ProGram - a Development Tool for GPSG Grammars

Roger Evans

ive Science Research Paper

no: CSRP 036

iversity of Sussex
ive Studies Programme
of Social Sciences

on BN1 9QN

ProGram - a Development Tool for GPSG Grammars*

Roger Evans,
Cognitive Studies Programme,
University of Sussex,
Falmer, Brighton,
East Sussex.
February 1985

Abstract

The "ProGram" Grammar Development System is a computational tool designed for linguists attempting to develop grammars expressed in the "Generalised Phrase Structure Grammar" (GPSG) formalism. The GPSG theory is complex enough that ensuring that any but a small grammar behaves as expected is a difficult task. ProGram aims to help overcome this problem by providing a computational representation for GPSG grammars, and tools which allow the linguist to explore the effects of different parts of the grammar on the analytic structures assigned by it. This paper describes the design and implementation of ProGram, and gives examples of its use on a small GPSG grammar.

This paper describes the result of research supported by the Social Science Research Council (UK), grant number HR 7829/1 to the University of Sussex (principal investigator: Gerald Gazdar).

* This paper will also appear in "Linguistics", late 1985.

ProGram - a Development Tool for GPSG Grammars

1. Introduction

Because natural languages are so rich, any serious attempt to provid realistic formal characterisation for even a small part of a language is f with the problem that the necessary complexity of the formalism makes theory difficult to test, evaluate and sometimes even express. By u computational aids in such cases, it should be possible to extend linguist's manipulative power, so that the development of interesting complex theories becomes much easier.

This paper describes the "ProGram" grammar development system [Evans Gazdar 1984] [1], an example of a computational tool designed to assist development of grammars expressed in the "Generalised Phrase Structure Gram (GPSG) formalism of Gazdar et al. (see, for example, [Gazdar and Pullum 1 [Gazdar et al. 1985]). GPSG aims to formally characterise the synta structure of natural languages [2], and has been developed to a point whe large number of interesting syntactic phenomena can be accommodated in integrated formal framework. But, unsurprisingly, its extensive formal cove is matched by its complexity. A typical [3] GPSG grammar has seven or e distinct components which interact in a complicated fashion, providi description of a language at a very high level. Although theoret development can continue by focussing attention on only a small portion of formalism at a time, confident prediction and validation of the cor syntactic coverage of any non-trivial GPSG grammar (and hence the gl validity of the theory) is difficult and, if undertaken manually, prone error. The ProGram system is designed to help overcome these problems, allo more thorough validation of experimental grammars, and at the same time ma it easier to write large grammars using existing GPSG techniques.

2. System Objectives

The principle design objective for the ProGram system was to produc GPSG grammar development tool which was usable and useful. For the system t usable, it needed to be accessible to linguists with little computati experience, easily portable from one machine or operating system to anot and practical (in particular, in terms of speed and size). These points largely implementation issues which will be discussed in more detail be For the system to be useful, it needed to provide facilities which actu help a linguist attempting to develop a grammar and which are convenien use. Such facilities can be loosely divided into two groups: facilities specifying and modifying a grammar using ProGram, and facilities for exami and testing the grammar once it has been specified.

The first step in the design of the grammar specification language was decide exactly which version of the GPSG formalism to use. Although there always been widespread informal agreement about the general shape of G tying the theory down to the formal detail which a computational tool requi in a manner which satisfied everyone concerned, could have posed a prob However, the appearance of [Gazdar and Pullum 1982] (henceforth GP82), w gave a thorough, formally adequate (more or less) and largely uncontrover account, provided an independent formulation of the theory suitable for us a basis for ProGram.

The most recent account of GPSG, [Gazdar et al. 1985], differs from GP8 in various ways. Most of these are details of convention definitions, etc which can be ignored for present purposes. The most significant change however, is worth noting here. GP82 (and ProGram) makes use of tree-structure categories and features. These are more fully described in section 4 below. [Gazdar et al. 1985], categories are expressed as feature sets (rather tha feature trees), allowing greater flexibility, but this facility is not provide in ProGram.

GP82 uses various notational conventions to describe different parts of grammar and some of these, such as the notation for feature definitions, a well described in the paper. Others are not: the interpretation of the 'hea category marker, for example, is less well specified, while the abbreviation category specifications and the implicit correspondence of categories in meta rules are examples of very vague conventions which appeal to linguist informal usable. The ProGram system could simply have ignored all of the extra notations and assumed only the rigorous formal descriptions actual required. This would have lead to a rule expressed informally as (2.1) (this GP82 example (32)) being expressed for ProGram as something like (2.2). [4]

```
(2.1)  < 5; VP --> V,NP >

(2.2)  [cat [bar 1] [head [major [verb +] [noun -]]]] -->
           [cat [bar [lexical 5]] [head [major [verb +] [noun -]]]],
           [cat [bar 2] [head [major [verb -] [noun +]]]]
```

Such an approach is clearly undesirable, and indeed hardly satisfies either the 'usefulness' criteria above (it is neither helpful nor convenient). T ideal case would allow the linguist to use all the notational devices, form and informal, that are normally used. Grammar specification in ProGram w designed with this in mind; at every point the aim was to make a grammar lo as much like the GP82 notation as possible. To achieve this, the inform conventions needed to be formalised as far as possible and abandoned otherwis While the aim has not been perfectly achieved in all cases, the exampl throughout this paper give some indication of the success, and the topic discussed more fully below. Compare (2.3), a typical ProGram immedia dominance rule, with (2.1) and (2.2).

```
(2.3)  5: vp --> v,np.
```

Having designed a suitable specification language for grammars, remained to decide what facilities for manipulating a grammar should provided. What sorts of things might a linguist want to do using the system? GPSG grammar is a high level specification of a language which is mediat through an ordinary context-free grammar [5] and one tends to think of grammar in terms of the context-free rules it generates, rather than direct in terms of the strings it allows. Thus the system should provide at the ve least some facilities for examining this underlying context-free grammar. straightforward listing of all the rules would not be very informative as the grammars are large and complex; some form of interactive exploration would more appropriate. The simplest approach would be to allow the linguist to a whether particular context-free rules could be derived from the grammar specified. The linguist could concentrate attention on likely problem areas the grammar, where spurious rules might be generated, or expected rul omitted.

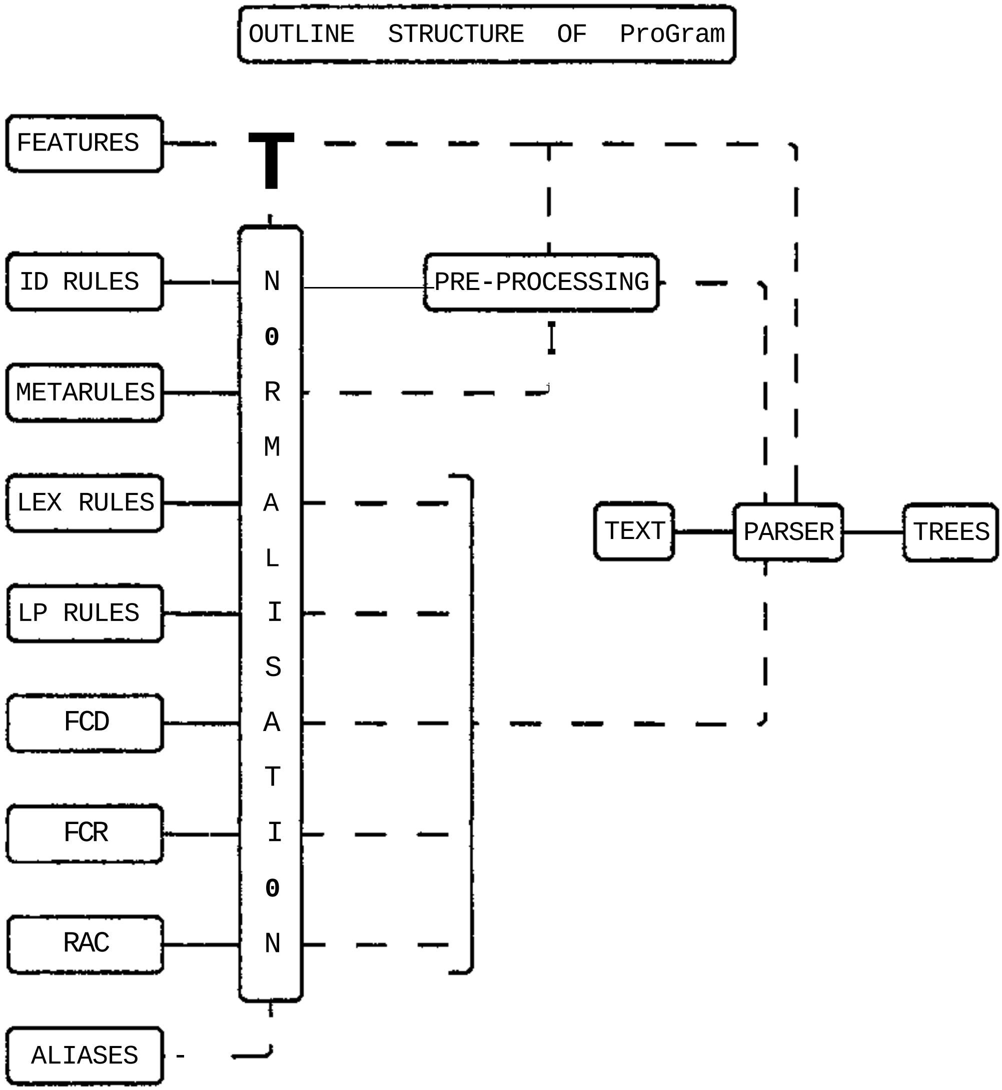
There are two main difficulties with this approach: firstly, it would
difficult and time consuming for the linguist to check out rules to the po
where she could be confident the grammar contained exactly the expected rul
and secondly, given a 'correct' grammar, it would still be difficult to pred
exactly what structures (if any) it assigns to given strings. This lat
problem suggests that the system is bound to need some sort of pars
facilities, even though parsing is not its main function. And this suggests
alternative approach where the parser is used as the principle diagnostic to
rather than as a peripheral utility.

Suppose the system provided a parser which produced analyses of strings
generating context-free rules as needed from the GPSG data. Rather than ask
about a context-free rule explicitly, the linguist attempts to parse a str
whose analysis depends on the rule. A successful parse means that the rule
in the grammar. In fact, it means that several associated rules (all the oth
used in the analysis) are in the grammar too. Conversely, ungrammatical phra
could be used to ensure that certain rules were not in the grammar. M
generally, the linguist would be able to test several different aspects of
grammar, and their interactions in assigning analyses, all at the same ti
and the results of the tests would be available in a relevant context (
would know not only that a rule is in the grammar, but also where in
analysis it can occur). Furthermore, grammar testing is achieved simply
providing test phrases, rather than making use of a complicated r
specification language.

Using a parser in this fashion is a common way of informally testing
grammar, and it often suffers from a lack of control and diagnos
information. Thus, for example, it is usually not possible to force the par
to attempt a particular analysis, or find out why an expected analysis did
occur. Where the context-free rules the parser is using are not immediat
available to the linguist, but instead are being generated from higher-le
data as the parser proceeds, such problems become more severe. But a speci
purpose parser, designed for the task of grammar development rather t
analysis per se, can incorporate extra features to overcome these proble
resulting in a useful development tool. It is on this view of the problem t
ProGram is based. The central component of the system is a parser wh
interprets a GPSG grammar and provides analyses of phrases, and which can
controlled by the user to allow more precise and detailed examination of cho
aspects of the grammar.

3. Overall Design

The following diagram shows the overall layout of ProGram. The boxes d
the left hand side, and the one marked TEXT, are all data which the user m
provide in some form. These are discussed in more detail in section 4. The
marked TREES contains the parse trees that are produced, and the remain
three boxes are the main functional units of the system.



Key: ———— Main data for process

_ — _ Background data for process

(All data flows left to right)

The basic procedure for developing a grammar is as follows. Having crea the initial data files for the grammar, the first task is to atte normalisation. Most of the data modules have to be translated into an inter format for use with the parser. The normalisation process carries out t translation, taking account of the built-in and user-specified notatio conventions (see ALIASES in section 4), and checking that all the data it

are well-formed and, as far as possible, sensible. The user is inform
problems encountered and the culprit files must be edited and re-n
Once all of these generally simple errors have been removed, the
data is ready for use. Most of the data modules are used direc
parser. The exceptions are the Immediate Dominance (ID) rules and
ProGram carries out metarule expansion before parsing; the parser on]
expanded set of ID rules. Furthermore, the Control Agreement Principl
Head Feature Convention [6] are also compiled into the ID rules befor
These three processes are collected together in the PRE-PROCESSING bo
diagram. After normalisation then, the user may wish to expand the
rules (or not, if there are no metarules, or they are being ignored
present) and carry out the convention processing on them. The ID
then also be ready for the parser.

Although some minor errors will have been detected in these
processes, the main diagnostic tool is the parser. Basic use of the
very simple: the LEX RULES module contains a lexicon, and the user
phrases made up of words in this lexicon to be parsed. The parser can
three different modes, corresponding to different levels of control,
mode, the parser runs completely independently, and produces one
parses for the phrase. The user may set various tracing switches to
parsing process, but has no control over it. MONITOR mode allows t
ratify all the major decisions the parser makes. After a possible c
been located, the user is informed and asked whether the parser sho
up the choice. The user can agree to the choice or ask for an a
choice (if any) to be sought. In this way, the actual calc
possibilities is still automatic, but the user can direct the parser
particular lines of search and so force particular analyses quickly,
mode the user specifies exactly what to do at each choice point, (for
which ID rule to attempt to apply). Whenever a choice fails (that is,
applied for some reason), the parser prints out diagnostic messages t
user exactly what went wrong (for example, the Foot Feature Principl
Thus the user tells the parser exactly what to do, and may force it
paths it would not normally follow at all, and so can find out what
the theory is blocking expected analyses.

ProGram commands can be given with virtually no knowledge of
all. For example, the command to parse phrases in a file 'sentence
the trees in 'treesl' might be (3.1), while (3.2) is the command to
set of ID rules from a file called 'idrules', putting the expanded se
into a file called 'idrules2', and using the file 'mrules' as the sou
(normalised) metarules.

(3.1) parse from sentences to treesl.

(3.2) expand from idrules to idrules2 using mrules for normed
metarules.

On the other hand, because this command structure is achieved
suitably defined Prolog operators, the more knowledgeable user can u
ordinary Prolog commands, and easily incorporate new commands which m
the existing ones.

In addition to the utilities themselves, ProGram documentation
three forms. The main documentation takes the form of "The ProGr
[Evans and Gazdar 1984], which gives detailed instructions for
system. The system itself provides a quick help facility consistin

explanations of most of the commands and facilities in ProGram. Because this i
built in to the standard system, it is always available regardless of loca
customisations etc. The more extensive 'help files' are also provided with th
system, and can be read at the terminal using a suitable file-reading o
editing package. In POPLOG, the screen editor, VED, is used to access thes
files in the same way as it is used for POPLOG help files. In other systems, i
may be more convenient to have printed copies of them, or to use the manual
which covers virtually the same material.

4. A ProGram Grammar

This section describes what a ProGram grammar actually looks like, i
other words, what goes into each of the nine boxes down the left-hand side o
the diagram above. The examples are taken from the example grammar which i
used in section 5 and is repeated in full in the appendix. Although thi
grammar is quite simple, it incorporates ID/LP format, the Head Featur
Convention, Control Agreement Principle, slash categories and metarules. Thi
section will refer to a few features of the implementation language, Prolog
for full details see [Clocksin and Mellish 1981].

FEATURES

The most basic data for the grammar is the specification of the featur
syntax, the FEATURES box. In GP82 a feature is a tree-structure with a root
labelled with the feature name, and sub-trees which are other features. Th
feature data specifies for each feature its name and the names of permissibl
sub-features. The notation used by ProGram is virtually the same as in GP82, s
that to specify a feature MINOR which takes a sub feature AGR, and either VFOR
or CASE (but not both) one would put (4.1) in the FEATURES file.

```
(4.1) feature [minor agr {vform case}].
```

A FEATURES file will have many statements of this sort, defining the range o
possible tree structures for features. Many of these features have specia
significance to ProGram (for example HEAD is used by the Head Featur
Convention), so the overall structure of the feature system is usually simila
to that given in GP82. The feature specification for this example grammar i
given in (4.2).

```
(4.2) feature [root   cat  foot].
      feature [cat    bar  head].
      feature [bar    {lexical  1  2}].
      feature [head   major  minor].
      feature [major  {v  n  d  p}].
      feature [minor  agr  {vform  pform  nform}].
      feature [agr    {sing  plur}].
      feature [vform  {fin  pass}].
      feature [pform  {by  to}].
      feature [foot   cat].
```

Additionally, the FEATURES data must specify which of these features i
permissible as an actual syntactic category. (4.3) specifies that the featur
ROOT is syntactic category, so that (4.4) is a valid category but (4.5) is not

(4.3) syncat root.

(4.4) [root [cat [bar 2] [head [major v]]] [foot]]

(4.5) [bar 2]

The feature specification gives ProGram the information required to d<
check categories in the other parts of the grammar. It also tells Pr<
possible other features there can be in a category specification whicl
complete. Thus (4.7) may be just a more detailed instance of (4.6).

(4.6) [root [cat [bar 2] [head [major v]]]]

(4.7) [root [cat [bar 2] [head [major v] [minor [agr plur]]]
[foot [cat [bar 2] [head [major n]]]]]

ID RULES AND ALIASES

ID rules are specified in terms of syntactic categories (in the
instances of the ROOT feature). The basic format of an ID rule is as
and (4.9) is an example.

(4.8) <name>: <cat> —> <cat>, ... , <cat>.

(4.9) s: [root [cat [bar 2] [head [major v]]]] —>
[root [cat [bar 2] [head [major n]]]],
[root [cat [bar 3] [head [major v]]]]*

This is a traditional s —> np,vp rule, but expressed assuming th<
specification above. As was noted above, such rules are a little unw:
ProGram provides a mechanism to make them more manageable, tl
specifications. An alias is an expression like (4.10).

(4.10) alias(s, [root [cat [bar 2] [head [major v]]]]).

This means, "if ever I write 's[1] as a category specification, I mean [
[bar 2] [head [major v]]]]". If in addition there are aliases such
and (4.12), rule (4.9) can be written as (4.13).

(4.11) alias(vp, [root [cat [bar 1] [head [major v]]]]).

(4.12) alias(np, [root [cat [bar 2] [head [major n]]]]).

(4.13) s: s —> np,vp.

This has exactly the same effect as the original (note that the "s" b<
colon is not affected by the alias - ProGram knows this is the name r;
a category).

More complex aliases can be written by using a functional notati<
(4.14). [7]

(4.14) alias(v(l), [root [cat [bar 1] [head [major v]]]]).
alias(v(2), [root [cat [bar 2] [head [major v]]]]).

Notice that these two aliases give alternative abbreviations for the

vp" above. This means that (4.9) could now be written as (4.15), making th
-bar information of the categories more explicit.

```
(4.15) s:  v(2) --> np, v(l).
```

Aliases make grammar writing far easier and they can be used wherever
ull category is required. The full set of aliases for the example grammar ar
;iven in the appendix and will be used henceforth, without comment except wher
he meaning is unclear.

Equipped with these aliases, some of the ID rules for the example gramma
lay be expressed as in (4.16).

```
(4.16) vpl: v(l) -->  h.
       vp2: v(l) -->  h,n(2).
       vp3: v(l) --> h,n(2),p(2,to).
       vp4: v(l) -->  h,v(l,pass).

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

       np2: n(2) --> h.
       nom: n(l) --> h.
```

There are several comments to be made about these rules. Firstly, the us
>f the alias "h". ProGram takes the definition of "head daughter category" t
>e "a daughter with unspecified MAJOR feature value and minimal bar level"
'he alias "h" stands for the category in (4.17) in which MAJOR has no sut
'eatures.

```
(4.17) [root [cat [bar lexical] [head [major]]]]
```

Jince all the other aliases (apart from h(l) etc. - higher level heads) c
specify the MAJOR feature value, use of "h" will generally guarantee prope
selection of the head daughter. A second feature of an ID rule is the rul
tame. This is used for lexical subcategorisation; lexical rules (see below) ar
llso labelled with ID rule names and an ID rule can only introduce lexica
terns occurring in a lexical rule of the same name.

The example grammar also contains two other ID rules, shown in (4.18).

```
(4.18) s: v(2) --> N2,H1    where N2 is n(2),
                                  HI is h(l),
                                  N2 controls HI.

       npl: n(2) --> DET,H1  where DET is d(lexical),
                                  HI is h(l),
                                  HI controls DET.
```

iere, Prolog variables N2,H1 and DET are being used to specify contrc
igreement. The "where" clauses must define the categories corresponding to *tl*
variables, and the control relationship between them, (For full details, se
[Evans and Gazdar 1984]). These annotations ensure that the categories satisf
:he Control Agreement Principle, that is, that their agreement features (th
feature AGR) are identical whenever the rule is applied.

METARULES

The example grammar contains the three metarules shown in (4.19). T implement a simple passive, the "slash termination metarule" for introdu holes and "that-less relatives" (see [Gazdar et al. 1982]). Note the use slash notation (A/B) which specifies category A with FOOT feature set to CAT of category B. This is an advanced but very convenient use of alia discussed more fully in [Evans and Gazdar 1984].

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


       stm1:        (C1 --> C2, ...    where C1 is [root],
                                             C2 is [root,[cat,[bar,2
              ==>
                     C1/C2 --> ...  .


       relcl:       (N1 --> ...        where N1 is n(1))
              ==>
                     N1 --> ... , v(2)/n(2).
```

Each metarule has a name and consists of two nameless ID rules which as above, except that the daughter lists may include "...", denoti "multiset variable" which can match any number of arbitrary categories in a rule. In (4.19) there are further instances of Prolog variables. In STM1 RELCL variables are introduced simply to ensure that identical catego appear in the input and output ID rules to the metarule. Notice that C defined as just [root] - that is, any category whatsoever. Similarly C2 is category of bar level 2. In PASS, the variables are used to specify that categories "match". This implements the informal notion of correspond between categories in the two rules. In this case the two mother categories forced to be the same (even where they are not explicitly specified in rule), except that the second one is passive. In general, a "matches" cl ensures that categories are identical except where they differ explicitly the metarule.

LEXICAL RULES

The lexicon contains a mapping from categories to lexical items. A typ lexical rule looks like (4.20).

```
(4.20) vp2(fin): v0(fin) ->- sees.
```

This sets up the word SEES as an instance of a v0(fin) category. The rule is used for lexical subcategorisation: such a v0(fin) can only be introduce a rule also labelled VP2 (above, the transitive vp rule). The extra (fin) the rule name allows the user to distinguish this lexical rule from ano (vp2(pass) for passive verbs) which may also be used in VP2. The full lex for this grammar is given in the appendix.

LINEAR PRECEDENCE RULES

The ID rules do not specify the orderings on the daughter cat
Instead this is done independently by the linear precedence (LP) ru
example grammar contains the LP rules shown in (4.21).

```
(4.21) [root [cat [bar lexical]]] <<
                { [root [cat [bar 1]]]
                  [root [cat [bar 2]]]  }.
       n(2) <<  v(1).
       [root ~foot] << [root [foot cat]].
```

The use of curly brackets in (4.21) is a notational abbreviation to
group of mutually unordered categories to be treated like a single cate

LP rules need not specify complete categories, and an orde
daughters is only valid if all the LP rules which match pairs of daugh
observed. Thus the first LP rule in (4.21) states that lexical categor
precede non-lexical (that is, bar level 1 or 2) categories, the sec
noun-phrases precede verb-phrases and the third that slashed categories
follow non-slashed categories (see [Farkas et al. 1983]). Note here th
~FOOT to mean "FOOT is unspecified".

FCD, FCR AND RAC RULES

The final three components of a ProGram grammar are Feature Coe
Defaults, Feature Co-occurrence Restrictions and Root Admissibility Con
The first two of these are as described in GP82: FCD's specify default
for categories in ID rules which have to be met by valid instantiat
FCR's specify constraints on the structure of legal categories in gener
example grammar contains one FCD and three FCR's as shown in (4.22).

```
(4.22) fcd( vform, [foot], fin, free ).

       fcr( major, [foot], n,  minor, [foot], nform ).
       fcr( major, [foot], p,  minor, [foot], pform ).
       fcr( major, [foot], v,  minor, [foot], vform ).
```

The FCD states that the default value for VFORM (but not the VFORM whic
sub-feature of FOOT), is FIN for lexical categories, and unconstra
phrasal categories. The FCR's ensure that the MINOR features VFORM,
NFORM only occur with appropriate MAJOR features. This excludes such il
categories as passive nouns.

The RAC's specify additional featural constraints on root catego
any complete parse tree produced by ProGram. They allow the user to
that certain parses are not interesting (for example, very unders
categories), perhaps because they could never be part of a larger ph
the simple example grammar there are no RAC's.

5. Examples using the grammar

This section provides examples of ProGram being used to expl
demonstration grammar given in the appendix. The examples will demonst
head feature handling, metarules, foot features, and FCD's. The gramm
includes control agreement and FCR's, but because of the simplicit

lexicon, these will not have any overt effect in the examples. The g given is more or less correct, in that it does what the author inten and in the examples ProGram will be used to explore different aspect rather than attempt to debug it further. There are, of cou inadequacies and gaps in its coverage, even of the small subset o permitted by its lexicon.

First of all, a brief description of the grammar. The specifications follow GP82 fairly closely. There are four MAJOR featu corresponding to verbal (V), nominal (N), prepositional (P), and dete categories, and the grammar employs a simple three level X-bar sys head features are AGR, which is used for the CAP, but will not be here (all lexical items are singular), and VFORM, PFORM, NFORM. mutually exclusive, and specify type-specific data - for verbs, passive, for prepositions, the actual preposition (like a termi feature) and for nouns, nothing. ~NFORM is used for determin simplicity. Three FCR's enforce the correct MAJOR/MINOR feature corre There are nine ID rules and three metarules as described above, expansion, the grammar contains twenty rules derived as follows:

Initial rules	One metarule	Two metarules
nom	relcl(nom)	stm1(pass(vp2))
np2	stm1(pp)	pass(stm1(vp3))
np1	pass(vp3)	stm1(pass(vp3))
pp	stm1(vp3)	stm1(pass(vp3))
vp4	stm1(vp3)	
vp3	pass(vp2)	
vp2	stm1(vp2)	
vp1		
s		

Notice that STM1 applies in two ways to VP3 (slashing either N(2) or and PASS(VP3) (slashing either P(2,to) or the optional Unfortunately, this gives rules with the same names, which may confusion. The linear precedence rules are quite restrictive (Englis strictly ordered): the only scope for variation is in VP3 where the the P(2,to) are unordered. Thus the grammar accepts both "give fido t "give to kim fido". Finally the single FCD blocks passive verbs f introduced except where they are specifically requested, namely in r are output from PASS.

Example 1: the FCD.

The single word SEEN has three parses, as a simple verb, as a pa phrase (as in "kim is seen") and as a passive verb phrase with a slas in "by fido kim is seen", although the grammar does not handle topic The FCD requires that SEEN cannot occur in the active verb positio automatic mode, a phrase such as "kim seen fido" has no parses. control mode, ProGram announces that it is the FCD which ensures this

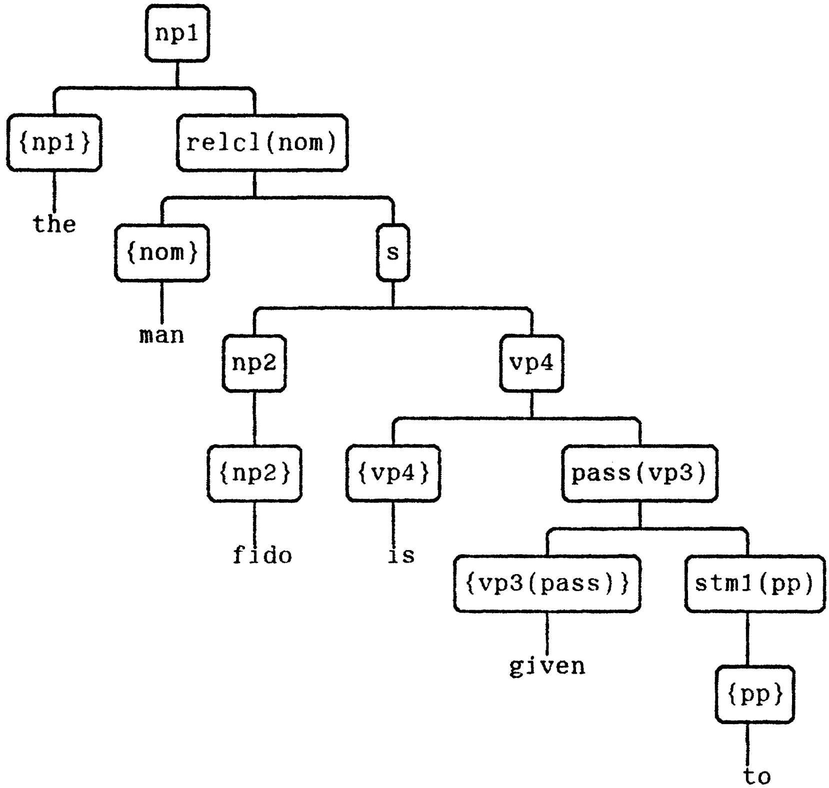
Parsing in CONTROL mode. |: kim seen fido	The initial sentence to p
--- LEX rule for FIDO |: np2 Located LEX rule np2	Select the lexical rule f

```
Current segment: {np2}                          The {} mean 'lexical item'.

--- ID rule to consume {np2} |: np2             Select ID rule NP2 to make an NP
    Located ID rule np2


Current segment: np2                            NP created successfully.

--- ID rule to consume np2 |:                   Typing nothing causes ProGram
    Leaving category np2 alone.                 to move left to the next word.

--- LEX rule for SEEN  |: vp2                   Select lexical rule for SEEN.
    Located LEX rule vp2(pass)                  ProGram reports the exact name.


Current segment: {vp2(pass)} np2                There are two categories,V and NP

--- ID rule to consume {vp2(pass)}  |: vp2  Try to build a VP ...
    Located ID rule vp2

Warning: FCD failed.
Involving: vform in [root, [cat, [bar, [lexical, [vp2]]], [head, [major,
     [v]], [minor, [agr, [sing]], [vform, [pass]]]]], [foot, [~]]]


*** No (more) possible matches for rule.
*** ID rule vp2 not applicable.                 ... and fail.
```

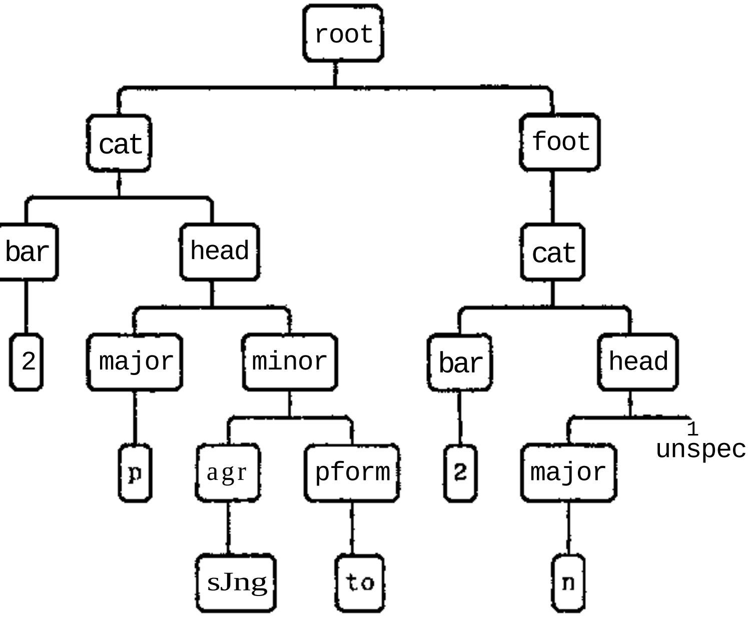
Disaster has struck: VP2 does not specify the VFORM feature for its head daughter. Because the head is lexical, defaults apply to the HEAD feature, so VFORM must be FIN. This conflicts with the vp2(pass) selected for SEEN. Note that the ID rule PASS(VP2) does specify VFORM - it is specified on the mother and transferred by the head feature convention, so the default is never checked in that case.

Example 2 - A parse tree.

ProGram displays parse trees at two levels, the actual parse tree structure and the feature tree structures for the categories. For example, the phrase "the man fido is given to" produces the following parse tree:



There is a departure from the usual conventions here: nodes are la the mother category, but by the name of the ID rule applied. Th the categories are themselves feature trees and so cannot easily in a compact form. Careful labelling of ID rules, however, ensu labels are sensible. Notice in the tree above that instances of ou three metarules occur, STM1(PP), PASS(VP3) and RELCL(NOM). obvious from this tree is that four of the nodes have a slashed NI on the S by the relative clause rule and consumed by the STM1(PP) one needs to look at the feature structure of a category. ProGram the feature tree of any node on request. As an example, the noc associated with a PP/NP category whose precise structure is as fo



lotice that the MINOR features of the slashed NP were never specified, denote n the tree by UNSPFC. It is also clear that attempting to use such a tree as tode label is rather impractical.

Ixample 3 - the Foot Feature Principle.

The parse tree given above is the only parse for the phrase "the man fid s given to" according to the example grammar. However, its sententia counterpart, "fido is given to the man", has two parses, as a straightforwar entence, and as a sentence with a slashed PP (the optional PP[by] i >ASS(VP3)). This second parse does not occur in the nominal form because th oot feature principle blocks it. It requires that "given" be consumed b JTM1(PASS(VP3)) as a VP/PP[by]. This rule does not require an object NP (PAS ias deleted that) but it does need a PP[to]. In this phrase, there is a PP[to] >ut it has a slashed NP as well. The foot feature principle must transfer thi NP to the mother, but it is incompatible with the /PP[by] already there. Thu ;he principle is violated. The following trace starts after the PP/NP ("to" tas been built, and "given" is being introduced.

```
--- LEX rule for GIVEN  |: V£3            Select VP3 for GIVEN
    Located LEX rule vp3(pass)
```

```
Current segment: (vp3(pass)} stml(pp)

____ID rule to consume {vp3(pass)> |: vp3   Select a VP3 rule to build a VP
    Located ID rule vp3
— Ok? |: n                                  There are six VP3 rules, we fore
                                            the parser to try a particular
    Located ID rule pass vp3                one (in fact, the last one).
— Ok? |: n

    Located ID rule stml vp3
— Ok? |: n                                   (reject all these)

    Located ID rule stml vp3
— Ok? |: n

    Located ID rule pass stml vp3
— Ok? |: n

    Located ID rule stml pass vp3          This is the one.
— Ok? |: y

Warning: FFP failed.
Involving: [foot, [cat, [bar, [2]], [head, [major, [p]], [minor, J, [pform,
     [by]]]]]]

*** No (more) possible matches for rule.
*** ID rule stml pass vp3 not applicable.
```

6» How ProGram works

The aim of this section is to give a brief description of some aspects the actual mechanics of the ProGram system. The bulk of ProGram is writtei Prolog [Clocksin and Mellish 198]] and conforms to the DEC10 Prolog stanc [Pereira et al. 1978]. Prolog was chosen as the implementation language bec< of its relative portability and popularity, particularly in Europe, increasingly elsewhere [9]. The core DEC10 compatible system contains all main facilities, but the POPLOG [Hardy 1982] version also includes extra PC libraries which redefine and augment some of these features, providing grapl displays and editing facilities which could not easily be added to the stanc system. As in section 4, the discussion will occasionally refer to s features of Prolog, and brief explanations will be given where necessary, interested reader is referred to [Clocksin and Mellish 1981] for full detai:

DATA NORMALISATION

As discussed in section 3, data normalisation transforms the grammar < into a standardised internal format, translating aliases and carrying superficial error checking, the examples show how valuable this extra layer translation is. The central task of the normaliser is the basic translatioi category expression. The formal representation of a feature (as defined GP82) is already a bit cumbersome for comfort. ProGram's intei representation is in fact slightly more cumbersome, because it requires 1 all coefficients of a feature be present, and in a fixed order. Omitted feal coefficients are represented by Prolog variables. Thus, given the feat specification of the examples above, the category in (6.1) ((representation) has alternative forms as in (6.2), but a unique ProGram foru n (6.3).

```
(6.1) [root [cat [bar 2] [head [major v]]]]

(6.2) [root [cat [head [major v]] [bar 2]]]
      [root [cat [bar [2]] [head [major [v]]]]]

(6.3) [root [cat [bar [2]] [head [major [v]] M]] F]
```

n (6.3) M and F are Prolog variables representing the absent MINOR and FOO features, and the ordering of coefficients is fixed to be the same as in th eature specification. Note also that features not explicitly defined in th eature specification are assumed to take no coefficients. Thus 2 and V hav he implicit definitions in (6.4).

```
(6.4) feature [2].
      feature [v].
```

he reason for standardising features thus is to make comparing and matchin eatures straightforward. Indeed, feature unification (as defined in GP82 ecomes just Prolog unification. In order to achieve this standard form 'roGram picks apart the category structure given, interpreting the aliases a ppropriate, and rebuilds the category with the translated coefficient nserted in the correct order. In the process it checks for unknown an ncorrectly positioned features etc..

Most of the other normalisation processes are re-arrangements of the dat nto a more convenient form. For example, the BAR coefficient is extracted fro jach category and stored separately, so that it is easy to tell whether th ategory is lexical. And in ID rules, the rule name is attached as an extr ;ub-feature value of LEXICAL (the lowest bar level). This is also done to th ategories in the lexicon rules, so that lexical subcategorisation is otherwis LUtomatic. A lexical category with LEXICAL feature VP1, say, will only match ule category with the same LEXICAL value, that is, a rule category in an I 'ule labelled VP1.

:ULE PRE-PROCESSING

The pre-processor takes the original set of ID rules and produces a ne ;et which have been closed under metarule expansion, and which include th :onstraints required by the head feature convention and the control agreemen convention. It is this latter set that are used by the parser, whic :onsequently can ignore these aspects of rule instantiation completely. Th eason these particular aspects are dealt with in this way is a combination c ffjciency and convenience.

Following Thompson [Thompson 1981], ProGram is designed on the view tha >re-expansion of the ID rules using the metarules is more sensible that oi^ ;he-fly expansion. The overheads of incrementally calculating and saving new I *ules are far outweighed by the simplicity of pre-expansion, and the attendan ibility to examine the expansion process in isolation, before any parsing i carried out.

The internal form of the metarules makes ID rule expansion particular1 >asy: if an ID rule can be successfully matched against the pattern of metarule, then Prolog variables in the pattern will be given values from the I rule. Some of these variables also occur in the target of the met once they have been set (by matching the pattern) the target rule can read off.

Consider the following example (using a simplified category nota metarule in (6.5) has an internal representation something like (6.6)

```
(6.5)  pass:       vp --> np, ...
               =>
                   vppas --> ...

(6.6)  metarule( pass,  rule(N,vp,[np|W]),  rule(pass(N),vpp
```

The Prolog variable N represents the name of the input rule, and W the multiset variable (... in (6.5)) [10]. Similarly, the ID rule in an internal representation like (6.8):

```
(6.7)  vp3: vp --> np, v, pp

(6.8)  rule(vp3, vp, [np, v, pp]).
```

Applying (6.5) to (6.7) consists of matching (6.8) with the first rul and in this simple example, this can be achieved by Prolog's un resulting in N being set to vp3 and W to [v, pp]. Thus the second rul becomes (6.9), which is just the internal form of (6.10):

```
(6.9)  rule(pass(vp3),vppas, [v,pp]).

(6.10)  pass(vp3): vppas --> v, pp
```

Finally (6.10) is exactly what one would expect the output of metaru to be, given the input ID rule "vp3".

The actual matching process in ProGram is more complex in that allow for regular expression operators and arbitrary orderings o categories. It also includes handling of the "matches" clauses, whe than unifying categories together in the usual fashion, a best-fi carried out - the categories must coincide except where differ actually specified. And, in accordance with the theory, metarule applied to lexical ID rules (ID rules with lexical heads), and ma applied to their own output. In (6.9) the name records the derivati of the rule, and this is used to ensure this latter constraint.

The processing of HFC and CAP in the pre-processing stage amount compilation of these principles into the rules. Unlike the other p these two can be implemented as Prolog's unification, which means the encoded directly into ID rules in such a way that they become simply of a rule just as "having two daughter categories" is a feature of a example, the mother and head daughter's HEAD features must be iden checks that they are identical where specified, and ensures that w variable is used to denote 'unspecified', the same variable is us mother and daughter (for that particular feature). This ensures that only ever be given the same value, so that the HEAD features are gua be the same. (Compare with the use of variables in aliases in section is, in fact, a mechanism commonly used, particularly for agreement ha Prolog parsing systems, for example in definite clause grammars [Pe Warren 1980]. Letting the Prolog system do the work, rather t

specific code in the parser, makes the handling of these principles effici
and totally transparent to the parser itself.

PROGRAM'S PARSER

Of the various aspects of the GPSG formalism, there are three wh
particularly suggest that the context-free grammars generated by GPSG gramm
are in general best suited to bottom-up parsing techniques. The basic issue
whether the mother category or the daughter categories are the best select
of the correct context-free rule to apply [11].

Unless the linear precedence rules impose a total ordering on
daughters of every rule, there will be rules which permit some freedom am
daughters. Such rules generate several context-free rules with identi
mother categories, but different daughter-lists. Thus a GPSG grammar wh
makes full use of ID/LP descriptions will contain in its underlying conte
free grammar many groups of rules with the same mother category, and so
mother category will be a bad selector of the correct context-free ru
Conversely, however, ID/LP does not significantly affect the extent to whic
given daughter set selects a context-free rule.

Similarly, the Foot Feature Principle in GPSG requires that the mothe
foot increment is the unification of the daughters' foot increments.
details of this statement are not important here, the basic point to notice
that part of the mother is specified by unifying (an operation similar to
union) parts of the daughters. Like ID/LP expansion, unification is
operation where many different configurations of 'inputs' (from daughte
specify the same 'output' (on the mother), so the problem described in
preceding paragraph arises again here.

Finally, the Conjunct Realisation Principle requires that the daughters
a coordinate structure are all extensions of the mother, that is, they are
just more detailed instances of the same category as the mother. Here
mother could be any category contained in (that is, less well specified th
the maximal intersection of a given daughter set, and this means that
general, the daughters do not fully define the mother. However, ProG
interprets the CRP as also requiring the mother to be maximal, and this me
that the same considerations applied above are relevant here - the daught
specify the mother but not vice versa.

These conclusions suggests that bottom-up parsing is generally the b
way to proceed with a GPSG grammar (whether fully expanded or not). The f
that the expansion (to a context-free grammar) has not been carried out
ProGram provides further support. The categories in the rules are only fu
specified once they have been incorporated into an analysis. A top-d
approach must deal with this under-specification by guesswork, or by delay
instantiation (and hence delaying validity checks etc.) until after
daughter categories have been built. In either case it is difficult to av
wasting time by following paths that cannot succeed due to earlier but as
unchecked violations. An alternative which overcomes this problem is to ca
out the checking incrementally, an approach which would require sophistica
inference mechanisms of the sort described in [Frisch 1985], but which are
readily available in current programming languages. A bottom-up parser, on
other hand, starts off with fully instantiated lexical categories, and w
care can ensure that categories are always fully specified and checked bef
they are used elsewhere.

Taking account of these arguments, ProGram parses bottom-up. The is very primitive, with no memory of partial constituents etc. It with the fully instantiated lexical categories, and attempts to fin which will consume some of them. It takes an ID rule as its basis, a it against some of the daughters, checking linear precedence, and conventions. After a successful match, the mother category itse complete and so, inductively, all daughters are always complete. particular, it is guaranteed that the CRP and FFP will never modify in any way and so the linear precedence checks can be carried out as daughter is consumed bu the ID rule, rather than waiting until the been found and FFP and CRP checks (which notionally precede the LP ch been done. This rule matching process is then repeated, the overall being a single rule which consumes all the current categories.

The fact that the simplicity of the algorithm permits such optim linear precedence etc. suggests that a more sophisticated algorithm have to be considerably more complex. The main reasons for keeping simple if inefficient scheme are that it is easy to follow when contr parser manually, and, because it uses Prolog's backtracking to do searching, it minimises its space requirement. The structures manipulates can be quite large, so this is a real consideration.

To give a brief idea of the algorithm, consider a sequence of as in (6.11) being parsed according to the simple grammar in (6.12). and subsequently, the bar represents the left boundary of the currently under consideration. The parser starts by considering the category.

```
(6.11) det  n  vtr  det  adj |  n

(6.12) s --> np,vp
       np --> det, nom
       nom --> n
       nom --> adj, nom
       vp --> v, np
```

The parser will apply a rule to the N producing a NOM, and fail to ap to that (alone). So instead the parser moves the bar left to look a category (6.13).

```
(6.13) det  n  vtr  det | adj  nom
```

The rule consuming the ADJ and the NOM is now applied, creating a again, the NOM cannot be built up any further, so another category i the current list (6.14).

```
(6.14) det  n  vtr | det nom
```

From here, an NP can be built (but nothing more), the bar moved left built. The bar moves again, giving (6.15).

```
(6.15) det | n vp
```

This time, a rule applies to the N which does not consume the VP, so remains untouched. Again, no further building is possible, the bar and the NP and then the S can be built. With only one category rem parse has been found.

At each stage of this procedure, there is one principle choice: whether t
ipply a rule (and if so which one) or move the bar left, to include anothe
ategory. It is the parser's behaviour at such choice points which differs fo
lifferent control modes. Exhaustive search in AUTO mode is achieved by usin
'rolog's backtracking - Prolog remembers each choice selected, and can back u
:o the last choice point and try a different alternative. Once all the rules
ind moving the bar left, have been attempted, Prolog backs up to the choic
efore and tries a different alternative there. In MONITOR mode, the use
'ilters the choices the parser makes, and in CONTROL mode, the choice is lef
ntirely up to the user. In the example, after producing the parse, the syste
ndoes the last choice (building the S) and tries something else. The situatio
s as in (6.16).

(6.16) | np vp

'here are no more rules to apply, and the bar cannot go further left, so th
ystem backs up again, undoing the NP giving (6.17). Again nothing can be done
nd the next back up moves the bar RIGHT to give (6.18).

(6.17) | det nom vp
(6.18) det | nom vp

n this way the parser will back up all the way back to the beginning. Sinc
he grammar is so simple, no new alternatives will be tried. In general, ther
ay be new possibilities to try at any stage, causing the parser to mov
'orwards again.

:ULE MATCHING

The principle task of the parsing algorithm described above is the rul
atching, which may be summarised as follows. The parser is faced with a lis
f categories (all those to the right of the bar) and a candidate ID rule. Th
laughter categories of the ID rule are to be matched with a leading (leftmost
ublist of the given categories such that

a) linear precedence constraints are observed
b) FCD specifications are fulfilled
c) CRP and FFP between the mother and daughters are satisfied
d) No essential daughters in the rule are not matched

f all these conditions hold, the rule has matched successfully, and the mothe
an replace the daughters in the list of given categories. Recall from abov
hat HFC, CAP and metarule processing can be ignored.

An ID rule contains a list of daughter categories which are unordered an
ay be annotated with operators "opt" (optional category), "*" (zero or mor
nstances of this category permitted) or "+" (one or more instances of thi
ategory permitted). Multiple instances need not be adjacent in the final rul
unless LP rules force them to) and if a multiple instance category is no
'ully instantiated, then each instance can be matched independently of th
thers. Such a rule is exhausted by a list of instantiated categories if ever
ssential category in the rule matches one in the list (and possibly some non
ssential ones do too). The basic ID rule matcher takes an ID rule and tries t
llocate instantiated daughters (taken from left to right) to rule daughter
in any order) until there are no essential rule daughters outstanding. As eac
laughter is matched LP checks between it and all the daughters already matche

are carried out (it must not be required to precede any of them). sufficient to ensure LP admissibility of the entire list. Rule dau given a status marker, which is suitably updated (according to their type) as they are consumed and used to check for rule exhaustion statuses are "requiring matching", "available for matching, but not matching" and "not available for matching. A rule is exhausted when are no categories still requiring matching.

When the rule is exhausted, the CRP and FFP processing is carrie complete the instantiation of the mother, and then FCR constraints ar FCD checks on the mother are deferred: since they depend on the categ as daughter in a rule, they cannot be carried out until the incorporated into the level above in the analysis.

FEATURE INSTANTIATION PRINCIPLES

The actual implementation of the remaining principles (FFP, CRP, and RAC) is straightforward. For the FFP, the daughter foot incr collected up as the rule-match proceeds, and compared with the mother at the end. The CRP routines 'walk' down the feature trees of all the in parallel, and as long as they coincide, the mother must agree with checking takes place at the same time as rule category matchin categories to be matched are examined in parallel to ensure that agrees with, but does not contribute to, the daughter, and non- features are checked for defaults. FCR and RAC checks are simple det particular feature value combinations on the now fully instantiated c

7. Evaluation and Conclusions

This section provides a brief appraisal of ProGram, eight mon completion. Although the system has found its way into several sites countries in that time, there has as yet been little feedback from trying to use it. Thus the comments in this section are based larg author's own experience of using and demonstrating the system, experiences of students and colleagues at Sussex [12].

The idea of using a controllable parser to 'debug' a grammar wo providing a tool which can show up problems with grammars that are located manually. The overall design of the system sacrifices muc interests of practicality. The segregation of data into different disc-based nature of the system, and indeed the whole normalisation could be more streamlined, or even completely transparent. To a certa ProGram could be improved as it is - the addition of more i normalisation commands, and other high-level commands, would undoub the system easier to use, although probably no faster. But the ove automatic normalisation and sufficiently flexible file management wo greater.

In general, the attempts to overcome the time and space problems been partially successful. ProGram is a large system and it takes amount of skill to avoid loading too much of the system at once, grammar development proceeds smoothly and quickly, and without incon other users or consuming processing quotas. However, with practice t can be acquired, after which the system is in general quite m Automatic parsing is a particular problem, and cannot always be avo only remedies seem to be: keep phrases as short as possible (for e

lot try to test large noun phrases inside complex sentences if it can b
voided - use proper nouns instead), make sure no words are duplicated in th
exicon, keep the number of ID rules currently loaded down where possible, an
un the parser in batch mode - perhaps using the cut-down parser, LIB PRS
ather than the full system.

Understanding the behaviour of the parser in MONITOR and CONTROL mode
ilso requires some effort. Most of the problem seems to be the right-to-lef
ehaviour (which can be picked up quite quickly), and the backtracking
lacktracking can produce some rather strange effects at times, althoug
roGram's behaviour is model Prolog chronological backtracking (no cuts etc.)
n control mode, ProGram announces when it is forced to backtrack, which make
he flow of processing easier to follow.

ProGram provides a variety of tracing and diagnostic facilities which ca
11 be very useful in various situations. There are, however, seriou
Missions, particularly in the CONTROL mode diagnostics. For example, it is no
lossible to look at the detailed structure of the categories currently unde
:onsideration during the parse. The WATCH flag provides parse tree information
»ut this seems far Jess useful. One of the least helpful diagnostics is the on
itating simply that the rule match failed ('ID rule not applicable'). This ca
tappen because the rule was totally inappropriate in the first place, becaus
:he output to some metarule was slightly incorrect, or because HFC or CAP hav
>een violated. Because these latter features of GPSG are handled in the pre
>rocessing stage, the parser simply cannot tell which is the problem. But i
he categories themselves were available, it would often be possible to se
;hat was wrong. More generally it is sometimes difficult for naive users t
rork out what the error messages mean: a given linear precedence problem, fo
example, might be caused by incorrect LP, ID rule or alias data (to mention bi
hree), and the parser does not help decide which.

The lexicon seems to cause users some trouble. ProGram expects only th
lost rudimentary of lexicons and places quite severe constraints on it: th
exical categories must be fully specified and it is inadvisable for a word t
>ccur more than once. Used sensibly, with only one or two words for eac
exical type, and different words for, say, different case nouns, it i
lanageable, although the phrases that can be produced may sometimes seem
ittle contrived. Unfortunately, users of the system find it difficult to thir
n these terms, and try to create large lexicons and omit features which ca
:ake any value (for example, CASE for proper nouns). This leads to disaste
since ProGram forces such features to stay unspecified, so that the categorie
lever match with any rule. A lexicon utility has been implemented by Kelle
[Keller 1984], but this has not yet been fully integrated into the ProGra
'ramework. It makes specification of a more general lexicon easier, and is
iseful aid to developing a lexicon suitable for use with ProGram, an
subsequently extending it to provide more realistic coverage.

The problem with the lexicon has a parallel with the parser itself. Th
ilgorithm is rather sensitive to the structure of the ID rules. For example, i
lislikes explicit optional categories, especially occurring as left-mos
laughter in a rule. It will try almost everything else before it gets round t
Lttaching such a category. It is far better behaved if optionality is encode
:n two separate rules, one containing the category, the other not. The syste
LS it is prefers grammars that basically resemble 'standard' GPSG grammars, an
i/ith these it is generally well-behaved.

The ProGram system was completed in April 1984, and since
subject to only minor modification and extension. The pro
active and so further development is not forseen. ProGram was
attempt to provide a computer representation of a GPSG grammar
attempt to specify and verify a computer representation of a g
the project was successful, providing a tool which offers effe
GPSG grammar writer. Aside from ProGram's value as a developme
existence of the parser ensures that the internal formalism of
adequately capture all the aspects of GPSG (as presented in
provides a proven base for other GPSG system implement
generators etc.).

totes

1] ProGram is a suite of Prolog [Clocksin and Hellish 1981] programs developed using the POPLOG programming environment [Hardy 1982] at th University of Sussex, supported by the Socia] Science Research Counci (UK), grant number HR 7829/1 to the University of Sussex (principa investigator: Gerald Gazdar). ProGram was designed and implemented b Roger Evans and Gerald Gazdar. ProGram is available free to intereste academic users - see below for details.

[2] GPSG also contains a formal semantic component, but this will not b discussed here - the ProGram system contains no semantics.

[3] As with any developing theory, several different variants of GPSG coexis at one time. Below, we shall commit ourselves to one particula formulation.

4] Throughout this paper, list notation [...] will be used without comma separating items. This is tidy, but NOT in keeping with Prolog syntax. Tli grammar in the appendix, however, does conform to Prolog syntacti specifications.

[5] We need not be concerned here about the possibility that the underlyin grammar is not context-free (although a full listing of an infinite rule set would undoubtedly cause a problem). The more recent proposal [Gazda et al. 1985] that a GPSG grammar does not specify a context-free grammar but rather it constrains the tree-set directly, is more interesting, an corresponds more closely to the way ProGram actually works. However th difference is not important to the present discussion.

^6] These conventions, and others mentioned below, are part of the GPSG theor and will be used without comment throughout this paper. The followin abbreviations will be used: HFC - head feature convention, CAP - contrc agreement principle, FFP - foot feature principle, CRP - conjunc realisation principle (see [Gazdar et al. 1982]), DAC - default assignmer convention, FCD - feature coefficient defauJts (FSD in [Gazdar et al 1985]), FCR - feature co-occurrence restrictions, RAC - root admissibilit conditions. The reader is referred to GP82 for more details.

[7] Aliases are in fact just ordinary Prolog clauses, so that tl" knowledgeable Prolog user can construct arbitrarily comple specifications. For example, the use of Prolog variables is particular] useful. The aliases in (4.14) are both subsumed by a single alias:

```
alias( v(N), [root [cat [bar N] [head [major v]]]]).
```

Here N is a Prolog variable (adopting the convention that Prolog variable start with an UPPER CASE letter), and the alias is valid for any value c N, but the same value must occur in both instances of N. (14) contain examples of this general alias with N=l and N=2 respectively. See th example grammar for many further instances of this mechanism.

[8] The following examples are real transcripts of ProGram use, slight] modified to remove unnecessary output etc. Input from the user i underlined.

[9] ProGram has been tested with varying degrees of thoroughness on DEC1

Prolog, CProlog and POPLOG Prolog systems.

[10] The Prolog notation [np|W] means "a list whose first element i whose other (currently unspecified) elements are in the varia matched with a list like [np,v,s], W would be set to [v,s].

[11] This distinction is somewhat simplistic: for example, the properties of the first daughter in a context-free rule will be in a predictive bottom-up parser. The general argument sti however, and the underlying question of what specific parsing are well suited to GPSG grammars (and hence, one might claim, t languages), has yet to be answered.

[12] The author is indebted to the experience and comments of ProGram Sussex, particularly Lynne Cahill and David Allport.

Availability

1. "The ProGram Manual" [Evans & Gazdar 1984] provides a introduction to the system, its features, and its use. It is av University of Sussex Cognitive Science Research Paper 35 (CSRP can be ordered from Ms. Judith Dennison, Cognitive Studies Progr E, University of Sussex, Falmer, Brighton BN1 9QN, for 7.5 including postage and packing.

2. ProGram is now part of the standard Sussex POPLOG system and without extra charge, in all academic issues and updates of system. POPLOG is available for VAX's under VMS, VAX's under Bleasdale BDC 680a's under Unix. Non-educational customers (UK & who want ProGram with POPLOG should order it through System Ltd., Systems House, 1 Pembroke Broadway, Camberley, Surrey GU15 62244).

3. Academic users of other Prolog systems can obtain a magnetic tap tar format) of the Prolog code of the ProGram system free, toget copy of "The ProGram Manual", provided they pay the tape, package, and handling costs (35 pounds). Copies can be ordere Alison Mudd, Cognitive Studies Programme, Arts E, University o Brighton BN1 9QN. A cheque for 35 pounds, made payable to "The of Sussex", should be enclosed with the order.

References

Clocksin and Mellish 1981: Programming in Prolog. Clocksin W. and C. Springer Verlag, 1981.

Evans and Gazdar 1984: The ProGram Manual. Evans R. and G. Gazdar. Studies Research Paper No: CSRP 035, University of Sussex, April

Farkas et al. 1983: Some revisions to the theory of features an instantiation. Farkas D., D. Flickinger, G. Gazdar, W. Ladusaw, J. Pinkham, G. Pullum and P. Sells. In Proceedings of the ICOT W Non-Transformational Grammars, 11-13 (Tokyo: Institute for New Computer Technology). 1983

Frisch 1985: Parsing with Restricted Quantification. A. Frisch. University of Sussex, January 1985

Gazdar et al. 1982: Coordinate structure and unbounded dependencies. Gazdar G., E. Klein, G. Pullum, and I. Sag. In M. Barlow, D. Flickinger & I.A. Sag (eds.) Developments in Generalised Phrase Structure Grammar: Stanford Working Papers in Grammatical Theory, Volume 2. Bloomington: Indiana University Linguistics Club, 38-68. 1982

Gazdar et al. 1985: Generalised Phrase Structure Grammar. Gazdar G., E. Klein, G. Pullum, and I. Sag Oxford: Blackwell, 1985.

Gazdar and Pullum 1982 (GP82): Generalised phrase structure grammar: a theoretical synopsis. Gazdar G. and G. Pullum. Mimeo, Indiana University Linguistics Club, August 1982.

Hardy 1982: The POPLOG Programming System. Hardy S., Cognitive Studies Research Paper No: CSRP 003, University of Sussex, November 1982

Keller 1984: A Lexicon Handler for the ProGram Grammar Development System. W. R. Keller Cognitive Studies Research Paper No: CSRP 040, University of Sussex, June 1984

Pereira et al. 1978: User's Guide to DECsystem-10 prolog. Pereira L. M., F. C. N. Pereira and D.H.D Warren. DAI Occasional paper 15, Department of Artificial Intelligence, University of Edinburgh. 1978

Pereira and Warren 1980: Definite Clause Grammars for Language Analysis - a survey of the formalism and a comparison with augmented transition networks. Pereira F. C. N., and D.H.D Warren. Artificial Intelligence 13:3 231-278. 1980

Thompson 1981: Handling metarules in a parser for GPSG. Edinburgh D.A.I. Research Paper No. 175. Also: In M. Barlow, D. Flickinger & I.A. Sag (eds.) Developments in Generalised Phrase Structure Grammar: Stanford Working Papers in Grammatical Theory, Volume 2. Bloomington: Indiana University Linguistics Club, 26-37.

Appendix - the example grammar

This appendix contains the complete grammar used in the paper. Note
version below conforms to Prolog syntax, (unlike the examples i
text).

Features

```
syncat root.

feature [root,  cat, foot].
feature [cat,   bar, head].
feature [bar,   {lexical, 1, 2}].
feature [head,  major, minor].
feature [major, {v, n, d, p>].
feature [minor, agr, {vform, pform, nform}].
feature [agr,   {sing, plur}].
feature [vform, {fin, pass}].
feature [pform, {by, to}].
feature [foot,  cat].
```

Aliases

```
alias(  v(N),   [root,[cat,[bar,N],[head,[major,v]]]] ).
alias(  n(N),   [root,[cat,[bar.N],[head,[major,n]]]] ).
alias(  d(N),   [root,[cat,[bar,N],[head,[major,d]]]] ).
alias(  p(N),   [root,[cat,[bar,N],[head,[major,p]]]] ).
alias(  h(N),   [root,[cat,[bar,N],[head,[major]]]] ).
alias(  v(L,N),[root,[cat,[bar,L],[head,[major,v],[minor,[vform,N]]]]
alias(  p(L,N),[root,[cat,[bar,L],[head,[major,p],[minor,[pform,N]]]]
alias(  h,     h(lexical)).
alias(X/Y, Z) :-
    normfeat(X,XN),normfeat(Y,YN),
    pathfor(foot,YN,f~'),
    pathfor(cat,YN,YCat),
    pathfor(foot,XN,[[catjYCat]]),
    Z  = protect(XN).

( The following aliases are mainly for the lexicon: )

alias(  nO,     [root,[cat,[bar,lexical],
                          [head,[major,n], [minor,[agr,sing],nform]]
alias(  dO,     [root,[cat,[bar,lexical],
                          [head,[major,d], [minor,[agr,sing],"nform]
alias(  vO(V),  [root,[cat,[bar,lexical],
                          [head,[major,v], [minor,[agr,sing],[vform,
alias(  pO(P),  [root,[cat,[bar,lexical],
                          [head,[major,p], [minor,[agr,sing],[pform,
```

ID rules

```
s: v(2) —> N2,H1  where N2 is n(2),
                        HI is h(l),
                        N2 controls HI.
```

```
vp1: v(1) -> h.
vp2: v(1) -> h,n(2).
vp3: v(1) -> h,n(2),p(2,to).
vp4: v(1) -> h,v(l,pass).
pp:  p(2) -> h,n(2).
np1: n(2) ~> DET,HI where DET is d(Jexical),
                          HI is h(l),
                          HI controls DET.
np2: n(2) -> h.
nom: n(l) -> h.
```

Metarules

```
pass:      (VP1 -> ... • n(2)          where VP1 is v(l))
      ==>
           (VP2 -> ... , opt(p(2,by)) where VP2 is v(l,pass),
                                            VP1 matches VP2).

stml:      (Cl -> C2, ...  where Cl is [root],
                                 C2 is [root,[cat,[bar,2]]])
      ==>
            C1/C2 -> ...  .

relcl:     (Nl -> ...                  where Nl is n(l))
      ==>
            Nl -> . . . , v(2)/n(2).
```

LP rules

```
[root,[cat,[bar,lexical]]] « { [root,[cat,[bar,1]]],
                                [root,[cat,[bar,2]]]  }.
n(2) « v(l).
[root,"foot] << [root,[foot,cat]].
```

FCD's

```
fcd( vform, [foot], fin, free ).
```

FCR's

```
fcr( major, [foot], n, minor, [foot], nform ),
fcr( major, [foot], p, minor, [foot], pform ).
fcr( major, [foot], v, minor, [foot], vform );
```

Lexical rules

```
vpl:        vO(fin)  ->-  jumps.
vp2(fin):   vO(fin)  ->-  sees.
vp2(pass):  vO(pass) ->-  seen.
vp3(fin):   vO(fin)  ->-  gives.
vp3(pass):  vO(pass) ->-  given.
vp4:        vO(fin)  ->-  is.
np2:        nO       ->-  kirn, fido.
```

```
nom:        n0        ->-  woman, man.
np1:        d0        ->-  the.
pp:         p0(to)    ->-  to.
pp:         p0(by)    ->-  by.
```