NOTICE WARNING CONCERNING COPYRIGHT RESTRICTIONS:
The copyright law of the United States (title 17, U.S. Code) governs the making of photocopies or other reproductions of copyrighted material. Any copying of this document without permission of its author may be prohibited by law.

The ProGram Manual

Roger Evans and Gerald Gazdar*

Cognitive Studies Programme

University of Sussex

April 1984

* The order of the authors' names is only coincidentally alphabetic: the first author wrote all of the ProGram code and almost all of the prose; the second author conceived the project, got the grant, managed the money, provided the occasional exemplar rule, handled public relations, devised nroff macros, did the copy editing, designed the cover, and wrote this footnote.

We are grateful to the SSRC (UK) [Grant HR 7829/1] for the financial support that made ProGram possible, to our AI colleagues at Sussex whose POPLOG programming environment made the creation of ProGram almost a pleasure, and to Andy Clews for the printer driver.

Copyright ©, University of Sussex, 1984. This document may be copied for noncommercial purposes, without further permission being sought.

SSX
035

The ProGram Manual

Table of Contents

Aval1abilitv

1. This manual is *University oi Sussex Cognitive Science Research Paper* 35 (CSRP 835) and can be ordered from Ms. Judith Dennison, Cognitive Studies Programme, Arts E, University of Sussex, Falmer, Brighton BN1 9QN, for 7.50 pounds, including postage and packing.

2. ProGram is part of the standard Sussex POPLOG system and included, without extra charge, in all academic issues and updates of the POPLOG system. POPLOG, is available to UK academic users for the sum of 500 pounds (special arrangements apply to holders of SERC AI grants who have a VAX running Unix) and is already in use at nearly 50 educational institutions in England and abroad. Existing UK academic POPLOG users can obtain a free update of the P0PL06 system, one which will include ProGram, in return for a magnetic tape sent to Mr. Jonathan Laventhol, POPLOG Group, Arts E, University of Sussex, Falmer, Brighton BN1 9QN. POPLOG is available for VAX's under VMS, VAX's under Unix, and Bleasdale BDC 680a's under Unix. It is scheduled to become available on PERQ's and 6EC 63's. Non-educational customers (UK & overseas) who want ProGram with POPLOG should order it through System Designers Ltd., Systems House, 1 Pembroke Broadway,. Camberley, Surrey GU15 3XH (0276 62244). This company makes POPLOG available to educational institutions in the USA for 995 dollars.

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

1. Introduction

ProGram is a suite of Prolog programs that are intended to permit the design, evaluation, and debugging of computer realizations of phrase structure grammars for large fragments of natural languages. The grammar representation language employed, that known as Generalized Phrase Structure Grammar or GPSG (Gazdar & Pullum 1982 - 'GP82', henceforth; Gazdar, Klein, Pullum and Sag 1982), is neutral with respect to parsing or generating sentences, and is capable of being used with a variety of programs. ProGram is thus a grammar interpreter, where the latter is, in this instance, construed as a software tool, namely a grammar development system for use by linguists or computer scientists developing GPSG's for large fragments of natural languages.

All the major parts of the grammar interpreter code are written in standard Prolog (see Clocksin & Mellish 1981), as provided within the POPLOG multilanguage program development environment (Hardy 1982, Hardy & Sloman 1982, Mellish & Hardy 1983). Only specialized optional modules (for example the tree drawing package) have been written in POP11. This is intended to maximise the portability of the main grammar development system, especially overseas where most relevant sites now have Prolog available, but usually not POP11. Installation of the system should be fairly simple on any machine of moderate size which supports Prolog.

We decided that the most perspicuous way of arriving at a grammatical representation language that would be entirely neutral with respect to the computational uses to which the grammar might subsequently be put was simply to build a grammar interpreter that would interpret the extant GPSG formalism in more or less exactly the way it was normally written (including all the abbreviatory and alias devices beloved of generative linguists). This decision means that linguists who can understand the GPSG formalism, but are quite naive computationally, can use the system immediately without the need to learn an additional grammar representation language.

The standard GPSG grammar notation employed in the project is entirely noncommittal with respect to the potential uses to which the grammar might be put, e.g. as a component of a sentence generator, recogniser or parser. And it is also noncommittal within these uses. So, for example, nothing in the grammar format (as opposed to the structure of the grammar itself) forces a parsing utilisation to be top-down rather than bottom-up, or left-right rather than right-left.

This neutrality also extends to the programming language environment. Although ProGram is written in Prolog, that fact does not in itself require that programs which use grammars developed with the ProGram system must be written in Prolog. No language-specific features of Prolog are exploited in the grammar formalism proper (in contrast to, e.g., Direct Clause Grammars (Pereira & Warren 1980)). Thus, for example, a grammar could be developed and debugged with ProGram, and then used, in its normalised form, by a chart parser written in POP11 or LISP, say.

There are two motivations for this the neutrality of the representation, one theoretical, the other practical. The GPSG framework is a theory about the structure of languages, it is not, in itself, a theory about how languages are produced, parsed or learned. In adapting the GPSG notation to a form in which it can be accessed as a component of possible theories of production, parsing, or grammatical inference, it is essential to ensure that the conversion process does not augment it with gratuitous biases in respect of such theories. The practical motivation is, of course, that a grammar format that is maximally neutral with respect to application and language environment is likely to attract the widest possible range of users.

1.1 Overview

The ProGram grammar development system is a collection of utility programs designed to aid the creation, debugging, and testing of phrase structure grammars for natural languages written within the notational conventions of GPSG. The GPSG framework is a formally precise, yet powerful and practical grammar representation language, one which has already been adopted by a big Hewlett Packard research project (Gawron et al. 1982) for the syntactic basis of a natural language understanding system. The specification of a GPSG grammar has a number of components which interact in complicated ways, and a GPSG grammar defines large and complex structural analyses. Thus the task of ensuring manually that a given grammar behaves as expected, assigning all and only the correct structures to any given phrase, is both time-consuming and prone to error.

The ProGram grammar development system is a computational tool to help overcome these problems. As such it can be of use both to the theoretical linguist who wishes to examine the behaviour of a grammar, and to the applied computational linguist, who is concerned that the grammar to be incorporated in an language understanding system or a language production system is internally consistent and incapable of assigning spurious analyses.

The system allows the complete specification of a grammar, from feature syntax upwards, in a form essentially identical to that given in GPSG, including ID rules, LP rules, metarules, feature defaults, etc., subject only to a few constraints on the features of special significance to the theory (HEAD, FOOT, etc.). The initial processing of this grammar includes wellformedness checks on the specification (for example, every category - a feature tree - must conform to the feature syntax specifications), but the main testing function of the system is a parser which can be interactively controlled, allowing specific bugs in the the user's grammar to be located and examined quickly.

The parser has three modes of operation: in automatic mode, the parser runs without intervention from the user, stopping either at the first parse found, or doing an exhaustive search for all parses. In monitor mode, the parse proceeds automatically, but checks with the user every time it makes a major decision (for example, what rule to apply). The user may accept the decision, reject it completely, or ask for alternatives, before the parser continues. In control mode, the parser asks the user to actually make the major decisions. For example, the user will be asked which rule to attempt to apply next. The system then

proceeds, and if the attempt fails, it produces diagnostic information (for example, 'Foot Feature Convention failed').

These three modes, the ability to select only a particular subset of input data (ID rules, etc.), the possibility of tracing the parse, and the clear presentation of the resulting structure, combine to make the system a flexible and useful grammar-testing tool.

A GPSG grammar, as far as ProGram is concerned, has up to nine components as follows:

1. Specification of feature syntax.
2. Immediate dominance rules (ID rules).
3. Metarules which operate on the ID rules.
4. Linear precedence rules (LP rules).
5. Feature coefficient default values (FCD's).
6. Feature cooccurrence restrictions (FCR's).
7. Feature aliasing data.
8. Root admissibility conditions (RAC's).
9. A lexicon.

Of these, the first six are exactly as characterized in the GPSG literature (see, e.g., GP82) and are discussed in chapter 2, below.

The feature aliasing data allows abbreviation of feature expressions, so that, for instance, a grammar writer can write S instead of, perhaps, [CAT1, [CAT, [BAR,2], [HEAD, [MAJOR, +V, -N]]]] which is likely to be the full, unabbreviated, GPSG form of the sentence category. Clearly, this makes rule specification and the like simpler and more readable.

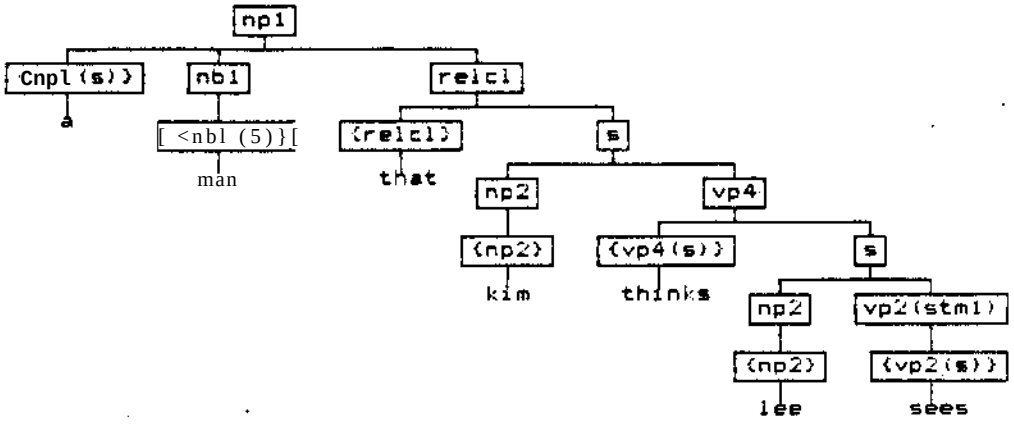
The root admissibility conditions allow the grammar writer to place restrictions on the sort of category which is acceptable as the root of a parse tree.

The lexicon provides a way of establishing a simple correspondence between words and the parts of speech to which they belong. However, provision is also made for a more sophisticated lexicon.

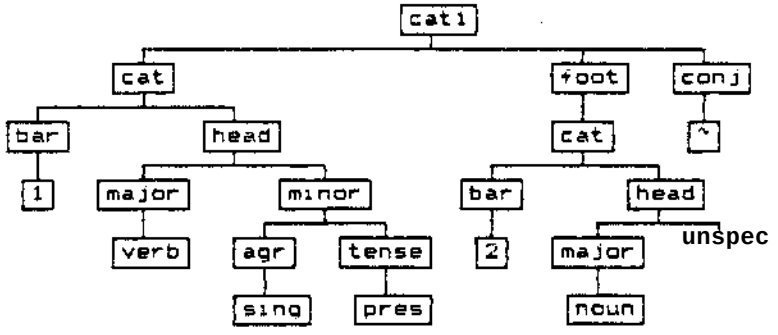
ProGram operates in two phases: data normalisation and parsing. The grammar components as specified above must all be normalised before they are used for parsing. Normalisation consists of translating the alias abbreviations (as above) and checking for valid feature specification (according to the feature syntax provided) and for valid datum (ID rule, metarule, etc.) specification. Various error messages and warnings may be produced during normalisation and these can then be corrected as necessary - usually only the data module concerned needs renormalising.

Parsing provides the main investigative function of the system. The parser uses the normalised data produced in the manner described above, and parses successive strings of words. All the major conventions described in the GPSG literature are implemented, including the Head Feature Convention, the Foot Feature Principle (and hence SLASH categories, etc.), the Control Agreement Principle, the Conjunct Realization Principle, lexical subcategorization and rule instantiation incorporating the notion of privilege.

ProGram provides the user with various means of displaying trees, the most perspicuous of these (only available under POPLOG) provides screen (and printer) output in the for* exemplified below.



Note that in the parse tree above, the nodes are labelled with the names of the rules which expand then, not with their category label. This is partly because in developing a grammar it is usually more useful to know what rule is responsible for a node than it is to know what its gross category is, and partly because the category label in a 6PS6 is itself a tree of a features, and not a simple monadic category name. To see the category associated with a particular node, one simply moves the cursor to the node and presses the VED PUSH key. The VDU will then display the structure of the category. Thus, for example, if we were to inspect the internal structure of the node labelled VP4 in the example above, then the system would display the following feature tree.



The various options available during parsing include:

1. **Reading sentences from teriinai or data file.**
2. **Writing parse trees to terminal or data file.**
3. **Displaying parse trees on teriinai in a readable fonat.**
4. **Finding first parse only, or all parses.**
5. **Parsing completely autoiaticaily.**
6. **Parsino with user advising at all crucial decisions.**
7. **Parsing «ith user in complete control over choices.**
8. **Tracing the operation of the parser.**

Using these options, the grammar writer can see, for example, what the parser is doing (in particular, perhaps, see where it is going wrong) or force the parser to try particular rules, either to produce unlikely parses more quickly than by exhaustive search, or to attempt to force wrong analyses. Furthermore, in control mode, the parser reports the reasons for failure when any particular user-selection fails.

ProGram is currently equipped with a simple demonstration grammar designed to illustrate the various features of the system. Subsequent releases of the system may include a larger grammar covering a significant portion of the syntactic constructs of English, and designed to provide a basis for serious research into English syntax, or to provide a grammatical foundation for computational systems intended to parse or produce English.

1.2 Using the system

This section describes the basic mechanics of how to use the ProGram grammar development system. It assumes the user has written a grammar and wishes to use ProGram to develop and test it. It also assumes the user has created text files containing the various components of the system using local editing utilities (e.g. VED in POPLOG). For full details of how the grammar should be represented, see chapter 2.

Details of how ProGram is loaded will be found in the help file LOCAL - these depend on the local setup. Several messages may (or may not) be displayed during the loading sequence. When the system is ready to use, the following messages will be displayed:

```
ProGram Grammar Development System
Version 1, 29/4/84

(For help information type:    help. )

?-
```

The '?-' is the Prolog prompt. It means the system is waiting for the user to type something. In some systems it may be different (e.g. ':-'), in some systems it may vary according to whether it is at 'command level' or inside a program (when it may become, for example, '|:'). Your Prolog reference manual will have details.

In this case, the '?-' means that the system is waiting for a command. A command consists of one or more words followed by a period. In ProGram, the first word is the main command and may be modified with additional data specification clauses. See chapter 5 for a list of the commands available, and chapter 3 for details of the data clauses.

In order to test the grammar, the user must first normalise all the components. See chapter 3 for full details. The first command might be:

```
normid from myrules.
```

This says 'normalise the ID rules in the data file MYRULES' (which the user has previously created as specified in chapter 2). The normalised rules are simply displayed on the screen, and any errors detected are reported. If there are errors (see chapter 5), the text file MYRULES should be edited and then the command tried again. When there are no errors, the data can be safely saved by giving the command:

```
normid from myrules to normdat.
```

This command normalises as before, but saves the results in a file called NORMDAT (which it creates if necessary).

A similar procedure must be undertaken to normalise all the components of the grammar, using the commands NORMLEX (normalise lexical category rules), NORMLP (normalised LP rules), etc.

At various times during these commands messages like

```
(loading normrules)
```

will appear. These are just to inform the user that system modules are being loaded (and hence there is a slight delay) and can be safely ignored. Also, the system may ask for the names of 'background' data files (as discussed in chapter 3). For example if the command given above really is the first command of the session, then the data files FEATURES and ALIASES will be requested. This could have been avoided by typing the command as:

```
normid from myrules using myfeats for
features using myalias for aliases.
```

It is quite permissible for commands to be spread across several lines, as long as words are not broken by line boundaries. The computer will continue to prompt for more until a period is typed.

If you mistype a command, or type a command which is not understood, the system will either produce an error, or else respond with 'no' and a new prompt. At the end of commands which are executed successfully, the system types 'yes' before the next prompt. Another prompt without a YES, NO or an error, probably means the period has been omitted. This can be corrected simply by typing a period.

Having normalised all the data, the grammar can be tested using the PARSE command (see chapter 4 for more details). Other utility commands may also be given whenever the system is at command level.

Note finally that, throughout this manual, terms which have a special significance to ProGram (e.g. commands, filenames, features, etc.) appear in upper case within the text. In the Prolog code itself, these terms (with the sole exception of Prolog variables) appear in lower case. Explicitly quoted expressions, and examples of interaction with the system (the latter being inset, usually), will appear in lower case.

2. Grammar Format

This chapter describes the format of a grammar for use with ProGram. As data for a Prolog program, the various parts must conform to Prolog's syntactic requirements (subject to operator declarations made in the system or by a user with knowledge of both Prolog and the system), and all the examples shown below do so conform except that items in angle brackets (< .... >) refer objects defined elsewhere and should not be taken literally. This file does not provide full details of Prolog syntax (for that, see Clocksin and Mellish, 1981), in fact, the topic is neglected except by example. However, the following point is of particular importance: the Prolog convention about variable names is that any word starting with an upper case letter is assumed to be a Prolog variable. Prolog variables have a place in the grammar specification, only in the case of ALIASES, as detailed below, but in general, feature names, rule names, etc., are constants and so must not start with an upper case letter.

A grammar may have up to nine components as listed below:

1. Specification of feature syntax.
2. Feature aliasing data.
3. Immediate Dominance rules (ID rules).
4. Metarules which operate on the ID rules.
5. Linear precedence rules (LP rules).
6. Feature coefficient default values (FCD's).
7. Feature cooccurrence restrictions (FCR's).
8. Root admissibilty conditions (RAC's).
9. A lexicon.

Not all of these components are compulsory, in fact (2) is purely a notational convenience (but it is very convenient). There follows a section on each component, and it is assumed for simplicity that each component will be a separate data file. This is not strictly necessary as the system allows the user to concatenate files and filter output data from them.

2.1 Feature syntax

The specification of feature syntax consists of a collection of assertions of three types:

```
feature <feature spec>.
boolean <feature name>.
syncat  <feature name>.
```

The first of these defines the coefficients of a feature in a form very similar to that used in GPSG: a <feature spec> is a list (enclosed in square brackets ([...]) whose first element is a <feature name> and whose other elements are either single <feature name>'s or lists of alternatives, enclosed in braces ({ ... }). A <feature name> is a Prolog atom, e.g. a word (one not starting with a capital letter), a number or a word followed by a number.

A feature specifications defines all the coefficients the feature can take (by giving the names of the features which may occur). Here are some examples of feature specifications:

```
feature [cat,bar,head].
feature [bar, {lexical,1,2,3}].
feature [head,major,{tense,case}].
```

A FEATURE is a tree (as described in GPSG) which conforms to the feature specifications, that is, a feature is a list whose first element is a ⟨feature name⟩ and whose other elements (if any) are all features permitted by the feature specification for that ⟨feature name⟩. The three specifications above permit all the following features (and more):

```
[bar, [lexical]]        (can choose any one of the options)
[bar, [1]]
[head,[case],[major]] (order of coefficients is irrelevant)
[cat]                   (coefficients are optional)
[foo]                   (not specified means 'no coefficients')
[cat, [bar,[1]],[head,[major],[agr]]]
```

As is apparent, these features can get quite cumbersome so the system has some built in conventions about features. One of them we have seen above - if a feature (FOO, say) is not specified its definition is assumed to be

```
feature [foo].
```

This was done with FOO above, but also with LEXICAL, 1, 2, 3, CASE, AGR, MAJOR - in fact all the leaf values in the feature trees. A second convention was not employed above: if a bare ⟨feature name⟩ is encountered where a feature is expected, it is assumed to be the feature of that name with no coefficients specified e.g. foo can be used for [foo]. This is very useful for readability, not to mention bracket-counting. The list of examples above could have been written as follows:

```
[bar, lexical]
[bar, 1]
[head,case,major]
cat
foo
[cat, [bar,1],[head,major,agr]]
```

The second type of feature definition, BOOLEAN, is also an abbreviation, but with an additional notational convenience thrown in. The assertion:

```
boolean ⟨feature name⟩.
```

is equivalent to

```
feature [⟨feature name⟩, +, -].
```

but + and - are treated specially in the system, to allow you to use them as prefixes. Thus if we add the following to our definitions above:

```
feature [major, n,v].
boolean n.
boolean v.
```

we are now allowed features of the following sort:

```
[cat, [head,[major,[+n,-v]]]]
[head, [major,-v],agr].
```

There is one further convention built in to the system. It is often desirable to specify the absence of a coefficient in a feature. Thus far, this is not easy to do - it cannot be just left out since that means 'unspecified' not 'absent' and during parsing, for instance, the head feature convention might decide to specify a value for it. So the system allows the use of the prefix ~ to mean absence. For example:

```
[head, [major,-v],~case]
```

This 'extra value' is available for all features, regardless of how many coefficients etc. they are supposed to take.

Finally the assertion

```
syncat <feature name>.
```

This is used by the rule handling routines to decide whether a feature may be used as a syntactic category or not. Any feature whose <feature name> is declared as a SYNCAT may be sed in an ID rule (etc.). Any other sort of feature gives an error.

2.2 Feature aliasing data

In section 2.1, a few feature definitions were provided by way of example. They are repeated here:

```
feature [cat,bar,head].
feature [bar, {lexical,1,2,3}].
feature [head,major,{agr,case}].
feature [major, n,v].
boolean n.
boolean v.
```

This is, approximately, a small part of the syntax given in GP82. Here are some examples of common categories expressed using this feature syntax:

```
np   =  [cat,[bar,2],[head,[major,+n,-v]]]
verb =  [cat,[bar,lexical],[head,[major,+v,-n]]]
s    =  [cat,[bar,2],[head,[major,+v,-n]]]
etc.
```

To produce a grammar of any size using these longwinded feature names would be laborious at best and the result would be largely incomprehensible. So ProGram provides a mechanism for abbreviating them - ALIASES. In the simplest case, an alias definition looks like this:

```
alias(<alias>,<feature>).
```

An alias in its simplest form just tells the system that a particular word (the <alias> above) it goinq to be used as an abbreviation for a feature (the <feature> above). Given these three aliases:

```
alias(np,[cat,[bar,2],[head,[<najor,+n,-vl]]).
aliasfverb,[cat,[bar,lexical],[head,[major,+v,-n333).
alias(sf[cat,[bar,2],[head,[major,•vf-n-331).
```

the grammar writer could freely use NP, VERB, and S in the rules of the grammar and the system itself would make all the relevant substitution.

By making use of Prolog variables and terms, tht aliasing can be made even more helpful. A Prolog variable is denoted by a word that starts witha upper case letter and serves to bind together parts of a definition or assertion without actually specifying any value. Prolog terms are more complex expressions than the simple words used up to now and are written in functional notation i.e. as a functor, which must be a word (not a variable) followed by one or more arguments (enclosed in parentheses) where each argument is itself a variable, a word (or number, etc.) or a term. The use of Prolog terms in aliasing is best shown by example;

```
alias(h(L), [cat,[bar,L3,[head,major3]).
alias<n(L,Agr), teat,[bar,L3,[head,[major,+n,-v3,[agrfAgr331>.
```

The first of these is an alias for a generic HEAD feature (unspecifed major coefficients - see GP82), the variable L is used to specify the bar level. The way to think of this variable behaving is like this:

> L has no value specified (i.e. it is a variable), but the alias is only valid for term/feature pairs with the same value (the 'value' of L) for all occurrences of L.

Thus this first alias allows the use of H(LEXICAL), H(l), H(2), H(3) in the grammar rules (actually it also allows silly values - e.g. H(FOQ) - but the feature syntax checker will reject such invalid coefficents for BAR).

In the second alias above, two variables have been used, to allow specification of bar level and agreement features thus, for example, N(2,SINB) now means a singular NP.

Aliases may also be applied to other aliases. For example, suppose the following alias was added to those above:

```
alias (h,h(lexical)).
```

This would allow the use of H to stand for [CAT,[BAR,LEXICAL], [HEAD,MAJOR]] via two alias assertions.

Aliasing in fact allows the complete power of Proloq for decoding aliases, and it is applied repeatedly wherever possible (including subfeatures)• A competent Prolog programmer can make use of this for more involved abbreviations. The demonstration grammars make use of a few of these nor advanced features, in particular, it is possible to

provide an alias for the SLASH notation (S/NP), etc. - see 6.2.

2.3 ID rules

The basic format for specifying immediate dominance rules is as follows:

```
<name> : <cat> --> <cat>, .... ,<cat>.
```

Here <name> is the name of the ID rule, and <cat> is a syntactic category, i.e. a feature/alias specification as above. There can be as many categories on the right-hand-side as desired, separated by commas and terminated by a period.

The <name> is a valid Prolog term, usually a word, number or a functor with one or two arguments. Examples of ID rules are:

```
vptr:   [cat, [bar,1],[head,[major,+v,-n]]] -->
        [cat, [bar,lexical],[head,[major,+v,-n]]],
        [cat, [bar,2],[head,[major,+n,-v]]].

s1: s --> np, vp.
np(propn): np --> n.
```

In the first, a rule for a transitive vp, no aliases are used, and it is quite cumbersome. In the second, alaises for s, np, vp are used to make things tidier. The third rule also uses aliases, and it has a complex term for the rule name. For reasons why you might want to do this, see below. Note that aliases can be mixed up with full category specifications at will, and that aliases are not detected in the rule name - there they are treated as literal words (etc.). Also, two rules may have the same name (but see the discussion of subcategorisation, below).

Different parts of the <name> are used for several things in the system, and the following points should be borne in mind when choosing rule-names:

(i) Identifying the ID rule

The principal use of the name is to identify the ID rule, for example in control mode when the user is asked to specify which rule to use. The user does not have to specify the full name - if the name is a complex term, then only the functor of the name need be specified - (the parser will try each matching rule in turn).

(ii) Identifying a node in a parse tree where the rule is applied

The name is used in the parse tree displays to label any node where the rule has been applied (it is impractical to label with the parent category, as usual with parse trees, since the category itself is typically itself a large feature tree). Thus it is useful to specify rule names with mnemonic significance for their LHS category. The name is used in full (i.e. functor and arguments) but in any word in the term, if the word contains an underline character, then it and all the characters in the word after it are not displayed. Thus the both the labels VP_INTR and VP_TR will appear, in parse tree displays only,

simply as VP.

(iii) Lexical subcategorisation

The rule name is used to implement the lexical subcategorisation feature. The name, or its functor if it is a complex tern, is added as a feature argument of [bar, lexical! in any lexical category in the rule (RHS). This means that the category Mill only match lexical items specified in a lexical rule (see lexical rules, below) with the same name/functor.

2,3,1 Regular expressions

The categories on the right hand side of a rule nay also have regular expression operators attached to them. Three operators are possible:

```
opt(<cat>)    -  <cat> is optional
<cat> *•      -  <cat> may be appear zero or many times
<cat> ••      -  <cat> may appear once or many times
```

2,3,2 Variables

A slightly more complicated form of an ID rule is as follows:

```
<name>:<cat> --> <cat>, ... ,<cat>  where <cond>, ... ,<cond> .
```

The additional WHERE clause allows the user to specify two sorts of things - variable definitions and control agreement. Variable definitions allow one to incorporate variables into ID rules. Normally, a rule with a variable (i.e. a word starting with a capital letter) in it will not normalise. In order to allow variables in a rule, a <cond> of the form

```
<var> is <feature>
```

must be provided in a WHERE clause. Thus the following rules are both acceptable (and do exactly the same job):

```
si: s –> np,vp.
si: s –> np,VP  where VP is vp.
```

Variables added like this (there can be several, with a <cond> for each) are used in two ways. Firstly, one sometimes wants to specify that two parts of a rule are identical, without completely specifying their value. The following rule incorporates subject-vp agreement in this way (note that layout is not important):

```
tl: s –> [cat,[bar,23,[head,[major,+n,-v3,AGR33,
         [cat,[bar, 13,[head,[major,+v,-n3,A6R]3
         where A6R is [agr].
```

Here we have specified a variable A6R which takes the same value in *HP* and VP and which is specified as the feature '[agr3'. The coefficients of 'Cagr3' are not specified, but must be the same in any use of this rule.

Doing agreement this way works, but is a little unwieldy, and the GPSG formalism contains the CAP which places special constraints on some [agr] features. Hence there is a better way of doing agreement using the CAP. This is the second sort of <cond> expression in a WHERE clause. It simply specifies that one category controls another. However, to do this, the categories themselves have to be expressed as variables (since they occur in two places in the rue). Thus subject-vp agreement is properly done like this:

```
s1: s --> NP,VP  where NP is np, VP is vp, NP controls VP.
```

Here we have two variables, NP and VP, which are specifed as 'np' and 'vp' respectively, and which are a control pair. Syntactically, control agreement is symmetrical - it doesn't matter which controls which. However, the CONTROLS clause must always follow the clauses defining its variables.

2.3.3 Heads

In order for the Head Feature Convention (HFC) to operate properly, ID rules must have heads. ProGram takes the following as the definition of a head:

> The head of an ID rule is the unique category (in the RHS of the rule), if there is one, with no coefficient of the feature MAJOR (although MAJOR itself must be present) and of minimal bar level (out of the RHS categories of the rule).

Some ID rules do not have heads, either because there is no category with unspecified MAJOR coefficient, or because there is more than one of minimal bar level. The HFC processing produces a warning message in such cases, but this need not be taken as an error.

It is often useful to define an alias for heads, i.e. an alias for a category with unspecified MAJOR. For example:

```
alias(h(B), [cat,[bar,B],[head,major]]).
```

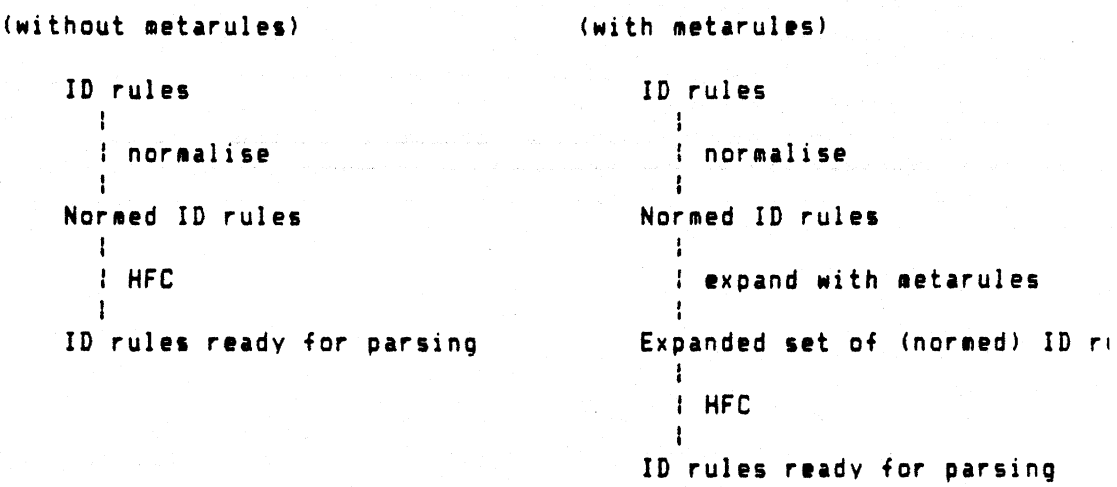
Using this alias, the head can be explicitly specified in a rule.

```
s1: s --> np,h(1).
vtr: vp --> h(lexical),np.
```

The HFC ensures that the H(1) in rule S1 becomes a VP (by inheriting its +V, -N from the S). Actually, including control agreement, the final form of this rule might be:

```
s1: s --> NP, H1 where NP is np, H1 is h(1), NP controls H1.
```

The definition of head is still as above - the alias itself doesn't ensure that no other category is a suitable candidate. The head feature convention operates as specified in GP82. The system's handling of HFC is not done during parsing, but as a preprocess which takes place between normalisation (of ID rules) and parsing. This preprocess is not incorporated into the normalisation because expansion using metarules (see below) must normally come between normalisation of ID rules and HFC processing. Thus the normal sequence of processing required is:



2.4 Metarules

Metarules are rules which map one ID rule into another. The simplest form of a metarule is:

⟨name⟩: ⟨ID rule⟩ ==> ⟨ID rule⟩ .

Here ⟨name⟩ is just as in ID rules and ⟨ID rule⟩ is an ID rule as specified above except that:

(i) The ⟨name⟩ and the colon following it are omitted.
(ii) If there is a WHERE clause, then the whole ID rule (including the WHERE clause) must be enclosed in parentheses.
(iii) A multiset variable may be included in the daughter list.

The grammar expansion process (see below) uses these metarules to expand the set of ID rules in the grammar. For each ID rule already in the grammar that fits the ID rule on the left hand side of the metarule, (i.e. none of the feature specifications contradict), an ID rule of the form on the right hand side is added to the grammar. In the simplest case, no communication between the two sides of the metarule is necessary, i.e. none of the information particular to the ID rule that fits the LHS is needed in the RHS. However, it is more likely that communication is required. There are two ways of achieving communication - by using multiset variables and by using MATCHES clauses.

2.4.1 Multiset variables

The daughter list of an ID rule in a metarule (and only in a metarule), may contain a multiset variable, written as

...

There may be at most one multiset variable in a rule (no more are ever needed, since the daughter lists are not ordered), and it may occur anywhere in the list. When comparing the LHS of the metarule with an ID rule, the multiset variable can match any number of categories (including none), and the categories do not have to be adjacent in the ID rule. So the particular categories specified in the metarule LHS are

mapped onto categories of the ID rule, and the multiset variable gets the rest. If the multiset variable also occurs in the RHS ID rule of the metarule, then all the categories which mapped into the multiset variable on the LHS are inserted into the new ID rule generated. An error will result if the RHS has a multiset variable and the LHS does not. Here is an example of the use of a multiset variable:

```
pass:   vp --> v, np, ...

        ==>

        vp --> vpass, opt(pp), ... .
```

This is a simple version of the passive metarule. VP, VPASS, etc. are assumed to be alias expressions. This metarule rule says

> For any ID rule which expands a VP as a V, an NP (the NP can occur anywhere in the daughter list of the ID rule), and possibly other stuff, create another ID rule for a VP expanding as a passive verb (VPASS), an optional prepositional phrase, and all the other stuff (remembering that ordering is specified independently in the LP rules).

A couple of technical points are worth noting: (i) the three dots (...) can sometimes get confused with other symbols (e.g. commas, the dot at the end of the rule, etc.). Thus it is advisable always to put spaces on either side of multiset variables. And (ii), multiset variables are sensitive to the use of regular expression operators elsewhere in the rule. In particular if an ID rule containing a C++ category (i.e. C must occur at least once) matches a metarule LHS of the form ? --> C, ... then a C** is put into the multiset variable (not a C++), since one C has already occurred.

2.4.2 Matching

Either of the ID rule descriptions in a metarule can take a WHERE clause. These WHERE clauses may not contain CONTROLS specifications, but they may contain IS specifications (defining variables) and MATCHES specifications. A MATCHES clause is used for the second sort of communication between the two ID rule descriptions in a metarule. It is conventional in GPSG metarules to assume a correspondence between like categories on either side of a metarule, except where differences are made explicit.

Consider the following auxiliary-initial metarule:

```
inv:    vp(aux, fin) --> h, vp

        ==>

        s(inv) --> h, s.
```

Here, H is an alias for a lexical head category (see the discussion above), VP(AUX, FIN), VP, S, etc., are alias expressions. This rule does not state that the S(INV) inherits everything from the VP(AUX,FIN), except BAR level, which is explicitly stated (via aliases) and similarly the daughter S inherits from the daughter VP. To do this we use a MATCHES clause. A MATCHES clause takes the same form as a CONTROLS

clause, and similarly it requires two IS clauses to define its variables, and they must precede it in the WHERE clause. The WHERE clause:

where VP1 is vp(aux, fin), SI is s(inv), VP1 matches SI.

defines the two variables VP1 and SI and then states that they match everywhere except where they differ. That is, all the bits of the categories which can be matched together and which are not explicitly specified in the metarule, must be identical in all uses of the metarule (so the created rule inherits them from the original rule).

The complete metarule might then look like this:

```
inv:    (VP1 –> h,VP2  where   VPt is vp(aux,fin)f
                               VP2isvp)
        ••).

        (81–> h, S2    where   SI is •(inv),
                               S2iss,
                               VP1 matches SI,
                               VP2 matches S2 ).
```

Again note that layout is not important. The manner in which clauses may be distributed among the two WHERE clauses is subject to the following rules:

(i) an IS clause must occur in the WHERE clause of the ID rule the variable is first used in, or earlier (i.e. LHS for a variable in RHS but not vice versa).

(ii) IS clauses defining variables must precede the MATCHES clauses they are used in.

2.4.3 Rulg names

The ID rules generated by metarules have names which encode the history of their metarule derivation. In the parse tree displays, these names are written in functional notation. For example:

VP1	name of an ID rule in the grammar;
PASS(VPl)	name of a rule produced by applying metarule PASS to VP1;
INV(PASS(VP1>)	name of a rule produced by applying PASS, then INV to VP1.

The rule-types that result may not be uniquely named - metarules can apply to rules in more than one **way.**

2.5 LP rules*

The linear precedence rules specify the ordering constraints on sister categories. The simplest form of lp rule is as follows:

```
<cat> << <cat>.
```

This simply states that one category must precede another (if they cooccur). The categories must be full category specifications (i.e. not just subfeatures, etc.), but may incorporate aliasing as desired.

Two other forms of LP statement are possible. Firstly, wherever you can have a single <cat>, you can have a collection, in braces {<cat>, ..., <cat>>. No ordering is specified among these categories (although other LP statements may do so). For example;

```
{n,v,p} << (np>,
```

is the same as

```
n << np.
v << np.
p << np.
```

Secondly, statements may be cascaded and transitivity is assumed. For example,

```
n << np << vp.
```

is the same as

```
n<<np.
np << vp.
n << vp.
```

Both these mechanisms can be mixed up arbitrarily. The normalisation process also expands the LP rules under the appropriate transitive closure.

The normalisation mechanism uses the expressions given in the rules (i.e. the alias expressions, etc.> to decide whether two categories are the same. Thus if the LP rules employ both the following expressions:

```
s
[cat,  [bar,23,[head,[major,+v,-n333
```

where one is simply an alias for the other, they will be viewed as different categories for the purposes of LP rule expansion. This will have no effect on the system's behaviour, but it will generate more LP rules than are required. In general, then, the user should avoid using alternative expressions for a single category in LP rules.

2.6 Feature coefficient defaults

Feature coefficient defaults specify default values for feature coefficents which are assigned if not privileged (see GPSG) or otherwise specified. The form of an FCD specification is:

fcd(<featname>,<excl>,<Lfeat>,<Pfeat>).

Here <featname> is the name of the feature whose default is to be defined, and <excl> is a list of feature names - see below. <Lfeat> and <Pfeat> are the lexical and phrasal default values respectively (features can take different defaults on lexical or phrasal (i.e. bar level at least 1) categories). Either can take any of the following forms:

the word FREE - meaning no default,
a feature specification for <featname> (or alias expression),
a feature name, the name of a coefficient of <featname>.

An FCD can only be specified for a given feature name in one place - some features (e.g. AGR) can appear in more than one place in the feature syntax, and hence in a given category. Only one of these instances can have defaults assigned for it. The exclusion list, <excl>, is used to direct the normalisation process to the correct instance. It consists of a list of feature names which must not be above the instance required in the feature tree. For example, an exclusion list [FOOT, CONJ] would mean that the default applies to an instance not in the FOOT feature or the CONJ feature. If there is only one instance of <featname>, <excl> can safely be left as []. Here is an example of an FCD:

fcd(case, [foot], free, ^case).

Here CASE defaults to FREE on lexical categories and ^CASE on phrasal categories. The stipulation of FOOT in the exclusion list, prevents the FCD applying to the instance of CASE in FOOT (e.g. [FOOT, [SLASH, [CAT, [HEAD, CASE]]]]) being used.

Any feature which is not privileged and which does not get a value, either from a rule or by default, is given the value UNSPEC.

2.7 Feature cooccurrence restrictions

Feature cooccurrence restrictions specify pairs of feature values which must or must not cooccur. The form of an FCR is:

fcr(<name1>,<excl1>,<feat1>,<name2>,<excl2>,<feat2>).

Here, <name1> and <excl1>, and <name2> and <excl2>, combine to specify particular instances of the features <name1> and <name2> as in the FCD's above. <feat1> and <feat2> can take values as in <Lfeat> and <Pfeat> above and additionally be one of the forms:

```
unspec
not(uniptc)
not (..«)
```

where ... is *an* <Lfeat> or <Pfeat> above. The single word UNSPEC as a value Deans that the feature value was not specified at all by feature instantiation. The restriction specifed is that if feature <na»el> has (doesn't have) the value specifed, then feature <na«e2> «ust have (not have) the value specifed. Here are soae examples of FCR's:

```
fcr(minor, [foot], vforr, n, [foot], -n).
fcrtainor, [foot], vfori, v, [foot], +v).
```

Together, these FCR's ensure that any category with a coefficient VFORM (assumed to be encoding of verb torphology, etc.) on MINOR really is a verb, i.e. -N, *V.

2.S Root admtcsibility conditions

The root admissibilty conditions are not part of the SPSS theory, but *are* used to prevent the parser from generating unwanted alternative parses for things. The for* of an RAC is:

```
rac(<name>, <excl>, <feat>).
```

Here the arguments are just like one half of an FCR specification (see above). Every time a tree is generated the RAC's are checked. If the root node of the tree has the feature <name> which has the value <feat> then the tree is rejected. Here is an example of an RAC:

```
rac(bar, [foot], not(unspec))•
```

This rac ensures that the bar level of any root category is not unspecified (it is possible sometimes to get unspecified bar levels when the Conjunct Realization Principle ('CRP' henceforth - see Gazdar, Klein, Pullum and Sag (1982) for discussion of the CRP) is being used).

2.9 The lexicon

.The lexicon consists of lexical category rules which provide the association between words and the parts of speech to which they belong. The ford of a lexical category rule is:

```
<na»e> i <cat> ->- <wordl>, ... ,<wordN> •
```

Here <na<*e> is JIS in ID rules (see above), <cat> is a feature expression (or alias, etc.) and <wordl>, ... ,<wordN> is just a list of words, separated by co»mas. A word aay not start with a capital letter. A lexical rule says that each of the words on the right hand side is an instance of the category on the left hand side, and *may* be used in any ID rule with the same name (or whose name's functor is the same). This latter point is the lexical subcategorisat ion mechanism, which can be turned off using the switch NOLEX (see chapter 5). Note that <cat> must be of bar level LEXICAL, otherwise a normalisation error occurs.

The lexical categories need to be fully specified in all features which are not specified by default, or by the corresponding category in the rules that introduce the category. Leaving a feature unspecified in the lexicon does not mean it can take any value, indeed, the value will be set to UNSPEC, and cannot be changed. Here are a couple of examples of lexical category rules:

```
npl: n ->- kirn, lee, sandy.
np2; n ->- book, house, hill.
```

These two both define nouns to be used in two different NP rules - the one for proper nouns (NP1) and one for ordinary count nouns. The relevant ID rules (night be:

```
npl s np –> n.
np2: np –> det, adj*» ,n.
```

assuming suitable aliases for NP, N, DET, AOJ, etc.

The third and fourth lexical rules below provide an example of the use of features in the lexicon which do not specifically occur in the ID rules.

```
vtr(s): v(s) ->- sees, likes, loves,
-vir(p): v(p) ->- see, like, love.
```

Suppose we have an ID rule:

```
vtr: vp ->- v, np.
```

Then the two lexical rules above would both apply to fill in the V, taking their number feature (S or P) with them. HFC, CAP, etc., could then operate to communicate this number feature to the VP, subject NP, and so on. In the lexical rules, VMS) and V(P) are examples of complex aliases. A suitable alias might be:

```
alias<v(NUMB),
[cat,tbar,lexical ],[head,[major,+vf-n3,Cagr,[number ,NUMB]]]]).
```

It is worth noting that the parser will run more efficiently, particularly in a search for all parses, if a given word has only one lexical category associated with it. In grammar testing applications, this should not be too inconvenient, although it means that, for example, it is best to have several non-overlapping groups of proper nouns for the different cases (nominative, accusative, etc.).

2,10 Assumptions about the GPSG formalism

ProGram makes various assumptions about use of feature names, etc., and places certain constraints on rule format, etc., which are not explicitly discussed in the GPSG literature. This section contains a summary of these constraints.

The following features *are* assumed to be present:

CAT - tht feature representing a basic 'saall' category (i.e. corresponding to CAT in 6P82, not CAT[1]). The only tssuted significance of CAT is that CONJ is not a subfeature of it (or at least, the instance of CONJ used for the CRP is not).

FOOT - the FOOT feature used by the Foot Feature Principle CFFP' hereafter, see GP82 for discussion). It need not be present if HOFQOT is on (see 5.4).

CONJ - feature used by CRP. Categories which have a CONJ feature which is not [s] (i.e. they *ire* not *CQNJ) are privileged - they do not take defaults. CONJ should not be a subfeature of CAT. See (v), below.

HEAD - the head feature used by HFC. This lust not be a subfeature of FOOT (although there aay be a head feature under FOOT, but the HFC will not appply to this).

MAJOR * used in the definition of a head category. Head categories aust have no coefficient for MAJOR specified.

BAR - used by HFC and lexical fubcategorization routines. Its first coefficient (in the feature syntax specification) lust be the bar level, taking values LEXICAL, 1, 2, 3, ... (positive integers only, signifying increasing level. LEXICAL * BI. All values to be used should be stipulated in the feature syntax specification.

LEXICAL * lowest bar level. This lust have no subfeatures in the feature syntax ,specification (although the systei will add one when doing lexical isubcategorization).

AGR - feature used by CAP - as with HEAD, instance §ust not be under FOOT.

If any of the above features are absent, then the corresponding principles, etc., simply Mill not work * most of the*} will fail. The switches NOCQNJ, NOFOOT, and so on, lay bl set to overcome this (see chapter 4).

There are also some restrictions on the format of rules and metarules that need to be observed:

(i) ID rule descriptions occurring in metarules may not contain CONTROL AGREEMENT clauses.

(ii) REPEATER categories (those marked with regular expression operators #* or ++> generate independent instances. In particular, using variables in a repeater category, or having a repeater as head category or part of a control pair, will fail (the category will not be constrained as expected).

(iii) Having variables in HEAD, FOOT or AGR can cause unpredictable results when using HFC, FFP, CAP. For example, FFP will sometimes 'lose' the bindings in a foot feature.

<iv) CONTROLS clauses in an ID rule should come after the associated variable binding clauses.

(v) The extension restriction in CRP does not apply to the -feature CONJ (i.e. the CONJ features of the daughters do not have to extend that of the mother). The mother's CONJ feature is alHays set to NCONJ. Also, CRP as implemented requires that the mother be the MAXIMAL category which the daughters all extend. A small modification to the source code Hill change this - see the note in CRP system module (.../program/sys/crp, in the UNIX version of ProGram). However, this modification will slow the parser down considerably.

3. Normalisation

To use ProGram, you need to have a GPSG grammar. This grammar can have up to nine components, and we shall assume that each component is in a different file (this is not strictly necessary, but it makes life easier). For full details of specifying these components, see chapter 2. We shall assume here that the components of our grammar have the following names:

FEATS1	feature syntax specification
ALIAS1	feature alias specification
IDRULES	immediate dominance rules
LEXICON	the lexicon
LP	linear precedence rules
METARULES	metarules
FCD	feature coefficient default specifications
FCR	feature co-occurence restrictions
RAC	root admissibility conditions

3.1 Normalisation of data modules

The first thing that has to be done is normalisation of these data modules. Normalisation uses the information in FEATURES and ALIASES to translate and check all the rest of the data. The data is translated into an internal form in which the aliases have been fully expanded and a few other minor bookkeeping functions have been carried out. If, in the process of doing this, the system detects an error, for example a badly specified feature, then a mishap message will be printed. In a few cases a warning message is printed when strange, but not actually illegal, data is encountered.

Normalisation has to be done before parsing or testing, since the parser uses the internal, normalised forms. The primitive normalisation commands are:

```
NORMID,NORMLEX,NORMLP,NORMMETA,NORMFCD,NORMFCR,NORMRAC
```

In each case the default input and output are the terminal, so if you just type

```
normid.
```

to the top-level (Prolog) prompt, you will be prompted to type an ID rule, which will get normalised and displayed. This is not usually what is required, it is more common to redirect the data using TO and FROM:

```
normid from idrules to nidrules.
```

This means 'read in all the rules from the file IDRULES, normalise them and write the normalised versions to the file NIDRULES'. In order to do this, the system needs to read the data modules for features and aliases. If these modules have not already been loaded (e.g. because you have already normalised something), then the system needs to know the appropriate file names and it will ask you for these names when it needs them, e.g.

```
FEATURES data is now required.
Please type in data file name :
```

You just type the name, followed by a period. You can also specify the relevant names in the original command employing a USING clause:

```
normid from idrules to nidrules using feats1
for features using alias1 for aliases.
```

The 'for features' bit is necessary - it tells the system that the file it is reading (FEATS1) is the data module 'features'. Without the FOR clause, the data would be read, but the system would still prompt for features data.

Notice that a command can spread over several lines if necessary - in general, Prolog does not care about new lines or spaces between words, so you can lay out data files (e.g. the ID rules) in any format you choose. The important thing is to remember the period at the end of each command/rule/feature specification. The only other point to note is that the system prints little messages as it loads system modules, just to keep you informed. For example, this is what would happen if we had just loaded the system and we typed:

```
?- normid from idrules to nidrules using feats1 for features.

(loading normrules)
(loading normid)
ALIASES data is now required.
Please type in data file name : alias1.

(loading normfeats)

<longish pause>

yes

?-
```

Three system modules were loaded (NORMRULES, NORMID, and NORMFEATS) and the aliases data was requested - the features data was given in the command. If we now type:

```
?- normlex from lexicon to nlexicon.
(loading normlex)

yes

?-
```

we see that only one module was loaded, since NORMFEATS, FEATURES and ALIASES, the other three modules required, have already been loaded.

If for some reason you do not want a data module to be loaded (e.g. your grammar is so simple there are no aliases), you can use the word NONE as a filename, either in the command, or when prompted.

As mentioned above, all the components of the grammar, apart from the features and aliases themselves, have to be normalised and the normalised data saved. Thus we could type:

```
?- normmeta Iron Metarules to nmetarules.
?- normfcr fro* fcr to nfcr.
<etc>
```

The MORECQMMS library provides some commands which do the normalisation •ore easily, by assuming standard nafies for all the data Modules - see chapter 5.

Normalisation of ID rules includes all the processing required for the Control Agreement Principle and lexical subcategorization. It does not include Head Feature Convention (HFC) processing (see 3.3, below). The MORECOMMS library provides commands which do normalisation and the HFC automatically, but if metarule expansion is required, this has to be done between the two. MORECOMMS provides a command to do that automatical 1y, too.

Before using the grammar, you may well want to expand it using the metarules. This should be done after normalisation, but before HFC processing. See 3.2, below.

By using the FILTER library, you can combine several data files together to be viewed as one. Thus, if your ID rules *are* split between several files, you can give a command like:

```
normid from idrulesl then idrules2 to nidrules.
```

which combines data from IDRULES1 and IDRULES2, normalises it and stores the result in NIDRULES. Similarly, you can select particular rules *iron* a file by using the SOMEOF filter (see 5.5.1 for details).

In general, normalisation does not display anything except mishaps and warnings on your screen. If you want to watch what is happening, set the switches TI and TO to on (see 5.4 for details). Note that Prolog syntax errors in your data files may not be apparent * in standard Prolog the error message gets written into the output data file, not the terminal (see 5.3).

The rest of this section consists of examples of each of the various grammar modules being normalised. Output is sent to the terminal (since it is not redirected) so that any syntax errors are picked up properly. After all the modules have been successfully normalised with no errors, the data can safely be saved in data files, either by individual commands as below, but with redirection (FROM) clauses, or using one of the commands N0RM_6RAM or NORN.HETA" from the MORECOMMS library (see 5.5.2).

For convenience,, tracing of INPUT data has been turned on, so that the reader can see what it is we are normalising.

```
?- ti on.                              SWITCH ON INPUT TRACING
trace_in is on
yes

?- normid from idrules.                NORMALISE SOME ID RULES

( loading normrules )
( loading normfeats )

PROGRAM NOW PROMPTS FOR SOME DATA MODULES, SINCE THEY WERE NOT SPECIFIED IN
THE COMMAND. NOTE THAT ONCE THEY ARE LOADED, PROGRAM NEVER PROMPTS AGAIN.
features data is now required
Please type in data file name !: features.
                                       REMEMBER THE FULL STOP
aliases data is now required
Please type in data file name !: aliases.
( loading normid )
```

ID rule normalisation can now proceed - notice that the traced input is interspersed with the output. Normally the tracing would be on the screen and the output data would be redirected to a file. The banner comment at the top, identifying the data module, is produced for all data files, with an appropriate heading. Also, if a header string is specified by the user, it appears in this banner. The /* and */ bracketing tells Prolog that the banner is a comment and not part of the data.

```
/*
ProGram Grammar Development System
Version 1, 29/4/84

      NORMED ID RULES
*/
                                       THIS IS THE INPUT:
s : v(2) --> _1 , _2 where _1 is n(2,[nom]) , _2 is h(1) , _1 controls
      _2.

                                       AND THIS IS THE NORMALISED RESULT:
baserule(s,no,[root, [cat, [bar, [2]], [head, [major, [v]], _1]], _2, _3],[c(c
      trl,_4,[2],no,[root, [cat, [bar, [2]], [head, [major, [n]], [minor,
      [agr ! _5], [case, [nominative]]]]], _6, _7]), c(ctrl,_8,[1],no,[root
      , [cat, [bar, [1]], [head, [major, _9], [minor, [agr ! _5] ! _10]]],
      _11, _12])],[ctrl controls ctrl]).

vp_1 : v(1) --> h.

baserule(vp_1,[],[root, [cat, [bar, [1]], [head, [major, [v]], _1]], _2,
      _3],[c(_4,_5,[lexical, [vp_1]],no,[root, [cat, [bar, [lexical,
      [vp_1]]], [head, [major, _6], _7]], _8, _9])],[]).

.... data omitted here ....
```

```
THE ERRORS FOLLOWING ARE EXPECTED, SINCE THE CATEGORIES IN THE RULE ARE VERY
UNDERSPECIFIED:

conj : [root, [cat, _1], _2] --> c(_2) , [root, [cat, _1], * conj] where
       _1 is bar , _2 is conj.

Warning: BAR coefficient not specified - no lexical check
Involving: [root, [cat, [bar, _1], _2], _3, [conj, [*]]]

baserule(conj,no,[root, [cat, [bar, _1], _2], _3, [conj, _4]],[c(_5,_6,[lexica
       l, [conj]],no,[root, [cat, [bar, [lexical, [conj]]], [head, [major,
       [conj, _4]], _7]], _8, _9]), c(_10,_11,_1,no,[root, [cat, [bar,
       _1], _12], _13, [conj, [*]]])],[]).

coord : root --> [root, [conj, neither]] , [root, [conj, nor]] ++.

Warning: No BAR coefficient found in category
Involving: [root, _1, _2, [conj, [neither]]]

Warning: No BAR coefficient found in category
Involving: [root, _1, _2, [conj, [nor]]]

baserule(coord,no,[root, _1, _2, _3],[c(_4,_5,no,no,[root, _6, _7, [conj,
       [neither]]]), c(_8,_9,no,plus,[root, _10, _11, [conj, [nor]]])],[]).

.... etc ...

end_of_file.                          GENERATED BECAUSE INPUT IS BEING TRACED

yes

ID RULES SUCCESSFULLY NORMALISED. NOW DO THE LEXICON IN A SIMILAR WAY:

?- noralex from lexicon.

( loading noralex )
/*
ProGram Grammar Development System
Version 1, 29/4/84

       NORMED LEXICON
*/
                                      INPUT DATA:
vp_1(sf) : v(0,[sing, fin]) ->- jumps , runs , sings.
                                      OUTPUT DATA:
lexrule(_1,lex(vp_1(sf)),[root, [cat, [bar, [lexical, [vp_1]]], [head,
       [major, [v]], [minor, [agr, [singular]], [vform, [finite], [auxiliary
       , [-]], [inverted, [-]]]]]], _2, _3]) :- once(member(_1,[jumps,
       runs, sings])).
```

```
vp_1(pf) : v(@,[plur, fin]) -)- jump , run , sing.

lexrule(_1,lex(vp_1(pf)),[root, [cat, [bar, [lexical, [vp_1]]], [head,
      [major, [v]], [minor, [agr, [plural]], [vform, [finite], [auxiliary,
      [-]], [inverted, [-]]]]]], _2, _3]) :- once(member(_1,[jump, run,
      sing])).

.... lots omitted ....

SLIGHTLY DIFFERENT OUTPUT FORMAT IF THERE IS ONLY ONE WORD:

pp(by) : p(@,by) -)- by.

lexrule(by,lex(pp(by)),[root, [cat, [bar, [lexical, [pp]]], [head, [major,
      [p, [by]]], _1]], _2, _3]).

.... etc. ....

conj(b) : c([conj, nor]) -)- nor.

lexrule(nor,lex(conj(b)),[root, [cat, [bar, [lexical, [conj]]], [head,
      [major, [conj, [nor]]], _1]], _2, _3]).

end_of_file.

yes
```

Now the LP data. This is slightly different. Because the LP rules are expanded as well as normalised, all the data is read in before any output is produced, so with TRACE_IN on, all the tracing output comes first. Also, there are usually more output data items than input ones.

```
?- normlp from lp.

( loading normlp )
                                        INPUT DATA (ALL OF IT):
[root, [cat, [bar, lexical]]] << {[root, [cat, [bar, 1]]] , [root, [cat,
      [bar, 2]]]}.
p(@) << n(2) << v(1).
a(1) << n(1) << p(2).

[root, [conj, neither]] << [root, [conj, nor]].
[root, [foot, nil]] << [root, [foot, cat]].
end_of_file.
                                        OUTPUT DATA:
lp([root, [cat, [bar, [lexical, _1]], _2], _3, _4],[root, [cat, [bar, [1]],
      _5], _6, _7]).
lp([root, [cat, [bar, [lexical, _1]], _2], _3, _4],[root, [cat, [bar, [2]],
      _5], _6, _7]).
```

```
lpUroot, [cat, [bar, [23], [head, lujor, [n]], .133, .2, .3],[root, [cat,
      [bar, II]], [head, [ujor, [v]], .411, .5, .4]).
lpUroot, (cat, [bar, [lexical, .13], [head, [•ajor, (p, .2]], .313, .4,
      .51,[root, (cat, [bar, [23~J, [head, [ujor, In33, >33, .7, .83).
lp([root, [cat, [bar, [133, [head, [lajor, (n33, .133, .2, _33,troot, cat,
      [bar, [23], [head, (•ajor, [p, .433, .533, .6, .73).
lp([root, [cat, [bar, [133, [head, [ujor, tall, .133, _2, .33,[root, teat,
      [bar, [133, [head, [ujor, In33, .433, .5, .63).

lp([root, .1, .2, [conj, [neither]]],[root, _3, _4, tconj, [nor333>.
lpltroot, .1, [foot, [nil]]', .23,[root, .3, [foot, [cat, .4, .533, .63).
ipUroot, [cat, [bar, [lexical, .133, [head, [ujor, [p, .233, .333, .4,
      .53,[root, [cat, [bar, [133, [head, [ujor, tv3), .633, .7, .63).
lp([root, [cat, [bar, [133, [head, tiajor, tail, .133, .2, .3],[root, (cat,
      (bar, [233, [head, dajor, [p, .413, .513, .6, .73).
yes

NETA-RULE NORHAL1SATION IS LIKE ID RULE NORMfiLISATION:

?- norueta froi tetarules.

( loading norueta )
/•
Pro6rai Graeiar Development Syitei
Version 1, 29/4/64

      HORNED HETA RULES
»/
pass : v(l) -> ... , n(2) «=) v(l,[passl) -> ... , opt(p(2,by>).

Ktarulelpass,baserule(.i,.2,[root, [cat, (bar, (133, [head, teajor, (v33,
      .333, .4, .53,[c^6,.7,[23,no,[root, [cat, [bar, (233, (head,
      [•ajor, In33, .833, .9, .H3)3,.ll),str(J2),baserule(.l,[pais
      ! .21,[root, (cat, (bar, [HI, [head, [ujor, [v]], (•inor, .13,
      [vfori, [passive], [auxiliary, (-33, (inverted, [-33331], .14,
      _151,(cl_16,_17,[2],opt,[root, [cat, (bar, [233, (head, lujor,
      tp, [by333, .1833, .19, .283) ! .123,.21)).

.... ÄC MM

yes

FCDf FCR AND RAC NORMALISA!IW ARE SIMILAR. TYPICAL COMMANDS ARE:

 ?- nortfeti froi led.'

 ?- nortfer frot lcr.

 ?- norirac froi rac.
```

3.2 Metarule application

This section describes how to apply the metarules to the ID rules in a ProGram grammar. Grammar expansion is achieved by calling the command EXPAND. EXPAND reads normalised ID rules from its input stream and produces the expanded grammar of (normalised) ID rules as its output. It assumes that the normalised metarules have already been loaded - normally they are specified by a USING clause in the command. Metarule application must take place before HFC processing (see 3.3, below).

A typical use of the EXPAND command might be:

```
?- expand from nidrules to eidrules
?- using nmeta for normed metarules.
```

This command takes the (normalised) ID rules from a file called NIDRULES and expands them using the (normalised) metarules in the file NMETA. The resulting grammar (including the original ID rules) is saved in a file called EIDRULES. Notice that the USING clause also specified which data module it was using (the NORMED METARULES module). Otherwise the system would have prompted the user for this information.

If the TRACE_OUT switch is set (see 5.4), then the names of the new rules are printed out as they are generated, together with the pass number: pass 1 contains the original rules, pass 2 contains those generated from pass 1, pass 3 contains those generated from pass 2, and so on. Each rule is printed out as follows:

```
<original name of rules> <meta rules applied>
```

For example:

```
pass 2 -  vp_1 pass stm1
```

announces a new rule generated on pass 2 by metarule STM1 from the pass 1 rule VP_1 PASS (i.e. result of metarule PASS applied to ID rule VP_1). Note that full names are used here - truncation of names, and also insertion of brackets to form a functional notation, are only done on parse displays.

The expansion algorithm makes repeated passes over the ID rules, attempting to apply each metarule to each of the ID rules generated on the pass before (but no metarule can apply to an ID rule twice). The EXPAND command produces its output in the order it generates it, i.e. the original ID rules first, followed by those produced on the first pass, then those produced on the second pass etc. Between each such block of rules a Prolog comment announcing the pass number is also put into the output stream. This is purely for the convenience of any user who might wish to examine the raw output.

Let us examine an example of doing metarule expansion on the demonstration grammar. The example uses the command NORM_AND_EXPAND provided in the MORECOMMS library (see 5.5.2) which normalises the ID rule and the metarules, does the expansion and then does HFC processing on the resulting rules. Standard data file names are assumed for this command, so no explicit request for features, aliases, etc., is made. Also, since the output data is written to data files (with standard

names), no output is normally produced at the terminal. But in this example we switch TRACE_OUT on first, so that output is displayed as well as saved.

```
?- t on.                              TURN TRACING ON
trace_out is on
yes
?- norm_and_expand.                   GIVE THE MAIN COMMAND

( loading normrules )                 SEVERAL SYSTEM MODULES ARE LOADED.
( loading normfeats )
( loading normid )

                                      THE REST OF THE OUTPUT IS TRACING
                                      FIRST, THE ID RULES, NORMALISED

baserule(s,no,[root, [cat, [bar, [2]], [head, [major, [v]], _1]], _2, _3],[c(c
      trl,_4,[2],no,[root, [cat, [bar, [2]], [head, [major, [n]], [minor,
      [agr | _5], [case, [nominative]]]]], _6, _7]), c(ctr],_8,[1],no,[root
      , [cat, [bar, [1]], [head, [major, _9], [minor, [agr | _5] | _10]]],
      _11, _12])],[ctrl controls ctrl])

.... lots left out here ....

baserule(pp,[],[root, [cat, [bar, [2]], [head, [major, [p, _1]], _2]],
      _3, _4],[c(_5,_6,[lexical, [pp]],no,[root, [cat, [bar, [lexical,
      [pp]]], [head, [major, _7], _8]], _9, _10]), c(_11,_12,[2],no,[root,
      [cat, [bar, [2]], [head, [major, [n]], _13]], _14, _15])],[])

Warning: BAR coefficient not specified - no lexical check
Involving: [root, [cat, [bar, _1], _2], _3, [conj, [*]]]

THIS ERROR AND THE FOLLOWING ONES ARE EXPECTED - THE COORDINATION RULES
DO NOT HAVE LEXICAL CATEGORIES, ETC.

baserule(conj,no,[root, [cat, [bar, _1], _2], _3, [conj, _4]],[c(_5,_6,[lexica
      l, [conj]],no,[root, [cat, [bar, [lexical, [conj]]], [head, [major,
      [conj, _4]], _7]], _8, _9]), c(_10,_11,_1,no,[root, [cat, [bar,
      _1], _12], _13, [conj, [*]]]])],[])

Warning: No BAR coefficient found in category
Involving: [root, _1, _2, [conj, [neither]]]

Warning: No BAR coefficient found in category
Involving: [root, _1, _2, [conj, [nor]]]

baserule(coord,no,[root, _1, _2, _3],[c(_4,_5,no,no,[root, _6, _7, [conj,
      [neither]]]), c(_8,_9,no,plus,[root, _10, _11, [conj, [nor]]])],[])

baserule(top,no,[root, [cat, [bar, [2]], [head, [major, [v]], _1]], _2,
      _3],[c(_4,_5,[2],no,[root, [cat, [bar, [2]], _6], [foot | nil],
      _7]), c(_8,_9,[2],no,[root, [cat, [bar, [2]], [head, [major, _10],
      _11]], [foot, [cat, [bar, [2]], _6]], _12])],[])
```

```
                                        NOW THE METARULES ARE NORMALISED
( loading noraieta )

....  trice output here ,,,,
                                        THE ID RULES 6ET EXPANDED
( loading expand )

pass 1 * top                            THIS TRACE ROUTPUT IS PRODUCED FOR EACH
pass 1 - coord                          RULE PRODUCED BY THE EXPANSION PROCESS.
pass i * conj
pass 1 - pp
pass 1 - ap_2                           FIRST OF ALL, THE INITIAL RULES
pass i * apj                            (BACKWARDS!)
pass 1 - nb_2
pass 1 - nbj

pass 1 * np.2
pass i - npj
pass 1 - vp_5
pass i - vp.4
pass 1 - vpJS
pass 1 - vp.2
pass 1 * vp.1
pass 1 - s

pass 2 - pp stil                        NOH THE RULES GENERATED ON THE SECOND PASS
pass 2 - nb.l relci                     FORMAT IS ID RULE NAHE FOLLOWED BY
pass 2 - np.l stil                      HETA-RULE NAHE (E.G. FIRST ONE IS A RULE
pass 2 - vp.5 inv                       RESULTIN6 FROM APPLYING STHI TO THE PP
pass 2 - vp_4 stii                      RULE)
pass 2 - vp_3 pass
pass 2 - vp_3 pass
pass 2 - vpj stii
pass 2 - vp.J stil

pass 2 * vp.2 pass
pass 2 - vp.2 stil
pass 3 - vp.2 stil pass                 PASS 3 CAN ONLY USE RULES GENERATED ON
pass 3 - vp_3 pass stil                 PASS 2. THE RULE 'vp_2 stil pass' IS THE
pass 3 - vp_3 pass stil                 RESULT OF APPLYING STHI TO THE PASS 2 RULE
pass 3 - vp_3 stil pass                 'vp.2 pass'
pass 3 - vp_3 stil pass
pass 3 * vp_3 stil pass
pass 3 - vp_3 stil pass
```

The output above is produced as the rules ire generated. The output below is the real traced output as the rules are written to the data 'file. Again, since there are rather a lot, most of then have been omitted.

```
... a couple emitted here ...

baserule(vp..2,[],(root, feat, [bar, tl]], [head, (iajor, N13, .1]}, .2,
       _3],Ec(_4,_5,(lexical, [vp_2]3,no,troot, teat, (bar, [lexical,
       (yp_2333, [head, dajor, _1), _733, _8, _9]>, c(_11,_11,(21,no,[root,
       (cat, (bar, [23], [head, (Mjor, [i>33, .12]], .13, _141>1,[]>

baserule(vp..3,[],[root, [cat, [bar, M l , [head, (ujor, Iv]), .1)3, .2,
       .3},i:c(.4,.S,(lexical, (vp_3)],no,[root, (cat, (bar, [lexical,
       (vp.311], [head, tiajor, .«, .711, .8, .91), c(.ll,.ll,[2J,no,[root,
       [cat, [bar, [2]], [head, taajor, [n33, .12]], .13, .14]), c(.15,.16,[
       2],no,[root, [cat, (bar, [213, [head, tiajor, In]], _17]], _18,
       .193)3,[»

... all the rest of the past 1 data (i.e. original rules) here ...

                                        THE FIRST NEW RULE:

baseruletppJstilMroot, [cat, [bar, [233, [head, lujor, [p, .13], .233,
       [foot, [cat, (bar, [233, [head, (lajor, [n33, [einor, .3, [case,
       t*33:i33], _43,Ic(_5,_6,tlexical, [pp]],no,[root, (cat,~[bar, [lexical
       , [pp]]], [head, tnjor, .73, .83], .9, _1I3)],_11)

... a fen mire left out ...

                                        THE VP RULES ABOVE GENERATE
                                        THE F0LL0HIN6 RULES ON PASS 2

baserule(vp..3,[pass],(root, [cat, [bar, M l], [head, [lajor, tv]3, (linor,
       .1, (vfori, [passive], [auxiliary, [-]], [inverted, [-]]]]]],
       .2, ,33,[c(.4,.5,[2],opt,[root, [cat, [bar, [2]], [head, Eiajor,
       [p, [by]}], .633, _7, 83), c(_9,_IB,[lexical, [vp_3J],no,[root,
       [cat, [bar, (lexical, ivp.3]]], [head, [ujor, .11], .12]], .13,
       .143), c(.lS,.16,(23,no,[root, (cat, (bar, (21], (head, Uajor,
       in]], .173], .18, .19])],.28)

baserule(vp.3,[pass],(root, (cat, (bar, [|}], [head, (ujor, [vJ], [einor,
       _1, (Won, [passive], [auxiliary, (-]], [inverted, [-133]}],
       .2, .33,(c(_4,.5,[2],opt,[root, [cat, (bar, [213, (head, [ujor,
       (p, (by]]], .63], .7, .83), c(.9,.ll,[lexical, (vp.33],no,[root,
       [cat, (bar, (lexical, ivp.3]]]^ (head, [ujor, .111, .1233, .13,
       _143), c(_15,_li,[23,no,[root, [cat, (bar, 1211, (head, [•ajor,
       tn]], .173], .18, .19])],.28)

baserule(vp.3,(stil3,(root, (cat, (bar, [1J], (head, dajor, [v33, .1]],
       (foot, [cat, (bar, [2]], (head, [ujor, [n]3, dinor, _2, (case,
       V m m ,  .31,(c(.4,.5,[lexical, [vp.3]3,no,(root, (cat, (bar,
       [lexical, tvp.3333, (head, (ujor., .63, .733, .8, .93), c(.ll,.ll,t23
       ,no,[root, [cat, [bar, (233, (head, (aajor, [nil, _1233, _13, _14])],_
       15)
```

```
baserule(vp_3,[stm1],[root, [cat, [bar, [1]], [head, [major, [v]], _1]],
      [foot, [cat, [bar, [2]], [head, [major, [n]], [minor, _2, [case,
      [*]]]]]], _3],[c(_4,_5,[lexical, [vp_3]],no,[root, [cat, [bar,
      [lexical, [vp_3]]], [head, [major, _6], _7]], _8, _9]), c(_10,_11,[2]
      ,no,[root, [cat, [bar, [2]], [head, [major, [n]], _12]], _13, _14])],_
      15)

baserule(vp_2,[pass],[root, [cat, [bar, [1]], [head, [major, [v]], [minor,
      _1, [vform, [passive], [auxiliary, [-]], [inverted, [-]]]]]],
      _2, _3],[c(_4,_5,[2],opt,[root, [cat, [bar, [2]], [head, [major,
      [p, [by]]], _6]], _7, _8]), c(_9,_10,[lexical, [vp_2]],no,[root,
      [cat, [bar, [lexical, [vp_2]]], [head, [major, _11], _12]], _13,
      _14])],_15)

baserule(vp_2,[stm1],[root, [cat, [bar, [1]], [head, [major, [v]], _1]],
      [foot, [cat, [bar, [2]], [head, [major, [n]], [minor, _2, [case,
      [*]]]]]], _3],[c(_4,_5,[lexical, [vp_2]],no,[root, [cat, [bar,
      [lexical, [vp_2]]], [head, [major, _6], _7]], _8, _9])],_10)
```

AND NOW THE PASS 3 RULES

```
baserule(vp_2,[stm1, pass],[root, [cat, [bar, [1]], [head, [major, [v]],
      [minor, _1, [vform, [passive], [auxiliary, [-]], [inverted, [-]]]]],
      [foot, [cat, [bar, [2]], [head, [major, [p, [by]]], [minor, _2,
      [case, [*]]]]]], _3],[c(_4,_5,[lexical, [vp_2]],no,[root, [cat,
      [bar, [lexical, [vp_2]]], [head, [major, _6], _7]], _8, _9])],_10)

baserule(vp_3,[pass, stm1],[root, [cat, [bar, [1]], [head, [major, [v]],
      [minor, _1, [vform, [passive], [auxiliary, [-]], [inverted, [-]]]]],
      _2, _3],[c(_4,_5,[2],opt,[root, [cat, [bar, [2]], [head, [major,
      [p, [by]]], _6]], _7, _8]), c(_9,_10,[lexical, [vp_3]],no,[root,
      [cat, [bar, [lexical, [vp_3]]], [head, [major, _11], _12]], _13,
      _14])],_15)

... etc ...
```

NOW HFC PROCESSING TAKES PLACE

```
( loading hfc )
baserule(s,no,[root, [cat, [bar, [2]], [head, [major, [v]], [minor, [agr
      | _1] | _2]]], _3, _4],[c(ctrl,_5,[2],no,[root, [cat, [bar, [2]],
      [head, [major, [n]], [minor, [agr | _1], [case, [nominative]]]]],
      _6, _7]), c(ctrl,no,[1],no,[root, [cat, [bar, [1]], [head, [major,
      [v]], [minor, [agr | _1] | _2]]], _8, _9])],[ctrl controls ctrl])

... all ok until we reach coordination rules ...

Warning: HFC failed
Involving: idrule(coord)

baserule(coord,no,[root, _1, _2, _3],[c(_4,_5,no,no,[root, _6, _7, [conj,
      [neither]]]), c(_8,_9,no,plus,[root, _10, _11, [conj, [nor]]])],[])
```

```
Warning: HFC failed
Involving: idrule(top)

baserule(top,no,[root, [cat, [bar, [2]], [head, [major, [v]], _1]], _2,
      _3],[c(_4,_5,[2],no,[root, [cat, [bar, [2]], _6], [foot | nil]],
      _7]), c(_8,_9,[2],no,[root, [cat, [bar, [2]], [head, [major, _10],
      _11]], [foot, [cat, [bar, [2]], _6]], _12])],[])

THE FAILURES ABOVE ARE EXPECTED  -   THE RULES DO NOT HAVE HEADS.
                                     NOW ON WITH THE NEW RULES:

baserule(pp,[stm1],[root, [cat, [bar, [2]], [head, [major, [p, _1]], _2]],
      [foot, [cat, [bar, [2]], [head, [major, [n]], [minor, _3, [case,
      [*]]]]]], _4],[c(_5,_6,[lexical, [pp]],no,[root, [cat, [bar, [lexical
      , [pp]]], [head, [major, [p, _1]], _2]], _7, _8])],_9)

... and so on, without any problems ...

yes                                  PROLOG REPORTS THE COMMAND
                                     SUCCESSFULLY COMPLETED
```

Notice that VP_3 and PASS generated two identical new rules in pass 2. This is because VP_3 has two identical NP's for the metarule to match against. ProGram makes no check that the rules generated are not identical, so users should filter duplicate rules out afterwards if they want to. Note that duplicate rules in early passes can lead to more duplication in later ones.

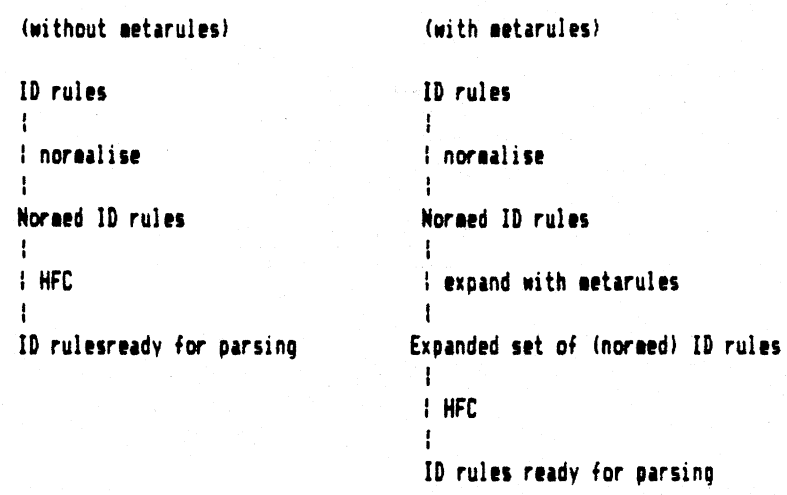
Notice also that the output from metarule expansion is rather unreadable. It is possible to decide from the data above whether all and only the expected rules were produced, but it is not easy. The rule-name output helps catch gross generation errors, but for finer testing, the parser is best used.

3.3 The Head Feature Convention

The head feature convention (HFC) operates in the manner specified in GP82. See 2.4.3 for details of how to specify a head category in an ID rule.

The system's handling of HFC is not done during parsing, but as a preprocess which takes place between normalisation (of ID rules) and parsing. This preprocess is not incorporated into the normalisation process because expansion using metarules (see 3.2) must normally come between normalisation of ID rules and the HFC processing. Thus the

normal sequence of processing required is:



The primitive command for doing the head feature processing is called HFC. It is used just like NORMID, etc. (see 3.1): it expects a file of normalised ID rules as input, and it produces a file of normalised ID rules with the featural consequences of the HFC incorporated. A typical use might be:

```
hfc from nidrules1 to nidrules2.
```

It does not require any other data modules to be loaded.

The MORECOMMS library (see 5.5.2) provides several commands which combine HFC with ID rule normalisation, metarule expansion, etc.

3.4 ProGram data files

This section contains information about the data files used by the system - what they must contain, how to access them in commands, the standard names for them, and so forth.

Most of the commands require two sorts of data - background data and input data - and produce output data. Background data consists of information about the grammar which is required to do the processing. Input data is what the processing is done on. Output data is the result. For example, the parser needs a grammar for background data, sentences for input data and produces parse trees for output data. These three sorts of data can be specified in the command using USING, FROM and TO clauses.

3.4.1 USING clauses

A USING clause takes the form:

```
using <input file>
```

It specifies that the <input file> given is to be used as background data. It causes the file to be loaded (in Prolog terms 'reconsulted') before the command it is attached to is executed. For example:

```
parse using foo.
```

means 'load file foo, and then parse'. Note that <input file> can be a complex expression using THEN and SOMEOF if the FILTER library has been loaded (see 5.5.1). A command can employ several USING clauses in order to load different files. A second form of the USING clause is

```
using <input file> for <data module name>
```

Here, the additional information is that this background data is to be taken as the data for <data module name>. The point of this is that each command knows which data modules it needs, and the system maintains a record of which data modules have been loaded. If a command is run and a needed data module has not been loaded, the user will be prompted for a file name for that data. By adding a FOR clause, the system is informed that the data module given has now been loaded, and so it does not ask for it.

For example, suppose nothing has been loaded, and we want to normalise some ID rules. The background data for this command consists of two modules, FEATURES and ALIASES. Here are two alternative ways to get them loaded:

i. `normid using myfeats for features using myalias for aliases.`

ii. `normid.`

```
FEATURES data is now required.
Please type in data file name ?- myfeats.
ALIASES data is now required.
Please type in data file name ?- myalias.
```

In (i), the data was specified in the command; in (ii), the system prompted for it. Note the period after the filenames in this latter case. Apart from in the commands themselves this is the only place where a period is required.

Now consider the following:

iii. `normid using myfeats for features using myalias.`

```
ALIASES data is now required.
Please type in data file name ?-
```

Despite the fact that alias data was loaded in the command, the system was not told that it was the alias data, so it asked for it anyway. If this happens, or if, in general, you do not want to specify data for a given module, type NONE instead of the file name.

USING clauses without a FOR can be used to load nonstandard data modules (see 5.5.3 for an example) and also for reloading data modules. It is not necessary to tell the system that a data module is being loaded if it thinks it is already loaded.

3.4.2 FROM clauses

A FROM clause takes the form:

from ⟨input file⟩

The clause specifies the source of input data for the command. As with USING clauses, ⟨input file⟩ may include THEN and SOMEOF operations. Normally, there is only one FROM clause in a command (if more than one is present, then only the leftmost applies). If there is no FROM clause, input is expected from the terminal. That is, the user is expected to type the data for the command in, using the correct syntax, terminated by a period, when prompted. To end the input data type

end.

Examples of FROM clauses:

i. parse from sent1.

ii. normid from idrules1 then idrules2 using myfeats for features.

In (i), the parser gets its sentences from the file SENT1, and trees are displayed on the terminal. In (ii), ID rules are taken from IDRULES1 and then from IDRULES2, and the data module MYFEATS is used for the features. If aliases have not been loaded already, the system will prompt.

3.4.3 TO clauses

A TO clause takes the form:

to ⟨file name⟩

The clause specifies where the output data is to be put. The ⟨file name⟩ must be a simple file name - THEN and SOMEOF may not be used. Tracing still goes to the terminal. Again, there should be at most one TO clause. If there is none, output is displayed on the terminal.

Examples of TO clauses:

i. normlex to nlex1.

ii. parse from sent1 to trees1.

In (i), input is from the terminal (the user types in lexical rules) and the normalised versions are saved in NLEX1. In (ii), the input is from SENT1 and the output is put in TREES1. Unless tracing is on, nothing will happen at the terminal at all.

3.5 Data modules required by commands

The following table gives the name, contents and standard file name of the various data modules used as input and background data by system commands. The standard file names are those assumed by the high level commands in the MORECOMMS library (see 5.5.2).

Module	Standard name	Contents
features	features	feature syntax specification
aliases	alaises	feature aliases specification
idrules	idrules	ID rule data
metarules	metarules	Meta-rule data
lexicon	lexicon	lexrule data
lprules	lp	lp rule data
fcd	fcd	fcd data
fcr	fcr	fcr data
rac	rac	rac data
normed idrules	nidrules	ID rules data (normalised) (+ HFC if used)
normed metarules	nmeta	metarule data (normalised)
normed lexrules	nlexicon	lexrules data (normalised)
normed lprules	nlp	lp rules (normalised)
normed fcd	nfcd	fcd data (normalised)
normed fcr	nfcr	fcr data (normalised)
normed rac	nrac	rac data (normalised)
text	<none>	sentences etc.
trees	<none>	parse trees

The following table gives the background, input and output data modules for the basic commands. The commands in the MORECOMMS library are composites of these.

Command	Background	Input	Output
normid	features,aliases	idrules	normed idrules
normmeta	features,aliases	metarules	normed metarules
normlex	features,aliases	lexrules	normed lexrules
normlp	features,aliases	lprules	normed lprules
normfcd	features,aliases	fcd	normed fcd
normfcr	features,aliases	fcr	normed fcr
normrac	features,aliases	rac	normed rac
hfc	<none>	normed idrules	normed idrules
expand	normed metarules	normed idrules	normed idrules
showdata	<none>	<any>	<no output>
parse	features, normed idrules, normed lexrules, normed lprules, normed fcd, normed fcr, normed rac.	text	trees

3.6 Structure of normali»gd **data** records

This section contains examples of normalised versions of each type of data object, together with a brief explanation of the function of each component. In general, intimate knowledge of these objects should only be needed as a last resort, for example, when errant behaviour cannot be tracked down using the parser.

3.6.1 ID rules

An ID rule is normalised into an instance of the predicate BASERULE. For example, the ID rule:

```
s : v<2) --> NP , VP where      NP is n(2,tnoa])f
                                VP is h(l)f
                                NP controls VP.
```

becomes:

baserule(s,no,[root, teat, [bar, [211, [head, [major, [v]J, J]J, _2, _31,[c(c trl,_4,[2],no,[root, teat, (bar, [21], [head, tiajor, [ft]], [minor, tagr ! J5], [case, tnominativeJ]]]], _6, .7]), c(ctrl,J,[l],no,[root , [cat("[bar, [1]], [head, tiajor, _9]f" tiiinor, [agr i J) i .11111, J l, .121)1,[Ctrl controls ctrlJ).

The components of this break down as follows (the number is the argument position in the tern, with the value in the above example also given in brackets):

1. (s) - the name of the ID rule (or the original ID rule, for rules generated with tetarules).

2. (no) - the aetarules which have been applied. NO leans Metarules cannot be applied to this rule'. For lexical rules, this object is the list of the naaes of the metarules which have been applied to produce this rule (initially U>.

3. ([root, teat, ...],_2,_31) - the normalised version of the mother category. Note that .1, .2, etc., are Prolog's own notation for variables.

4. ([c(....),c(....)l> * the daughter category list - each category is represented by a term with functor C. See belott for a description.

5. ([ctrl controls Ctrl]) - an internal structure which is used for the CAP processing* By the time the rule is normalised, it is redundant.

The daughter category terms have functor C and five arguments as follows: (examples from first daughter category above).

1. (ctrl) - flag used for CAP processing.

2. (.4) - flag used to tark the HEAD daughter * should be a variable until HFC processing is done,
3. (t2D- coefficient of BAR in the category.

4. (no) - the category type. The four possible values *are* NO (in ordinary category), OPT (in optional category), STAR (a Kieene § category) and PLUS (a Kleene • category),

5. ([root, (cat, ...,], .6, .7]) * the normalised feature tree for the category.

3.6.2 The lexicon

Lexical rules normalise into clauses for the predicate LEXRULE which takes three arguments (see also 2.9). There are two forts - depending on whether the rule specifies one word or several words.

The following lexical rule:

```
vp_l(sf) i v(B,tsing, fin]) ->- jumps, runs, lings,
```

normalises to:

```
lexrule(J,le«<vpJ(sf)),troot, teat, tbar, [lexical, tvp.l]]), [head,
    [aajor, [v]J, [ainor, [agr, [singular]], [vfort, [finite], [auxiliary
    , Ml, [inverted, l-]]]]]], .2, JH1-once(tetber(J,£jutps,
    runs, sings])).
```

which is of the form

```
lexrule(W,<name of rule>f<normalised feature tree)) 5-
        once (member (N,<list of words)).
```

A rule with just one word, such as

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

normalises to:

```
lexrule(fay,lex(pp(by)),[root, [cat, [bar, [lexical, [pp]]],
    [head, [tajor, [p, [by]]], .1]], .2, .31).
```

which is of the form:

```
1txrule(<word>, <nafie of rule), Normalised feature tree)).
```

3.6.3 LP rules

A given linear precedence rule produces at least one clause of the predicate LP with two arguments, both normalised feature trees. Each instance specifies that categories matching the first argument precede categories matching the second argument. Note that one LP rule will in general produce several LP clauses by itself, and mav interact with

other LP rules via transitivity.

The following LP rule

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

would produce the following two LP clauses:

```
lp([root, [cat, [bar, [lexical, _1]], _2], _3, _4],[root, [cat, [bar, [1]],
      _5], _6, _7]).
lp([root, [cat, [bar, [lexical, _1]], _2], _3, _4],[root, [cat, [bar, [2]],
      _5], _6, _7]).
```

3.6.4 Metarules

A normalised metarule is an instance of the predicate METARULE with four arguments. The following metarule:

```
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.
```

normalises to the following structure:

```
metarule(inv,baserule(_1,_2,[root, [cat, [bar, [1]], [head, [major, [v]],
      [minor, _3, [vform, [finite], [auxiliary, [+]], [inverted, [+]]]]]],
      _4, _5],[c(_6,_7,[lexical, [_8]],no,[root, [cat,[bar, [lexical,
      [_8]]], [head, [major, _9], _10]], _11, _12]), c(_13,_14,[1],no,[root
      , [cat, [bar, [1]], [head, [major, [v]], _15]], _16, _17])],_18),no,ba
      serule(_1,[inv | _2],[root, [cat, [bar, [2]], [head, [major, [v]],
      [minor, _3, [vform, [finite], [auxiliary, [+]], [inverted, [+]]]]]],
      _4, _5],[c(_19,_20,[lexical, [_21]],no,[root, [cat, [bar, [lexical,
      [_21]]], [head, [major, _22], _23]], _24, _25]), c(_26,_27,[2],no,[ro
      ot, [cat, [bar, [2]], [head, [major, [v]], _15]], _16, _17])],_28)).
```

The arguments are:

1. (inv) - the name of the metarule.

2. (baserule(....)) - the normalised version of the first ID rule

3. (no) - either NO or STR(VAR) depending on whether there is a string variable in the rule. If so, the categories it matches will be unified with VAR when the metarule is used. This variable may also appear in the daughter list of the second baserule.
4. (baserule(....)) - the normalised version of the second ID rule.

The ID rules normalise in the same way as ordinary ID rules, except for the first two arguments (name and metarule list). These are unified appropriately to automatically produce the right name in any rule generated. Similarly, any MATCHES clause will produce unifications within the metarule. These can be detected by more than one instance of a variable in the rule (i.e. two variables with the same number). For example variables _15,_16,_17 appear in two places in the rule above. The rule only applies when both instances of a given variable are the same.

3.6.5 Feature coefficient defaults

FCD records normalise into instances of the predicate NCFD. For example:

```
fcd(case,[foot],acc,free).
```

normalises to:

```
nfcd(case,[root, [cat, _1, [head, _2, [minor, _3, [case | _4] | _5] | _6]
        | _7] | _8],_4,[[*]],_9).
```

The arguments are:

1. (case) - the name of the feature.

2. ([root ....]) - the path to the feature through a category (a minimally instantiated feature tree identifying the given feature. The exclusion list ([foot] in this case) is used to locate the correct path.

3. (_4) - the variable in the path that corresponds to the coefficent being specified.

4. ([[*]]) - the lexical default value of the coefficient.

5. (_9) - the phrasal default value of the coefficient.

The default values are normalised, except in the special case FREE which becomes a variable.

3.6.6 Feature cooccurence restrictions

FCR's normalise to a clause for the predicate NFCR. The clause succeeds only if the restriction is violated. For example:

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

normalises to:

```
nfcr(J) ;- fcrttstU * [root, [cat, .2, [hud, titjor, CvJ] ! .31 ! .43
      ! _5],not(.l « [root, [cat, .6, [head, _7, tiinor, _B, lease,
      j)) I .ill ! .11"] i J21M \ !.
```

This is of the fore

```
nfcr(Cat) s- fcrtesUCat * Pathl, not<Cat « Path2)),!.
```

FCRTEST is a system predicate (in the MONITOR »odule). PATH1 and PATH2 *are* paths leading to the feature specified (as in FCD's above), with the coefficient filled in as specified. The equalities (unifications) succeed if the CAT has the coefficient given. NOT is inserted if it is present in the original FCR, to negate the condition.

3.6.7 Root admissibi1itv conditions

RAC's behave like FCR's - they generate clauses that «ust fail. The clauses contain paths as above. The example is the special case UNSPEC, which nay also be used with FCR's.

```
rac(case,[foot],not(unspec)).
```

normalises tot

```
nrac(J) J- not (not (_1 > [root, [cat, .2, [head, .3, [ainor, .4, [cast,
      unspecJ i .51 ! Jl ! .73 I JJ)) , !•
```

The general form is

```
nrac(Cat) ?- not(Cat * Path),!,
```

with possibly an extra NOT thrown in for negation.

4. The Parser

The parsing module of Program provides the main tool for investigating grammars. The basic function of the parser is to accept phrases of the language (i.e. strings of words which occur in the lexical rules) and attempt to parse them according to the grammar specified. The basic command to invoke the parser is 'parse' and, as with the normalisation commands, input and output files may be specified. The input consists of strings of words separated by spaces and terminated by end-of-line. The output depends on the setting of the switch SHOW (see chapter 5). If SHOW is OFF, the output is the raw Prolog representation of the parse tree (which can be read back in if desired, e.g. for use with 'showdata'). If SHOW is ON, then the output employs an interactive display mechanism which allows the user to examine the tree (note: since it is interactive, output should be to the terminal whenever SHOW is ON). By using TRACE_OUT, with SHOW off, the user can examine and save the parse tree if they desire.

Examples of calling parse:

```
parse.
```

will display input and output on the terminal;

```
show on, parse.
```

will display input and output on the terminal, and provide an interactive display of the parse trees;

```
trace_out on.
show off.
parse from data to trees.
```

will parse the sentences to be found in a file called DATA, display the resulting parse trees, and save these parse trees in a file called TREES.

The parser needs several data files to be loaded (ID rules, lexicon, etc.) before it will run. It will ask for these if they are not already loaded or specified in a USING clause.

The parser has its own special data reading routine, which means you do not have to worry about commas, periods, upper case letters, etc., - you just type the sentence as normal. However, this means that the filter options - SOMEOF and THEN - (see chapter 5) cannot be used with parser input.

4.1 Parsing modes

The parser operates in three distinct modes, which are described below. Initially, the parser is in AUTO mode, but the mode may be changed by giving one of the mode switching commands:

```
control on.
monitor on.
auto on.
```

This should be done before running the parser (although it is possible to switch mode inside the parser, when in monitor or control modes).

The difference between the modes lies in how the parser makes its decisions during parsing. For a complete understanding of the parser's behaviour, it is necessary to understand Prolog's backtracking search strategy (for those interested, see 4.4, below). However, the brief discussion that follows should be sufficient for most users of the system.

At any point in the parsing process, the parser has a sequence of syntactic categories which span the sentence, or part of the sentence. It has to find an ID rule which combines some of these categories into one larger category. There may be several ways to do this, using different ID rules and different groups of the categories. In the process of searching for all parses, the parser must try all possible ways. Some of them lead to dead ends, perhaps because the parser ends up with only two categories which span the whole sentence but which cannot be combined (a successful parse is one where there is only one category left, spanning the whole sentence - the root of the parse tree). So, at any point there are two things the parser can do: try to find a new way to combine categories, or decide that it is a dead end and go back to the decision before and try to find a new way there (that is what is called 'backtracking'). The different modes allow different levels of control over the choices made and decisions about backtracking.

4.1.1 AUTO mode

In auto mode, the parser runs completely automatically. The only control over the parsing operation is choosing whether you want only the first parse, or all parses (the ONEPARSE switch - see below). The parsing process can be watched by setting the WATCH switch (see below), but it cannot be controlled. If SHOW or TRACE_OUT is selected, then the user gets the chance to examine each tree as it is produced.

4.1.2 MONITOR mode

In monitor mode, the parser reports each decision it makes to the user. There are two sorts of decision - choosing a lexical category for a word and choosing an ID rule to apply. In each case the parser makes sure the choice is applicable first and then reports the name of the rule (ID or lexical) chosen.

Monitor mode can be used in two ways, controlled by the CHECK switch. If CHECK is on (as it is initially), then after reporting the choice, the user is asked if it is acceptable. If the user says yes, the parser continues, if no, then it looks for another acceptable choice. If there isn't another one it backtracks to the decision before. Normal exhaustive search for all parses involves trying all acceptable choices at each stage, and by rejecting an acceptable choice in this way, the user can prevent the parser from doing work that would, for instance, lead to parses already known to be correct. The user can thus quickly force the parser to explore the possibility of particular analyses.

If CHECK is off, the parser does not ask the user at each stage, and monitor mode simply provides a more detailed trace of the parser's operation than when running with WATCH on alone.

4.1.3 CONTROL mode

In monitor mode, the parser processes until it reaches a decision which can be used (e.g. an ID rule which matches the categories available). In control mode, the user actually tells the parser which rule to try. Each time the parser reaches a choice point, for example, which ID rule to try, it asks the user for the name of an ID rule. It then attempts to use the rule specified. If it succeeds, then parsing continues on to the next choice point, if it fails, then failure is reported, together with the reason for failure, when it is known (for example, failure of the Foot Feature Principle, or default checking, etc.) and the user is asked to specify another ID rule. Alternatively the user can say 'no more possibilities' forcing the parser to backtrack to the choice before.

When the user specifies a name of a rule, the name does not have to be exact. The name of a lexical rule or ID rule can be a simple word, or a complex term (see chapter 2). In control mode, the name specified by the user can be the whole name, or the functor of the name. This means that the name can sometimes match more than one rule. the system will try each in turn, but before it does, the user is told which one it is using. If the CHECK flag is on, the user may reject the selection if desired.

See below for examples of the different modes in operation, and the printout produced by each of them.

4.2 The WATCH switch

In monitor and control modes, the parser displays the categories it is currently working on - a collection of categories which spans a right subsequence of the input string (the 'current segment'). This information is usually sufficient for the user to keep track of what is going on. The WATCH switch causes the parser to display the complete partial trees after each scan, rather than just the names of the root nodes. WATCH works in all modes.

4.3 Summary of the parsing switches

The following switches are relevant to the parser:

AUTO, MONITOR, CONTROL	change the parse mode.
ONEPARSE	when ON, stop after the first parse, when OFF look for ALL parses.
SHOW	use the interactive tree-displaying routines for each successful parse.

TRACE_OUT	write data normally and also SHOW it on the screen.
CHECK	controls whether the user can check choices made by the parser.
WATCH	when ON, the parser displays partial trees between passes.

4.4 Outline of the parsing algorithm

This section contains a brief description of the parsing algorithm. This algorithm is designed specifically for ease of use in the ProGram system. It is not intended as an efficient free-standing parser for GPSG grammars. Understanding the algorithm is not a prerequisite for using the system, but may be of interest to those wishing to understand the workings of the system, modify it, etc. It assumes a fair knowledge of the workings of Prolog.

The algorithm is presented as a schema for the main predicates in a Prolog implementation.

FIND_PARSE is the main parsing predicate. Given a list of words as its first argument, it returns a list of parse trees spanning all the words. Backtracking produces alternatives. A valid parse is one where the output list of trees has only one member.

```
find_parse([],[]).
find_parse([Word|Words],[NewTree|RemTrees]) :-
        /* parse the tail */
        find_parse(Words,OldTrees),
        /* look up a category for the head */
        lexcat(Word,LexTree),
        /* try to combine new cat with the rest */
        tryrule(LexTree,OldTrees,NewTree,RemTrees).
```

TRYRULE tries to combines the new tree (Tree) with a leading sublist of the old trees (OldTrees), giving a new resulting tree (ResTree) and the remaining old trees (ResTrees). If it succeeds, it will try again on the resulting tree. If not, it just leaves the tree alone. (Hence the parser produces one big tree before several little ones).

```
tryrule(Tree,OldTrees,ResTree,ResTrees) :-
        /* apply a rule, giving a new tree and
        some of the old ones left over */
        rulefor([Tree|OldTrees],NewTree,RemTrees),
        /* combine the new tree with the remainder to get result */
        tryrule(NewTree,RemTrees,ResTree,ResTrees).
tryrule(T,0,T,0).
/* no rule applies, results=inputs */
```

The predicate LEXCAT returns a parse tree for the word it is given - i.e. a tree whose root is the appropriate syntactic category and whose substructure is whatever is desired (in ProGram, simply a leaf consisting of the word itself). Backtracking produces alternatives.

The predicate RULEFOR takes a list of trees and tries to find a rule which will consume a leading (LEFTMOST) sublist, returning the new tree constructed and the remaining trees of the original list. It is here that all the principles, linear precedence, default setting, etc., get done. Backtracking produces alternatives.

Predicates LEXCAT and RULEFOR contain all the mode-dependent processing. The WATCH information is printed out immediately before RULEFOR in TRYRULE (although, for clarity, the predicate has been omitted above).

4.5 Using the system

This section contains an example of a typical session using the ProGram system to explore the demonstration grammar. It assumes that the grammar data is syntactically correct, but not normalised. Note that the demonstration grammar is less than descriptively adequate, even for the small subset of English that it purports to cover. Some of its failings will be revealed and discussed below.

We shall start inside ProGram, with the DEMO directory specified as the grammar directory: either by being in the demo directory when ProGram was started up, or by using customised predicates to set the grammar directory (e.g., in POPLOG, the predicate GRAMMAR).

```
?- nora_gram.                  FIRST OF ALL, NORMALISE THE
                               GRAMMAR WITHOUT METARULE
                               EXPANSION - TO TEST OUT THE
                               BASIC PHRASES, ETC.

yes
?- show on.                    SWITCH SHOW ON - INITIALLY WE
                               WANT TO LOOK AT THE TREES,
                               NOT SAVE THEM.

SHOW is on.
yes

?- go.                         NOW LOAD THE GRAMMAR AND START
                               PARSING (IN AUTOMATIC MODE)
```

We start with some simple phrases. Note that automatic exaustive parsing is quite slow, so it is best to try small things out first, to get some idea of what won't go wrong in bigger phrases. The actual trees are not included here, but the following table indicates the number of parses obtained.

```
!: kii                      2 PARSES - AS NOUN OR AS NP
i: sees bert                i PARSE AS VP ONLY
!: bert sees                NO PARSES - 'bert' IS ACCUSATIVE
!: kii sees bert            1 PARSE
it sees bert kit            NO PARSES - VP-NP LP RELATION IS NOT AVAILABLE

Is in i book                1 PARSE - PP's ARE OK
Is a tin in i book          1 PARSE
Is sees i i*n in a book     2 PARSES, PP MODIFIES NP OR VP
Is neither kit nor sindy    3 PARSES
is neither bert nor bill    3 PARSES
```

Because 'kira','sandy' etc. are ambiguous between noun and noun-phrase, in the demonstration grammar, there are three possible parses here.

```
is neither sees bert nor loves bill
                            1 PARSE
i; neither sees nor loves bert 1 PARSE
```

But there is no comparable ambiguity here, so only a single parse results.

So the basic phrases seem sound - obviously we could test some more, and perhaps a good way of checking out a lot is to run the system as a batch job, saving the trees by giving the command:

```
go to trees.
```

to get the data saved in the file 'trees'. Later, the data can be examined with the commands

```
showdata from trees.
```

Having decided the basic phrases are sound, we can continue by expanding the grammar with the metarules and doing some more complicated things:

```
?- nora_aeti.           NORMALISE GRAMMAR A6AIN, INCLUDING
                        METARULES THIS TIHE. THIS IS A BIT
                        WASTEFUL, SINCE MANY COMPONENTS ARE
yes                     UNCHANGED, BUT IT IS CONVENIENT

?- go.                  PARSE A6AIN IN AUTOMATIC NODE

Parsing in AUTO lode.

is kit is seen          AN INNOCUOUS EN0U6H SENTENCE BUT...
```

Here we get two problems - firstly, ProGram produces three parses and secondly it takes a very long time to do it. We shall tackle the first one first. The problem is with the VP 'is seen'. The three parses produced involve (i) a VP, (ii) a VP/PP (this is correct - the optional PPCby) has been slashed) and *(Hi)* a VP/NP. This latter case is unexpected. Examining the feature trees for the categories reveals that 'seen' has been parsed as a VP/NP derived from rule VP.2 (V<2) –> H, N<2>>. The problem, then, is that VP_2 should not produce PASSIVE VP rules - only the metarule PASS should do that. But nothing in the rule

blocks the passive feature. We should have a default for the VFORM feature of FINITE. The FCD we require is

```
fcd(vform,[foot],finite,finite).
```

Now we could simply add this to the FCD file of the demonstration grammar and renormalise the FCD's. Another way to do it, and one which does not disturb the existing demonstration grammar too much, is to create a new file with this FCD in it and then add it to the given FCD's. How the new file is created depends on the particular computer environment (see chapter 5 for details). We shall assume that a new file NEWFCD has been created with the FCD above in it. The following commands will incorporate the new FCD into the grammar that the parser is using:

```
?- normfcd from newfcd to newfcd2.
                    THIS COMMAND NORMALISES THE NEW FCD.
                    ASSUMING THE GRAMMAR IS ALREADY LOADED,
                    WE WILL ONLY BE ASKED FOR THE ALIASES DATA.
ALIASES data is now required.

Please type in data file name
|: aliases.         WE TYPE 'aliases.'
yes

?- go using nfcd then newfcd2.
                    THIS COMMAND RESTARTS PARSING BUT USING
                    nfcd AND newfcd2 AS FCD DATA.
```

Now, 'kim is seen' will only produce two parses as expected. The other problem mentioned above is the time. In general, automatic exhaustive parsing with more than a few rules tends to be slow. There are several ways to overcome this problem. First of all, avoid using words which have more than one lexical entry (indeed, the demonstration grammar deliberately has no such words) since a lot of time will be spent checking that the second (usually inappropriate) entry cannot be used. Secondly, keep the phrases as short as possible - for example, once NP syntax has been tested satisfactorily, it is often sufficient to use proper names in NP positions without loss of generality. Thirdly, keep down the number of ID rules in use. It is often possible to restrict attention to a particular subset of the rules, or to expand using only a subset of the metarules. There are several ways of cutting out unwanted rules, as briefly described below:

(i) Organise the ID rules into separate data files, rather than all in one. Thus there may be a file of VP rules, a file of NP rules, etc. These can be normalised separately and then combined as desired (using THEN in a USING clause), or combined (using THEN) before normalising into one file.

(ii) Filter the ID rules (or the normalised ID rules) using a SOMEOF clause. The command:

```
parse using someof nidrules for normed idrules.
```

will prompt the user to accept or reject each normed ID rule in turn. It is sometimes worth filtering a file and simply saving the result for later use. COPYDATA can be used for this. The command:

```
copydata from someof nidrules to nidrules2.
```

will let you save a subset of the normed ID rules in the new file NIDRULES2. This is also a good way of deleting accidental duplicate rules generated by the metarules. See chapter 5 for more details.

(iii) Metarules can be split similarly, but remember that, to ensure that all the appropriate rules are generated, expansion must use all the metarules that are to be used. That is, you may not expand with one set, and then with another and simply combine the result. There are usually only a few metarules and it is often more practical to simply filter them as in (ii) above. Either filter the metarules in an expand command:

```
expand from idrules using someof metarules.
```

or filter the set of normalised expanded idrules, removing any produced by a given metarule.

In our case we can easily reduce the load for testing passives. Once the grammar has been loaded once (after the initial LOAD or GO command), we can give the following command to filter the idrules:

```
go using someof nidrules.
```

with the following results:

```
idrule(s) !: y          FILTER PROMPTS WITH FIRST RULE - ACCEPT IT
idrule([vp_1]) !: n     REJECT SECOND RULE (INTRANSITIVES)
idrule([vp_2]) !: y     ACCEPT
idrule([vp_3]) !: n     REJECT
idrule([vp_4]) !: n     REJECT
idrule([vp_5]) !: y     ACCEPT
idrule(vp_pp) !: y      ACCEPT (GIVES US THREE PARSES FOR 'is seen by bert')
idrule([np_1]) !: n     REJECT (ONLY PROPER NOUNS FOR NOW)

idrule([np_2]) !: y     ACCEPT
idrule([nb_1]) !: skipto pp
                        SKIP ON TO THE PP RULE
idrule(nb_2) idrule([ap_1]) idrule([ap_2])
                        THESE ARE THE RULES SKIPPED
idrule(coord) !: n
idrule(conj) !: n       DON'T WANT COORDINATION STUFF
idrule(top) !: y        BUT KEEP TOPICALISATION

idrule([pp, sta1]) !: y
                        WANT RESULTS OF sta1 AND pass FOR pp AND vp_2
idrule([nb_1, relcl]) !: skipto vp_2 pass
                        HERE WE SPECIFY METARULE HISTORY TOO

idrule([np_1, sta1]) idrule([vp_5, inv]) idrule([vp_4, sta1]) idrule([vp_3,
pass]) idrule([vp_3, pass]) idrule([vp_3, sta1]) idrule([vp_3,
sta1])
```

```
idrule([vp_2, stm1]) !: y
idrule([vp_2, stm1, pass]) !: y
idrule([vp_3, pass, stm1]) !: end
                    WE DON'T WANT ANYTHING ELSE
Parsing in AUTO mode.
!:                  READY TO PARSE WITH ID RULES SELECTED ABOVE
```

There are other ways to overcome the speed problem in some cases. If the required parse is known to be the first one, then switching ONEPARSE on will stop the subsequent exhaustive search. Alternatively, don't use automatic mode. For example, in monitor mode we can tell the parser not to explore a particular alternative if we wish. Consider the string 'a man lee sees'. This has at least two parses - as a topicalised sentence or as a noun phrase containing a relative clause. Automatic parsing produces the the topicalised sentence first. The following use of monitor mode (with all the ID rules loaded) forces the parser straight to the noun phrase parse:

```
?- m on.            TURN ON MONITOR MODE
Current mode is MONITOR
yes

?- go.              START PARSING

Parsing in MONITOR mode.

!: a man kim sees
Using lexrule vp_2(sf) for SEES
--- Ok? (type y, yes or nothing to accept) !:y
                    LEXRULE vp2(sf) HAS BEEN CHOSEN -
                    WE ACCEPT THE CHOICE
Current segment: (vp_2(sf))

Using ID rule vp_2 stm1 to consume 1 categories.
--- Ok? (type y, yes or nothing to accept) !:y
                    THE CHOICES ARE IN THE SAME ORDER AS AUTOMATIC
                    PARSING - WE ACCEPT ALL THE EARLY ONES
Current segment: stm1(vp_2)

Leaving category vp_2 stm1 alone.
--- Ok? (type y, yes or nothing to accept) !:y

Using lexrule np_2(prop_nom) for KIM
--- Ok? (type y, yes or nothing to accept) !:y

Current segment: (np_2(prop_nom)) stm1(vp_2)

Using ID rule np_2 to consume 1 categories.
--- Ok? (type y, yes or nothing to accept) !:y

Current segment: np_2 stm1(vp_2)
```

```
Using ID rule s to consume 2 categories.
--- Ok? (type y, yes or nothing to accept) !:y

Current segment: s

Leaving category s alone.
--- Ok? (type y, yes or nothing to accept) !:y

Using lexrule nb_1(s) for MAN
--- Ok? (type y, yes or nothing to accept) !:y

Current segment: (nb_1(s)) s

Using ID rule nb_1 to consume 1 categories.
--- Ok? (type y, yes or nothing to accept) !: n

                    HERE WE ARE - IF WE ACCEPT THIS CHOICE,
                    WE GET THE INITIAL NP BUILT FIRST, AND
                    THEN TOPICALISATION IS USED. WE REJECT
                    IT, FORCING THE S/NP TO COMBINE NOW
Using ID rule nb_1 relcl to consume 2 categories.
--- Ok? (type y, yes or nothing to accept) !:y
                    NOW WE CAN LET IT CONTINUE TO THE PARSE
Current segment: relcl(nb_1)

Leaving category nb_1 relcl alone.
--- Ok? (type y, yes or nothing to accept) !:y

Using lexrule np_1(sdet) for A
--- Ok? (type y, yes or nothing to accept) !:y

Current segment: (np_1(sdet)) relcl(nb_1)

Using ID rule np_1 to consume 2 categories.
--- Ok? (type y, yes or nothing to accept) !:y

Current segment: np_1

Leaving category np_1 alone.
--- Ok? (type y, yes or nothing to accept) !:y
                    THE PARSE AS AN NP:

node(np_1, [root, [cat, [bar, [2]], [head, [major, [n]], [minor, [agr,
[singular]], [case, [*]]]]], [foot, [cat, unspec, [head, unspec,
                    ... 23 LINES OMITTED HERE! ...
[conj, [*]]], sees)])])]).
                    NOW THE PARSER STARTS BACKTRACKING -
                    LOOKING FOR ALTERNATIVES TO THE CHOICES
                    MADE. IN THIS CASE, SINCE WE ARE HAPPY
                    WITH OUR PARSE, WE TELL IT TO STOP
```

```
node(np_2,[root, [cat, [bar, [2]], [head, [major, [n]], [minor, [agr, [singula
r]], [case, [*]]]]], [foot], unspec],[node((np_2(prop_acc)),[root,
[cat, [bar, [lexical, [np_2]]], [head, [major, [n]], [minor, [agr,
[singular]], [case, [*]]]]], [foot, unspec], [conj, [*]]],bill)]).

node((np_2(prop_acc)),[root, [cat, [bar, [lexical, [np_2]]], [head, [major,
[n]], [minor, [agr, [singular]], [case, [*]]]]], unspec, unspec],bill
).

                        AH! TWO PARSES AS EXPECTED
|: sees bill           TRY THE VP

node(vp_2,[root, [cat, [bar, [1]], [head, [major, [v]], [minor, [agr, [singula
r]], [vform, [finite], [auxiliary, [-]], [inverted, [-]]]]]], [foot],
unspec],[node((vp_2(sf)),[root, [cat, [bar, [lexical, [vp_2]]],
[head, [major, [v]], [minor, [agr, [singular]], [vform, [finite],
[auxiliary, [-]], [inverted, [-]]]]]], [foot, unspec], [conj,
[*]]],sees), node(np_2,[root, [cat, [bar, [2]], [head, [major,
[n]], [minor, [agr, [singular]], [case, [*]]]]], [foot, unspec],
[conj, [*]]],[node((np_2(prop_acc)),[root, [cat, [bar, [lexical,
[np_2]]], [head, [major, [n]], [minor, [agr, [singular]], [case,
[*]]]]], [foot, unspec], [conj, [*]]],bill)])]).

|: end                  VP WAS OK - STOP PARSING FOR NOW
yes                     PROLOG SAYS '60 command successful'
?- showdata from rac.   LETS HAVE A LOOK AT THE RAC'S - MAYBE
                        THEY ARE BLOCKING THE OTHER PARSE OF 'kim'

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

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

                        NO - NOTHING FUNNY THERE
yes
?- c on.                OK - LET'S PARSE IN CONTROL MODE TO SEE
                        WHAT IS WRONG
Current mode is CONTROL
yes
?- parse.               ONCE THE GRAMMAR IS LOADED, 'parse' AND
                        'go' ARE SYNONYMOUS.

Parsing in CONTROL mode.
|: kim                  TRY 'kim' AGAIN
--- LEX rule for KIM    |: b
                        CONTROL MODE ASKS WHICH LEXICAL RULE WE
                        WANT TO USE - CAN'T REMEMBER WHAT IT IS
                        CALLED. PROLOG'S BREAK FACILITY ALLOWS US
                        TO GO BACK TO TOP LEVEL TEMPORARILY. VERY
                        USEFUL.
[break]
```

```
*** No (more) choices for np_1
                    BACKTRACKING
Leaving category (np_1(sdet)) alone.
--- Ok? (type y, yes or nothing to accept) |: q

Data files closed.
;;; [execution aborted]
Setprolog

?-                  IF WE HAD WANTED TO STAY INSIDE THE PARSER,
                    REPEATED REJECTION OF RULES WOULD HAVE
                    EXITED AS FAST AS WAS POSSIBLE.
```

Finally, from the point of view of the system, CONTROL mode is fastest since it does no searching - it just asks the user what to do. For an example of control mode in use, see below.

4.6 Testing a grammar

This section contains an example of the parser in use as a grammar tester. The grammar being tested is the demonstration grammar but without any metarule expansion having been done - in the initial stages of testing the extra rules generated by metarules are best left out, since they slow things down a bit. The grammar bug is as follows: in the FCD specification for CASE, the lexical and phrasal defaults are the wrong way round. A look at the FCD data file will show how this could easily come about - in fact, this bug, and the debugging process shown below, genuinely occurred while testing the demonstration grammar.

```
ProGram Grammar Development System
Version 1, 29/4/84

(For help information type:    help. )

?- go.              LOAD UP THE GRAMMAR (ALREADY NORMALISED)

Parsing in AUTO mode.  AUTO MODE IS THE DEFAULT

|: kim sees bill     TRY TO PARSE A SIMPLE SENTENCE
|: kim               OH DEAR - NO PARSES (SECOND PROMPT
                     DISPLAYED WITHOUT ANY TREES)
                     TRY JUST 'kim'.
node((np_2(prop_nom)),[root, [cat, [bar, [lexical, [np_2]]], [head, [major,
[n]], [minor, [agr, [singular]], [case, [nominative]]]]], unspec,
unspec],kim).

                     THAT'S FUNNY - SHOULD BE TWO PARSES - AS
                     A WORD (THE ONE WE GOT) AND AS AN NP
|: bill              TRY 'bill' ('kim' AND 'bill' DIFFER IN THE
                     LEXICON ONLY BY CASE FEATURE)
```

```
?- thoNdati froa lexicon.
                        HAVE A LOOK AT THE LEXICON

vp.Hsf) s v(l,ttinq, fin]) ->- juaps , runs , linos

...etc...               LOTS OF STUFF LEFT OUT HERE

np.2(prop_noa) t ndjsing, noa]) ->- kia , sandy , lee

np.2(prop_acc) i nil,(sing, ace]) ->- bill , ben , bert

... etc ...             OK, SO HE NEED THE NP.2 RULES

yes

?- ^Z                   CONTROL Z ENDS THE BREAK ...
[end break]
- LEX rule for KIH  i: np.2
                        AND HE ARE BACK IN THE PARSE.
                        DON'T HAVE TO 6IVE THE FULL NAHE, IT HILL
                        OFFER ALL THE ALTERNATIVES (ONLY ONE IN
                        THIS CASE)
Located LEX rule np.2(prop_noa)
- Ok? (type y, yes or nothing to accept)  i:y
                        HE ACCEPT THE RULE

Current segient: (tip_2(prop.noa))
                        REPORTS THE CURRENT STATE.
                        NOW HE NEED TO 6IVE THE NAHE OF AN ID RULE
                        HHICH HILL CONSUME OUR NOUN - ITS CALLED
                        NP.2 AS HELL.
- ID rule to consuae (np_2(prop.noa)) (at least)!: np.2

Located ID rule np.2
- Ok? (type y, yes or nothing to accept)  i:y
                        ACCEPT IT A6A1N
turning: FCD failed.
Involving: case in [root, [cat, (bar, (lexical, [np.2]]3, [head, [sajor,
[n]], tainor, tagr, [singular]], [case, [noainativi]]]]], [foot
! .1], tcomj, [*]]]

                        AHA! THAT RULE SHOULD HAVE WORKED BUT IT
                        DIDN'T, BECAUSE THERE IS SOHETH1N& WRONG
                        HITH THE CASE DEFAULT IN THE NOUN
»<i No (tore) possible Batches for rule.
                        NO OTHER HAY FOR THE RULE TO HORK
HI ID rule np_2 not applicable.
                        6IVE UP ON THE RULE
```

```
Current segment: (np_2(prop_nom))
                        BACK TO WHERE WE CHOSE THAT ID RULE
                        NO OTHERS TO TRY, SO QUIT
--- ID rule to consume (np_2(prop_nom)) (at least) !: q
Data files closed.
;;; [execution aborted]
Setprolog

yes                     OK, SO THE FCD'S MUST BE WRONG -
?- showdata from fcd.   LET US HAVE A LOOK

fcd(case,[foot],acc,free)
fcd(inverted,[foot],- inverted,free)
fcd(auxiliary,[foot],- auxiliary,free)
yes
                        THEY LOOK ALRIGHT. case IS THE PROBLEM,
                        LET'S JUST CHECK OUT THE HELP FILE

?- help grammars.
<LOOK AT *GRAMMARS>
                        AH! THE case COEFFICIENTS ARE THE WRONG WAY
                        ROUND, IT SHOULD BE fcd(case,[foot],free,acc).
                        SO IT IS TRYING TO FORCE acc ONTO THE NOUN
                        BETTER FIX THAT
<EDIT THE FCD FILE>

?- normfcd from fcd to nfcd.
                        NOW RENORMALISE THE FCD'S
( loading normmark )
( loading normfeats )
                        THE PARSER HAS LOADED THE FEATURES,
                        BUT NOT THE ALIASES
ALIASES data is now required.
Please type in data file name !: aliases.

yes
?- go using nfcd.       RESTART, BUT LOAD THE NEW nfcd FILE FIRST

Parsing in CONTROL mode.
                        STILL IN CONTROL MODE
!: kim
--- LEX rule for KIM  !: np_2
Located LEX rule np_2(prop_nom)
--- Ok? (type y, yes or nothing to accept) !:y

Current segment: (np_2(prop_nom))

--- ID rule to consume (np_2(prop_nom)) (at least)!: np_2
Located ID rule np_2
--- Ok? (type y, yes or nothing to accept) !:y

                        AS BEFORE UP TO NOW - THIS TIME IT WORKS
Current segment: np_2
```

```
                        NO NEED FOR ANY MORE - QUIT
--- ID rule to consume np_2 (at least)!: q
Data files closed.
;;; [execution aborted]
Setprolog

?- a on.                BACK TO AUTOMATIC MODE

Current mode is AUTO
yes

?- go.

Parsing in AUTO mode.

!: kim sees bill        TRY THE SENTENCE AGAIN


node(s,[root, [cat, [bar, [2]], [head, [major, [v]], [minor, [agr, [singular]]
, [vform, [finite], [auxiliary, [-]], [inverted, [-]]]]]], [foot],
unspec],[node(np_2,[root, [cat, [bar, [2]], [head, [major, [n]],
[minor, [agr, [singular]], [case, [nominative]]]]], [foot, unspec],
[conj, [*]]],[node((np_2(prop_nom)),[root, [cat, [bar, [lexical,
[np_2]]], [head, [major, [n]], [minor, [agr, [singular]], [case,
[nominative]]]]], [foot, unspec], [conj, [*]]],kim)]), node(vp_2,[roo
t, [cat, [bar, [1]], [head, [major, [v]], [minor, [agr, [singular]],
[vform, [finite], [auxiliary, [-]], [inverted, [-]]]]], [foot,
unspec], [conj, [*]]],[node((vp_2(sf)),[root, [cat, [bar, [lexical,
[vp_2]]], [head, [major, [v]], [minor, [agr, [singular]], [vform,
[finite], [auxiliary, [-]], [inverted, [-]]]]], [foot, unspec],
[conj, [*]]],sees), node(np_2,[root, [cat, [bar, [2]], [head,
[major, [n]], [minor, [agr, [singular]], [case, [*]]]]], [foot,
unspec], [conj, [*]]],[node((np_2(prop_acc)),[root, [cat, [bar,
[lexical, [np_2]]], [head, [major, [n]], [minor, [agr, [singular]],
[case, [*]]]]], [foot, unspec], [conj, [*]]],bill)])])]).

!: end                  IT WORKED - EXACTLY ONE PARSE. STOP
                        PARSING
yes
?- end.                 END SESSION

Exiting ProGram.
```

4.7 Displaying parse trees

In normal use, without the SHOW switch on, the parser produces parse trees which are quite unreadable. For example, the tree for 'kim sees

bill ' looks like this:

node(s,[root, [cat, (bar, (211, [head, [eajor, [vl], [einor, [agr, [singular]]
f [vfora, [finite], [auxiliary, [-]], [inverted, [-]]]»], [foot],
unspec],[node(np_2,[root, [cat, [bar, [2]], (head, [§ajorf [n]],
[•inor, [agr, [singular]], [case, [noeinativel]]]], [foot, unspec],
[conj, [*]]],[node({npj(prop_noa)>,[root, [cat, [bar, [lexical,
[np_2]]J, [head, [aajor, [ft]], [aiior, [agr, [singular]], [case,
[nosinative]]}]], [foot, unspec], [conj, [*]]],kie)]), node(vp_2,[roo
t, [cat, [bar, [1]], [head, iiajor, [vli, [itnor, [agr, [singular]],
[vfora, [finite], [auxiliary, [-]], [inverted, [-]]]]]], [foot,
unspec], [conj, [*]]],[node({vp_2(sf)},[root, [cat, [bar, [lexical,
[vp_2]J], [head, [aajor, tv]], [iinor, [agr, [singular]], [vfort,
[finite], [auxiliary, [-]], [inverted, [-]]]]]], [foot, unspec],
[conj, [%]]],sees), node(np.2,[root, [cat, [bar, [2]], [head,
[•ajor, [nil, [sinor, [agr, [singular]], [case, [*]]]]}, [foot,
unspec}, Iconj, [%]]],[node({np_2(prop.acc)},[root, [cat, [bar,
[lexical, [np.2]]], [head, Ciajor, [n]J, ttinor, [agr, [singular]],
[case, [v]]]]]f [foot, unspec], [conj, [*]]],bill)])])]).

This is, of course, the internal representation of the tree, and for «ost people it has only two uses:

(i) it is the for«t that the tree is saved in a data file in, for exaaple the coAnand

parse to treedat.

will store trees like this. They can then be viewed at leisure using

showdata fro« treedat.

(without needing the SHOW switch on -see below).

(ii) One can count how aany of then there are, to see if the right number of parses has been found.

The foraat of a tree is actually very siaple. A tree is a ten of the fora:

node(N,N,D)

where N is the naae of the node (the full name, without truncation), M is the feature tree for the ©other and D is a list of the daughter subtrees (also instances of NODE), in left-right order, except when the node is lexical, in which case D is the word itself.

A itore readable interactive tree-display routine is provided by the FILTER library, and can be selected by switching SHOW on (see chapter 5). It is also used by the trace printer (when TRACE.OUT is on) and the command SHOWDATA.

With the interactive display routine selected, the parse tree above is displayed as follows:

```
1: s
2: .  np
3: .  .  {np(prop)}
.  .  .  kim
4: .  vp
5: .  .  {vp(sf)}
.  .  .  sees
6: .  .  np
7: .  .  .  {np(prop)}
.  .  .  .  bill
```

The tree itself is in 'indentation format' - for each tree, the root node name is printed, followed by the subtrees (also in indentation format) indented several spaces. The dots are to help keep track of the indentation. The labels of the nodes are as discussed in chapter 2 - they are the names of the ID rule applied at that node, but anything beyond an underline character is omitted. Thus the node labelled VP here derives from the rule named VP_2. If the tree had been labelled conventionally, i.e. with each node labelled by the category associated with it, then this label would be:

```
[root, [cat, [bar, 1], [head,
[major, v],[minor, ....]]], _1, _2]
```

Notice also that names of LEXICAL rules are enclosed in braces { } to distinguish them. The node-naming conventions used by ProGram allow the basic parse tree to be readable, and give the grammar writer control over the labels used.

To view the internal structure of a category, one uses the numbers given at the start of each line (except lines for word-nodes, which have no internal structure). Once the main tree has been displayed, ProGram will prompt the user as follows:

```
Examine option ::
```

There are several things that can be typed - nothing (i.e. just <return>) means 'finished examining tree - go back to parser', T means 'display the main tree again' and a number means 'display the internal category structure of this node (as numbered in the main tree)'. As always, one can also type HELP.

In this case, typing '7' would result in the following tree being displayed:

```
root
.  cat
•  .  bar
•  .  .  lexical
•  •  .  •  np_2
.  .  head
.  .  .  ftajor
•  •  .  •  n
.  .  .  •inor
.  .  .  .  agr
.  .  .  .  .singular
.  .  .  .  case
.  .  .  .  .
.  foot
.  •  unspec
.  conj
.  .  *
```

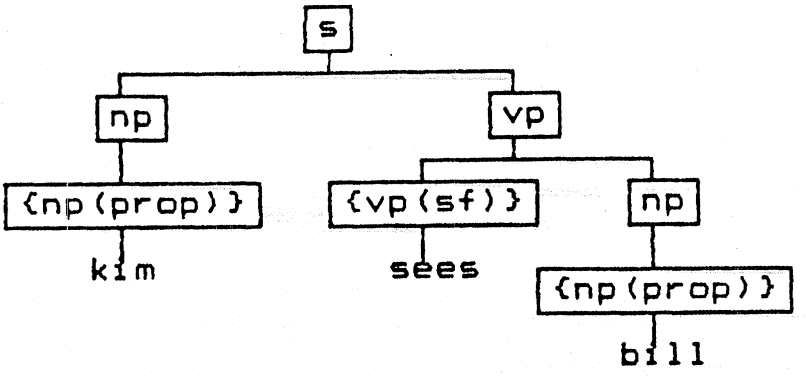
This is the feature tree for the noun 'bill' (using lexical rule NP_2(PRQP_ACC) which get displayed as (NP2(PR0P)}). The format is the same, except that feature names are used to label nodes, and no nuebers are given, since there is no internal structure to these node labels*

The user can now select another node or the main tree etc. Node 1 (the S node) looks like this:

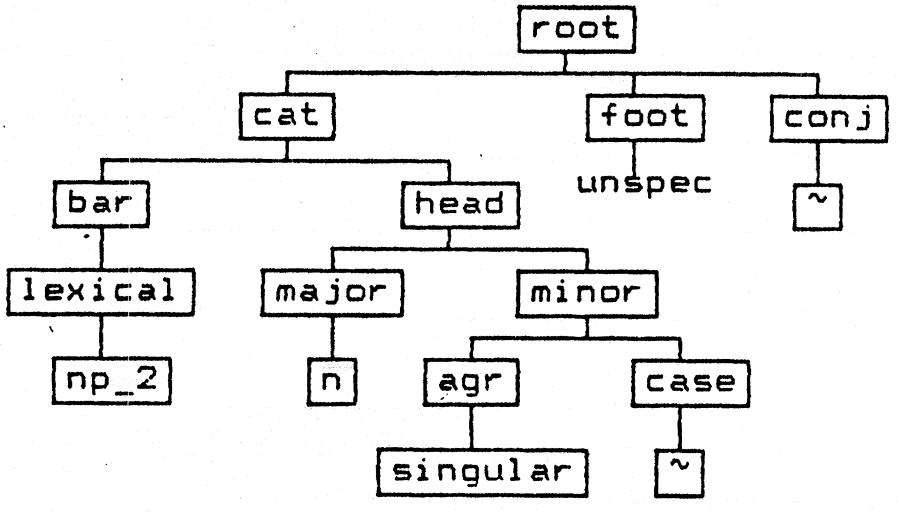
```
Examine option is 1
root
.  cat
.  t  bar
.  .  .  2
.  .  head
.  .  .  major
.  a  .  .  v
.  .  .  minor
.  .  .  .  agr
.  .  .  .  .  singular
.  .  .  .  vform
.  .  .  .  .     finite
.  .  .  .  .     auxiliary
.  .  .  .  .  .  -
.  .  .  .  .  inverted
.  .  .  .  .  .  -
.  foot
.  unspec
```

4.8 Th» SHOWTREE package

Users of the system under P0PL06 can take advantage of a prettier display routine (for certain VDU's only) which makes use of the POP1i parse tree display library (the SHOWTREE library). By loading the custom module SHOWTREE after the FILTER library, the display predicate is redefined to display a tree as follows:



The labelling conventions for nodes are as above. The tree is displayed using the POPLOG screen editor VED and most of the keys on the keyboard are redefined or disabled. The new functions for keys are given in your LOCAL help file. They permit the user to move the cursor from node to node in the tree. A further key (the VED PUSH key) causes the category structure tree of the current node to be displayed. For example (the node for 'bill' again):



The VED POP key restores the main tree again, and the user can jump between main tree and feature trees at will. The VED END-OF-FILE key returns control to the parser.

5. The System

5.1 Summary of system commands

This section summarises the commands available in the basic system. Further commands are discussed in section 5.5.2, below.

NORMID	normalise ID rules (see chapter 3).
NORMLEX	normalise lexicon (see chapter 3).
NORMMETA	normalise metarules (see chapter 3).
NORMFCD	normalise FCD data (see chapter 3).
NORMFCR	normalise FCR data (see chapter 3).
NORMRAC	normalise RAC data (see chapter 3).
NORMLP	normalise LP data (see chapter 3).
EXPAND	expand ID rules using metarules (see chapter 3).
PARSE	run parser (see chapter 4).
STATUS	display switch settings (see section 5.4).
S ON	turn switch/switches S on (see section 5.4).
S OFF	turn switch/switches S off (see section 5.4).
LIB X	load library module X (see section 5.5).
HELP	display general help information (see section 5.2).
HELP X	display help information on topic X (see section 5.2).
ID	show identification string.
FORGET ALL	forget all data modules.
END	quit ProGram, return to operating system.

SHOWDATA - display the data in a data file using the SHOW display option. Data is read from the input and displayed as though the SHOW switch were on. Particularly useful for looking at stored parse trees.

COPYDATA - copy data from one file to another. This command does nothing to the data at all, but can sometimes be useful with the SOMEOF filter (see 5.5.1).

ID X - set identification string to X. The identification string is printed in the header for all data files created by the system. Typical information might be date, user's name, etc.

CLOSE - close all data files. Occasionally, due to syntax errors, etc., a data file may be left in an open state and reading thus commences half way through. This command ensures all data files are closed.

FORGET X - forget data module X. This command causes the system to forget that the specified module has been loaded, so it will require it to be reloaded before attempting to use data from that module.

5.2 Help facilities

Information and help with ProGram cones in two torus. Host of the information about using the system is contained in help files. If the system is running under PDPLOG, the help files will probably be online and accessible in the same way as the POPLOG help files. Throughout the help files, references to other help files are preceded by '•'.

The system also has a limited built-in help facility for access while in use. This provides short explanations and references to the help files. The command which accesses this information is

help <keyword>.

where <keyword> is the topic required. If the keyword is omitted, general help information is printed. Note the period at the end of the command. A fair range of keywords related to the system can be used, and the command

help index,

gives you a list of the keywords covered*

In most cases where the system requires the user to type something (e.g. a data module name, a parsing decision, etc.), the user may also type HELP or HELP <keyword>, this time without a period (periods are only required at the command level). If no keyword is specified, help about what you can do at this point in the processing is provided. Thus, for example, to get the general information about help files found above, you would have to type HELP HELP.

Thus the general rule is 'if in doubt, type HELP, and if that doesn't work, then type HELP followed by a period.'

There are also a group of help files which contain example of ProGram in use. Host of the examples assume the Prolog-only version of the system, but there are also a few which discuss some of the libraries in the P0PL06 version.

All the examples are based on the demonstration grammar which is to be found in chapter 6. See your #LOCAL help file for details of the actual location of the demonstration grammar on your machine.

The example files all follow the tame general format - they consist of examples of terminal interaction with ProGram interspersed with comments. In general, large block comments are just ordinary text while short comments actually within, output etc., are in upper case. The sample output text is as produced by the system running under UNIX POPLOG, with the demonstration grammar for data. A few extra blank lines have also been inserted for clarity.

The example help files are:

EGDATA	what the normalised data looks like.
EGEXPAND	expanding using the metarules.
EGNORM	data normalisation examples.
EGPARSE	typical interaction with the system
EGTEST	using the parser to find a bug.
EGTREE	various ways of displaying the parse trees.

5.3 Errors, causes and corrections

When using ProGram, various sorts of errors and warning messages may be encountered. This section attempts to explain and diagnose some of them.

First of all, it is important to be able to distinguish errors produced by ProGram from errors produced by Prolog or some other part of your computer system.

5.3.1 ProGram errors

ProGram errors come in two forms:

```
        mishap: <some error message>
        involving: <objects that caused the problem>
or
        warning: <some error message>
        involving: <objects that caused the problem>
```

In the first case, the error is bad enough that the command which was being done has to be aborted, and the system returns to command level. Here are a couple of examples, both involving normalisation. If the normalisation of some rule fails, then one of the following error messages may be generated:

```
        mishap: normalisation failure
        involving:  < .... >
```

when the rule is illformed, or of the wrong sort - an ID rule in the lexicon, for example; or

```
        mishap: Feature specification error
        involving:  < .... >
```

when there is something wrong with the feature syntax specification.

The second sort of error is not really an error, but just a warning. Processing continues after the message has been produced. Here are some examples of this type of warning.

```
        warning: HFC failed
        involving: < .... >
```

This will appear when the HFC routine cannot find a head in an ID rule.

```
warning: FFP failed
involving: < .... >
```

In CONTROL mode, this is how the parser reports the problems it has using the rule chosen.

Hishaps, then, need correction, usually by editing the text files with the grammar data in. Warnings do not need correction, although they nay sometimes be an indication of a Mistake.

5.3.2 System errors

Any other error messages, not specifically of the for* indicated above, *d^re* produced by the system. Exactly what they look like depends on the Prolog system being used, but a typical POPLOG system error tight be:

```
;;; PR0L06 SYNTAX ERROR - Expecting a separator, operator or closing bracket
;;; ITEM: baz
;;; PARSING: foo ...
```

This particular type of error, a syntax error, can occur if a command is Mistyped, or not understood by the system (although sometimes the response in this latter case will simply be NO - see 4.5). Either way, the user can carry on typing in cowhands afterwards - nothing will have been disturbed.

Prolog syntax errors can also occur if data in the data files has been mistyped. For example, consider the following pair of ID rules:

```
si:     s –> np, vp
vpl:    vp –> v np.
```

There are two syntax errors here (i) a Kissing period after the first rule, and (ii> a missing comma in the second rule between V and NP. Both of these will generate an error like the example above. Another common cause of syntax errors is not matching up round or square brackets properly.

Prolog systems vary on whether they can recover from syntax errors. Some of them return to command level straight after the first one, others continue, trying to locate as many syntax errors as possible. Often, the first error detected may not be the only one in the file - the user should expect to have to try loading the file several times, correcting errors between each load, to ensure that all errors have been found.

Suppose the following command is given:

```
normid from idrules to normedrules.
```

If there are any syntax errors in the data file IDRULES, then a syntax error will be produced. Unfortunately, due to the design of Prolog, the error message will not be printed on the terminal - it will be put into the output data file NORMEDRULES instead. So the user may not realize anything has gone wrong. For this reason, it is advisable to give the command as:

normid from idrules.

This will send output to the terminal, first of all, just to check that there are no syntax errors. Once all syntax errors have been cleared, the earlier command can be given safely to store the normalised data.

Apart from syntax errors, the only other sorts of errors one should expect are errors involving data files, for example, trying to read data from a nonexistent file, or giving a file a name which is not permitted by the operating system. Again, the particular error produced will vary. A POPLOG example is:

```
;;; PROLOG ERROR - CAN'T OPEN FILE (no such file or directory)
;;; INVOLVING: idrules
```

This will appear if you attempt to use a file IDRULES which doesn't exist.

5.4 Switches

Various features within ProGram are controlled by switches which can be turned on and off by the user. The switches fall into three categories: trace options, parse modes and general flags. The flags, acceptable abbreviations for them, and their functions are given below:

Switch name	Abbreviation	Function
TRACE_OUT	TRACE, TO, T	controls tracing of output data (data displayed as it is written)
TRACE_IN	TI	controls tracing of input data (data displayed as it is read)
SHOW	S	controls output form (SHOW on produces friendlier, but not machine-readable output)
WATCH	W	controls the parser's scan mechanism (WATCH on produces full trees each scan)
AUTO	A	switch on automatic mode
MONITOR	MON, M	switch on monitor mode
CONTROL	CTRL, C	switch on control mode
CHECK	CH	controls user-checking of parser decisions
ONEPARSE	O	controls whether to stop at first parse
NOCONJ	NOC	controls whether Conjunct Realization Principle is used (ON = not used)
NOFOOT	NOF	controls whether Foot Feature Principle is used (ON = not used)
NOHEAD	NOH	controls whether Head Feature Convention is used (ON = not used)
NOLEX	NOL	controls whether lexical subcategorization is used (ON = not used)
NOBAR	NOB	controls whether bar levels are being used

LEJCHETA	L	controls whether tetarules only apply to ID rules that introduce lexical categories (when ON) or to all ID rules (when OFF).

Switches nay be turned on or off using the ON and OFF predicates, for examples

```
check on.
oneparse off.

auto, nohead on.
c, o, chf to off.
```

Several switches can be changed in one statement, and the abbreviations given Bay also be used if desired. The current setting of the switches can be viewed with the command

```
status.
```

The initial setting of switches is:

```
Parse mode AUTO
CHECK ON
LEXMETA ON
All others OFF
```

This can be changed if desired by setting switches in the custom module (see 5.8.2). Note that the parse modes cannot be turned OFF, you can only switch to a different mode.

5.5 Libraries

This section describes the library files in the system. Library files are optional parts of the system which can be loaded in by the user with the LIB command:

```
lib <libraryname>.
```

Alternatively they can be included automatically by having the assertion

```
included ib (<1 ibraryname>)).
```

in the customisation file (see 5.6). In a few cases, they are loaded automatically (e.g., the FILTER library loads if you select the WATCH option).

The *LOCAL help file should contain details of any libraries local to your own installation. Local libraries can be loaded either with

```
custom <1ibraryname).
```

as a command, or by putting

```
include(custom(<libraryname>)).
```

in your customisation file.

5.5.1 FILTER

The FILTER library provides more sophisticated IO handling than is available in the basic system monitor. Except when using the system in batch mode, or on a machine which does not have a large address space, it is usual to have FILTER included automatically, i.e. in the customisation file. It contains all the trace handling routines, as well as the filtered input routines described below. Note that this means that unless FILTER is loaded, the trace switches do nothing.

The FILTER library provides two mechanisms for combining and controlling input data files. Using the basic system only, an input data file specification is simply a filename in a FROM or USING clause (see 3.4 for more details). For example, to normalise ID rules from a data file called NPRULES the command would be:

```
normid from nprules.
```

Using the FILTER library, the input can be specified as the concatenation of two (or more) files, using the word THEN - for example:

```
normid from nprules then verbrules to normed.
```

This command normalises the rules in the file NPRULES and then the rules in the file VERBRULES. All the resulting normalised rules are put in a file called NORMED. THEN can also be used in a USING clause - for example:

```
parse using normed1 then normed2 for normed ID rules.
```

Being able to concatenate files in this fashion allows the user to write a grammar modularly, with different files for different parts of the grammar. They can then be tested in isolation before being combined into a full system, using THEN concatenation.

The second feature provided by lib filter is the interactive filter, invoked by the word SOMEOF - for example:

```
normlex from someof mylex.
```

This command normalises data from the file MYLEX, but interposes an interactive filter between the file and the program. This means that every time the program reads a new rule from the data file, the name of the rule is reported to the user. The user then has several options:

1. accept the rule - so the program uses it
2. reject the rule - go and read another one and report that
3. look for a rule with a specific name and use that
4. start again (at beginning of the file)
5. behave as though the end of the file has been reached

The SOMEOF filter allows the user to be selective about what data within a file is to be used, so that, for example, particularly troublesome rules can be easily isolated for testing. For an example of this filter in use, see 4.5.

THEN and SOMEOF can be freely combined - for example:

```
parse using foo then someof baz then bing.
```

This command uses file FOO, then BAZ with the filter interposed, and then BING. Brackets allow the grouping to be rearranged:

```
parse using foo then someof (baz then bing).
```

This command uses the filter on both BAZ and BING.

Suppose the ID rules for the grammar have been developed in two files, which have, perhaps, been tested separately. Let's say the normalised VP rules are in NVPRULES and all the rest are in NIDRULES. To load them both for parsing, the following command could be given:

```
parse using nvprules then nidrules for normed idrules.
```

Alternatively, using the command:

```
load.
```

to load up the whole grammar would cause NIDRULES to be loaded (since it has the standard name) but not NVPRULES. But if we subsequently gave the command:

```
parse using nvprules then nidrules.
```

all would be well. We do not need a FOR clause since the system already knows about the normed ID rules. However, the following commands would not work:

i.
```
load.
parse using nvprules.
```

ii.
```
load using nvprules then nidrules.
```

In the first case, NVPRULES would replace the existing rules, so NIDRULES would effectively not have been loaded. In the second case, all the rules get loaded, but then the LOAD command reloads NIDRULES only, again replacing all the existing rules.

5.5.2 MORECOMMS

The commands built in to the basic system are just the system primitives, NORMID, NORMLEX, PARSE, etc. The MORECOMMS library provides some compound commands which make common operations easier. It also provides a good illustration of how the user can build up more complex commands if desired. The commands provided are as follows:

NORMHFC - normalise ID rules and do HFC processing on them. If metarules are not being used, these two operations can safely be combined (see 3.3).

NORM_IDRULES - do NORMHFC using the standard filenames.

NORM_MISC - normalise lexicon, LP rules, FCD's, FCR's, RAC's from standard filenames.

NORM_AND_EXPAND - normalise ID rules and metarules, expand the grammar and do HFC on the result. Standard filenames used throughout.

NORM_GRAM - normalise a whole grammar (without metarules), using standard filenames.

NORM_META - normalise (and expand) a whole grammar (with metarules), using standard filenames throughout.

LOAD - load in all the normalised data required for parsing, using standard file names.

GO - load up data (if necessary) and run the parser.

DEBUG - set switches for convenient debugging.

The commands which use standard filenames do not need to have their inputs and outputs specified (any specification will be ignored). The standard filenames for all data modules are given in 3.5.

5.5.3 NEWWORDS

The NEWWORDS library provides a simple mechanism for dealing with specific unknown words. The library should be loaded after the (normalised) lexicon has been loaded. For each new word to be used, a clause 'newword(<word>).' must be provided. This can be done in several ways. The command

```
assert(newword(foo))
```

adds 'foo' as a new word. Alternatively, a data file containing several new words can be constructed. A data file (WORDS1, say) with some new words in it might look like this:

```
newword(grengle).
newword(foo).
newword(thunk).
```

Such a file could then be loaded with a USING clause:

```
parse using words1.
```

Note that no FOR clause is possible since WQRDS1 is not a standard data nodule. Now, whenever the parser comes across a word which is not in the lexicon, but which is specified as a NEWWORD, it will try parsing using every category in the lexicon for that word. Any successful parse will produce a candidate for the syntactic category of the word. Unfortunately, the parser does not remember the successful categories at present.

The NEWWORDS library is provided for the benefit of users who are interested in experimenting with the automatic acquisition of lexical subcategorisation information from texts.

As one might expect, using NEWWORDS slows the parser down a lot, since there are many alternatives to be considered. It works best with only one new word in the sentence, and the nearer the beginning that word appears, the better.

5.6 Customising the system

The ProGra® system has been designed with portability in mind, using only 'standard' Prolog for the main system and libraries. The system makes no assumptions about the operating system and file handling mechanisms - these aspects, as well as the inclusion of nonstandard modules (for example, the SHOWTREE module in the POPLOG version, which makes use of P0P11 libraries), must be locally tailored in the CUSTOM module. This section describes what customisation is possible, and how to achieve it.

There is only one file in ProGram which needs modification to suit local requirements, and that is the PROGRAM module itself. As the comments at the head of the module explain, a predicate CUSTOMISATION.FILE must be defined, whose function is to REC0N5ULT the customisation file described below. This predicate has to be specifically defined taking into account the format of filenames in the local operating system. For example, under P0PL06 UNIX, the clause might be:

```
customisation_file i- reconsult('$program/custom/custom.pi ' ) .
```

and under POPLOB VMS it might be:

```
customisation_file :~ reconsult ('program:[custom]custom.pi ' ) .
```

Note that the actual name of the customisation file is irrelevant.

The only other possible change to the PROGRAM module is the use of the operator ':-' to mean 'top-level-invoke'. In some systems, this may be a different symbol, e.g. '?-'• It appears twice in the PROGRAM module, and it also appears in the library module NEWWORDS. There are no other changes to be made to the main system, all other customisation takes place in the customisation file. Note that the customisation file is loaded before the operator declarations have been made, and so must not presuppose that infix operators, etc., have been defined.

The customisation file is simply a file of Prolog code which gets reconsulted when the system loads. Thus it can be used to define new local commands, etc., simply by providing clauses for them (see below for details of redefining system commands, however). It can also provide new definitions for three predicates which are used by the system: FILENAME, MODULENAME, and INCLUDE.

The predicate FILENAME takes two arguments. The first is the name of a file, as given to the system by the user (i.e. as typed in a command). It is a Prolog atom. The second should be instantiated by the predicate to the actual filename for that file, suitable for use in a SEE or RECONSULT. If no FILENAME clause is specified, the default clause does nothing, i.e. it returns the filename exactly as given.

For example, in a system (e.g. VMS) where filenames have file types, and the conventional type for Prolog files is '.pl', not redefining FILENAME would lead to the necessity of commands like:

```
parse from 'foo.pl' using 'id.pl' for normed idrules.
```

But if we define FILENAME to add the .pl automatically, as follows:

```
filename(X,Y) :- name(X,XN), append(XN,'.pl',YN),name(Y,YN).
```

then we could write

```
parse from foo using id for normed idrules.
```

Similarly, we could redefine FILENAME to get grammar files from a specific directory, which would get automatically prefixed onto every file name.

MODULENAME is similarly a two place predicate which serves the same function as FILENAME, but for system modules. The first argument takes different forms according to what sort the module is as follows:

1st arg.	Module sort
lib(foo)	library module foo
custom(foo)	customised module foo
foo	system module foo

This allows the different classes of module to be stored in separate subdirectories if desired. The default clause for MODULENAME just returns the name of the module (i.e. it strips off LIB, CUSTOM, etc.).

INCLUDE is a predicate with one argument, which should be a module name as in the table above. When the ProGram system loads, only the MONITOR and COMMAND modules are loaded automatically (although most other modules will be loaded automatically as soon as they are needed). Modules specified in an INCLUDE clause are also loaded at system-load time. This can be useful for several reasons. (i) If you are precompiling the system (e.g. a POPLOG 'saved image'), and you want various bits of the system included in it, this is the clean way to get them in. (ii) Some modules (e.g. MORECOMMS, FILTER) don't load automatically and have to be loaded explicitly anyway. (iii) You may prefer to have modules loaded at the start, rather than interrupting

your work half way through a session. (iv) All the INCLUDE clauses are handled after the monitor and commands have been loaded, and in the order that you specify. This means that if you want to redefine system commands, you can do it in a separate customised module and INCLUDE it. You cannot do it in the main customisation file, since this is loaded first, and so the system definitions Mould overwrite yours, not vice versa.

For example, the POPLOG customisation module SHOWTREE must be loaded after the library module FILTER, since it redefines a predicate in the latter. This can be done by putting the following clauses into the customisation file:

```
include(lib(filter>).
include(custom(showtree)).
```

Finally, a customisation file can set switch values, if the initial default settings (all OFF except CHECK, parsing mode « AUTO) do not suit. Clauses like

```
?- o on.
?- ch off.
?- c on.
```

in the customisation file, will achieve this. Note that the prompts shown here need to be typed into the file exactly as shown.

For an example of a fairly sophisticated customisation file, see the UNXcustom module, which customises Program for POPLOG on VAX UNIX (see appendix 1).

5.7 The lexicon interface

ProGram is mostly concerned with grammar rules. The lexicon is provided mainly for convenience - so that ordinary sentences can be used, rather than strings of syntactic categories. In particular, the built in lexicon handling is relatively simple-minded, and would not cope well with a realistically large lexicon resulting, for example, from all the morphological forms of a large number of words. The more experienced user may well wish to design lexicon handling routines (in Prolog or, in the POPLOG version, P0P11 or even LISP) which access a larger lexicon more efficiently. This section gives details of the interface between ProGram and its lexicon to enable such routines to be designed.

The lexicon lookup in the parser consists of a call of the predicate LEXRULE with three arguments as follows:

```
lexrule(Nord,Name,Cat)
```

The arguments are:

Word - the word being looked up (already instantiated).
Naie - this is a ten of the fore lex(M) where N is the naie of the lexical rule used. N should be set by the lexicon routine, and is used for display in parse trees.
Cat - the syntactic cateqory of the Uord. This is set by the lexicon routine and should be a fully normalised, fully specified category, according to the feature syntax specification.

Alternative syntactic categories for a word are obtained by Prolog backtracking, such alternatives represent alternative solutions to the LEXRULE predicate.

All the parse mode handling behaves just as usual - in control iode the user is asked to specify a lexical rule naie before lookup* This is not passed to lexrule, but after lexrule has returned, the user-specified name (say UN) is compared with the returned name (N, above). Only if UN * N or functor(N, UN, J is the rule used, otherwise the parser askes for an alternative rule.

The lexical category must be fully normalised (see chapter 3). The system provides a predicate NORMFEAT that does this, i.e. given a category C which is specified using aliases, etc., the call

```
normfeat(C,NC)
```

returns NC as the normalised version. NORMFEAT gives a Aishap if it cannot normalise - in particular, unbound variables cannot be normalised, and so normalising a normalised category will often fail in this way.

The lexical coefficient feature in ProGra* is handled by the lexicon normaliser. This means that any new lexicon lookup routine must set the LEXICAL feature itself. It oust be set to the name of the ID rule (or the functor of the name, if the name is a complex term) in which the word is introduced. Alternatively, one can set the switch NDLEX to stop all lexical coefficient checking. The system module NORMLEX (.../program/sys/normlex in the UNIX version of ProGram) contains more information and some potentially useful predicates.

Any new routines that redefine this predicate should be loaded as the normed lexicon data module, i.e. replacing the normal normed lexicon. The built in lexicon handler behaves as follows: NORMLEX is given rules of the form

```
vtr: vlex(sing) ->- likes, loves, sees.
```

and produces clauses like:

```
lexrule(X,lex(vtr), ... ) :- once(member(X,tlikes,loves,sees])).
```

where ... is the normalised version of VLEX(SING) (an aliased category) with its lexical coefficient set. These clauses are normally all written to a data file which is loaded as normed lexicon.

5.8 The structure of the file ivitgrn

This section contains an index to all the files in ProGram. The index is organised in the saute way as the files are in the POPLOG UNIX system, namely as a collection of subdirectories under the directory Iprogra*.

5.8.1 PROGRAM: the top level

custoa	* directory containing cuitoiised libraries
deto	- directory containing demonstration graatar
help	- directory containing aain help files
lib	- directory containing optional systea libraries
sys	- directory containing aain systea files

5.8.2 CUSTOMx customised libraries

PL06custoa.pl	- customisation for Prolog-only systea
UNXcustoa.pl	- custoaisation file for VAX UNIX POPLW
UNXakgds	- UNIX (cshl coaaand file to build saved iaage
VKScustoa	- custoaisation file for VAX VMS POPLOB
seetree.p	- POP11 library to interface to LIB SHOWTREE
showparse.p	- P0P11 library to interface Pro6raa to SEETREE
shoittree.pl	- PRDL06 interface to SHOHPARSE
usercoaa.pl	* library of systes-dependent coaaands
vedfiles.pl	- library for siaple VED interface

5.8.3 DEMO: a demonstration grammar

aliases	fed	for	features
idrules	lexicon	lp	aetarules
nfed	nfer	nidruies	nlexicon
nip	naeta	nrac	rac

5.8.4 HELPs the help files

assuaed	- assuaptions about 6PS6 foraalisa
coaaands	* suaaary of basic systea coaaands
custoa	- ho* to custoaise the systea to local requireaents
data	- details of data file organisation and use
errors	- notes on different sorts of error and cures

example	- examples of the system in use:
egdata	
egexpand	
egfilter	
egnorm	
egparse	
egtest	
egtree	
gpsg	- a bibliography of recent work in GPSG
grammars	- details of grammar specification for the system
help	- information about help facilities
hfc	- details on head feature convention processing
index	- an index to the system
intro	- introduction to the system documentation
lexicon	- notes on the interface to the lexicon
libraries	- details of the optional system libraries
local	- local installation notes
metarules	- details of use of metarules
norm	- how to normalise grammar data in the system
overview	- overview of system functions
use	- basic use of the system
parsing	- how to use the parser for grammar testing
poplog	- relevant sources of information on POPLOG
switches	- the system switches and what they do
systems	- details of other computational GPSG systems
use	- basic use of the system

5.8.5 LIB: optional system libraries

filter	- more sophisticated data file handling routines
morecomms	- some higher level system commands
newwords	- library to handle unknown words

5.8.6 SYS: the main system routines

comands	- basic top level commands
crp	- Conjunct Realisation Principle routines
expand	- grammar expansion (using metarules)
fcd	- FCD and category matching routines
ffp	- Foot Feature Principle routines
helpinfo	- data for built-in HELP facility
hfc	- Head Feature Convention routines
monitor	- basic system monitor - utility routines etc

normfeats	- main feature normalisation routines
normid	- ID rule normalisation routines
normlex	- lexicon normalisation routines
normlp	- LP rule normalisation routines
normmark	- FCD, FCR and RAC normalisation routines
normmeta	- metarule normalisation routines
normrules	- utility routines for normalisation
parse	- main parsing module
program	- main system loader

6. A Demonstration Grammar

6.1 Features

The top-levels of feature structure take a standard form.

```
feature [root,  cat, foot, conj].
feature [cat,   bar, head].
feature [bar,   {lexical, 1, 2}].
feature [head,  major, minor].
```

Major features - one for each phrasal type. Prepositions have a sub-feature which is a terminal symbol feature.

```
feature [major, {v, n, a, p, conj}].
feature [p,     {by, to, in, on, with}].
```

Minor features.

```
feature [minor, agr, {case, vform}].
feature [agr,   {singular, plural}].
feature [vform, {finite,passive}, auxiliary, inverted].
feature [case,  {nominative, possessive}].
```

The foot feature only has one candidate coefficent - a CAT for doing SLASH categories.

```
feature [foot,  cat].
```

Very simple conjunct features - just two possibilities.

```
feature [conj,  {neither, nor}].
```

The last two are boolean features

```
boolean auxiliary.
boolean inverted.
```

6.2 Aliases

The aliases below 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]]]] ).
```

The following aliases let you specify minor features too if desired - note that a preposition version is not included, simply because it is not used in the grammar at all.

```
aiias(  v(N,M),  [root,teat,[bar,N],[head,[major,v3,
                 CninorIH3333 ).
aliast  n(N,M), [root,[cat,Cbar,N3,[head,[major,n],
                 CminorlHJ]]] >.
alias(  a(N,M),  [root,[cat,[bar,N],[head,[major,a],
                 CAinorIH3333 )•
```

The next group of aliaies is designed to be used as the minor features arguent in the aliases above. The main reason that the actual feature names are the full words is so that the abbreviations can be used here. Thus, for example V(1,C**AUX**]) means a VP which is FINITE and ^AUXILIARY.

```
alias(  sing,  [agr,singular]).
alias<  plur,  [agr,plural1).
alias(  no«,   [case,no»inativel)•
alias(  ace,   ^case).
alias(  aux,   Cvform, finite, -^auxiliary]).
alias(  inv,   [vforn, finite, ^inverted]).
alias(  fin,   [vfom, finite, -auxiliary, -inverted!).
aliasf  pass,  [vform, passive, -auxiliary, -inverted]).
```

Me can provide a prepositional alias to simplify the stipulation of particular prepositions*

```
alias*  p(N,P)$_f$[root,[cat,[bar,N],[head,[«ajor,[p,P]]]]] ).
```

And a similar one to sinplify the stipulation of conjunction words.

```
alias(  c(C),  [root,teat,[bar,lexical],[head,[major,C]]]] ).
```

The following aliases allow the use of the H (for HEAD) notation.

```
aliasf  h(N),  [root,[cat,[bar,N],[head,[major]]]] ).
alias(  h,     h(lexical)).
```

The system requires the lowest bar level to be called LEXICAL - the alias below allows us to use 0 instead if we wish.

```
alias(  0,     lexical).
```

Finally, an alias which lets us use the slash notation for slash categories. Prolog understands expressions like X/Y as the same sort of thing as V(X,Y), only using V' instead of V. What the alias does is complicated - it is not just a straight translation: NORHFEAT does normalisation on feature expressions, PATHFOR locates the coefficients of particular features, and PROTECT ensures that the resulting feature is not normalised again (things must not be normalised twice).

```
alias(X/Y, Z) *-
        normfeat(X,XN),normfeat(Y,YN),
        pathfor(foot,YN,   ' * ' ),
        pathfor(cat,YN,YCat),
        pathfor(foot,XN,[tcat!YCat]3),
        Z  « protect(XN).
```

6.3 ID rules

The sentence rule uses variables to specify control.

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

In the VP rules that follow, notice the underline character (_) in the rule labels. We want them to have different names, so that lexical subcategorisation works, but everything after the underline gets omitted from the parse tree displays, etc., so the label appears as just VP.

```
vp_1: v(1) --> h.
vp_2: v(1) --> h,n(2).
vp_3: v(1) --> h,n(2),n(2).
vp_4: v(1) --> h,v(2).
vp_5: v(1,[aux]) --> h,v(1,[pass]).

vp_pp: v(1) --> h(1),p(2).
```

There are two NP rules - one for common nouns, the other for proper nouns and pronouns.

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

The N1 category allows for the introduction of adjectives.

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

Adjective phrases allow adjective modifiers.

```
ap_1: a(1) --> h.
ap_2: a(1) --> a(0),h(1).
```

And the prepositional phrase rule is straightforward.

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

A couple of rules that together permit one type of coordinate structure.

```
coord: root --> [root, [conj,neither]], [root, [conj, nor]]++ .

conj:   [root, [cat,B], C] --> c(C), [root,[cat,B],^conj]
        where B is bar, C is conj.
```

Finally, a rule for topicalisation. The topicalised category and the slashed category must have the same major feature. Note that M must be defined before it is used in the WHERE clause.

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

6.4 Metarules

It is sensible to lay metarules out so that they are easy to read. Here is passive.

```
pass:   v(1) --> ... , n(2)

        ==>

        v(1,[pass]) --> ... , opt(p(2,by)).
```

And here is 'subject-auxiliary inversion'. We need variables to specify correspondences between the categories (using MATCHES).

```
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 following metarule allows slash categories to terminate in a 'missing' item of the appropriate kind.

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

        ==>

        C1/C2 --> ...  .
```

Finally, a metarule for introducing that-less relatives.

```
relcl:  (N1 --> ...     where N1 is n(1))

        ==>

        N1 --> ... , v(2)/n(2).
```

6.5 LP rules

A lexical category precedes a non-lexical.

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

NP's precede VP's, adjectives precede nouns, nouns precede their PP modifiers.

```
n(2) << v(1).
a(1) << n(1) << p(2).
```

Conjuncts that begin with 'neither' precede those that begin with 'nor'.

```
[root,[conj,neither]] << [root,[conj,nor]].
```

And the final LP rule forces slash categories to be final in their constituents.

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

6.6 Feature coefficient defaults

Our demonstration grammar contains just three FCD's, giving defaults for the minor features. The first says that accusative is the unmarked phrasal case, and the second and third stipulate that unmarked lexical items are neither +inverted, nor +auxiliary. Note that specifying FOOT in the exclusion list forces the defaults onto the real minor features, and prevents the defaults applying to minor features appearing within a foot feature.

```
        feature         excl     lexical         phrasal
        -------         ----     -------         -------
fcd(    case,           [foot],  free,           acc      ).
fcd(    inverted,       [foot],  -inverted,      free     ).
fcd(    auxiliary,      [foot],  -auxiliary,     free     ).
```

6.7 Feature cooccurrence restrictions

Verbs are never marked for case.

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

Not even in a slashed category.

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

If something is inverted, then it is an auxiliary.

```
fcr(    inverted, [foot], +inverted, vform, [foot],
        [vform, finite, +auxiliary]).
```

6.8 Root admissibility conditions

Those given just prohibit categories with unspecifed major feature or bar level.

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

6.9 The lexicon

The demonstration lexicon is straightforward. The only points to note are (i) that the rule labels are complex terms. Only the bit outside the brackets is used for lexical subcategorisation, etc. The rest serves to identify each lexical rule uniquely, so that we can locate them properly when in CONTROL mode. And, (ii) we have to specify all the features we want specified (we cannot expect defaults to set them). Defaults check settings, but they will not add extra information to a category.

```
vp_1(sf): v(0,[sing,fin]) ->- jumps,runs,sings.
vp_1(pf): v(0,[plur,fin]) ->- jump, run, sing.

vp_2(sf): v(0,[sing,fin]) ->-  loves,sees,closes.
vp_2(pf): v(0,[plur,fin]) ->-  love, see, close.
vp_2(ps): v(0,[pass])     ->-  loved,seen,closed.

vp_3(sf): v(0,[sing,fin]) ->-  hands, gives,buys.
vp_3(pf): v(0,[plur,fin]) ->-  hand,  give, buy.
vp_3(ps): v(0,[pass])     ->-  handed,given,bought.

vp_4(sf): v(0,[sing,fin]) ->-  thinks, believes,knows.
vp_4(pf): v(0,[plur,fin]) ->-  think,  believe, know.
vp_4(ps): v(0,[pass])     ->-  thought,believed,knew.

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

np_2(prop_nom):  n(0,[sing,nom]) ->- kim,sandy,lee.
np_2(prop_acc):  n(0,[sing,acc]) ->- bill,ben,bert.
np_2(s_nom):  n(0,[sing,nom]) ->- he,she.
np_2(s_acc):  n(0,[sing,acc]) ->- him,her.
np_2(p_nom):  n(0,[plur,nom]) ->- they.
np_2(p_acc):  n(0,[plur,acc]) ->- them.

nb_1(s_acc): n(0,[sing,acc]) ->- book,man,woman.
nb_1(p_acc): n(0,[plur,acc]) ->- books,men,women.

nb_1(s_nom): n(0,[sing,nom]) ->- tree,boy,girl.
nb_1(p_nom): n(0,[plur,nom]) ->- trees,boys,girls.

np_1(sdet): a(0,[sing]) ->- a,an,every.
np_1(pdet): a(0,[plur]) ->- all,some.

ap_1: a(0) ->- red,blue,green,yellow.
ap_2: a(0) ->- very,bright,dark.
```

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

conj(a): c([conj,neither]) ->- neither.
conj(b): c([conj,nor]) ->- nor.
```

References

Clocksin, William, and Christopher Hellish (1981) *Pr ogr awwivg in Prolog.* Berlin: Springer-Verlag.

6awron, Jean Hark, Jonathan King, John lamping, Egon Loebner, Anne Paulson, Geoffrey Pullum, Ivan Sag & Thomas Wasow (1982) The 6PS6 linguistics system. 74-81. Also distributed as *Hewlett Packard Computer Science Technical Hot** CSL-82-5.

Gaidar, Gerald, and Geoffrey Pullum (1982) *Generalized phrase structure grawwar: a theoret ical synopsis,* Bloomington; Indiana University Linguistics Club mimeo. Also available as *University of Sussex Cognitive Science Research Paper 1* (CSRP 007).

Gazdar, Gerald, Ewan Klein, 6eoffrey Pullum, and Ivan Sag (1982) Coordinate structure and unbounded dependencies. In H. Barlow, D. Flickinger *I* I.A. Sag (eds.) *developments in Generalized Phrase Structure Grawwar: Stanford Working Papers in Grawaatical Theory, Volute 2.* Biooraington: Indiana University Linguistics Club, 38-68. Also available as *University of Sussex Cognitive Science Research Paper* 6 (CSRP 006).

Hardy, Steven (1982) The POPLOG programming system. *University of Sussex Cognitive Science Research Paper* 3 (CSRP 003).

Hardy, Steven, and Aaron Slogan (1982) POPLOG: a multi-purpose, multi-language program development environment. Himeo, Cognitive Studies Program, University of Sussex.

Hellish, Christopher, and Steven Hardy (1983) Integrating Prolog into the POPLOG environment. *Proceedings of the Eighth International Joint Conference on Artificial Intelligence,* 533-535.

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

Appendix 1

The system under Unix POPLOG

This appendix deals with the use of the Unix POPLOG version of the system (likely, in our view, to be the version in most common use). By the time a reader gets the system, it may be slightly out of date, so consult your own GHELP LOCAL help file for accurate information.

ProGram help files are accessed just like POPLOG help files (i.e. using VED) by giving the command

```
ghelp <filename>
```

to POP11 or PROLOG or VED, in the usual way. Since ProGram uses the word 'help', the usual Prolog 'help' command (to access Prolog help files) is renamed 'hlp'. The POP11 'help' command remains unchanged.

Data file names are automatically suffixed with '.pl'. They are prefixed with a directory which can be set with the GRAMMAR command (see below). It defaults to the current directory, unless specifed when the system is run (see below).

Extra commands available are:

TIME	- display the current date and time.
SETID	- set the ID string to be the current date and time.
GRAMMAR	- examine or change the grammar directory. The grammar directory should be a normal UNIX directory specification (ending with a '/ '). If the directory is already correct, type nothing (i.e. just hit <RETURN>).

In the POPLOG version of ProGram, there is an additional library, SHOWTREE, which redefines the parse-tree display routines to use a customised version of the POPLOG (POP11) library SHOWTREE (see POPLOG's help file on SHOWTREE). This is only used when the SHOW or TRACE_OUT switch is ON. The main tree is displayed graphically on the screen (using VED) and the keyboard is disabled except for the following keys, whose functions are redefined as follows:

CURSOR UP	- move up to parent node
CURSOR DOWN	- move down to middle daughter node
CURSOR DOWN-LEFT	- move down to leftmost daughter node
CURSOR DOWN-RIGHT	- move down to rightmost daughter node
CURSOR LEFT	- move to next sister left
CURSOR RIGHT	- move to next sister right
POSITION PUSH	- display feature tree for this node
POSITION POP	- display main parse tree
SCREEN REFRESH	- screen refresh
TOP OF FILE	- print tree (if possible)
END OF FILE	- return to parser

An extra command is also provided:

```
tree_printable.
```

This causes subsequent trees to be displayed in a printable format. Alternatively the user can specify a P0P11 procedure TREE_PRINTER which will display the tree itself (accessing the VED buffer globally). The tree will only be printable if one of these options is selected.

Another library, VEDFILES, does automatic checking of data files which are currently being used and which are also being edited in VED. It ensures that the lost up to date version is always loaded.

The ProGram system is situated under the directory denoted by the environment variable '^program'. The system files are distributed as documented in the customisation module, which is

```
$program/custom/UNXcustom.pl
```

The custom library (*program/custom) contain an executable (CSHELL) file UNXmkgds which rebuilds a saved image (fprogram/custom/image.psv) of the system. The saved image thus created may be run from CSHELL with the command:

```
prolog -gds
```

An optional extra argument may also be given, which is taken as the initial setting of the grammar directory (see GRAMMAR above).

```
popll -gds Vgraml
```

If the extra argument it missing, then the current directory is used.

A system-wide command UNIX command 'program' is provided to make access to this saved image easier. To run ProGram, all you need to do is type:

```
program
```

or

```
program <grammar directory)
```

to the CSHELL.

To run ProSram from inside Prolog, type:

```
consult('$program/sys/program.pi       ').
```

There will be several loading messages and then the system will start up as described in 4.5.

Appendix 2

Recent PSG implementations

1. John Bear

All paths, left corner chart parser. Uses features for agreement and unbounded dependencies. Relative clauses, questions, existentials. No semantics. Language: Interlisp. Machine (OS): DEC20 (TOPS 20).

Linguistics Research Center, P.O. Box 7247, University Station, University of Texas, Austin, TX 78712, USA.

Bear, John (1981) Gaps as syntactic features. MA dissertation, University of Texas at Austin. Published by IULC, Bloomington, IN., in 1982.

Bear, John and Lauri Karttunen (1979) PSG: a simple phrase structure parser. *Texas Linguistic Forum* 15, 1-46.

2. Hewlett Packard

Top-down parser and transducer yielding first order logic translations. Includes metarules, features, some feature instantiation principles, slash categories, but not ID/LP. Intended as portabl front-end for databases, and currently hooked up to relational database in HPRL (a development of FRL). System currently undergoing thorough revision and redesign. Language: LISP (PSL). Machines (OS): VAX 11/780 (UNIX), HP 9836 (NMODE).

Geoffrey K. Pullum, Daniel P. Flickinger, Carl Pollard, Derek Proudian, Ivan A. Sag, Thomas Wasow, (and formerly also Jean Mark Gawron and Anne E. Paulson). Computer Science Laboratory, Hewlett Packard Company, 1501 Page Mill Road, Palo Alto, CA 94304, USA.

Gawron, Jean Mark, Jonathan King, John Lamping, Egon Loebner, Anne Paulson, Geoffrey Pullum, Ivan Sag & Thomas Wasow (1982) The GPSG linguistics system. *Proceedings of the 20th Annual Meeting of the Association for Computational Linguistics,* 74-81. Also distributed as *Hewlett Packard Computer Science Technical Note* CSL-82-5.

3. Mark Johnson

Suite of small programs: sentence generator employing features, instantiation, ID/LP; feature package defining unification, increment, etc.; LR(1) parser. Language: FranzLisp. Machine (OS): VAX 11/780 (UNIX).

Department of Linguistics, University of California at San Diego, La Jolla, CA 92093, USA.

4. James Kilbury

Modified Earley-Shieber parser usinf a "first" relation for the ID/LP formalism. Extension to all aspects of the GPSG framework including direct parsing with metarules is planned. Parser to produce semantic representations for in an AI system with German language interface. ID/LP parser. Language: Waterloo PROLOG, Version 1.4. Machine (OS): ITEL AS/5-7031 [IBM 370] (VM/SP).

Technische Universitat Berlin, Fachbereich Informatik (20), Institut fur Angewandte Informatik, Projekt KIT, Sekr. FR 5-8, Franklinstrasse 28/29, D-1000 Berlin 10, West Germany.

Kilbury, James (1984a) A modification of the Earley-Shieber algorithm for direct parsing of ID/LP grammars. Unpublished paper, Technische Universitat Berlin.

Kilbury, James (1984b) GPSG-based parsing and generation. To appear in Claus-Rainer Rollinger (ed.) *Probleme des (Text-)Verstehens - Ansatze der Kunstlichen Intelligenz.* Tubingen: Max Niemeyer.

5. Francis Jeffry Pelletier

Recursive descent parser. Incorporates metarules, slash categories. Provides intensional logic translations. Doesn't incorporate features or ID/LP. Language: SNOBOL (SPITBOL dialect). Machine (OS): Amdahl 470 (MTS).

Department of Philosophy, University of Alberta, Edmonton, Canada T6G 2H1.

6. Stephen G. Pulman

RTN based parser operating either depth or breadth first. Compiles metarules (not ID/LP) into RTN and then optimises. Slash categories included, but not other featural information. Minimal semantics associated with one test grammar. Language: POP11. Machine (OS): VAX 11/780 (VMS).

Linguistics, School of English and American Studies, University of East Anglia, Norwich NR4 7TJ, UK.

Pulman, Stephen (1983a) Generalised phrase structure grammar, Earley's algorithm, and the minimisation of recursion. In K. Sparck-Jones & Y. Wilks (eds.) *Automatic Natural Language Parsing.* Chichester: Ellis Horwood, 117-131.

Pulman, Stephen (1983b) Computational linguistics and language teaching. MS, UEA.

7. Lenhart K. Schubert

Left corner parser, with pruning of syntactically or semantically unusual alternatives. Incorporates features and morphological analysis, coordination and slash categories. Provides first order logic translations. Intended as a front end for a question-answering system with access to a logic-based semantic net. Doesn't incorporate metarules. Languages: LISP and PASCAL versions. Machine (OS): Amdahl 470/V8 (MTS).

Department of Computing Science, University of Alberta, Edmonton, Canada T6G 2H1.

Schubert, Lenhart (1982) An approach to the syntax and semantics of affixes in 'conventionalized' phrase structure grammar. *Proceedings of the 4th Biennial Conference of the Canadian Society for Computational Studies of Intelligence*, 189-195.

Schubert, Lenhart, and Jeffry Pelletier (1982) From English to logic: Context-free computation of 'conventional' logical translation. *American Journal of Computational Linguistics* 8, 27-44.

8. Hidetoshi Shirai

Deterministic parser based on PARSIFAL. Incorporates metarules, raising constructions, and unbounded dependencies. Montague semantics. Language: LISP. Machine (OS): Hitac M200H (VOS 3).

Department of Mathematical Engineering and Instrumentation Physics, Faculty of Engineering, University of Tokyo, Hongo 7-1-2, Bunkyo-ku, Tokyo 113, JAPAN.

Shirai, Hidetoshi (1983) Deterministic parser. In *Proceedings of the Workshop on Non-Transformational Grammars.* Tokyo: ICOT, 57-61.

9. SRI International (PATR-I)

CKY parser, feature system allows Boolean combinations of feature equalities interpreted on the fly, no metarules, semantics converted to first-order logic and passed to a theorem prover. Language: INTERLISP. Machine (OS): DEC20 (TOPS 20).

Stuart Shieber and Stan Rosenschein, SRI International, 333 Ravenswood Avenue, Menlo Park, CA 94025, USA.

Rosenschein, Stanley, and Stuart M. Shieber (1982) Translating English into logical form. *Proceedings of the 20th Annual Meeting of the Association for Computational Linguistics*, 1-8.

10. SRI International (PATR-II)

Parser: CKY (LISP), Earley's algorithm (Prolog); feature system: directed acyclic graph structures, semantics embedded in feature system; morphological analysis by method of Kimmo Koskenniemi (LISP, Lauri Karttunen) Languages - 3 implementations of the PATR-II formalism: INTERLISP (DEC20), Prolog (DEC20), ZETALISP (Symbolics 3600). Machines (OS): DEC20 (TOPS 20), Symbolics 3600.

Stuart Shieber, SRI International, 333 Ravenswood Avenue, Menlo Park, CA 94025, USA.

Koskenniemi, Kimmo (1983) A two level model for morphological analysis. *Proceedings of the 8th International Joint Conference on Artificial Intelligence,* 683-685.

Shieber, Stuart (1983a) Sentence disambiguation by a shift-reduce parsing technique. Technical Note 281, SRI International. Also in *Proceedings of the 21st Annual Meeting of the Association for Computational Linguistics,* 113-118. And in *Proceedings of the 8th International Joint Conference on Artificial Intelligence,* 699-703.

Shieber, Stuart (1983b) Direct parsing of ID/LP grammars. Technical Note 291, SRI International.

Shieber, Stuart, Susan Stucky, Hans Uszkoreit, and Jane Robinson (1983) Formal constraints on metarules. Technical Note 283, SRI International. Also in *Proceedings of the 21st Annual Meeting of the Association for Computational Linguistics,* 22-27.

Stucky, Susan (1983) Metarules as meta-node-admissibility conditions. Technical Note 304, SRI International.

11. Henry Thompson and John Phillips

Chart parser (intended for grammar testing). Incorporates all aspects of the 1982 GPSG framework: features, metarules, feature instantiation, coordination, etc. Semantics currently being implemented. Language: UCI LISP, FranzLisp. Machines (OS): DEC10 (Tops10), VAX 11/780 (UNIX).

Department of Artificial Intelligence, University of Edinburgh, Hope Park Square, Edinburgh EH8 9NW, UK.

Thompson, Henry (1981) Chart parsing and rule schemata in PSG. *Proceedings of the 19th Annual Meeting of the Association for Computational Linguistics,* 167-172.

Thompson, Henry (1982) 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 Generalized Phrase Structure Grammar: Stanford Working Papers in Grammatical Theory, Volume 2.* Bloomington: Indiana University Linguistics Club, 26-37.

Thoupson, Henry (1983) Crossed serial dependencies: a low-power parseable extension to BPS6. *Proceedings of the 21st* /Annaai *Meeting of the Association for Computational Linguistics,* 16-21.

Thompson, Henry, & John Phillips (1984) An implementation of GPSG within the MCHART parsing framework. Unpublished paper, Department *oi* Artificial Intelligence, University of Edinburgh.

Appendix 3

Recent Bibliography on NJL.JPSBJLJB

This bibliography includes diiifrta.ti.pni and published papers and monographs not already included in the References or in Appendix 2, above. It excludes unpublished work, and work written in languages other than English.

Anward, Jan (1982) Basic Swedish. In E. Engdahl and E. Ejerhed (eds.) *Readings on Unbounded Dependencies in* Scandinavian *Languages,* Acta Universitatis Umensis, Umea Studies in the Humanities 43. Stockholm: Almqvist & Wiksell, 47-75.

Bachi, Efimon, and Barbara Partee (1980) Anaphora and semantic structure. In J. Kreiman fcA.E. Ojeda (eds.) *Papers from* the *Parasession on Pronouns and Anaphora** Chicago: Chicago Linguistic Society, 1-28.

Beeken, Jeannine (1983) *Generalized phrase structure grawwar: theorie en praktijk* (in Dutch). Leuven: Departement Linguistlek, KUL.

Bissantz, Annette (19B3> The syntactic conditions on *be* reduction in BPSG. In J.F. Richardson, M. Marks, and A. Chukernan (eds.) *Papers from the Parasession on the Interplay o1 Phonology, Morphology, and Syntax.* Chicago; Chicago Linguistic Society, 28-37.

Borsley, Robert (1983a) A note on the Generalized Left Branch Condition. *Linguistic Inquiry* 14, 169-174.

Borsley, Robert <19B3b> A Welsh agreement process and the status of VP and S. In S. Gazdar, E.H. Klein, and 6.K. Pullum (eds.) *Order, Concord and Constituency.* Foris Publications, Dordrecht, 57-74.

Borsley, Robert (1984) On the nonexistence o-f VP's. In W. de Geest fc Y. Putseys (eds.) *International Conference on Sentential Cowplewentation* . Dordrecht; Foris.

Cann, Ronald (1983) An approach to the Latin accusative and infinitive. In G. Gazdar, E.H. Klein, and G.K. Pullum (eds,) *Order, Concord and Constituency.* Foris Publications, Dordrecht, 113-137.

Chunq, Sandra, and James McCloskey (1983) On the interpretation o-f certain island facts in GPSG. *Linguistic Inquiry* 14, 704-713.

Crain, Stephen, and Janet Fodqr (1984) How can grammars help parsers? In D. Dowty, L. Karttunen and A. Zwicky (eds.) *Hatural language processing: psycholinguistic, computational, and theoretical perspectives.* New York; Cambridge University Press.

Culy, Christopher (1983) An extension *oi* phrase structure rules and its application to natural language. MA thesis, Stanford University.

Dahl, Osten (1983) On the nature of bound pronouns. *Papers froi the Institute of Linguistics University o1 Stockholm,* Publication 48.

Dowty, David (1980) Comments on the paper by Bach and Partee. In J. Kreiman & A.E. Ojeda (eds.) *Papers from the Parasession on Pronouns and Anaphora.* Chicago: Chicago Linguistic Society, 29-40.

Dowty, David (1982) More on the categorial analysis of grammatical relations. In A. Zaenen (ed.) *Subjects and Other Subjects: Proceedings of the Harvard Conference on Grammatical Relations.* Bloomington: Indiana University Linguistics Club. Also in *Ohio State University Working Papers in Linguistics* 26, 102-133.

Dowty, David, & Belinda Brodie (1984) A semantic analysis of "floated" quantifiers in GPSG. In *Proceedings of the Third West Coast Conference on Formal Linguistics.* Stanford: Stanford Linguistics Department, nn-nn.

Engdahl, Elisabet (1982a) Constituent questions, topicalization, and surface structure interpretation. In D. Flickinger, M. Macken, and N. Wiegand (eds.) *Proceedings of the First West Coast Conference on Formal Linguistics.* Stanford: Stanford Linguistics Department, 256-267.

Engdahl, Elisabet (1982b) A note on the use of lambda-conversion in generalized phrase structure grammar. *Linguistics and Philosophy* 4, 505-515.

Espinal i Farre, Maria Teresa (1981) The auxiliary in Catalan. MA Dissertation, University of London.

Farkas, Donka, and Almerindo Ojeda (1983) Agreement and coordinate NP's. To appear in *Linguistics.*

Farkas, Donka, Daniel Flickinger, Gerald Gazdar, William Ladusaw, Almerindo Ojeda, Jessie Pinkham, Geoffrey Pullum and Peter Sells (1983) Some revisions to the theory of features and feature instantiation. In *Proceedings of the ICOT Workshop on Non-Transformational Grammars,* 11-13 (Tokyo: Institute for New Generation Computer Technology).

Finer, Daniel (1982) A nontransformational relation between causatives and non-causatives in French. In D. Flickinger, M. Macken, and N. Wiegand (eds.) *Proceedings of the First West Coast Conference on Formal Linguistics.* Stanford: Stanford Linguistics Department, 47-59.

Flickinger, Daniel (1983) Lexical heads and phrasal gaps. In M. Barlow, D. Flickinger, and M. Westcoat (eds.) *Proceedings of the Second West Coast Conference on Formal Linguistics.* Stanford: Stanford Linguistics Department.

Fodor, Janet (1983) Phrase structure parsing and the island constraints. *Linguistics and Philosophy* 6, 163-223.

Gazdar, Gerald (1980a) A cross-categorial semantics for coordination. *Linguistics and Philosophy* 3, 407-409.

Gazdar, Gerald (1980b) A phrase structure syntax for comparative clauses. In T. Hoekstra, H.v.d. Hulst, and M. Moortgat (eds.) *Lexical Grammar.* Foris Publications, Dordrecht, 165-179. Also in *GLOT* 2, 379-393 (1979)

Gazdar, Gerald (1981a) Unbounded dependencies and coordinate structure. *Linguistic Inquiry* 12, 155-184.

Gazdar, Gerald (1981b) On syntactic categories. *Philosophical Transactions (Series B) of the Royal Society* 295, 267-283.

Gazdar, Gerald (1982) Phrase structure grammar. In P. Jacobson & G.K. Pullum (eds.) *The Nature of Syntactic Representation.* D. Reidel, Dordrecht, 131-186.

Gazdar, Gerald, and Geoffrey Pullum (1981) Subcategorization, constituent order and the notion "head". In M. Moortgat, H.v.d. Hulst and T. Hoekstra (eds.) *The Scope of Lexical Rules.* Foris Publications, Dordrecht, 107-123.

Gazdar, Gerald, and Geoffrey Pullum (1982) "Easy to solve". *Linguistic Analysis* 11, 265-267.

Gazdar, Gerald, and Ivan Sag (1981) Passive and reflexives in phrase structure grammar. In J. Groenendijk, T. Janssen, and M. Stokhof (eds.) *Formal Methods in the Study of Language.* Mathematical Centre Tracts, Amsterdam, 131-152.

Gazdar, Gerald, Geoffrey Pullum, and Ivan Sag (1982) Auxiliaries and related phenomena in a restrictive theory of grammar. *Language* 58, 591-638.

Gazdar, Gerald, Geoffrey Pullum, Ivan Sag, and Tom Wasow (1982) Coordination and transformational grammar. *Linguistic Inquiry* 13, 663-676.

Georgopoulos, Carol (1983) Trace and resumptive pronouns in Palauan. In J.F. Richardson, M. Marks, and A. Chukerman (eds.) *Papers from the Parasession on the Interplay of Phonology, Morphology, and Syntax.* Chicago: Chicago Linguistic Society, 134-105.

Gunji, Takao (1981) A phrase structural analysis of the Japanese language. MA dissertation, Ohio State University.

Gunji, Takao (1982) Apparent object control of reflexives in a restrictive theory of grammar. *Papers in Japanese Linguistics* 8, 63-78.

Gunji, Takao (1983a) Generalized phrase structure grammar and Japanese reflexivization. *Linguistics and Philosophy* 6, 115-156.

Gunji, Takao (1983b) Control of gaps and reflexives in Japanese. In *Proceedings of the Second Japanese-Korean Joint Workshop on Formal Grammar,* 151-186 (Logico-Linguistic Society of Japan).

Gunji, Takao (1983c) Topicalization in Japanese. In *Proceedings of the ICOT Workshop on Hon-Transfornational Grawwars,* 21-27 (Tokyo: Institute for New Generation Computer Technology).

Harada, Vasunari (1961) Reduced coordination and trans*ormations, a review of current approaches to semantic regularities. *Linguistic Research, forking Papers in English Linguistics* 1, 64-74 (Tokyo University English Linguistics Association).

Harlow, Stephen (1983) Celtic relatives. *York Papers in Linguistics* 10, 77-121.

Hoekstra, Teun (1981) The base and the lexicon in lexical grammar. In S. Daalder and M. Gerritsen (eds.) *Linguistics in the Netherlands 1981.* Amsterdam: North Holland, 93-102.

Hoekstra, Teun, Harry van der Hulst, and Michael Hoortgat (1980) Introduction to *Lexical Grawwar.* Dordrecht: Foris, 1-48.

Horrocks, Geoffrey (1983) The order of constituents in Modern Greek. In G. Gazdar, E.H. Klein, and G.K. Pulium (eds.) *Order, Concord and Constituency.* Foris Publications, Dordrecht, 95-112.

Horrocks, 6eoffrey (1984) The ECP, X'-theory and the 'pro-drop' parameter. In N. de Geest i V. Putseys (eds.) *International Conference on Sentential Cowplewentation,* Dordrecht: Foris.

Ikeya, Akira (1983) Japanese honorific systems in generalized phrase structure grammar. In *Proceedings of the ICOT Workshop on Non-Transforwational Grawnars,* 17-20 (Tokyo: Istitute for New Generation Computer Technology).

Jacobson, Pauline (1982a) Evidence for gaps. In P. Jacobson & G.K. Pullum (eds.) *The Nature of Syntactic Representation.* D. Reidel, Dordrecht, 187-228.

Jacobson, Pauline (1982b) Visser revisited. *Papers fro* the 18th Regional Meeting of the Chicago Linguistic Society,* 218-243.

Jacobson, Pauline (1983) Connectivity in generalized phrase structure grammar. To appear in *Natural Language and Linguistic Theory.*

Joshi, Aravind (1983) Factoring recursion and dependencies: an aspect of tree-adjoining grammars (TAG) and a comparison of some formal properties of TAGs, GPSGs, PLGs, and LFGs. *Proceedings of the 21st Annual Heeting of the Association for Cowputational Linguistics,* 7-15.

Joshi, Aravind (1984) How auch context-sensitivity is required to provide reasonable structural descriptions: tree adjoining grammars. In D. Dowty, L. Karttunen and A. Zwicky (eds.) *Hatural language processings psycholinguistic, computational, and theoretical perspectives,* New York: Cambridge University Press.

Joshi, Aravind, and Leon Levy (1982) Phrase structure trees bear more fruit than you would have thought. *American Journal of Coaputational Linguistics* 8, 1-11.

Karttunen, Lauri (1981) Unbounded dependencies: slash categories vs. dotted lines. In J. Groenendijk, T. Janssen, and M. Stokhof (eds.) *Formal Methods in the Study of Language.* Mathematical Centre Tracts, Amsterdam, 323-342.

Kay, Martin (1983) When meta-rules are not meta-rules. In K. Sparck-Jones & Y. Wilks (eds.) *Automatic Natural Language Parsing.* Chichester: Ellis Horwood, 94-116. Also: In M. Barlow, D. Flickinger & I.A. Sag (eds.) *Developments in Generalized Phrase Structure Grammar: Stanford Working Papers in Grammatical Theory, Volume 2.* Bloomington: Indiana University Linguistics Club, 69-91.

Klein, Ewan (1980) A semantics for positive and comparative adjectives. *Linguistics and Philosophy* 4, 1-45.

Klein, Ewan (1981a) The interpretation of adjectival, nominal, and adverbial comparatives. In J. Groenendijk, T. Janssen, and M. Stokhof (eds.) *Formal Methods in the Study of Language.* Mathematical Centre Tracts, Amsterdam, 381-398.

Klein, Ewan (1981b) The syntax and semantics of nominal comparatives In M. Moneglia (ed.) *Atti de Seminario su Tempo e Verbale Strutture Quantificate in Forma Logica.* Presso l'Accademia della Crusca, Florence, 223-253.

Klein, Ewan (1982) The interpretation of adjectival comparatives. *Journal of Linguistics* 18, 113-136.

Klein, Ewan (1983) Transduction of discourse representations. *York Papers in Linguistics* 10, 123-145.

Klein, Ewan, and Ivan Sag (1982) Semantic type and control. In M. Barlow, D. Flickinger & I.A. Sag (eds.) *Developments in Generalized Phrase Structure Grammar: Stanford Working Papers in Grammatical Theory, Volume 2.* Bloomington: Indiana University Linguistics Club, 1-25. Also: to appear in *Linguistics and Philosophy* 6.

Konolige, Kurt (1980) Capturing linguistic generalizations with metarules in an annotated phrase-structure grammar. In *Proceedings of the 18th Annual Meeting of the Association for Computational Linguistics,* 43-48.

Maclaran, Rose (1982) The semantics and pragmatics of the English demonstratives. PhD dissertation, Cornell University.

Maling, Joan, and Annie Zaenen (1982) A phrase structure account of Scandinavian extraction phenomena. In P. Jacobson & G.K. Pullum (eds.) *The Nature of Syntactic Representation.* D. Reidel, Dordrecht, 229-282.

Monzon, Christina (1979) A constituent stbucture rule grammar of the Spanish clitic positioning in complex and simple sentences. MA Dissertation, University of Texas, Austin.

Moortgat, Michael (1981) Subcategorization and the notion 'lexical head'. In S. Daalder and M. Gerritsen (eds.) *Linguistics in the Netherlands 1981.* Amsterdam: North Holland, 45-54.

Hoortgat, Michael (19S4) A Fregean restriction on metarules. In *Proceedings of the Fourteenth Annual Heeting of the Horth Eastern Linguistic Society.*

Napoli, D.J. (1963) Comparative ellipsis: a phrase structure analysis. *Linguistic Inquiry* 14, 675-694.

Nerbonne, John (1983) Temporalia and strict lexicalism. In J.F. Richardson, M. Marks, and A. Chukerman (eds.) *Papers fro* the Parasession on the Interplay of Phonology, tiorphology, and Syntax.* Chicago: Chicago Linguistic Society, 162-172.

Nozawa, Hideii (1984) A note on generalized phrase structure grammar. *Journal of the Faculty of Foreign Studies* 17, 45-78, Aichi Prefectural University.

Partee, Barbara, and Emmon Bach (1981) Quantification, pronouns, and VP anaphora. In J. Groenendijk, T. Janssen, and M. Stokhof (eds.) *Forwal Hethods in the Study of Language.* Mathematical Centre Tracts, Amsterdam, 445-481.

Partee, Barbara, and Mats Rooth (1983) Generalized conjunction and type ambiguity. To appear in Ch. Schwartze (ed.) *Heaning, Use, and Interpretation of Language.* Berlin: de Gruyter.

Pollard, Carl and Ivan Sag (1983) Reflexives and reciprocals in English: an alternative to the binding theory. In M. Barlow, D. Flickinger, and M. Westcoat (eds.) *Proceedings of the Second Hest Coast Conference on Formal Linguistics.* Stanford: Stanford Linguistics Department.

Pulluffl, Geoffrey (1982) Free word order and phrase structure rules. In James Pustejovsky and Peter Sells (eds.) *Proceedings of the Twelfth Annual Heeting of the Horth Eastern Linguistic Society,* 209-220. Graduate Linguistics Student Association, University of Massachusetts, Amherst, Mass.

Pullum, Geoffrey (1983a) Context-freeness and the computer processing of human languages. *Proceedings of the 21st Annual Heeting of the Association for Cowputational Linguistics,* 1-6.

Pullum, Geoffrey and Gerald Gazdar (1982) Natural languages and context free languages. *Linguistics and Philosophy* 4, 471-504.

Richardson, John (1982) Constituency and sublexical syntax. *Papers fro* the 18th Regional Heeting of the Chicago Linguistic Society,* 466-476.

Rooth, Mats, and Barbara Partee (1982) Conjunction, type ambiguity, and wide scope "or". In D. Flickinger, M. Macken, and N. Wiegand (eds.) *Proceedings of the First Hest Coast Conference on Portal Linguistics.* Stanford: Stanford Linguistics Department, 353-362.

Ross, Kenneth (1981) Parsing English phrase structure. PhD dissertation, University of Massachusetts at Amherst.

Sag, Ivan (1982a) A semantic theory of "NP-movement" dependencies. In P. Jacobson & G.K. Pullum (eds.) *The Nature of Syntactic Representation.* D. Reidel, Dordrecht, 427-466.

Sag, Ivan (1982b) Coordination, extraction, and generalized phrase structure. *Linguistic Inquiry* 13, 329-336.

Sag, Ivan (1983) On parasitic gaps. *Linguistics and Philosophy* 6, 35-45. Also in In D. Flickinger, M. Macken, and N. Wiegand (eds.) *Proceedings of the First West Coast Conference on Formal Linguistics.* Stanford: Stanford Linguistics Department, 35-46 (1982).

Sag, Ivan, and Ewan Klein (1982) The syntax and semantics of English expletive pronoun constructions. In M. Barlow, D. Flickinger & I.A. Sag (eds.) *Developments in Generalized Phrase Structure Grammar: Stanford Working Papers in Grammatical Theory, Volume 2.* Bloomington: Indiana University Linguistics Club, 92-136

Saito, Mamoru (1980) An analysis of the *tough* construction in Japanese. MA Dissertation, Stanford University.

Sampson, Geoffrey (1983) Context-free parsing and the adequacy of context-free grammars. In Margaret King (ed.) *Parsing natural language.* London: Academic Press, 151-170.

Schachter, Paul, and Susan Mordechay (1983) A phrase structure account of "nonconstituent" coordination. In M. Barlow, D. Flickinger, and M. Westcoat (eds.) *Proceedings of the Second West Coast Conference on Formal Linguistics.* Stanford: Stanford Linguistics Department.

Sells, Peter (1983) Relative clauses in Irish and Welsh. *York Papers in Linguistics* 10, 159-172.

Stucky, Susan (1981a) Free word order languages, free constituent order languages, and the gray area in between. In V. A. Burke and J. Pustejovsky (eds.) *Proceedings of the 11th Annual Meeting of the North Eastern Linguistic Society,* Department of Linguistics, University of Massachusetts, Amherst.

Stucky, Susan (1981b) Word order variation in Makua: a phrase structure grammar analysis. PhD Dissertation, University of Illinois at Urbana-Champaign.

Stucky, Susan (1982) Linearization rules and typology. In D. Flickinger, M. Macken, and N. Wiegand (eds.) *Proceedings of the First West Coast Conference on Formal Linguistics.* Stanford: Stanford Linguistics Department, 60-70.

Stucky, Susan (1983) Verb phrase constituency and linear order in Makua. In G. Gazdar, E.H. Klein, and G.K. Pullum (eds.) *Order, Concord and Constituency.* Foris Publications, Dordrecht, 75-94.

Udo, Mariko (1982) The Japanese VP system. MA thesis, University College London.

Uszkoreit, Hans (1982) German word order in GPSG. In D. Flickinger, M. Macken, and N. Wiegand (eds.) *Proceedings of the First West Coast Conference on Formal Linguistics.* Stanford: Stanford Linguistics Department, 137-148.

Uszkoreit, Hans (1983) A framework for processing partially free word order. *Proceedings of the 21st Annual Meeting of the Association for Computational Linguistics,* 106-112.

Wasow, Tom, Ivan Sag, and Geoffrey Nunberg (1982) Idioms: an interim report. In *Preprints of the Plenary Session Papers, The XIIIth International Congress of Linguists,* August 29 - September 4, 1982. Tokyo: CIPL.

Weeda, Donald (1981) Tenseless that-clauses in generalized phrase structure grammar. *Papers from the Seventeenth Regional Meeting of the Chicago Linguistic Society,* 404-410.