application by requiring rules from one class to be attempted before rules from another class could be used. Future experimentation may suggest the addition of other features to the system. Perhaps some of these mechanisms might be motivated from a more theoretical point of view, resulting in the construction of a "minimally adequate set" of features (Dahl, 1985). Nonetheless, the decision of which control mechanism to use, and how to use it, rests with the person who constructs the grammar.

Even with the use of procedural control, FIGG can not process GPSG grammars as efficiently as SAUMER, and even the ProGram system can outperform FIGG with respect to processing ID/LP grammars. The inefficiency of FIGG is its greatest weakness. An efficient grammar processor could result in efficient interpreted processing of ID/LP or GPSG grammars. Further development of FIGG should include optimisation of some of the code. Frequently used routines, such as *gap* and the rule translation routines, could be altered from the concise descriptions presented in this

thesis into a more efficent, but less readable, format. Parallel *gap processing* could also lead to implementation efficiencies (Dahl and Abramson, 1984). The bottom-up shift reduce parser of FIGG is an essentially complete processor of unrestricted GGs. However, this parser's major limitation is its inability to process grammars which permit bottom-up cycles unless procedural control is inserted into the rules to prevent the cycles. Recall that this means that derivations of the form $\alpha\omega\beta$ ==>* ω are not allowed. Since epsilon productions, *nt* --> ϵ, are often convenient to use in many grammars, FIGG might be modified to automatically *normalise* grammars containing these productions into equivalent grammars that do not. Although the top-down parser is more efficient than its bottom-up counterpart, it is only a GG parser. In this thesis, the top-down parser was shown to be a reasonably efficent processor of a totally free word order grammar. But it can not prbcess grammars produced by the ID/LP-UGG conversion procedure, nor can it use the gapping grammars produced from GPSG metarules. Both of these inadequacies are due to the nested head problem. Left recursive rules can be processed by this parser, but the grammar must often be modified due to the complicated restrictions imposed on the format of these rules. Nonetheless, the development of the top-down parser solved some of the problems associated with the GG1 implementation, and resulted in a parser that can operate efficiently on a subset of unrestricted GGs. It is apparent that further research into more efficient and complete unrestricted GG parsers is required.

Finally, the applications of GGs and unrestricted GGs have thus far been limited to sentence parsing. Research into the use of these grammars for sentence generation would also be appropriate.

Appendix A

Sample Terminal Session

This is a sample terminal session on a SUN Workstation using the latin grammar.

```
po% figg
C-Prolog version 1.5

FIGG 2.3

> /* Let us first see how the system translates some rules */
>
> +display
Display mode
> s(decl) --> np(Agr1), vp(Agr2), {agree(Agr1,Agr2)}.

reduce([np(_17)^1,vp(_21)^1|_121],_76):-
    agree(_17,_21),_121=_102,_78=_102;sr_parse([s(decl)^_106],_78,_76) true
> +topdown.
Top-Down Parser
> s(decl) --> np(Agr1), vp(Agr2), {agree(Agr1,Agr2)}.

s([],decl,_81,_82):-np(_110,_17,_81,_112),vp(_140,_21,_112,_82),agree(_17,_21)
>
> /* Now let us read in a grammar */
> -display.
Generate mode
> [grammar].

FIGG: grammar consulted 4136 bytes  2.68333 seconds.
> start_symbol  sentence(Tree)  /  (write('Parse Tree'),writetree(0,Tree)).
> parse.
Parse Mode: All Parses
? vir est.
Sentence:  vir est.
Total 0.0833333 sec.

? >
?> /* Oops, forgot to read in some Prolog clauses that we need */
?>
?> prolog [plog].
plog consulted 1992 bytes 1 sec.
?> ^D
?
```

```
? vir est.
Sentence:  vir est.
Parse Tree
s(
    np(
        n(vir)
    )
    vp(
        v(est)
    )
)  in 0.283333 sec.
Total 0.483334 sec.

? vir bonus vetus puellam amat.
Sentence:  vir bonus vetus puellam amat.
Parse Tree
s(
    np(
        n(vir)
        adjs(
            adj(bonus)
            adj(vetus)
        )
    )
    vp(
        np(
            n(puellam)
        )
        v(amat)
    )
)  in 1.93333 sec.
Parse Tree
s(
    np(
        n(vir)
        adjs(
            adj(vetus)
            adj(bonus)
        )
    )
    vp(
        np(
            n(puellam)
        )
        v(amat)
    )
)  in 0.699997 sec.
Total 3.11667 sec.

? >
?> +oneparse.
One parse mode
?> ^D
```

```
?
? vir bonus vetus puellam amat.
Sentence:  vir bonus vetus puellam amat.
Parse Tree
s(
    np(
        n(vir)
        adjs(
            adj(bonus)
            adj(vetus)
        )
    )
    vp(
        np(
            n(puellam)
        )
        v(amat)
    )
)  in 1.86666 sec.
Total 1.91667 sec.

? ^D
>
> /* Now lets try other parser */
>
> clear
> flags.
Bottom-Up Parser
Parse Mode: All Parses
Generate Mode
>
> [grammar1].   /* The start symbol is defined in this file */

FIGG: grammar1 consulted 3468 bytes  2.3 seconds.
>
>
> +oneparse.
One parse mode
> parse.
Parse Mode: One Parse
? vir est.
Sentence:  vir est.
Parse Tree:
s(
    np(
        n(vir)
    )
    vp(
        v(est)
    )
)  in 0.399994 sec.
Total 0.483343 sec.
```

```
? ^D
> ^D
[ FIGG execution halted ]

[ Prolog execution halted ]
po%
```

Appendix B

Direct Processing of ID/LP Grammars

B.1. ProGram Specification and Test Results

ProGram Specification

The following is a description of the generalised phrase structure grammar that is equivalent to the ID/LP grammar specified in (5.37). This specification is processed by the ProGram system.

```
/* Specify the feature syntax */

syncat  root.
feature [root,  cat].
feature [cat,   bar, head].
feature [bar,   {lexical, 1, 2}].
feature [head,  major].
feature [major, {v, n, p}].

/* Define some abbreviations for certain feature combinations */

alias(  v(N),  [root,[cat,[bar,N],[head,[major,v]]]] ).
alias(  n(N),  [root,[cat,[bar,N],[head,[major,n]]]] ).
alias(  p(N),  [root,[cat,[bar,N],[head,[major,p]]]] ).
alias(  h(N),  [root,[cat,[bar,N],[head,[major]]]] ).
alias(  0,     lexical).
alias(  h,     h(lexical)).

/* The ID rules */

s:  v(2) --> n(2),h(1).
vp: v(1) --> h.
vp: v(1) --> h,n(2).
vp: v(1) --> h,v(2).
vp: v(1) --> h,n(2),n(2).
vp: v(1) --> h,n(2),p(2).
vp: v(1) --> h,n(2),v(2).
np: n(2) --> h.
pp: p(2) --> h,n(2).
```

```
/* Lexical rules */

np(n): n(0) -> n.
vp(v): v(0) -> v.
pp(p): p(0) -> p.

/* linear precedence relations */

n(2) << v(1).
v(0) << v(2).
p(0) << n(2).

/* prohibit categories with unspecified major or bar level */

rac( bar,   [], not(unspec)).
rac( major, [], not(unspec)).
```

Test Results

The parse times for the individual sentences shown in (5.38), along with the total processing times are given below.

```
Sentence: n v
1st Parse:      3.13334 sec.
Total           11.5339 sec.

Sentence: n v n.
1st Parse:      11.95 sec.
Total:          35.7176 sec.

Sentence: n n v.
1st Parse:      20.25 sec.
Total:          33.8001 sec.

Sentence: n v n v.
1st Parse:      11.95 sec.
Total:          108.234 sec.

Sentence: n v n v n.
1st Parse:      22.0833 sec.
Total:          310.1 sec.

Sentence: n v n n v.
1st Parse:      22.6834 sec.
Total:          303.25 sec.

Sentence: n v n p n.
1st Parse:      23.4333 sec.
Total:          118.3 sec.

Sentence: n v p n n.
```

1st Parse: 17.4334 sec.
Total: 91.8502 sec.

Sentence: n n v p n.
1st Parse: 32.35 sec.
Total: 112.067 sec.

Sentence: n n p n v.
1st Parse: 68.0333 sec.
Total: 97.5344 sec.

Sentence: n p n v n.
1st Parse: 67.45 sec.
Total: 124.667 sec.

Sentence: n p n n v.
1st Parse: 89.35 sec.
Total: 117.702 sec.

Sentence: n v n v n v.
1st Parse: 70.3666 sec.
Total: 909.933 sec.

Sentence: n v n n n v.
1st Parse: 154.683 sec.
Total: 691.183 sec.

Sentence: n v n n v n.
1st Parse: 288.233 sec.
Total: 851.817 sec.

Sentence: n v n v n n
1st Parse: 87.8666 sec.
Total: 927.083 sec.

Sentence: n n v n v n
1st Parse: 218.433 sec.
Total: 1017.5 sec.

Sentence: n n v n n v
1st Parse: 396.683 sec.
Total: 984.317 sec.

B.2. FIGG Specification and Test Results

The FIGG Specification

The following rules and clauses are used by FIGG to process those sentences allowed by the grammar in (5.37).

```
op(700, yf, '):              /* Define the reverse quote */

start_symbol s.

/* The ID rules */

r(1):   s  --> mk, np', vp', mk, {id_rule}.
r(2):   vp --> v, {inter_rule, id_rule}.
r(3):   vp --> mk, v', np', mk, {id_rule}.
r(4):   vp --> mk, v', s', mk, {id_rule}.
r(5):   vp --> mk, v', np', pp', mk, {id_rule}.
r(6):   vp --> mk, v', np', s', mk, {id_rule}.
r(7):   np --> n, {inter_rule, id_rule}.
r(8):   pp --> mk, p', np', mk, {id_rule}.
r(lex): X --> [X], {inter_rule, id_rule}.

/* The two extra gapping rules */

gap(G), !, X' --> X' ', !, gap(lp(X),G).

mk^r(_), gap(G1), mk^r(_) --> {inter_rule}, gap([X,Y|Z]),
  {double_quote([X,Y|Z],G1)}.

prolog [user].  /* The following is processed by C-Prolog, not by FIGG */

/* The linear precedence relations */

lp(np,vp).
lp(p,np).
lp(v,s).
lp(mk,_).

/* Pairs of symbols which may both appear on the right hand side of some rule */

pair(X,X).
pair(np,vp).
pair(p,np).
pair(v,np).
pair(v,s).
pair(v,pp).
pair(np,pp).
pair(np,s).
```

```
/* And the definitions of the procedures */

no_LPs([],X) :- !.

no_LPs([Hd'|Rest], X) :-
   ( pair(Hd,X) ; pair(X,Hd) ), !
 , not lp(Hd,X)
 , no_LPs(Rest,X).

no_LP(X',Y) :- not lp(X,Y), !, ( pair(Y,X) ; pair(X,Y) ), !.

double_quote([],[]).

double_quote([Hd|Rest],[(Hd')'|NewRest]) :-
   double_quote(Rest,NewRest).

inter_rule :- not inrule, !, ( assert(inrule) ; abolish(inrule,0), fail ).

id_rule :- abolish(inrule,0) ; assert(inrule) , fail.
```

The Test Results

The actual output produced by FIGG while parsing the sentences (5.38) is shown below.

```
Sentence:  n v.
Parse found   in 0.283333 sec.
Total 1.93334 sec.

Sentence:  n v n.
Parse found   in 1.95 sec.
Total 13.7333 sec.

Sentence:  n n v.
Parse found   in 5.46666 sec.
Total 14.0167 sec.

Sentence:  n v n v.
Parse found   in 2.18335 sec.
Total 90.2667 sec.

Sentence:  n v n v n.
Parse found   in 6.73328 sec.
Parse found   in 57.9501 sec.
Total 503.4 sec.

Sentence:  n v n n v.
Parse found   in 6.3667 sec.
Parse found   in 170.9 sec.
Total 488.55 sec.
```

```
Sentence:  n v n p n.
Parse found    in 6.08398 sec.
Total 264.617 sec.

Sentence:  n v p n n.
Parse found    in 5.06738 sec.
Total 266.617 sec.

Sentence:  n n v p n.
Parse found    in 12.8672 sec.
Total 274.117 sec.

Sentence:  n n p n v.
Parse found    in 105.383 sec.
Total 257.867 sec.

Sentence:  n p n v n.
Parse found    in 55.4844 sec.
Total 283.117 sec.

Sentence:  n p n n v.
Parse found    in 123.467 sec.
Total 284.251 sec.

Sentence:  n v n v n v.
Parse found    in 64.1166 sec.
Total 2967.95 sec.

Sentence:  n v n n n v.
Parse found    in 922.418 sec.
Total 2546.57 sec.

Sentence:  n v n n v n.
Parse found    in 327.016 sec.
Parse found    in 173.137 sec.
Total 2591.35 sec.

Sentence:  n v n v n n.
Parse found    in 150.816 sec.
Total 2530.4 sec.

Sentence:  n n v n v n.
Parse found    in 378.813 sec.
Total 2765.94 sec.

Sentence:  n n v n n v.
Parse found    in 970.617 sec.
Total 2640.29 sec.

end-of-file
```

Appendix C

Direct Processing of GPSG Metarules

C.1. ProGram GPSG Grammar

```
/*  This grammar was adapted from a ProGram Demonstration Grammar (Evans and
    Gazdar, 1984)

/*  Features  */

syncat root.

feature [root,  cat, foot, conj].
feature [cat,   bar, head].
feature [bar,   {lexical, 1, 2}].
feature [head,  major, minor].
feature [major, {v, n, a, p, conj, rel}].
feature [p,     {by, to, in, on, with}].
feature [minor, agr, {case, vform}].
feature [agr,   {singular, plural}].
feature [vform, {finite,passive,base,infinitive}, auxiliary, inverted].
feature [case,  {nominative, possessive}].
feature [foot,  cat].
boolean auxiliary.
boolean inverted.


/* Aliases - aliases let you write v(2) to mean a basic verbal category of
   bar level 2 (similarly for other bar levels, and for nouns, adjectives and
   prepositions) */

alias(  v(N),   [root,[cat,[bar,N],[head,[major,v]]]] ).
alias(  n(N),   [root,[cat,[bar,N],[head,[major,n]]]] ).
alias(  a(N),   [root,[cat,[bar,N],[head,[major,a]]]] ).
alias(  p(N),   [root,[cat,[bar,N],[head,[major,p]]]] ).
alias(  v(N,M),  [root,[cat,[bar,N],[head,[major,v],[minor|M]]]] ).
alias(  n(N,M),  [root,[cat,[bar,N],[head,[major,n],[minor|M]]]] ).
alias(  a(N,M),  [root,[cat,[bar,N],[head,[major,a],[minor|M]]]] ).
alias(  h(N),   [root,[cat,[bar,N],[head,[major]]]] ).
alias(  p(N,P),[root,[cat,[bar,N],[head,[major,[p,P]]]]] ).
alias(  0,      lexical).
alias(  h,      h(lexical)).
```

```
alias(  sing,  [agr,singular]).
alias(  plur,  [agr,plural]).
alias(  nom,   [case,nominative]).
alias(  acc,   ~case).
alias(  aux,   [vform, finite, +auxiliary, -inverted]).
alias(  inv,   [vform, finite, +auxiliary,+inverted]).
alias(  fin,   [vform, finite, -auxiliary, -inverted]).
alias(  bse,   [vform, base, -inverted]).
alias(  bse1,  [vform, base, +auxiliary, -inverted]).
alias(  bse2,  [vform, base, -auxiliary, -inverted]).
alias(  inf,   [vform, infinitive, -auxiliary, -inverted]).
alias(  pass,  [vform, passive, -auxiliary, -inverted]).

/* an alias for slash categories */

alias(X/Y, Z) :-
    normfeat(X,XN),normfeat(Y,YN),
    pathfor(foot,YN,'~'),
    pathfor(cat,YN,YCat),
    pathfor(foot,XN,[[cat|YCat]]),
    Z  = protect(XN).


/* The Immediate Dominance Rules */

s: v(2) --> N2,H1 where N2 is n(2,[nom]),
                        H1 is h(1),
                        N2 controls H1.

vp_1: v(1) --> h.
vp_2: v(1) --> h,n(2).
vp_3: v(1) --> h,n(2),n(2).
vp_3: v(1) --> h,n(2),p(2,to).
vp_4: v(1,[aux]) --> h,v(1,[bse]).
vp_5: v(1,[aux]) --> h,v(1,[pass]).
vp_6: v(1,[bse1]) --> h,v(1,[pass]).
vp_7: v(1) --> h,n(2),v(1,[inf]).
vp_8: v(1) --> h,v(1,[inf]).
vp_9: v(1,[inf]) --> h,v(1,[bse]).
vp_B: v(1) --> h,v(2,[bse2]).

np_1: n(2) --> DET,H1          where DET is a(0),
                                     H1 is h(1),
                                     H1 controls DET.
np_2: n(2) --> h.

nb_1: n(1) --> h.
nb_2: n(1) --> a(1),h(1).

ap_1: a(1) --> h.

pp:  p(2) --> h,n(2).
```

```
/* prepositional phrase topicalisation rule */

top: v(2) --> C1, h(2)/C2
        where   M is [major,p],
                C1 is [root,[cat,[bar,2],[head,M]]],
                C2 is [root,[cat,[bar,2],[head,M]]].


/* The Metarules */

/* passive: */

pass:       (VP1 --> ... , n(2)          where VP1 is v(1))
        ==>
            (VP2 --> ... , opt(p(2,by)) where VP2 is v(1,[pass]),
                                              VP1 matches VP2).


/* aux inversion */

inv:        (VP1 --> h,VP2    where VP1 is v(1,[aux]),VP2 is v(1))
        ==>
            (S1  --> h,S2     where S1 is v(2,[inv]), S2 is v(2),
                                    S1 matches VP1,
                                    S2 matches VP2).

/* the simplest slash termination meta-rule */

stm1:       (C1 --> C2, ...   where C1 is [root],
                                    C2 is [root,[cat,[bar,2],
                                      [head,[minor,~case]]]])
        ==>
            C1/C2 --> ...  .


/* Linear Precedence Relations */

/* Any lexical category precedes a non-lexical */

[root,[cat,[bar,lexical]]] << { [root,[cat,[bar,1]]],
                                [root,[cat,[bar,2]]]  }.

/* a few straightforward precedences */

p(0) << n(2) << v(1).
a(1) << n(1) << p(2).
n(2) << p(2,to).

/* slash categories always last */

[root,~foot] << [root,[foot,cat]].
```

```
/* Feature Co-efficient Defaults */

/* just three fcd's, giving defaults for the minor features. Specifying FOOT
   in the exclusion list forces the defaults onto the real minors, not the
   slashed category (if any) minors */

/*    feature     excl      lexical     phrasal
      _______     ____      _______     _______
*/
fcd(  case,       [foot],   free,       acc      ).
fcd(  inverted,   [foot],   -inverted,  free     ).
fcd(  auxiliary,  [foot],   -auxiliary, free     ).


/* Feature Co-occurrence Restrictions */

/* +v --> ~case */

fcr(   major,    [foot],    v,
       minor,    [foot],    not(case)).

/* +v --> ~case in slashed categories too */

fcr(   foot,     [],        [foot,[cat,[head,[major,v]]]],
       foot,     [],        not([foot,[cat,[head,[minor,case]]]])).

/* +inv --> +aux */
fcr(   inverted, [foot],    +inverted,
       vform,    [foot],    [vform, finite, +auxiliary]).


/* Prohibit categories with unspecified major or bar level */

rac( bar,   [foot], not(unspec)).
rac( major, [foot], not(unspec)).


/* The lexicon */

vp_1(sf): v(0,[sing,fin]) -> runs,flies.
vp_1(bs): v(0,[bse])      -> run, fly.

vp_2(sf): v(0,[sing,fin]) -> loves,takes,sees.
vp_2(bs): v(0,[bse])      -> take, love, see.
vp_2(ps): v(0,[pass])     -> loved,taken,seen.

vp_3(sf): v(0,[sing,fin]) -> gives.
vp_3(bs): v(0,[bse])      -> give.
vp_3(ps): v(0,[pass])     -> given.

vp_4(sf): v(0,[sing,inv]) -> does.
vp_4(pf): v(0,[plur,inv]) -> do.
```

```
vp_5(sf): v(0,[sing,aux]) -> is.

vp_6(bs): v(0,[bse1])     -> be.

vp_7(sf): v(0,[sing,fin]) -> expects.
vp_7(bs): v(0,[bse])      -> expect.
vp_7(ps): v(0,[pass])     -> expected.

vp_8(sf): v(0,[sing,fin]) -> wants.
vp_8(bs): v(0,[bse])      -> want.
vp_8(ps): v(0,[pass])     -> wanted.

vp_9(in): v(0,[inf]) -> to.

vp_B(sf): v(0,[sing,fin]) -> sees.
vp_B(bs): v(0,[bse])      -> see.
vp_B(ps): v(0,[pass])     -> seen.

np_2(prop_nom):  n(0,[sing,nom]) -> john,math101.
np_2(prop_acc):  n(0,[sing,acc]) -> mary,marvin,cmpt101.

nb_1(s_acc): n(0,[sing,acc]) -> ball,girl,fly.

nb_1(s_nom): n(0,[sing,nom]) -> book,boy,house.

np_1(sdet): a(0,[sing]) -> the.

ap_1: a(0) -> big,little,house.

pp(by): p(0,by) -> by.
pp(to): p(0,to) -> to.
```

C.2. Unrestricted Gapping Grammar for GPSG Grammar

```
start_symbol s(Type,_,_,nil).

/* For sentences, the first argument states if it's inverted, the second
   states what form of the finite verb it contains, or if it's the infinitive,
   past participle, etc.  The third argument states if its active or passive,
   while the last argument is for the "foot feature".  In our case, it's "nil"
   for non-slash categories, and the "missing" constituent for slash categories.
*/

s:       s(-inv,Agr,Voice,Foot) --> np(A,subj,WH,nil), vp(Agr,Voice,Foot)
                                    , {agree(A,Agr), rule}.
pp_top: s(INV,Agr,Voice,nil)   --> pp(to,nil), s(INV,Agr,Voice,pp(to,nil)), {rule}.

s_a:     s(-inv,Agr,Voice,np(A,subj,WH,nil)) --> vp(Agr,Voice,Foot), {rule}.
np_top: s(-inv,Agr,Voice,nil)  --> np(A,obj,+wh,nil),
  s(+inv,Agr,Voice,np(A,obj,WH,nil))
                                    , {rule}.
```

```
/* For noun phrases, the first argument is for agreement, the second for case,
   the third is +wh for "wh" noun phrases (like "what"), and the last argument
   is once again for slash categories */

np_1: np(Agr,Case,WH,nil)  --> det(WH), np1(Agr,Case,-wh,-pn), {rule}.
np_2:  np(sg3,Case,WH,nil)  --> noun(Agr,Case,WH,PN), {PN \== -pn}, {rule}.
np_rel: np(Agr,Case1,WH,nil) --> np(Agr,Case1,WH,nil), relpro(Agr,Case)
                              , s(decl,np(Agr,Case,-wh,nil)), {rule}.

np1_1:  np1(Agr,Case,-wh,-pn) --> noun(Agr,Case,-wh,-pn), {rule}.
np1_2:  np1(Agr,Case,-wh,-pn) --> adjp, noun(Agr,Case,-wh,-pn), {rule}.

adjp:   adjp --> adj, {rule}.

/* Verb phrases have arguments for agreement, voice, and slash categories. */

vp_1:   vp(Agr,active,nil)  --> v(_,Agr,intrans,active), {rule}.
vp_2:   vp(Agr,active,nil)  --> v(_,Agr,trans,active), np(_,obj,WH,nil), {rule}.
vp_3:   vp(Agr,active,nil)  --> v(_,Agr,trans,active), np(_,obj,WH,nil),
  np(_,obj,WH,nil), {rule}.
vp_4:   vp(Agr,active,Foot) --> v(_,Agr,trans,active), vp_inf(Voice,Foot), {rule}.
vp_5:   vp(Agr,active,Foot) --> v(see,Agr,trans,active), s(-inv,bse,active,Foot),
  {rule}.
vp_pp:  vp(Agr,active,nil)  --> v(_,Agr,trans,active), np(_,obj,WH,nil),
  pp(to,nil), {rule}.
vp_A:   vp(Agr,active,Foot) --> v(_,Agr,trans,active), np(_,obj,WH,nil),
  vp_inf(Voice,Foot), {rule}.
vp_B:   vp(Agr,active,Foot) --> aux(do,Agr), vp(bse,active,Foot), {rule}.
vp_C:   vp(Agr,passive,Foot)--> aux(be,Agr), vp(pass,passive,Foot), {rule}.

vp_2a:  vp(Agr,active,np(_,obj,WH,nil)) --> v(_,Agr,trans,active), {rule}.
vp_3a:  vp(Agr,active,np(_,obj,WH,nil)) --> v(_,Agr,trans,active),
  np(_,obj,WH,nil), {rule}.
vp_Aa:  vp(Agr,active,np(_,obj,WH,nil)) --> v(_,Agr,trans,active),
  vp_inf(Voice,nil), {rule}.
vp_ppa: vp(Agr,active,np(_,obj,WH,nil)) --> v(_,Agr,trans,active), pp(to,nil),
  {rule}.
vp_ppb: vp(Agr,active,np(_,obj,WH,nil)) --> v(_,Agr,trans,active), np(_,obj,WH,nil)
                                          , pp(to,np(_,obj,WH,nil)), {rule}.
vp_ppc: vp(Agr,active,pp(to,nil)) --> v(_,Agr,trans,active), np(_,obj,WH,nil),
  {rule}.

vp_inf(Voice,Foot)  --> [to], vp(bse,Voice,Foot), {rule}.

pp1:    pp(to,nil) --> [to], np(_,obj,WH,nil), {rule}.
pp2:    pp(by,nil) --> [by], np(_,obj,WH,nil), {rule}.
pp1a:   pp(to,np(_,obj,-wh,nil)) --> [to], {rule}.
pp2a:   pp(by,np(_,obj,-wh,nil)) --> [by], {rule}.

/*  The lexical rules */

v(expect,sg3,trans,active) --> [expects], {rule}.
v(expect,bse,trans,active) --> [expect], {rule}.
```

```
v(expect,trans,pstprt) --> [expected], {rule}.

v(fly,sg3,intrans,active) --> [flies], {rule}.
v(fly,bse,intrans,active) --> [fly], {rule}.

v(give,sg3,trans,active) --> [gives], {rule}.
v(give,bse,trans,active) --> [give], {rule}.
v(give,trans,pstprt) --> [given], {rule}.

v(love,sg3,trans,active) --> [loves], {rule}.
v(love,bse,trans,active) --> [love], {rule}.
v(love,trans,pstprt) --> [loved], {rule}.

v(run,sg3,intrans,active) --> [runs], {rule}.
v(run,bse,intrans,active) --> [run], {rule}.

v(see,sg3,trans,active) --> [sees], {rule}.
v(see,bse,trans,active) --> [see], {rule}.
v(see,trans,pstprt) --> [seen], {rule}.

v(take,sg3,trans,active) --> [takes], {rule}.
v(take,bse,trans,active) --> [take], {rule}.
v(take,trans,pstprt) --> [taken], {rule}.

v(want,sg3,trans,active) --> [wants], {rule}.
v(want,bse,trans,active) --> [want], {rule}.
v(want,trans,pstprt) --> [wanted], {rule}.

aux(be,sg3) --> [is], {rule}.
aux(be,bse) --> [be], {rule}.

aux(do,sg3) --> [does], {rule}.

det(-wh) --> [the], {rule}.
det(+wh) --> [what], {rule}.
det(+wh) --> [which], {rule}.

noun(sg3,_,-wh,-pn) --> [ball], {rule}.
noun(sg3,_,-wh,-pn) --> [book], {rule}.
noun(sg3,_,-wh,-pn) --> [boy], {rule}.
noun(sg3,_,-wh,+pn) --> [cmpt101], {rule}.
noun(sg3,_,-wh,-pn) --> [fly], {rule}.
noun(sg3,_,-wh,-pn) --> [girl], {rule}.
noun(sg3,_,-wh,-pn) --> [house], {rule}.
noun(sg3,_,-wh,+pn) --> ['John'], {rule}.
noun(sg3,_,-wh,+pn) --> [math101], {rule}.
noun(sg3,_,-wh,+pn) --> ['Marvin'], {rule}.
noun(sg3,_,-wh,+pn) --> ['Mary'], {rule}.
noun(_,_,+wh,_) --> [what], {rule}.
noun(_,subj,+wh,_) --> [who], {rule}.
noun(_,obj,+wh,_) --> [whom], {rule}.

adj --> [big], {rule}.
```

```
adj --> [house], {rule}.
adj --> [little], {rule}.

relpro(sg3,_)  --> [that], {rule}.
relpro(_,subj) --> [who], {rule}.
relpro(_,obj)  --> [whom], {rule}.

/* active-passive  metarules */

    vp(pass,passive,Foot) --> mk(X), vp(Agr,active,Foot), mk(X), {lhs}.

    {numgen(X)}, mk(X), v(_,Agr,trans,active)^D, np(_,obj,WH,nil)^D, gap(G)^D,
  mk(X)
-->  v(_,trans,pstprt), {rhs}, gap([np(_,_,_,_),pp(_,_),vpinf(_,_)],G).

    {numgen(X)}, mk(X), v(_,Agr,trans,active)^D, np(_,obj,WH,nil)^D, gap(G)^D,
  mk(X)
-->  v(_,trans,pstprt), {rhs}, gap([np(_,_,_,_),pp(_,_),vpinf(_,_)],G), pp(by,nil).

/* sentence inversion metarule */

s(+inv,sg3,Voice,Foot)  --> mk(X), vp(Agr,Voice,Foot), mk(X), {lhs}.

    {numgen(X)}, mk(X), aux(Type,Agr)^D, vp(A,V,Foot)^D, mk(X)
-->  aux(Type,Agr), s(-inv,A,V,Foot), {rhs}.

/* And read in some Prolog definitions that we need */

prolog [user].

virtual([],[]).

rhs :- virtual([],_), !, ( virtual_push(right) ; virtual_pop(right), fail ).

lhs :- virtual([_|_],[]), !, ( virtual_pop(left) ; virtual_push(left), fail ).

rule :- virtual([],[]), !.
rule :- virtual([],_) -> ( virtual_swap(left) ; virtual_swap(right), fail ).

virtual_push(left)  :- virtual(X,[]), abolish(virtual,2),
  assert(virtual([x|X],[])),!.
virtual_push(right) :- virtual([],X), abolish(virtual,2),
  assert(virtual([],[x|X])),!.

virtual_pop(left)   :- virtual([x|X],[]), abolish(virtual,2),
  assert(virtual(X,[])),!.
virtual_pop(right)  :- virtual([],[x|X]), abolish(virtual,2),
  assert(virtual([],X)),!.

virtual_swap(left)  :- virtual([],X), abolish(virtual,2), assert(virtual(X,[])), !.
virtual_swap(right) :- virtual(X,[]), abolish(virtual,2), assert(virtual([],X)), !.

agree(_,_).    /* We aren't worrying about person number agreement right now */
```

Appendix D

FIGG Source Code

```
/*
                    FIGG 2.3

   Developed by Fred Popowich at Simon Fraser University, Burnaby B.C.,
during 1984 and 1985.  It was originally based on a Gapping Grammar
parser developed by Veronica Dahl.  The system commands for FIGG were
developed in parallel with those of SAUMER.

Copyright c ,  Fred Popowich, 1985.  FIGG can be used for non-commercial
purposes provided that the author and the Laboratory for Computer and
Communications Research at Simon Fraser University are appropriately
acknowledged.

        Laboratory for Computer and Communications Research
                    Simon Fraser University
                        Burnaby, B.C.
                      CANADA  V5A 1S6
*/

/* The operators used by FIGG */

 :- op(1180, fx, calc).          /* obsolete - replaced by the "display" flag */
 :- op(1170, xfx, -->).
 :- op(1150, xfx, \).            /* for left recursion removal specification */
 :- op(1101, xfy, ++).           /* parallel OR */
 :- op(1050, xfx, :).            /* to separate the rule name from the rule */

 :- op(990, fx, forall).         /* some meta-control constructs */
 :- op(970, xfx, in).

 :- op(800, fx, start_symbol).   /* to specify the grammar's start symbol */
 :- op(800, fx, lexicon).        /* to specify the lexicon files */
 :- op(800, fx, prolog).         /* to specify prolog files */
 :- op(800, fx, morpher).        /* to specify the morpher files */
 :- op(800, fx, input).          /* to specify the parse input file */
 :- op(800, fx, output).         /* to specify the parse output file */
 :- op(800, fx, close).          /* to flush and close a file */
 :- op(800, xf, #).              /* obsolete */
 :- op(800, xfx, @).             /* for recursive arguments, top-down parser */
 :- op(800, fx, ~).              /* NOT prefix for abbreviatory gaps */
 :- op(750, xfx, t).             /* to specify the dominating rule number */
```

```
  :- op(400, xf, (!)).          /* denotes local cut */

  :- op(800, fx, <<).           /* used in AAA lexicon for SAUMER */
  :- op(997, xfy, &&).
  :- op(996, xfy, lmda).


/* Definition of parallel-OR */

_1 ++ _2 :- _1 , (_2 ; true).
_1 ++ _2 :- _2.

/*
                The FIGG Command Interpreter

This file processes all top level input to FIGG.
*/

figg :-   nl , writeln('FIGG 2.3') , nl, !, nofileerrors, figg1('> ').

/*
   To allow us to read input from other files, the filenames can be entered
   as a list.
*/

figg([]) :- !.

figg([Hd|Rest]) :- ! ,
  ( seeing(Old)
  , see(Hd)             /* open input file */
  , Space0 is heapused
  , Time0 is cputime
  , figg1('')
  , seen
  , see(Old)
  , Space is heapused - Space0
  , Time is cputime - Time0
  , write('FIGG: '), write(Hd), write(' consulted '), write(Space)
  , write(' bytes  '), write(Time), writeln(' seconds.')

  ; write('Unable to open '), writeln(Hd)
  )
, figg(Rest), !.

figg(_body) :- !,               /* the main control procedure */
   control(_body, _body1) , !
 , ( _body1 ; writeln('FAIL') ), !.

figg1(Prompt)  :-
   prompt(Old, Prompt)
 , repeat
 , read(X)
 , ( X = 'end_of_file'
```

```
    , prompt(_,Old)
    , ( Prompt == '> ' -> (nl, writeln('[ FIGG execution halted ]'), halt) ; nl )
    ; figg(X), fail
    ).

references([]).          /* Pointers to Prolog rules generated from DCG rules */

/* The FIGG commands */

control(- display, (abolish(calc_mode,0), writeln('Generate mode'))) :- !.

control(+ display, (assert(calc_mode), writeln('Display mode'))) :- !.

control(- oneparse, (abolish(one_parse,0), writeln('All parses mode'))) :- !.

control(+ oneparse, (assert(one_parse), writeln('One parse mode'))) :- !.

control(- topdown, (abolish(top_down,0), writeln('Bottom-up Parser'))) :- !.

control(+ topdown, (assert(top_down), writeln('Top-Down Parser'))) :- !.

control(flags, (parser,prsmode,calcmode)) :- !.          /* Display the flags */

/* Is a dominator set for the input sentence */

control(sentence=Var, abolish(s_dom,1)) :- var(Var), !.

control(sentence=Dominator, (abolish(s_dom,1), assert(s_dom(Dominator)))) :- !.

control(clear,(
   references(References)
  , abolish(top_down,0), abolish(references,1), abolish(startsym,2)
  , abolish(reduce,3), abolish(s_dom,1), abolish(one_parse,0)
  , abolish(calc_mode,0), assert(references([]))
  , control((forall Ref in References, erase(Ref)), Code), Code) ) :- !.

/*
   The "parse" command.  If any arguments are given, open the appropriate
   input and output files.  They will remain opened until explicitly closed
*/

control(parse, (prsmode,main('? '))) :- !.

control((parse,Args), (prsmode, seeing(See), telling(Tell), Code, main('? '),
  see(See), tell(Tell))) :- !,
   control(Args,Code).

control((input File), ((see(File) ; writeln('Unable to open input file'), fail),
  !)) :- !.

control((output File), ((tell(File) ; writeln('Unable to open output file'), fail),
  !)) :- !.
```

```
control((close File), ((tell(File), told ; writeln('Unable to access file'), fail),
  !)) :- !.


/* For top down parser, call td_convert */

control((calc Rule), (td_convert(Rule,Rule1), writeln(Rule1)) )  :- top_down, !.

control((A --> B), (td_convert((A --> B),Rule1),assertr(Rule1))) :- top_down,!.

/*
   For bottom up (shift reduce) parser, invoke the "sr_trans" routine
   to process rules, or to display processed rules.
*/

control((calc Rule), ( sr_trans(Rule,Rule1) , writeln(Rule1) ) ) :- !.

control(Rule!, (sr_trans(Rule!,Rule1), assertr(Rule1))) :- !.

control((A --> B), (sr_trans((A --> B),Rule1), assertr(Rule1))) :- !.


/* And the various other commands */

control(( prolog Files ),  Files ) :- !, Files=[_|_].

control(  lexicon, (abolish(lookup,2), assert((lookup(X,Y) :- name(Y,X))))) :- !.

control(( lexicon Files ),  (abolish(lookup,2), Files) ) :- !, Files=[_|_].

control(( morpher Files ),  Files ) :- !, Files=[_|_].

control(( start_symbol S / Sem ),  assert(startsym(S,Sem)) ) :- !.

control(( start_symbol S ),  assert(startsym(S,_)) ) :- !.


/* And finally, for rule schemata we have... */

control((forall _x in [M|N], _body),  loop(_x, [M|N], _body1)) :- !,
   control(_body, _body1).

control((forall _p , _body), (_p , call1(_body1) ,  fail ; true)) :-
   nonvar(_p) , !
 , control(_body, _body1).

loop(X, [], Body) :- !.

loop(X, [Y|Rest], Body) :- !,
   ( X=Y, call1(Body), fail
   ; loop(X, Rest, Body) ).
```

```
/*   And we also allow the following Prolog syntax   */

control((_a , _b), (_a1 , _b1)) :- !, control(_a, _a1), control(_b, _b1).

control((_a ; _b), (_a1 ; _b1)) :- !, control(_a, _a1), control(_b, _b1).

control(not _a, not _a1) :- !, control(_a, _a1).

control(!, !) :- !.

control(_a, _a) :- !.

parser :- !,
   top_down -> writeln('Top-Down Parser') ; writeln('Bottom-Up Parser').

prsmode :- !,
   write('Parse Mode: ')
 , one_parse -> writeln('One Parse') ; writeln('All Parses').

calcmode :- !,
   calc_mode -> writeln('Display Mode') ; writeln('Generate Mode').


/*
The Prolog code for the Shift-Reduce Parser for FIGG. (Bottom-Up)
Convert all rules into "reduce(Stack,NewStack)" clauses.

For non-context-free rules, we treat them as transformations on the input.
If they put the new information on the stack, rather than into the "input",
other applicable rules would not get a chance to apply, since the addition
of more than one symbol to the stack may prevent their use. An equivalent
solution would be to "reduce" after adding each new "stack symbol".
(By putting it into the input though, we let "parse" worry about this adding
of symbols, one at a time.  The rule:

   s, x --> [a], x

would be translated as:

   reduce([[a],x|Stack], NewStack) :- sr_parse([s,x], Stack, NewStack).

Although this is not the most efficient, it provides a
more symmetric translation algorithm possible since ALL rules will be
translated into clauses of this form.

If the left hand side contains a cut symbol, then all symbols to the right of
the cut will be added to the stack, while the others will be return through
the NewInput argument to be added to the stack one by one. The rule:

  s, t, !, x, y --> [a], x

would be translated as:
```

```
   reduce([[a],x|Stack], NewStack) :- sr_parse([s,t], [x,y|Stack], NewStack).

 For context-free rules of the form:

    s --> [a], s.

 the following clause will result.

    reduce([[a],s|Stack], NewStack) :- sr_parse([s], Stack, NewStack)
*/

/*  Preprocess for rules contained in a cut */

sr_trans((Rule)!, NewRule) :- !,
   sr_trans(!, Rule, NewRule).

sr_trans(Rule, NewRule) :-
   sr_trans(true, Rule, NewRule).


sr_trans(Cut, (Lhs-->Rhs), (reduce(RhsStack,NewStack) :- Code)):-
   rulename(Lhs, Lhs1, RuleName)
 , lhs(Lhs1, ForInput, ForStack)
 , sr_trans('$undef', ForStack, LhsStack, StackBase, StackCode)
 , sr_trans('$undef', ForInput, Input, [], InputCode)
 , sr_trans(RuleName, Rhs, RhsStack, StackBase, RhsCode)
 , combine(InputCode, StackCode, LhsCode)
 , combine(RhsCode, LhsCode, RhsLhsCode)
 , combine(RhsLhsCode, sr_parse(Input,LhsStack,NewStack), NewCode)
 , '$flatconj'((NewCode,Cut), Code).


/* If a Rule Name is provided,  use it !!!   Otherwise, generate one */

rulename(A, A, RuleName) :- var(A), !, namegen(RuleName).

rulename((RuleName:Lhs), Lhs, RuleName) :- !.

rulename(Lhs, Lhs, RuleName) :- !, namegen(RuleName).

namegen(RuleName) :-
   name('$name',X)
 , numgen(Suffix)
 , name(Suffix,Y)
 , '$append'(X,Y,Z)
 , name(RuleName,Z), !.

/*
    If there is a cut in the left hand side of a rule, then the stuff before
    it is ForInput and the stuff to the right is ForStack.  If there is no cut,
    then everything is ForInput.
```

```
                lhs(Lhs, ForInput, ForStack)
*/
lhs(Var, Var, {true}) :- var(Var), !.

lhs((!,Code), {true}, Code) :- !.

lhs((A,B), (A,Rest), ForStack) :- !, lhs(B, Rest, ForStack).

lhs(!, {true}, {true}) :- !.

lhs(A, A, {true}).

/*
   sr_trans(RuleName, FIGG_Rule, Stack, StackBase, Code)

  For a FIGG rule, modify the stack appropriately and return any Prolog code
to be executed at that point in the rule.  The RuleName will be undefined
'$undef' for the left hand side of the FIGG rule
*/

/* For variables in the rules */

sr_trans('$undef', X, [X+Dom|R], R, true) :- var(X), !.
sr_trans('$undef', X+Dom, [X+Dom|R], R, true) :- var(X), !.
sr_trans(Name, X, [X+Name|R], R, true) :- var(X), !.


sr_trans(RuleNo, (X,Y), L, R, New) :- !,
   sr_trans(RuleNo, X, L, L1, NewX)
 , sr_trans(RuleNo, Y, L1, R, NewY)
 , combine(NewX,NewY,New).


sr_trans(_, {true}, L, R, (L=R)) :- !.          /* Special Case */

sr_trans(_, {Code}, L, R, (Code, L=R)) :- !.    /* We want to execute the code
                                                   before matching the rest of
                                                   the rule  */
sr_trans(_, !, L, R, (!, L=R)) :- !.

sr_trans(No, (Expr)!, L, R, call1(NewExpr)) :- !,  /* Local Cut */
   sr_trans(No, Expr, L, R, NewExpr).

/* For gaps on the left and on the right sides of rules. */

sr_trans('$undef', Gap+Dom, L, R, NewGap) :-
   Gap =.. [gap|Args], !
 , '$append'(Args, [L,R], NewArgs)
 , NewGap =.. [gapD,Dom|NewArgs].       /* use gapD, which inserts the ruleno */

sr_trans(No, Gap, L, R, NewGap) :-
   Gap =.. [gap|Args], !
 , '$append'(Args, [L,R], NewArgs)
```

```
    , NewGap =.. [gapD,No|NewArgs].


/* For terminal symbols appearing on the left hand side of rules... */

sr_trans('$undef', [Term]†Dom, [[Term]†Dom|R], R, true) :- !.
sr_trans('$undef', [Term], [[Term]†Dom|R], R, true) :- !.
sr_trans('$undef', [], R, R, true) :- !.

/* ... and on the right */

sr_trans(No, [Term], [[Term]†No|R], R, true) :- !.


/* For nonterminals on the left hand side... */

sr_trans('$undef', NonTerm†Dom, [NonTerm†Dom|R], R, true) :- !.
sr_trans('$undef', NonTerm, [NonTerm†Dom|R], R, true) :- !.

/* ... and for right hand sides */

sr_trans(No, NonTerm, [NonTerm†No|R], R, true) :- !.


/*
   And the revised gap predicate, with dominators.  If a rule number is
   provided, use it as the dominator for all the elements in the gap.

   There are five different definitions of gap.
        gap(+,G)  ==    increasing gap
        gap(-,G)  ==    decreasing gap
        gap(lp(X),G) == ensures that the linear precedence relation g < X
                        is not present forall g in G.
        gap(List,G) ==  ensures that the gap contains only elements of List
                        (abbreviatory gap)
        gap(~Excl,G) == ensures that the DOES NOT contain elements of Excl
*/

gapD(No,Gap) --> gapD(No,+,Gap).

gapD(_,lp(X),[]) --> [].
gapD(_,+,[])     --> [].
gapD(_,[_|_],[]) --> [].
gapD(_,~[_|_],[]) --> [].

gapD(No,[X|Y],[Word|Rest]) --> !, {No \== '$undef' -> Dom=No; true},
                               [Word†Dom], !,
                               {element(Word,[X|Y])}, gapD(No,[X|Y],Rest).

gapD(No,~[X|Y],[Word|Rest])--> !, {No \== '$undef' -> Dom=No; true},
                               [Word†Dom], !,
                               {not element(Word,[X|Y])}, gapD(No,~[X|Y],Rest).
```

```
gapD(No,lp(X),[Word|Rest]) --> !, {No \== '$undef' -> Dom=No; true},
                               [Word↑Dom], !,
                               {no_LP(Word,X)}, gapD(No,lp(X),Rest).

gapD(No,Sign,[Word|Rest]) --> {No \== '$undef' -> Dom=No; true},
                              [Word↑Dom],gapD(No,Sign,Rest).
gapD(_,-,[])-->[].

/*  And now the actual parser  */

sr(Input, FinalStack) :-
   prepare(Input,Input1)
 , sr_parse(Input1, [], FinalStack).

sr_parse([Word|Rest], Stack, NewerStack) :-
   sr_parse(Rest, Stack, NewStack)
 , reduce([Word|NewStack], NewerStack).


sr_parse([], Stack, Stack) :- !.


/* If a rule was specified to dominate the input, then use its number */

prepare(Phrase, NewPhrase) :-
   s_dom(Dom), prepare(Dom, Phrase, NewPhrase)
 ; prepare('$undef', Phrase, NewPhrase).

prepare('$undef', [X|Rest],[[X]↑_|NewRest]) :- !, prepare('$undef',Rest,NewRest).
prepare(Dom, [X|Rest],[[X]↑Dom|NewRest]) :- !, prepare(Dom,Rest,NewRest).
prepare(_,[],[]).


/* For processing a gapping grammar for top down parsing.  */

 td_convert((A,B --> C),Clause) :- !,
  rec_arg([],A,ARA,A1),                  /* remove the "recursive argument */
  td_convert1(A1,(c_nonterm-->C),CClause),      /* check for left recursion */
  clauseparts(CClause,CHead,CBody),
  CHead=..[c_nonterm,CRA,X,Z],
  pseudo_te(B,B1),                       /* contruct list of pseudo-terminals */
  expand_term1((b_nonterm-->B1),BClause),
  clauseparts(BClause,BHead,BBody),
  BHead=..[b_nonterm,BRA,Y,Z],
  A1=..[Pred|Args],
  '$append'(Args,[X,Y],NewArgs),
  NewA=..[Pred,ARA|NewArgs],
  '$and'(CBody,BBody,Body),
  formclause(NewA,Body,Clause),
  norm_rule(3,NewA).             /* generate a "normalising rule" if necessary */

/* for processing rules with a single non-terminal on the left */
```

```
td_convert((A-->B),Clause) :-
 td_convert1(A,(A-->B),Clause),
 clauseparts(Clause,Head,Body),
 norm_rule(3,Head).

/* For processing left recursion */

td_convert1(Lhs,(A --> B,C),Clause) :-
 unifiable(Lhs,B), !,                    /* we do have left recursion */
 findclause(C,First,NewC,N1),            /* find a clause to evaluate first */
 not unifiable(Lhs,First), !,            /* make sure it's ok */
 td_convert2((A-->B,NewC),First,N1,Clause).

/* if the user specifies the symbol first checked during left recursion */

td_convert1(Lhs,(A --> First\B),Clause) :- !,
 not unifiable(Lhs,First),
 findclause(B,First,NewB,N1),            /* find it in the rhs. of the rule */
 td_convert2((A-->NewB),First,N1,Clause).

/*
   generate a unique pseudo-terminal name, (N), for this rule.  Then, prevent
   this pseudo-terminal from being used when we are looking for "First".
   (notice use of 3 argument "gap" predicate).  N1 is unique to for each call
   of the rule.
*/

td_convert2((A-->B),First,N1,Clause) :-
 numgen(N),
 expand_term1((x_nonterm --> gap(G,[N|N1],List), First), XClause),
 clauseparts(XClause,XHead,XBody),
 XHead =.. [x_nonterm,XRA,X1,X2],

 /*
    If we do find "First", we put it into a pseudo terminal which will be found
    later.  It is marked and won't be found in a subsequent left recursion
    removal attempt.
 */

 expand_term1((r_nonterm-->B),RightClause),
 clauseparts(RightClause,RightHead,RightBody),
 RightHead =.. [r_nonterm,RRA,NewInput,R2],
 Body = (XBody,numgen(N1),'$append'(G,[te(List,First)|X2],NewInput),RightBody),
 rec_arg([],A,ARA,A1),
 A1=..[Pred|Args],
 '$append'(Args,[X1,R2],NewArgs),
 NewA=..[Pred,ARA|NewArgs],
 formclause(NewA,Body,Clause), !.

td_convert1(_,Dcg,Clause) :- !, expand_term1(Dcg,Clause).

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

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

gap(Gap) --> gap(+,Gap).

gap(+,[])-->[].
gap(+,[Word|Rest])-->[Word],gap(+,Rest).

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

/*
  This gap predicate succeeds unless the 2nd argument is an "element1" of the
  List of a "te" pseudo-terminal that is the first element following the gap.
  If, a legal pseudo-terminal follows the gap, augment the "invalid number
  list" (3rd argument) with our new "N".  Otherwise, merely return a list
  consisting of N.
*/

gap([],N,_)-->[te(List,_)], {element1(N,List)}, !, {fail}.
gap([],N,L)-->gap1(N,L).
gap([Word|Rest],N,L)-->[Word],gap(Rest,N,L).

gap1(N,[N|List],[te(List,A)|X],[te(List,A)|X]) :- !.
gap1(N,[N],X,X) :- !.

/*
  element1 succeeds if the head of the first argument is the head of some
  element in the list (2nd arg)
*/

element1(X,[]) :- !, fail.

element1([X|X1], [[Y|_]|Rest]) :- X = Y, ! ; element1([X|X1], Rest).

/* convert any nonterminals into pseudo-terminals, with the exception
   of gaps. Generate new axioms if necessary. */

pseudo_te((X,Y), (NewX,NewY)) :- !,
 pseudo_te1(X, NewX),
 pseudo_te(Y, NewY).

pseudo_te(X, NewX) :- !,
 pseudo_te1(X, NewX).

pseudo_te1([X|Y], [X|Y]) :- !.

pseudo_te1(gap(G), gap(G)) :- !.

pseudo_te1(gap(Sign,G), gap(Sign,G)) :- !.

/* determine if a recursive argument was given */
```

```
pseudo_te1(X@RA, [NewX]) :- !,
 NewX =.. [te,RA,X],
 norm_rule(0,X).

pseudo_te1(X, [NewX]) :- !,
 NewX =.. [te,[],X],
 norm_rule(0,X).

/* for the specified non terminal, add a rule to the appropriate pseudo
   nonterminal if it doesn't already exist. N will be three if the last
   two "parsing arguments", and "recursion argument", have already
   been added.  Otherwise, it will be 0
*/

norm_rule(N,X) :- !,
 functor(X,F,Arity),
 A2 is Arity - N,
 listlen(A2,List),
 F1 =.. [F|List],
 Term =.. [te,RA,F1],  /* te(RA,F1)  1st arg is for left recursion removal */
 '$append'([F,RA|List],[S0,S1],NewX1),
 NewX =.. NewX1,
 assertu((NewX :- c(S0,Term,S1))) .

listlen(0,[]) :- !.

listlen(N,[_|Rest]) :- !, N1 is N - 1, listlen(N1, Rest).

/*
the args correspond to the initial body, the found clause, the new body,
and the unique integer associated with this application of the rule.
*/

findclause((gap(G),Rest), Found, (gap(G),NewRest), N1) :- !,
 findclause(Rest,Found,NewRest,N1).

findclause((gap(Sign,G),Rest), Found, (gap(Sign,G),NewRest), N1) :- !,
 findclause(Rest,Found,NewRest,N1).

findclause(([X|Y],Rest), Found, ([X|Y],NewRest), N1) :- !,
 findclause(Rest,Found,NewRest,N1).

findclause(({X},Rest), Found, ({X},NewRest), N1) :- !,
 findclause(Rest,Found,NewRest,N1).

/*
   For the term that is selected, modify the rule to force the
   "Term --> te(_,Term)" rule to be used. (it's the only one that has a non-
   empty list as its first argument).  Also, ensure that we match the correct
   pseudo-terminal. (that's why we need the "N1")
*/

findclause((Term@RA,Rest), Term, ({RA=[_|_]}, Term@RA, {element([_|N1],RA)},
```

```
  Rest), N1).

 findclause((Term,Rest), Term, ({RA=[_|_]}, Term@RA, {element([_|N1],RA)}, Rest),
  N1) :-
  not Term=(_@_).

 findclause((Term,Rest), Found, (Term,NewRest), N1) :- !,
  findclause(Rest,Found,NewRest,N1).

 findclause(gap(G), Term, _, _) :- !, fail.

 findclause(gap(Sign,G), Term, _, _) :- !, fail.

 findclause([_|_], Term, _, _) :- !, fail.

 findclause(Term@RA, Term, ({RA=[_|_]}, Term@RA, {element([_|N1],RA)}), N1) :- !.

 findclause(Term, Term, ({RA=[_|_]}, Term@RA, {element([_|N1],RA)}), N1) :- !,
  not Term=(_@_).
/*
   This section of code is used to process the rules for use by the top-
   down parser.  For context-free rules only please...

   Adapted from PLOG:DCGS@UBCG_MTS, and from Clockson and Mellish
*/

expand_term1((Lhs --> Rhs), (_p :- _q)) :-
     '$dcglhs'(Lhs,_s0,_s,_p) , !,
     '$dcgrhs'(Rhs,_s0,_s,_q1) , !,
     '$flatconj'(_q1,_q) , !.

'$dcglhs'(NT,_s0,_s,_p) :-
  nonvar(NT) ,
  rec_arg([],NT,X,NewNT) ,      /* if there is a recursive argument */
  '$tag'(X,NewNT,_s0,_s,_p).

/* determine the correct recursive argument */

rec_arg(_, NT, _, NT) :- var(NT) , !.

rec_arg(_, NT @ RA, RA, NT) :- !.

rec_arg(RA, NT, RA, NT) :- !.

'$dcgrhs'((X1 , X2)!,_s0,_s,call1(_p)) :- !,   /* and the now right hand side */
  '$dcgrhs'(X1,_s0,_s1,_p1) ,
  '$dcgrhs'(X2,_s1,_s,_p2) ,
  '$and'(_p1,_p2,_p).

'$dcgrhs'((X1 , X2),_s0,_s,_p) :- ! ,
  '$dcgrhs'(X1,_s0,_s1,_p1) ,
  '$dcgrhs'(X2,_s1,_s,_p2) ,
  '$and'(_p1,_p2,_p).
```

```
'$dcgrhs'((_x1 ; _x2),_s0,_s,(_p1 ; _p2)) :- ! ,
  '$dcgor'(_x1,_s0,_s,_p1) ,
  '$dcgor'(_x2,_s0,_s,_p2).

'$dcgrhs'( {_p} ,_s,_s,_p) :- !.

'$dcgrhs'(!,_s,_s,!) :- !.

'$dcgrhs'(_ts,_s0,_s,true) :-
  '$islist'(_ts) , ! ,
  '$append'(_ts,_s,_s0).

'$dcgrhs'(_x,_s0,_s,_p) :- !,
  rec_arg(_,_x,RA,X),
  '$tag'(RA,X,_s0,_s,_p).

'$dcgor'(_x,_s0,_s,_p) :-
  '$dcgrhs'(_x,_s0a,_s,_pa) ,
  '$dcgor1'(_s,_s0,_s0a,_p,_pa).
'$dcgor1'(_s,_s0,_s0a,_p,_pa) :-
  var(_s0a) , _s0a \== _s , ! , _s0=_s0a , _p=_pa.
'$dcgor1'(_s,_s0,_s0a,_p,_pa) :-
  _p = ((_s0 = _s0a),_pa).

/* special case for the gap predicate */

'$tag'(RA,_x,_s0,_s,_p) :-
  _x =.. [gap|_args] , !,
  '$append'(_args, [_s0,_s], _newargs) ,
  _p =.. [gap|_newargs] .

'$tag'(RA,_x,_s0,_s,_p) :-
  _x =.. [_f|_args] ,
  '$append'(_args, [_s0,_s], _newargs) ,
  _p =.. [_f,RA|_newargs] .

/*  The parser */

main(Prompt) :-
   prompt(Old, Prompt)    /* Save the old prompt */
 , readline(Line)
 , ( Line = 'end_of_file', writeln('end-of-file'), prompt(_,Old)
   ; call1(process(Line)), fail
   ).

/*            Process the given sentence                */

process(_line) :-
 ( ( startsym(S,Code)              /* determine the start symbol */
   ; writeln('Start Symbol has not been defined')
   , fail
   )
 , ( top_down ->
```

```
      S =.. [Hd|SList]
    , append(SList, [Sentence,[]], NewSList)
    , NewS =.. [Hd,_|NewSList]
    ; assertuz(reduce(X,X))          /* for bottom up parsing */
    , NewS = sr(Sentence,[(S)t_])
    )
  , !
  , Time0 is cputime
  , lexical(_line, Sentence)
  , write('Sentence: '), writeline(Sentence)
  , abolish(numseed,1), assert(numseed(1))        /* reset the numseed */
  , Time1 is cputime
  , assert(time(Time1))
  , ( ( one_parse  -> NewS, process1(Code)                    /* one parse */
                ;  figg((forall NewS, process1(Code)))        /* all parses */
    )
    ; writeln('Analysis of Sentence fails')
    )
  , abolish(time,1)
  , Time is cputime - Time0
  , write('Total '), write(Time), writeln(' sec.')
) , nl, !.

process1(Code) :-                          /* Called after a successful parse */
 ( not var(Code) -> Code ;  write('Parse found ') )
 , write('  ')
 , Time2 is cputime
 , time(T1)
 , retract(time(T1))
 , assert(time(Time2))
 , Atime is Time2 - T1
 , write('in '), write(Atime), writeln(' sec.').

/*
  Look up the words in the lexicon.   The definition of "lookup" is supplied
  by execution of the "lexicon" FIGG command with no arguments.
*/

lexical([_lword|_rest], [_word|_sentence]) :-
   lookup(_lword, _word) , !
 , lexical(_rest, _sentence).

lexical([], []).

/* And now we have the utility routines used by FIGG and SAUMER. */

retract1(X) :-      /* retracts only one axiom */
   retract(X) , !.

append([A|B], C, [A|D]) :- append(B, C, D).
append([], X, X).

numseed(1).
```

```
numgen(_n)  :-              /* generate a unique number */
   numseed(_n)
 , retract(numseed(_n))
 , _newnum is _n + 1
 ; assert(numseed(_newnum)) , !.

reverse([_hd|_rest], _tmp, _rev)  :-        /* reverse a string */
   reverse(_rest, [_hd|_tmp], _rev).

reverse([], _rev, _rev).


askyes :-                  /* Succeeds if a word is entered that starts with "y" */
   readwd([_ch|_],_) , !
 , _ch = 121.

skel(Term) :-
   nonvar(Term) , functor(Term, _, N) , N > 0.

ucletter([_ch]) :- !,              /* is it an upper cse letter */
   integer(_ch)
 , 65 =< _ch
 , _ch =< 90.

addax(Rule) :- !, assert(Rule), write('New Axiom: '), writeln(Rule).

element(Var,_) :- var(Var), !.     /* Succeed if argument is a variable */

element(_elem, []) :- ! , fail.    /* is the argument an element of the list? */

element(_elem, [_hd|_rest]) :-
   _elem = _hd , !
 ; element(_elem, _rest).

/* drop the suffix from a word */

dropsuf(N1-Suf, N1) :- !.

dropsuf(N1, N1) :- !.

islist([]) :- !.

islist([_|_]) :- !.

/* Are they unifiable after dropping modifiers */

unifiable(Term1, Term2) :-
   (Term1 = Term1a@RA ; Term1 = Term1a),!
 , (Term2 = Term2a@RA ; Term2 = Term2a),!
 , Term1a =.. [Hd|List1]
 , Term2a =.. [Hd|List2]
 , unif_list(List1, List2).
```

```
unif_list([],[]) :- !.

unif_list([Hd1|Rest1], [Hd2|Rest2]) :-
   atom(Hd1)
 , atom(Hd2), !
 , Hd1 = Hd2
 , unif_list(Rest1, Rest2).

unif_list([_|Rest1], [_|Rest2]) :- !,
   unif_list(Rest1, Rest2).

/* Asserts the clause if it does not exist already. (assertunion) */

assertuz((Head :- Body)) :-
   clause(Head,Body), !.

assertuz(Clause) :-
   clause(Clause,true), !.

assertuz(Clause) :- !,
   assertz1(Clause).

assertu((Head :- Body)) :-
   clause(Head,Body), !.

assertu(Clause) :-
   clause(Clause,true), !.

assertu(Clause) :- !,
   asserta(Clause).

assertr(Rule) :-  calc_mode -> nl, writeln(Rule) ; assertz1(Rule).

assertz1(Rule) :- !,
   references(Refs)
 , abolish(references,1)
 , assertz(Rule,Ref)
 , assert(references([Ref|Refs])).

call1(X) :- X, !.

/*
  convert from the case mask notation of the AAA SAUMER lexicon to the
  "nom", "acc", or _.

   [N,x,_] ==> nom;  [x,A,_] ==> acc;  [N,A,_] ==> _.
*/

caseconvert([N,A,_], nom) :-
   var(N), atom(A), !.

caseconvert([N,A,_], acc) :-
   var(A), atom(N), !.
```

```
caseconvert([N,A,_], _) :-
   var(A), var(N), !.

/* Auxiliary predicates */

'$and'(true,_p,_p) :- !.
'$and'(_p,true,_p) :- !.
'$and'(_p,_q,(_p , _q)).

combine(true,_p,_p) :- !.
combine(_p,true,_p) :- !.
combine(_p,_q,(_p , _q)).

'$flatconj'(_a , _a) :- var(_a) , !.
'$flatconj'((_a , _b),_c) :- ! ,
  '$fc1'(_a,_c,_r) ,
  '$flatconj'(_b,_r).
'$flatconj'(_a,_a).

'$fc1'(_a,(_a , _r),_r) :- var(_a) , !.
'$fc1'((_a , _b),_c,_r) :- ! ,
  '$fc1'(_a,_c,_r1) ,
  '$fc1'(_b,_r1,_r).
'$fc1'(_a,(_a,_r),_r).

'$islist'([]) :- !.
'$islist'([_x|_y]).

'$append'([_a|_b],_c,[_a|_d]) :- '$append'(_b,_c,_d).
'$append'([],_x,_x).

/*
  And now a collection of various I/O routines.

  We first define the input routines.  Readwd gets every character
  up to the next special character and puts it into the first arg.
  The break character is put into the 2nd argument.  Readwd1 is
  to stop the recursion.  Readline forms a list of all the words
  input until a period is seen. Each word is maintained as a list,
  since this format is used some some morphological analysis routines
  that are floating around from the SAUMER system.
*/

readwd(_x, _last)              :-   repeat, get0(_ch), readwd1(_ch, _x,_last).

readwd1(-1, [], -1)            :-   read('end_of_file'), !.

readwd1(_ch, [], _ch)          :-   symbol(_ch) , !.

readwd1(_ch, [_ch|_x], _last)  :-   readwd(_x, _last) , !.

symbol(10).                    /* newline */
symbol(32).                    /* space */
```

```
symbol(33).                    /* exclamation mark */
symbol(44).                    /* comma */
symbol(46).                    /* period */
symbol(62).                    /* > the "Figg command" symbol */

readline(_line) :-
   readwd(_word, _last)
 , readline1(_word, _line, _last).

/* to allow periods after initials */

readline1(_word, [_word|_rest], 46)  :-
   ucletter(_word)
 , readline(_rest) , !.

/* to call Figg command processor (with a nice prefix) */

readline1([], Rest, 62) :- !,
   get0(10)
 , figg1('?> '), !
 , readline(Rest).

readline1([], 'end_of_file', -1) :- !.          /* end of file */

readline1([], [], 46) :- get0(10), !.           /* end of sentence */
readline1(_word, [_word], 46) :- get0(10), !.

/* ignore blanks */

readline1([], _rest, _)  :-  readline(_rest) , !.

readline1(_word, [_word|_rest], _) :-  readline(_rest) , !.


/* and for writing out a list, with a blank before each word, we have... */

writeline([_word|_rest]) :- write(' '), write(_word), writeline(_rest).

writeline([]) :- write('.'), nl.


/* For writing out trees */

writetree(N,Var) :- var(Var), !, tab(N), write(Var).

writetree(N,[Hd|Rest]) :- !, writetree(N,Hd), writetree(N,Rest).

writetree(N,[]) :- !.

writetree(N,Term) :-
   Term =.. [Hd|List]
 , list_of_atoms(List), !
 , nl, tab(N), write(Term).
```

```
writetree(N,Term) :-
    Term =.. [Hd|List]
  , nl, tab(N)
  , write(Hd), write('(')
  , NewN is N+4
  , writetree(NewN,List)
  , nl, tab(N), write(')').

list_of_atoms([]) :- !.

list_of_atoms([Hd|Rest]) :- !, atom(Hd), list_of_atoms(Rest).


writeln(X) :-  write(X) , nl.          /* A handy critter */
```

References

Aho, A.V. and Ullman, J.D. *The Theory of Parsing, Translation and Compiling, Volume 1: Parsing.* Englewood Cliffs, N.J.:Prentice Hall Inc., 1972.

Although this book describes a large variety of parsing methods, it was referenced for its detailed (and formal) description of a shift-reduce parser for context-free languages.

Berwick, R.C. and Weinberg, A.S. Parsing Efficiency, Computational Complexity, and the Evaluation of Grammatical Theories. *Linguistic Inquiry*, Spring 1982, *13(2)*, 165-191.

The authors argue that inefficiency is not a necessary consequence of non-context-free language parsing. They mention that grammar size can shrink with more powerful formalisms, and that the parsing efficiency depends on the grammatical format. There are numerous arguments from the cognitive and biological points of view.

Bien, J.S., Laus-Maczynska, K. and Szpakowicz, S. *Parsing Free Word Order Languages in Prolog*, pages 346-349. COLING 80, Proceedings of the 8th International Conference on Computational Linguistics, 1980. Also appears in J.S. Bien (Ed.), Papers in Computational Linguistics I, Institute of Informatics, University of Warsaw.

Modifications to metamorphosis grammars are described which allow "floating terminals." The modified system is used for some preliminary experimentation with the description of Polish syntax.

Clocksin, W.F. and Mellish, C.S. *Programming in Prolog.* Berlin-Heidelberg-NewYork:Springer-Verlag, 1981.

A thorough description of Prolog, complete with a tutorial, and numerous examples and exercises. There is one entire chapter devoted to "using grammar rules".

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

The author provides a formal introduction of metamorphosis grammars, along with a short introduction to Prolog. He supplies a method for using normalised metamorphosis grammars to parse or synthesize sentences. A compiler and a conversation system which use metamorphosis grammars are also described.

Dahl, V. *Current Trends in Logic Grammars.* Technical Report TR-83-2, Department of Computing Science, Simon Fraser University, 1983.

This survey of some logic grammar formalisms inspired the format of chapter two of Popowich's M.Sc. thesis. After a short introduction to logic grammars, the different formalisms are described and compared.

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

This second paper on Gapping Grammars continues where the earlier paper left off. It describes applications of these grammars in both formal language and linguistic domains. In particular, their use for describing coordination, free word order, and right extraposition is examined.

Dahl, V. personal communication. Dept. of Computing Science, Simon Fraser University, 1985.

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

This paper introduces gapping grammars and provides the motivation for their development. Two implementations are discussed, one which is general but inefficient, and another which is more efficient but less general.

Evans, R. and Gazdar, G. *The ProGram Manual.* Cognitive Science Programme, University of Sussex, 1984.

A reference manual for the ProGram system, which is a grammar development system based on generalised phrase structure grammars.

Gazdar, G. Phrase Structure Grammar. In P. Jacobson and G.K. Pullum (Ed.), *The Nature of Syntactic Representation*, D.Reidel, Dortrecht, 1981.

The first work to describe what are now called generalised phrase structure grammars. He proposes a variant of context-free grammars for use in the description of natural language. Phrase structure rules are used as "node admissibility conditions" on well formed trees. Properties of these grammars include: complex (structured) grammar symbols, metarules which operate on rules yielding new rules, and a semantic rule associated with each phrase structure rule.

Gazdar, G. and Pullum, G.K. *Generalized Phrase Structure Grammar: A Theoretical Synopsis.* Technical Report, Indiana University Linguistics Club, Bloomington Indiana, August 1982.

This paper provides a more detailed description of generalised phrase structure grammars than was provided in the earlier paper. Immediate dominance/ linear

precedence rules are introduced into the formalism, with the metarules now operating on these rules instead of on phrase structure rules. The slash categories of the earlier paper are replaced through the introduction of a "foot" feature into the complex grammar symbols. The semantic component of grammar rules is not discussed.

Greibach, S. and Hopcroft, J. Scattered Context Grammars. *Journal of Computer and System Sciences*, 1969, *3*, 233-247.

The authors formally introduce "scattered context grammars" and show that the languages described by these grammars are a subset of context sensitive languages. Their motivation for introducing these grammars was to eliminate the need for semantically useless nonterminal symbols whose only purpose was to "[transmit] information between widely separated parts of a sentence".

Griswold, R.E., Poage, J.F. and Polonsky, I.P. *The SNOBOL4 Programming Language*. Prentice-Hall Inc., Englewoods Cliffs, NJ, 1971.

This book describes the syntax of the programming language SNOBOL, and gives numerous examples of SNOBOL programs.

Hopcroft, J.E. and Ullman, J.D. *Introduction to Automata Theory, Languages and Computation*. Addison-Wesley Publisher Co. Ltd., 1979.

The first part of this book provides a good formal introduction to the languages of the Chomsky hierarchy. In particular, its definitions of grammars and their associated languages, and the description of "ambiguous" grammars were useful.

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.

The first part of the paper describes tree adjoining grammars, outlines their capacity to describe the "usual transformational relations", and illustrates their capability to describe various dependencies present in natural language. Then, the grammar formalisms mentioned in the title are compared based on whether or not they are powerful enough to describe some selected languages that exhibit various "patterns of dependencies".

Korfhage, R.R. *Logic and Algorithms*. John Wiley & Sons, Inc., 1966.

A section of this book gives a formal definition of a Markov algorithm. Several examples of these rule based algorithms that operate on strings are provided.

McCawley, J.D. *Everything that linguists have always wanted to know about logic but were ashamed to ask*. The University of Chicago Press, 1981.

An entertaining, but thorough, introduction to various logic formalisms, along with a look at their applications to linguistics.

Pereira, F.C.N. Extraposition Grammars. *American Journal of Computational Linguistics*, 1981, *7(4)*, 243-256.

The author introduces a logic grammar formalism called extraposition grammars as an extension of definite clause grammars. These grammars are shown to provide concise descriptions for left extraposition of sentential constituents. A processor of these grammars is also provided.

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

This manual describes variant of Prolog in which FIGG is written.

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

This paper provides a detailed description of definite clause grammars. The authors argue that these grammars can be used for efficient analysis of language. A comparison between definite clause grammars and augmented transition networks is also included.

Popowich, F. *Unrestricted Gapping Grammars.* Proceedings of the Ninth International Joint Conference on Artificial Intelligence, 1985.

Unrestricted gapping grammars are introduced, along with the FIGG implementation. The use of procedural control both to improve parsing efficiency and to restrict the language decribed by a grammar is advocated, and is supported by test results.

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

After providing a short introduction to unrestricted gapping grammars and FIGG, this paper describes the ID/LP-UGG conversion procedure. Some empirical results are also given.

Popowich, F. *SAUMER: Sentence Analysis Using MEtaRules.* Proceedings of the 2nd Meeting of the European Chapter of the Association for Computational Linguistics, March, 1985.

This work describes the SAUMER system, which is based on the early (1981) GPSG formalism. An outline of the syntax of the SAUMER Specification Language is provided, along with details about the implementation, and results from some applications of the system.

Popowich, F. *The SAUMER User's Manual.* Technical Report TR-85-3 and LCCR TR-85-4, Department of Computing Science, Simon Fraser University, 1985.

This manual describes how to use and SAUMER system, and provides some sample grammars.

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

This book is an easily understandable introduction to the Chomsky's Extended Standard Theory of transformational grammar. It includes details about Government and Binding.

Saint-Dizier, P. Long Distance Dependency Constraints in Gapping Grammars. INRIA Research Report, I.R.I.S.A. — Universite de Rennes, forthcoming.

Shieber, S.M. Direct Parsing of ID/LP Grammars. draft, 1982.

The author describes a modification of Early's algorithm which permits direct parsing of ID/LP grammars without their conversion into their corresponding context-free grammar. A proof of correctness is supplied, along with an argument that the time complexity of the algorithm is $O(n^3)$.

Shieber, S.M., Stucky, S.U., Uszkoreit, H. and Robinson, J.J. *Formal Constraints on Metarules*, pages 22-27. Proceedings of the 21th Annual Meeting of the Association for Computational Linguistics, June, 1983.

The authors outline some methods for constraining metarules to make their use with phrase structure grammars "computationally safe" They show the weaknesses of certain constraint methods, and suggest some directions in which "the ultimate solution" may lie.

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

Part of this short paper discusses a simple Prolog implementation of a shift reduce parser for context-free languages. This parser inspired the development of the shift reduce parser of FIGG.

Thompson, H. *Handling Metarules in a Parser for GPSG*. Technical Report D.A.I. No. 175, Department of Artificial Intelligence, University of Edinburgh, 1982.

Based on the early (1981) description of GPSG. The author proposes a method for making the application of metarules "computationally safe" and argues that the grammar should be "expanded" before any parsing is attempted. Also provided is a method for expanding the grammar.