Technical Report

UCAM-CL-TR-127
ISSN 1476-2986

Number 127

UNIVERSITY OF CAMBRIDGE
Computer Laboratory

A development environment for large natural language grammars

John Carroll, Bran Boguraev, Claire Grover,
Ted Briscoe

February 1988

15 JJ Thomson Avenue
Cambridge CB3 0FD
United Kingdom
phone +44 1223 763500

https://www.cl.cam.ac.uk/

Technical reports published by the University of Cambridge
Computer Laboratory are freely available via the Internet:

https://www.cl.cam.ac.uk/techreports/

ISSN 1476-2986

A Development Environment for Large Natural Language Grammars

John Carroll, Bran Boguraev

Computer Laboratory, University of Cambridge
Pembroke Street, Cambridge CB2 3QG, England

Claire Grover, Ted Briscoe

Department of Linguistics, University of Lancaster
Bailrigg, Lancaster LA1 4YT, England

February 1988

The Grammar Development Environment (GDE) is a powerful software tool designed to help a linguist or grammarian experiment with and develop large Natural Language grammars. (It is also, however, being used to help teach students on courses in Computational Linguistics). This report describes the grammatical formalism employed by the GDE, and contains detailed instructions on how to use the system.

Source code for a Common Lisp version of the software is available from the University of Edinburgh Artificial Intelligence Applications Institute[1].

[1] This work was supported by research grant GR/D/05554 from the U.K. Science and Engineering Research Council under the Alvey Information Technology initiative.

Contents

1. Introduction

The Grammar Development Environment (GDE) is a software system which supports a linguist or grammarian during the process of developing a Natural Language grammar. It provides facilities for defining and editing rules written in a metagrammatical formalism, and for building a lexicon which is compatible with the grammar. A parser, a generator and tools for inspecting the grammar from a number of viewpoints help the user to test and debug the grammar. Although the tools provided by the GDE are necessarily quite diverse, they are fully integrated and are accessed through a consistent and easy to learn set of commands. Section 3 describes the commands available, and sections 4 to 6 go into more detail about three of the major components: the parser, the generator, and the morphological analyser.

The metagrammatical formalism is similar to Generalized Phrase Structure Grammar (GPSG) (Gazdar *et al*, 1985), although it is interpreted somewhat differently. The GDE compiles the rules in the metagrammar into an 'object' grammar which is a unification grammar. An object grammar of this type is usually quite large and difficult to understand: specifying it in terms of the various rule types of the metagrammar is much more economical, understandable, and makes syntactic generalisations easy to express. Section 2 specifies the formalism in detail, but the example session with the GDE given below should give a flavour of it, and of the type of interaction which takes place when developing a grammar.

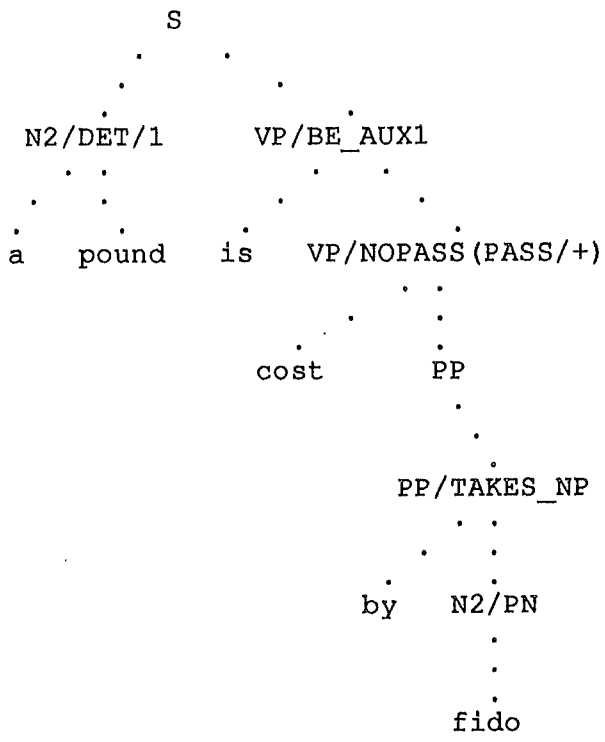
1.1 An Example GDE Session

The following is an extract from an actual GDE session, showing how the GDE helps a user to specify and debug a grammar. The details of the interaction are not important, as they will be explained in full later on in this document. The user first reads in a file containing an existing (small) grammar[1], and tries to parse a couple of sentences. (User input is **in this style**).

```
Gde> read gram.rules
File read

Gde> parse
Compiling grammar...
17 ID rules, 1 metarules, 9 propagation rules, 4 default rules
*** Warning, multiple match between VP/TAKES_2NP and PASS
31 expanded ID rules, 32 linearised rules

Parse>> fido costs a pound

80 msec CPU, 1000 msec elapsed, 367 conses
15 edges generated
1 parse

((fido) (costs (a pound)))

Parse>> a pound is cost by fido

140 msec CPU, 1000 msec elapsed, 756 conses
32 edges generated
1 parse

((a pound) (is (cost ((by (fido))))))
```

[1] Appendix 1 contains a listing of the grammar used in this example session.

```
Parse>> view rules
```



The second sentence should not have received a parse, so there must be a bug in the grammar. Viewing the parse tree immediately suggests where the problem lies: a rule called VP/NOPASS(PASS/+) appears at the place where the parse should have failed. The name of the rule also suggests that something is wrong, since it says that the rule was formed from the metarule PASS applied to the ID rule VP/NOPASS. The user goes on to exit the parser, examine the ID rule and metarule involved, and restrict the applicability of the latter.

```
Parse>> q

Gde > view id *NOPASS

VP/NOPASS : VP --> H[SUBCAT NOPASS], N2[+PRD].

Gde> view metarule PASS

PASS : VP --> W, N2. ==> VP[Pas] --> W, (P2[PFORM BY]).

Gde> input
Construct type? meta
Metarule declaration? PASS :
> VP --> W, N2[-PRD]. ==>
>   VP[Pas] --> W, (P2[PFORM BY]).

Replace existing definition (y/n)? y

Gde> names id *(PASS)

VP/TAKES_NP(PASS/-)  VP/TAKES_NP(PASS/+)  VP/DITR(PASS/-)
VP/DITR(PASS/+)      VP/TAKES_2NP(PASS/-) VP/TAKES_2NP(PASS/+)
VP/OR(PASS/-)        VP/OR(PASS/+)
```

The VP/NOPASS ID rule no longer appears in the list of rules resulting from the updated PASS metarule; the rules that do appear are the expected ones. Carrying on, an attempt to parse the last

sentence indeed fails as it should.

```
Gde> p
Compiling grammar...
17 ID rules, 1 metarules, 9 propagation rules, 4 default rules
25 expanded ID rules, 26 linearised rules

Parse>> previous
(a pound is cost by fido)

80 msec CPU, 1000 msec elapsed, 478 conses
21 edges generated
No parses

Parse>> q

Gde> generator N2

Gen>> sentence s 2
a dog
dog a
kim

Gen>> q

Gde> write gram.rules
Backing up file gram.rules
Writing file gram.rules
```

An exhaustive generation of all noun phrase structures licensed by the grammar indicates that the rule introducing determiners may be overgenerating, but the user decides to ignore this for the time being, and write the changed grammar back to disk. The GDE first saves the existing version of the file in case the user later wants to refer back to it. Appendix 5 contains an introductory guide to the GDE, with brief descriptions of some of the most basic commands (such as those used in this example), and a selective overview of the metagrammar formalism.

1.2 Background

The GDE was written to support the development of a large grammar of English (Briscoe *et al.*, 1987a; Grover *et al.*, 1988), one of the Alvey Natural Language Tools projects. (Briscoe *et al.* (1987b) give a summary of this project). The other collaborating projects have implemented a GPSG parser (Phillips and Thompson, 1987) and a dictionary and analyser system (Russell *et al.*, 1986); the analyser and the parser form part of the GDE. Although it is possible to use these two subsystems separately from the rest of the GDE by calling them directly through LISP functions, this mode of use will not be described here.

The morphological analyser and parser were originally written in the Franz Lisp dialect, and the GDE in Cambridge Lisp. All three components have since been ported to Common Lisp. Those implementations of Common Lisp in which they have been tested are listed in appendix 4, along with timings for performing typical tasks.

This report describes version 1.20 of the GDE. A magnetic tape containing the Common Lisp source code of this version of the GDE, together with the morphological analyser and parser, is available to U.K. Universities and to commercial participants of Alvey projects, and is distributed by the University of Edinburgh Artificial Intelligence Applications Institute (80 South Bridge, Edinburgh, U.K.).

2. The Metagrammatical Formalism

This section defines the metagrammatical formalism: the syntax used to declare the various types of rule in the formalism, and the way in which the formalism is interpreted. The formalism is similar to GPSG (Gazdar *et al*, 1985). There are, however, a few differences, motivated by a desire for more expressiveness and flexibility. Thus for example the user may bypass ID/LP format by including pure phrase structure rules in the metagrammar, and may define different feature passing conventions from GPSG by writing rules which explicitly state propagation regimes. As mentioned previously, the formalism is interpreted differently from GPSG. In GPSG, rules are defined declaratively as applying simultaneously in the projection from Immediate Dominance (ID) rules to local trees. The concept of simultaneous application is conceptually rather difficult. The interpretation of a GDE metagrammar is easier, however, since a well-defined, temporally ordered expansion procedure (described in section 2.14 below) is used to compile the metagrammar into an 'object' grammar, a simple unification grammar.

Although it is possible to define a grammar directly at the object grammar level, the metagrammatical formalism contains several types of rule to help the grammar writer capture linguistic generalisations. A metagrammar may contain any number of each type of rule, and each rule is defined to the GDE in a declaration.

The rest of this section describes the rule types in the formalism, for each giving a BNF specification of the syntax expected for their declarations. The following conventions apply in the specifications:

(1) A vertical bar ('|') is used to separate alternatives on the right hand side (RHS) of a BNF production. Items in parentheses ('(' ')') are optional.

(2) Terminals are underlined.

(3) Non-terminals are enclosed in angle brackets ('<' '>'). A non-terminal may be repeated one or more times if it is followed by a '+' and zero or more times if followed by a '*'.

(4) All items generated by the '+' and '*' iteration indicators should be separated by commas although this is not explicitly expressed in the specifications; in some cases separation by just spaces changes the meaning.

(5) Comments (introduced by ';' and terminating at the end of the line) may appear between any two terminal symbols.

Spaces, newlines and other layout characters in declarations are ignored. However, the casing of feature and rule names, feature values etc. matters, so that for example, a feature with name n would be treated as being distinct from one with name N.

2.1 Feature Declarations

Feature Declarations define the feature system used in the grammar. Each such declaration enumerates the values a specified feature may have. The feature system supported by the GDE is very similar to that assumed by several contemporary grammatical theories, but extends some of them, such as GPSG, in allowing features to take a variable value. The variable value ranges over the set of actual values as declared. Features are used to form categories, a category being an unordered collection of features, each feature in the category having a value. The BNF description of the syntax of a feature declaration is:

<feature-declaration> ::= <feature-name> { <feature-value>+ } |
<feature-name> CAT
<feature-name> ::= <atomic-symbol>
<feature-value> ::= <atomic-symbol>

Variable values need not be declared in the list of possible values a feature may have. Declaring a feature as CAT indicates that the feature, if it does not have a variable value, will have a category as its value; the value may never be an ordinary atomic value. A feature that is not category-valued may be declared as having any number (strictly greater than zero) of possible values, but if elsewhere in the grammar the feature appears with a value that is not in the list of possible values for the feature, the GDE will report an error. The example feature declarations below state that the feature BAR will always have one of the values 0, 1 and 2, and that AGR is a category valued feature.

```
BAR {0, 1, 2}
AGR CAT
```

2.2 Set Declarations

Feature Set Declarations define groups of features which behave in the same manner with respect to feature value defaulting, feature value propagation and so on. In rules which perform these functions the name of the set may be used as a more readable way of referring to the whole collection of features.

<set-declaration> ::= <set-name> = { <feature-name>+ }
<set-name> ::= <atomic-symbol>
<feature-name> ::= <atomic-symbol>

For example, the features PLU, PER and CASE could be grouped together in the set NOMINALHEAD. This would be expressed by:

```
NOMINALHEAD = {PLU, PER, CASE}
```

2.3 Alias Declarations

Aliases are another convenient abbreviatory device. They may be used to name categories and feature complexes, and used in rules to avoid having to write out in full all the feature / value pairs in a category.

<alias-declaration> ::= <alias-name> = <category> .
<alias-name> ::= <atomic-symbol>

See section 2.13 for the definition of <category>. For example, the two alias declarations below would allow the category [N +, V -, BAR 2, PLU +, PER 3] to be written as N2[+PLU, PER 3].

```
N2 = [N +, V -, BAR 2].
+PLU = [PLU +].
```

2.4 Category and Lexical Category Declarations

Category Declarations and Lexical Category Declarations define a particular category as consisting of a given set of features. These declarations are used to flesh out into more fully specified categories the partially specified categories which typically appear in ID rules and the

definitions of words. When a category declaration is applicable to a category forming part of an ID rule or word definition, those features in the category declaration which are not present in the ID rule or word definition are added to it with a variable value.

<category-declaration> ::= <category-name> :
    (<feature-name>) <pattern-category> => <feature-set>
<category-name> ::= <atomic-symbol>
<feature-set> ::= { <feature-name>+ } | <set-name>
<feature-name> ::= <atomic-symbol>
<set-name> ::= <atomic-symbol>

See section 2.13 for the definition of <pattern-category>. Category declarations without the optional feature name ensure that top level categories which match the given pattern category contain the set of features specified. When the optional feature at the beginning of the declaration is present, the procedure is applied to all categories which are the value of this feature. Ordinary category declarations apply only to categories in ID rules: lexical category declarations are identical in form, but apply to word definitions (section 2.12) as well as ID rules.

Set names may be used on the RHS of category declarations, as in the first example below. In the second, if AGR is a category valued feature, then the N2 category inside a category such as V2[AGR N2], will have the features PER and PLU added to it.

```
VAR_NOUN : [N +, V -] => NOMINALHEAD
AGR_N2 : AGR N2 => {PER, PLU}
```

2.5 Extension Declarations

Some features, such as SLASH, are not part of the 'basic' make-up of a category (in the way that, for instance, the feature PER might be in nominal categories). These 'extension' features, which will therefore not appear in any category declaration, may be declared as such using the Extension Declaration. Doing so does not affect the form of the compiled grammar, but acts mainly as a convenient reminder of feature usage for the grammar writer.

<extension-declaration> ::= { <feature-name>+ } | <set-name>
<feature-name> ::= <atomic-symbol>
<set-name> ::= <atomic-symbol>

When an extension declaration is input, the GDE checks that every feature not in the extension set is in at least one category declaration. A warning is printed if any feature fails this test. For example,

```
{WH, SLASH}
```

declares WH and SLASH as extension features, and these two features are now not expected to appear in a category declaration (although it is not an error if they do).

2.6 Immediate Dominance Rule Declarations

Immediate Dominance (ID) rules encode permissable dominance relations in phrase structure rules. Dominance is all they encode; other properties of phrase structure rules (such as the ordering of the categories in them) are determined by other types of rule in the grammar.

<idrule-declaration> ::= <idrule-name> : <category> --> <rhs-term>+ .
<idrule-name> ::= <atomic-symbol>
<rhs-term> ::= <category> | (<category>) | (<category>)+ |
    (<category>)*

See section 2.13 for the definition of <category>. If the RHS categories in the rule are separated by commas, then the rule will later be subject to linear precedence rules; if the categories are separated by just spaces the rule is taken to be already linearised (and treated as a pure Phrase Structure rule, section 2.7). The first ID rule below states that a verb phrase may consist of a verb subcategorised for NP, and a noun phrase. The second rule contains an optional prepositional phrase.

```
VP/TAKES_NP : VP --> H[SUBCAT NP], N2.
VP/SSR : VP --> H[SUBCAT SSR], ( P2[to] ), VP[TO].
```

2.7 Phrase Structure Rules

Phrase Structure (PS) rules are similar in form to ID rules, except that the commas between categories should be omitted. PS rules would typically be employed where it was wished to bypass LP rules, for example in the following rule encoding 'heavy movement' in English.

```
Heavy_NP_Shift : VP --> H[SUBCAT NP_PP] PP NP[+Heavy].
```

2.8 Propagation Rule Declarations

Propagation rules define how features propagate between mother and daughter categories and between two or more daughter categories in ID and PS rules. The effect of propagation rules is to bind variables, instantiate values of features, or add new features with variable values to rules in the 'object' grammar. Propagation rules can be used to encode particular feature propagation principles, such as the various versions of the Head Feature Convention proposed for GPSG.

```
<proprule-declaration> ::= <proprule-name> :
      <pattern-rule> <value-restrictions> (, F in <set-restriction>) .
<value-restrictions> ::= <value-restriction> (= <value-restriction>)+
<value-restriction> ::= F [ <category-index> ] |
      <feature-name> [ <category-index> ]
<category-index> ::= <integer> | <integer> [ <feature-name> ]
<set-restriction> ::= [ <feature-name>+ ] | <set-name>
<feature-name> ::= <atomic-symbol>
<set-name> ::= <atomic-symbol>
```

See section 2.13 below for the definition of <pattern-rule>. The category-index in a feature value restriction indexes the categories in the first part of the propagation rule, so that an index of 0 refers to the rule mother, 1 refers to the daughter that appears first in the propagation rule declaration, and so on. Thus in the example propagation rule below, the 0 refers to the [N +, V -] category, and the 1 refers to the [H +] one. (The meaning of the U metavariable is described below in section 2.10).

```
HFC_NOMINAL :
   [N +, V -] --> [H +], U. F(0) = F(1), F in NOMINALHEAD.
```

If the rule pattern part of a propagation rule matches an ID rule, the value restrictions are applied for each feature in the set restriction. The value restrictions are considered in order and the value of each feature in each of the indexed categories in the ID rule is taken to be either the value of that feature in the first category for which is specified, or a variable value if the feature is specified in none of the categories. For example, the pattern part of the HFC_NOMINAL propagation rule above matches the ID rule

```
N2/DET : N2 --> DetN, H[SUBCAT NULL].
```

and applying the propagation rule would result in each feature in the set NOMINALHEAD being added with the same variable value to the N2 mother and the head daughter.

2.9 Default Rule Declarations

Default rules allow the grammar writer to assign default values for specified features in a particular ID (or PS) rule environment. A default rule will, however, have no effect if the specified features already have values (assigned perhaps initially in the original ID rule definition, or subsequently as a result of the application of a propagation rule or another default rule).

```
<defrule-declaration> ::= <defrule-name> :
      <pattern-rule> <value-assignment> (, F in <set-restriction>) .
<value-assignment> ::= F ( <category-index> ) = <feature-value> |
      <feature-name> ( <category-index> ) = <feature-value>
<category-index> ::= <integer> | <integer> [ <feature-name> ]
<set-restriction> ::= { <feature-name>+ } | <set-name>
<feature-name> ::= <atomic-symbol>
<feature-value> ::= <atomic-symbol> | @
<set-name> ::= <atomic-symbol>
```

See section 2.13 for the definition of <pattern-rule>. As in propagation rules, the category-index in a feature value assignment indexes the categories in the first part of the default rule. The action of default rules is also somewhat similar to that of propagation rules, the difference being that default rules assign default (usually non-variable) values to one or more features in a specified ID rule category, whereas propagation rules tie together the values of features in two or more specified categories.

After the pattern part of a default rule has matched an ID rule, each feature in the value assignment is added with the given value to the indexed category if it is not yet specified there or has only a variable value. A feature in a category which is the value of a category valued feature may be defaulted by subscripting the numeric index with, in square brackets, the name of the category valued feature. This is illustrated in the second of the default rules below, where the features in AGRFEATS are added with variable values to the (N2) category which is the value of the SLASH feature in the S daughter.

```
RHS_N2_POSS : [] --> N2, U. POSS(1) = -.
SLASH_N2A :
   S --> S[H +, SLASH N2], U. F(1[SLASH]) = @, F in AGRFEATS.
```

2.10 Metarule Declarations

Metarules are a principled way of automatically and systematically enlarging the object grammar on the basis of the set of ID and PS rules initially produced by the grammar writer. A metarule consists of, on the LHS a pattern, and on the RHS the skeleton of a new rule; for every existing ID or PS rule that matches the metarule pattern, a new rule based on the skeleton is added to the object grammar.

```
<metarule-declaration> ::=
      <metarule-name> : <pattern-rule> ==> <metarule-rhs> .
<metarule-name> ::= <atomic-symbol>
<metarule-rhs> ::= <category> --> <rhs-term>+ .
<rhs-term> ::= <category> | W | U | ( <category> ) |
      ( <category> )+ | ( <category> )*
```

See section 2.13 for the definitions of <pattern-rule> and <category>. The categories in the rule may be separated by just spaces, rather than commas, and this signifies that the rule is to be applied only to PS rules and to already linearised ID rules. The W and U metavariables match zero or more rule categories, the W category variable marking the metarule as only being applicable to lexical ID rules, and the U variable marking it as being unrestricted. If neither of the W or the U variables appears in a metarule, then the rule is assumed to be unrestricted. The

lexical / non-lexical distinction as applied here is intended only for GPSG-type grammars; a lexical category in this context is taken to be one that is BAR 0 and specified for the feature SUBCAT.

The precise operation of metarule application is best illustrated with an example. If a metarule called PASS (for deriving passive verb phrases from active ones) is defined as

```
PASS : VP --> W, N2. ==> VP[PAS] --> W, ( P2[by] ).
```

then it will match the ID rule

```
VP/TAKES_NP : VP --> H[SUBCAT NP], N2.
```

The correspondences between the ID rule and the LHS of the metarule are worked out (in this case mother VP with VP, and daughters N2 with N2, H[SUBCAT NP] with W), and a new ID rule is built, each category being the combination of the corresponding input ID rule and RHS metarule categories. The combination operation is similar to unification, with the difference that if a feature occurs with different values in the two categories, the value of the feature in the metarule category takes precedence. Thus the mother of the new ID rule will be VP[PAS] (the unification of VP[PAS] with the VP in the ID rule), and the daughters will be H[SUBCAT NP] (the W metavariable remains unchanged) and an optional P2[by] (which is a new category added by the metarule). The original ID rule N2 category does not appear in the new ID rule since it does not appear on the metarule RHS. Thus the new ID rule is:

```
VP/TAKES_NP(PASS) : VP[PAS] --> H[SUBCAT NP], ( P2[by] ).
```

2.11 Linear Precedence Rule Declarations

Linear Precedence (LP) rules specify permissible precedence relations among daughter categories in ID rules.

<lprule-declaration> ::=
    <lprule-name> : <pattern-category> (< <pattern-category>)+ .
<lprule-name> ::= <atomic-symbol>

See section 2.13 below for the definition of <pattern-category>. Thus the first LP rule in the following examples

```
L1 : [SUBCAT] < [~SUBCAT].
L2 : [N +] < P2 < V2.
L3 : [CONJ (both, either, neither, NULL)] <
        [CONJ (and, but, nor, or)].
```

says that an ID rule category that is specified for the feature SUBCAT will always occur before one that is not so specified.

2.12 Word Declarations

Words are not part of the metagrammatical formalism, but may be defined to help use the parser and the generator in the GDE to test a grammar under development. A word declaration consists of the word, followed by one or more syntactic categories to be associated with the word.

<word-definition> ::= <word> : <category>+ .
<word> ::= <atomic-symbol>

See section 2.13 for the syntax of <category>. For example, the plural and possessive forms of

the noun `cat` (ignoring the effects of apostrophes) could be defined as

```
cats : N[-POSS, PLU +, PRO -, PN -, SUBCAT NULL],
       N[+POSS, PRO -, PN -, SUBCAT NULL].
```

It is often not necessary to specify feature / value pairs where the value is a variable, since when the definition of a word is retrieved for use by the GDE parser or generator, lexical category rules (section 2.4) are applied to the definition of the word to flesh out the categories in it.

2.13 Rule Patterns and Grammatical Categories

<pattern-rule> ::= <pattern-category> --> <rhs-pattern-item>+ .
<rhs-pattern-item> ::= <pattern-category> | W | U |
(<pattern-category>) | (<pattern-category>)+ |
(<category>)*

<pattern-category> ::=
<p-feature-bundle> | <alias-name> (<p-feature-bundle>)
<p-feature-bundle> ::= [<p-feature-specification>*]
<p-feature-specification> ::=
<feature-name> <p-feature-value> |
<feature-name> (<p-feature-value>+)
<feature-name> | ~ <feature-name> | <alias-name>
<p-feature-value> ::=
<atomic-symbol> | <pattern-category> | @ | @ <atomic-symbol>

<category> ::= <feature-bundle> | <alias-name> (<feature-bundle>)
<feature-bundle> ::= [<feature-specification>*]
<feature-specification> ::=
<feature-name> <feature-value> | <alias-name>
<feature-value> ::= <atomic-symbol> | <category> | @ <atomic-symbol>

<feature-name> ::= <atomic-symbol>
<alias-name> ::= <atomic-symbol>

Pattern rules occur in propagation rules, default rules, and as the LHS of metarules. Pattern categories may occur as part of pattern rules, and also in category declarations and LP rules. Pattern categories are like ordinary categories except that they may also include feature names with a list of possible values (e.g. `[CONJ (and, but, nor, or)]`), feature names with values which may only match variables (e.g. `[N @]`), feature names with unspecified values (e.g. `[N]`) which successfully match that feature with any non-variable value, and also feature names specified to be not present (e.g. [~N]). Examples of ordinary categories are

```
[BAR 2, +N, -V]
N2
X2[SLASH N2, PER @x]
```

and of pattern categories are

```
[~N, ~V]
N2[SLASH]
```

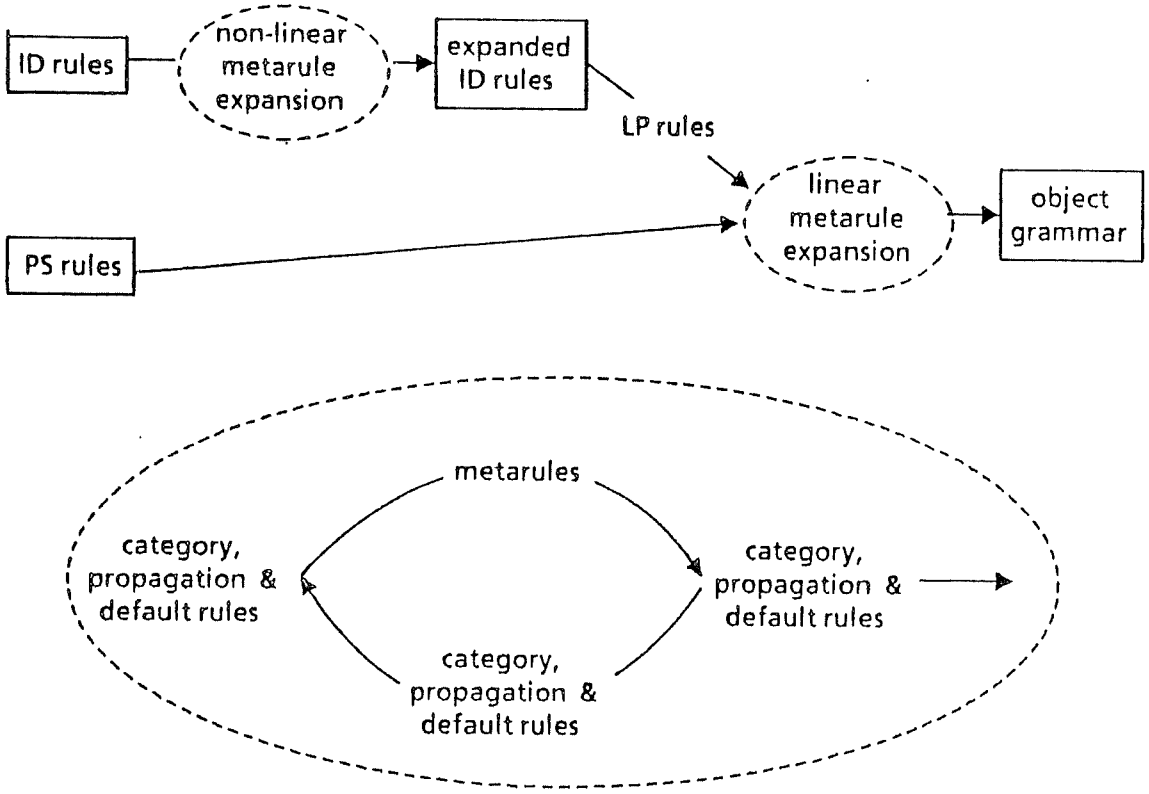
Feature values beginning with `@` are special values which behave as variables. Propagation rules may tie their value to that of other features, default rules may give them a definite value, and in the parser and generator they are allowed to unify with any value. They are treated slightly differently from proper values in the process of matching against pattern categories: for instance the pattern `[PER]` (the feature `PER` with an unspecified value) will not match a category such as the `X2` category in the example above which contains `PER` with a variable value; the feature

specification must include a 'proper' value for the match to succeed.

It is rather unfortunate that inside a feature bundle a feature with an unspecified value is notated in exactly the same way as an alias. Any ambiguity (where a symbol is both the name of a feature and of an alias) is decided in favour of the interpretation containing the alias. The GDE allows the user to view the category after any aliases that were present have been expanded out into feature bundles: doing this will show what course has been taken.

2.14 The Metagrammar Compilation Procedure

The object grammar is produced from the metagrammar by first 'normalising' it, that is, expanding out all aliases and sets in it into feature bundles and lists of features respectively, and splitting each ID and PS rule which contains optional categories into two rules, one with the optional category and one without. The next stage is the application first of propagation rules, followed by default rules and category declarations, to ID rules, and then the application of the non-linear metarules one-by-one in order to the set of fleshed out ID rules. (Thus the order in which the metarules were declared is significant). After each metarule has applied, propagation, default and category rules are applied to any new ID rules and these are added to the original set before the next metarule is applied. The resulting expanded set of rules is then linearised according to the LP rules, merged with any PS rules that may have been defined, and finally linear metarules are applied to the complete pool of PS rules (again with feature propagation, defaulting and fleshing out of categories at all stages). This process is summarised below:



Propagation rules are normally applied before default rules, but this behaviour may be changed in the GDE by altering the value of a flag (section 3.4). Two other aspects of metagrammar expansion behaviour may also be controlled by flags. The first is whether multiple identical metarule expansions of the same ID rule are reduced to just one. Multiple identical expansions may occur if a metarule matches an ID rule in more than one way; if this happens a

warning message is printed out. For example, the PASS metarule of section 2.10,

```
PASS : VP --> W, N2. ==> VP[PAS] --> W, ( P2[by] ).
```

would match an ID rule introducing ditransitive verbs, such as

```
VP/DITR : VP --> H[SUBCAT DITR], N2, N2.
```

in two ways: one way with the W metavariable covering the head and one of the noun phrases, and the other way in which it covers the head and the other noun phrase. If the flag were set to OFF, then the metarule would produce two identical output rules. This behaviour is probably not what is desired (since the parser would then give duplicate parses), so this flag initially has the value ON.

The second, similar aspect of behaviour is whether multiple identical linearisations are reduced to just one. Multiple linearisations occur if an ID rule contains two categories which are identical (the VP/DITR rule above for instance). A warning message is printed out in this case, and also in the case where the LP rules acting together completely block all linearisations of a particular ID rule.

3. Commands

Section 1.1 presented a typical GDE session in which a few of the more basic GDE commands were issued. This section describes all of the commands accepted by the GDE, with examples showing a few of the ways in which they might be used.

The general style of interaction is for the GDE to print a prompt when it is waiting for a command or other input from the user, the user to type something followed by a carriage return, and then the GDE to perform an appropriate action or prompt for more information. The user may abandon a command when the GDE is is prompting for more input by not giving any, and just typing a carriage return on its own. All commands and options may be abbreviated, usually to their first letters, but in some cases to the first two or three; just enough to disambiguate the command name or option from other alternatives. Commands may be typed in either upper or lower case (or even a mixture). Casing, however, does matter for the names and bodies of declarations.

Commands prompt the user for more information if they need it; the expert user may bypass the extra interaction involved by typing all the information he or she knows a command will need on the same line. However, if there is not enough, if some of it was incorrect, or if the system wants the answer to an important question (such as whether to delete a declaration) the command will issue a prompt, ignoring the extra input.

Several of the commands take an option representing the type of declaration or construct they are to act on. This can be one of (minimum abbreviations in upper case): "COMment", "FEature", "Set", "Alias", "CAtegory", "Lexical Category", "EXtension", "Idrule", "PSrule", "PRoprule", "Defrule", "MEtarule", "LPrule", or "Word". The "comment" construct allows a comment block to be associated with each file which contains grammar declarations and definitions. More specific comments may be put inside individual definitions. The same commands which act on metagrammatical constructs may also be used to manipulate the morphology system (section 6) constructs "ENtry completion rule", "MUltiplication rule", and "CONsistency check".

Every declaration or rule in a grammar has a name, that given to it by the grammar writer when the declaration was defined. Thus an ID rule introducing proper names might be called `N2/PN`, and be declared as

```
N2/PN : N2 --> H[SUBCAT NULL, PN +].
```

Declarations have names so that each declaration may conveniently be referred to in GDE commands. Some commands accept a pattern as a way of specifying a list of several construct names. Patterns are like normal names, except that they may include "?", meaning that any character is allowable at that position, and "*", meaning that any sequence of zero or more characters is allowable. Patterns which start with the character "=" select those constructs which actually contain the item following the "=" in their bodies. Specifying "altered" as the pattern has the special meaning of the names of all the constructs that have been changed since they were last saved to a file on disk.

More than one such pattern may be conjoined by '&', subsequent patterns filtering the results of previous ones. E.g.

```
=AGR & S*
```

specifies those declarations containing `AGR` whose names begin with `S`, and

```
=N2 & altered
```

specifies those which contain `N2` and which have been altered. Patterns preceded by '=' may

include the '*' and '?' characters, and the patterns behave in a manner analogous to when these characters are included in construct names.

The rest of this section describes the commands which are available in the GDE. The commands may be split into five major groups on the basis of their function:

(1) basic commands such as those for inputting and deleting grammar rules,

(2) file management commands allowing the user to save rules to disk and then later reload them,

(3) grammar management commands operating globally on the grammar: reordering metarules for instance,

(4) commands connected with the morphological analyser sub-system,

(5) and finally miscellaneous commands.

3.1 Basic Commands

3.1.1 Input construct-type declaration

The "input" command allows the user to define a construct of the given type. Declarations may be split over several lines; a prompt (">") is issued for each continuation line. If the construct has already been defined, the GDE displays the existing definition and asks if the construct should really be redefined and the old definition destroyed. If the construct is not yet defined , the GDE asks for the name of the file with which the declaration should be associated, and to which it should be written when the metagrammar is later saved to disk. E.g.

```
input id VP/TAKES_NP : VP → H[SUBCAT TAKES_NP], NP.
```

In general, the order in which rules and declarations are input does not matter. The one exception is that any features taking category values must be declared as such before any constructs which use those features are defined, otherwise the GDE may interpret the constructs incorrectly and give misleading results later. So, for example, the feature AGR, if it is category-valued should be defined in a feature declaration as AGR CAT before using it in an ID rule.

Comments (introduced by ";" and terminating at the end of the line) may be associated with any declaration and can be placed anywhere inside the declaration. When subsequently displayed, the comments (if there was more than one) are concatenated and put just after the declaration name. Comments which are not inside a declaration (i.e. at the top level) may be entered to the GDE and associated with a file using the "input comment" command. When the "write" command is issued, their file is saved and they are all joined together and put at the beginning of it.

Declarations already in a file on disk may be input by issuing a command to read the file (section 3.2) into the GDE session. In this case each declaration must be preceded by its type, FEATURE for example.

3.1.2 Edit construct-type pattern

The "edit" command allows a rule or declaration in the grammar to be edited. The precise details of the command's operation depend on the type of terminal or workstation being used.

3.1.3 Delete construct-type pattern

The "delete" command may be used to delete a declaration in the grammar. Several declarations may be deleted at once by giving the command a pattern containing "*", "?" or "=". Each declaration is displayed, and the user asked for confirmation that it should actually be deleted. E.g.

```
del lp L1
D alias *P
```

To ensure consistency, the GDE will not permit the deletion of features, sets or aliases that are used anywhere else in the grammar; if this is being attempted, an error message is printed out giving the names of the declarations involved.

3.1.4 View options construct-type pattern

The "view" command allows the user to inspect the metagrammar and object grammar; the command prints out all declarations of the given type which correspond to the pattern. One or more of "Normalised", "FUlly instantiated" and "LInearised" may be specifed as options. E.g.

```
V set MAJOR
v id VP*
v id =FIN
v comment
```

The command may be called with the special construct type "all" (e.g. `view all altered`), and in this case the command ranges over all construct types (features, aliases, sets, ID rules and so on) in the grammar.

If the "normalised" option is used then the normalised version of the declaration is displayed. Normalisation consists of expanding all references to sets and aliases into their constituent features or feature value pairs. This option is valid for all construct types except comments, features and sets. The "fully instantiated" option is valid only for ID and PS rules and words, and causes the given definitions to be displayed after category statements, propagation and default rules (in the case of ID and PS rules) or lexical category statements, morphology system entry completion rules etc. (in the case of words) have been applied. The "linearised" option is only valid for ID rules; it causes all linearised forms of the specified rule to be displayed. This option is the one to use to view linear ID rules or the results of the application of linear metarules. More than one of these options may be given in the same view command for some types of construct.

If an ID or PS rule pattern contains parentheses it is taken to refer to the matching rule after metarule expansion, e.g.

```
VP/TAKES_NP(PAS,STM1) (the "," is optional)
```

Specific subsets of expanded rules may be viewed using wildcarding inside the pattern parentheses, e.g.

```
VP/TAKES_NP(*) all expansions using all metarules
VP/TAKES_NP(* SAI) those where SAI was the last metarule applied
VP/TAKES_NP(* SAI *) those where SAI was applied at some point
```

In some cases, a metarule may match an ID or PS rule in more than one way. A distinct rule is generated for each match; the resulting rules are assigned names of the form `VP/DITR(PASS/1)`, `VP/DITR(PASS/2)`. Similarly, if the LP rules allow more than one linearisation of an ID rule, the names of resulting rules are of the form `VP/DITR/1` etc., or `VP/DITR(PASS/1)/1` etc.. ID and PS rules which were input with optional categories, or ones resulting from a metarule expansion which introduced an optional category are split into two

rules with names like `VP/DITR(PASS/+)`, for the rule with the optional category, and `VP/DITR(PASS/-)` for the rule without. Similarly, rules containing Kleene star categories are split into one rule without the category, and one where the category is specified to occur at least once.

Some more examples of view commands are:

```
view n li id *(*)
view alias +*
view fe TAKES
view meta altered
```

3.1.5 Names option construct-type pattern

The "names" command is similar to the "view" command except that instead of printing out the definitions of the declarations specified by the pattern, it prints just their names. If there are many names, they are printed in several columns; the ordering of the declarations is the same as that of the names, reading a line at a time from left to right across the columns. Possible options for ID rules are "Linearised" showing which rules in a given set have multiple linearisations, and "Normalised", showing which rules were split into +/- pairs because they contained optional categories.

3.1.6 Parse option

Invokes the parser top loop at which sentences may be typed to be parsed in order to test the grammar. A special set of commands specific to the parser (for displaying parse trees and so on) is available inside this top loop. These commands are described in section 4. If the option "uncache" is specified to the "parse" command, then all internal cached data is discarded (see the description of the command "uncache" below) before the parser command loop is entered. This is a convenient way of reducing the frequency of garbage collection during parsing, but at the cost of not being able to inspect rules in the object grammar without having to wait for the GDE to recompile the relevant part of the metagrammar from scratch.

3.1.7 Generate category

Invokes the generator top loop. The top loop makes available a specialised set of commands (described in section 5) to control the generator.

3.1.8 Quit

Exits from the GDE, first asking for confirmation; it is all too easy to type 'q' by mistake! The command requests additional confirmation if there are declarations in the grammar that are new or have been altered and not yet written to a file on disk.

3.2 File Management Commands

3.2.1 Read filename

Reads in and defines the rules and declarations contained in the given file. The operation is abandoned if a syntax error occurs, or if any of the items in the file are already defined.

3.2.2 Write filename

Writes to disk the definitions associated with the given file. If the file already exists and is being written for the first time in this GDE session then it is first backed up by copying it to a file with the same name, but of type 'bak'. If the file name given to the command is specified as "*", then all declarations in the grammar are written back to their respective files.

3.2.3 FIles

This command prints out the names of the grammar files that have been read in so far in the current GDE session.

3.2.4 Move

Starts a dialogue which allows a declaration to be renamed and / or associated with a different file.

3.3 Grammar Management Commands

3.3.1 Order construct-type

The "order" command allows the definitions of any construct type to be re-ordered. For most constructs the only significance this has is in determining in what order definitions of that particular type appear when written to file. For features, however, the order determines the structure of the internal tree data-structures formed by the parser and generator so that they can look rules up efficiently. (The lookup will be more efficient if more discriminating features appear before less discriminating ones). The orders of metarules, propagation, default and category rules determine the order in which the individual rules of these types are applied to ID and PS rules. The command starts up a top loop containing the commands: "View", "Move", "Help" and "Quit".

3.3.2 FOrget filename

Literally 'forgets' about a file and its contents, effectively putting the grammar into a state in which the declarations in the file appear never to have existed. As with the delete command, "forget" will complain if the file contains declarations of features, sets or aliases which are used anywhere, since forgetting any of these would make the grammar inconsistent.

This command, in conjunction with "read", makes it easy to debug a grammar incrementally by allowing preset groups of rules to be conveniently removed from the grammar and then, when desired, quickly added back into it. "Forget" is also useful if an error occurs while a file is being read in from disk. (An error is possible only if the file had been text-edited and left containing a syntax error). In this case, the best course of action is to 'forget' the file (so that constructs defined before the error occurred are discarded), re-edit the file to correct the error, and then read the file in again.

3.3.3 CLear

Clears the grammar currently under development.

3.4 Miscellaneous Commands

3.4.1 SEt flag-name value

Various aspects of the behaviour of the GDE are controlled by flags. The "set" command allows the user to change the values of these flags. In most cases, flags should have either the value ON or the value OFF. The flags are:

PRop before default	Controls the order in which propagation and default rules are applied to ID rules.
Addition checking	Whether an error is signalled if a metarule, when about to add a feature value pair to an ID rule category finds that the feature already present.
Multiple Expansions	Controls whether multiple identical ID rules resulting from metarule expansion are reduced to just one in the compiled version of the grammar.
Multiple Linearisations	Controls whether multiple identical linearisations of an ID rule are reduced to just one.
MOrphology system	Controls whether the morphology package is loaded if the GDE does not know about a word. Values for this flag should be either OFF, or the initial part of the filenames of the source and compiled morphology files. See section 6.1.
Fast morph lookup	When a word is looked up in the morphology system lexicon, this flag controls whether a 'fast lookup' takes place, in which an attempt is first made to look the word up as a morpheme, and if successful not also to attempt a full analysis of the word.
Ecrs before multiply	Controls the order in which entry completion rules and multiplication rules are applied to word definitions.
TErm unification	Whether the parser and generator match grammatical categories using term unification or unrestricted unification. Note that this flag is completely distinct, and has nothing to do with whether the morphological analyser is interpreting its word grammar using term or unrestricted unification.
TOp category	When ON, the parser accepts only extensions of the category [T +] as valid parses. Otherwise it accepts any category which covers the whole of the input.

3.4.2 FLags

Displays the current settings of all the flags.

3.4.3 COmpile

Creates a complete set of context free rules by metarule expanding, propagating features in, applying default rules and category statements to, and linearising every ID rule in the system. See section 2.14 for a description of this metagrammar compilation process. Some statistics concerning the size of the grammar are printed out. The resulting context free rules are not printed; they are simply stored for future use by the parser and generator.

3.4.4 Uncache

A primary requirement for a grammar development environment, particularly one such as the GDE which supports a high-level metagrammatical formalism, is speed of compilation to form the

object grammar. To this end, the GDE maintains several internal data-structures representing partially compiled portions of the current metagrammar, so that minor changes to the grammar do not force the GDE to recompile the whole grammar from scratch. The "uncache" command deletes all these internal intermediate data-structures generated by the GDE. This cached data describing the grammar is of no concern to the user of the GDE, who thus is unlikely to need to use this command, except perhaps to force the next recompilation of the grammar to take place from scratch in order to force any warning messages issued by the process to be displayed.

3.4.5 DUmp option filename

Compiles the grammar and prints the resulting PS rules to the given file in a format (with the "unreadable" option) suitable for input to a stand-alone version of the Alvey Tools Projects parser. If the option is "readable" then the output is similar to that produced calling the "view" command with the "linearised" and "normalised" options.

3.4.6 DWords option filename

Prints to the given file all the words in the GDE lexicon, and those retrieved from the morphology system in the current session. The intention is that a stand-alone version of the Tools Projects parser without an interface to the morphology system could be run just using these words. As in the "dump" command, the option may be specified as either "readable" or "unreadable".

3.4.7 Help

Displays a page of information on commands available in the GDE.

3.4.8 SHell

Invokes an operating system command shell from within the GDE.

3.4.9 ! lisp-expression

Prints the result of evaluating the given lisp expression.

4. The Parser

The parser is invoked by the "parse" command (section 3.1), and is a chart parser using a bottom-up strategy, a modified version of the parser produced by the Alvey Natural Language Tools parser project (Phillips, 1986)[1]. If the "top category" flag (section 3.4) is OFF, any category which covers all the input is accepted as a valid parse; if the flag is ON then only parses whose root category is an extension of `[T +]` are accepted. At the end of each parse, statistics are printed giving CPU and elapsed time, and the number of chart edges generated during the parse. In addition, bracketings of the words in the sentence are printed, one for each valid parse. The following is a summary of the commands that the top loop accepts.

View option

The option should be one of "Words", "Rules", "Full", "CAtegory", "COmmon", "Sentence", "Parsed", "Edges", "ACtive", "INactive", "IA", or "AI".

The first three options display the parse tree(s) resulting from parsing the last sentence. If the "words" option is specified, the output is just a bracketing of the words in the sentence; with the "rules" option, each parse tree is displayed graphically with the name of a rule at non-terminal nodes and a word at terminal nodes in the tree; the "full" option additionally displays the category at each node (but with features having uninstantiated values suppressed to save space on the screen). The "category" option may be used to look at a selected category in a parse tree without this suppression of variable values. The option first displays a menu consisting of rule names, each representing a parse tree node: selecting one causes the category at that node to be printed out. Each of these options may additionally be given a number (for example 1 meaning first, 2 meaning second) indicating that only that particular tree is to be displayed. E.g.

```
view rules
view full 2
view category 1
```

When there are multiple parses for a sentence, the "common" option may be used to find out which subtrees are shared between two or more of the parses. Common sub-trees are indicated in a table, with parse numbers horizontally across the top, and rule names with corresponding word ranges vertically down the left hand side.

The "sentence" option prints out the words in the last sentence parsed, the words numbered to correspond with the numbering of vertices in the chart; the "parsed" option prints out all the sentences parsed in the current session. The remaining options output detailed information to help in debugging the grammar. The options either display all the edges in the chart ("edges"), all active edges ("active"), all inactive edges ("inactive"), incoming inactive and outgoing active at a vertex ("ia"), and *vice versa* ("ai"). The option name may be followed by one or two numbers to select the subset of edges starting at the vertex specified by the first number (and optionally finishing at the vertex specified by the second). E.g.

```
view inact 0 1
view ia 3
```

An edge in the chart when displayed consists of four fields: the first is either the single letter 'A' (indicating that the edge is active, and is expecting one or more constituents to its right) or 'I' (inactive meaning the edge represents a complete constituent). The second field contains two

[1] The modifications made to the version of the parser used by the GDE were made to enable chart edges to be printed out after a parse for grammar debugging purposes, to fix a couple of bugs, and to allow a choice of either term unification or unrestricted unification of grammatical categories.

numbers, the start and end vertices of the edge. What follows depends on the type of the edge, i.e. whether it is active or inactive. If the edge is active, the next field is the name of the rule that has found one or more of its daughters, but is still expecting to find more; the categories expected are shown after the rule name. So, for example, the edge

```
A  0 -> 1      S -->
                   [N -, V +, AGR [N +, V -, BAR 2], PRD -, BAR 2,
                    FIN +, SUBJ -]
```

is an active edge, starting at vertex 0 and finishing at vertex 1 (and thus has been able to consume the first word in the sentence), the name of the rule which introduced this edge is `S`, and it is still looking for one (verbal) category.

If the edge is inactive, the third field may again be a rule name, or it may be a word, and is followed by the category of the complete constituent found. The first edge below representing the word `fido` will have been created when the word was found in the sentence to be parsed, and the category following it will be its definition. The second edge is also a complete constituent, the name of the rule in the 'object' grammar being `N2/PN`, and the category of the constituent is `[N +, V -, BAR 2, PN +]`.

```
I  0 -> 1      fido
                   [N +, V -, BAR 0, SUBCAT NULL]
I  0 -> 1      N2/PN
                   [N +, V -, BAR 2, PN +]
```

Previous

Attempts to parse the previous sentence again.

Fparse option input-filename output-filename

Parses all the sentences in the given input file. Each sentence should finish with a full stop or question mark so the GDE can tell where one sentence ends and the next begins. The output file argument need not be given. If it is, then all messages from the parser are directed to that file, otherwise they are just sent to the screen as usual. If output is to a file and the file already exists, it is first backed up. The form of output produced depends on the option specified immediately after the command: "brief" gives just numbers of parses for each sentence, whereas "verbose" prints timing statistics and word bracketings for each parse.

Write option filename

If the option is "rules" or "full", the parse trees from the last sentence successfully parsed are written to a set of files (with names of the form file1, file2 etc.) in a way that should allow large trees to be studied by pasting hardcopy printouts of the files side by side. With the "parsed" option, the command writes all the sentences parsed in the current session to the given file in a format acceptable for input to the "fparse" command. Using the "parsed" option makes it easy to build a corpus of sentences representing the coverage of the grammar being developed.

Help

Displays a page of information on commands available in the parser.

Quit

Exits from the parser top loop.

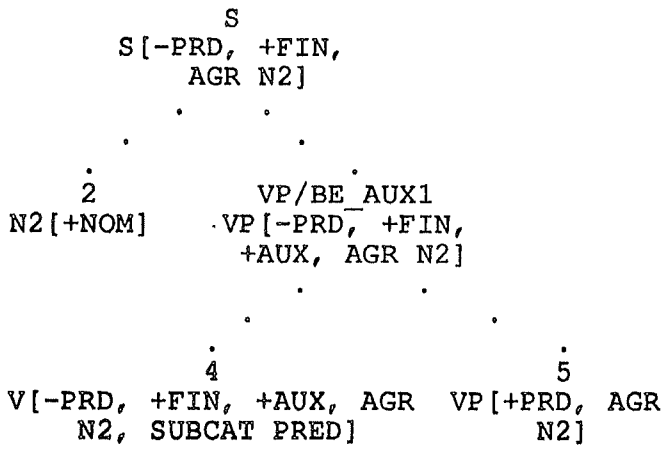
anything else

The input is taken to be a sentence to be parsed.

5. The Generator

The generator is invoked by the "generate" command (section 3.1). A category may be specified as part of the command, and is taken to be the top node of the tree to be generated. (If no category is given, the category of the top node defaults to `[T +, BAR 2]`).

The generator may essentially be run in two modes: manual and automatic. In the former, the user can incrementally and interactively build a syntax tree by expanding one node in the tree at a time; in the latter mode the GDE exhaustively generates all lexical strings starting from the current state of the syntax tree. The generator is a particularly useful tool for detecting and tracking down sources of overgeneration in a grammar. When operating in manual mode, overgeneration shows up as inappropriate rules unexpectedly being applicable to a node being expanded while building the generator tree. For example, in the GDE session in section 1.1, the incorrect `VP/NOPASS(PASS/+)` rule might have been spotted after starting to generate with top category `S` when the situation



had been reached. After issuing the command `expand 5`, the GDE would have printed the names of the rules in the object grammar that were applicable to node 5,

```
...
14. VP/TAKES_TWONP(PASS/1/-)  15. VP/TAKES_TWONP
16. VP/NOPASS̄(PASS/+)         17. VP/NOPASS̄(PASS/-)
18. VP/NOPASS                 19. VP/DITR(PASS/+)
...
```

and the user would probably have noticed the spurious `VP/NOPASS(PASS/+)` rule. Whether the rule was noticed or not, starting automatic generation from this point would have resulted in sentences like

```
fido is cost
fido is cost by fido
```

again showing up even more clearly the overgeneration in the initial grammar. The following are the commands which may be used to control the generator.

View option

The option should be one of "Sentence", "Words", "Rules" or "Full". All of these options display the current state of the generator tree, but differ in the amount of detail they show. If the "sentence" option is specified, the word (or category if the node has not been expanded that far)

on each leaf node in the tree is output; the "words" option brackets them according to the structure of the tree. The "rules" option shows the tree graphically with the name of a rule or word at each node; the "full" option additionally displays the category at each node, but with features with (so far) uninstantiated values suppressed.

Expand node-name pattern

Expands a generator tree node, with a given ID rule if the node is not lexical, or a word if it is. The new tree is then displayed. E.g.

```
expand 2 *
e 3 VP/TAKES_NP
e 10 The
```

If there is more than one rule or word that both matches the pattern and is applicable to the given node, the names of the applicable rules or words are printed out, indexed by a number, and the user asked for the number corresponding to the rule or word that should be applied.

Sentence option maximum-length

Starts automatic generation. Each node in the tree is expanded until it is lexical; if the GDE knows an appropriate word, the word is used to fill in the node. The generator then prints out the tree (by calling the view command, passing the option) and then backtracks to try and find further trees. The generator will not apply any PS rule more than once down any branch of the tree. This helps to control the process, as does the maximum length parameter (a positive number) which puts a limit on the length in words of the sentences to be generated.

Clear

Clears the generator tree. The tree is maintained over entries to and exits from the generator, except when the optional category to the "generate" command is actually specified.

Help

Displays a page of information on commands available in the generator.

Quit

Exits from the generator top loop.

6. Using the Morphological Analyser with the GDE

The GDE includes version 3.0 (with the unrestricted unification word grammar option) of the Alvey Natural Language Tools Morphological Analyser (Russell *et al*, 1986; Ritchie *et al*, 1987). The development of the GDE and the morphological analyser were originally, however, quite separate (although collaborative) projects. A consequence of this is that the two systems may be used independently of each other. So, on the one hand, the morphological analyser has its own top level loop from which commands to look up words, compile spelling rules and so on are available, and on the other, words may be defined to the GDE and their definitions retrieved later by the parser and generator.

6.1 The Interface to the Morphology System

The flag "morphology system" (section 3.4) controls whether the analyser may be invoked to provide the definition of a word. If the flag is OFF, the GDE signals an error if an attempt is made to look up a word which has not been directly defined to the GDE (using the "input word" command). Otherwise, the flag is assumed to contain the initial part[1] of the names of the files holding the compiled spelling rule, word grammar and lexicon files; these files will be loaded, and the morphological analyser called to look up the word. If the word has been directly defined to the GDE, however, any definitions of it that may be provided by the morphology system will automatically be overriden, even if the flag is ON.

The type of lookup that the morphological analyser performs is controlled by the flag "fast morph lookup". If the flag is ON, the analyser first tries to look the word up as a simple word, and if it is successful returns this result and does not attempt a full analysis of the word. If the flag is OFF, the analyser always attempts a full analysis. So, for example, if the word `believes` is in the lexicon as a simple word, and the flag is ON, then a lookup of `believes` will return its definition as a simple word, and not that definition plus ones resulting from also treating it as `believe` with the suffix `s`. When a word is retrieved using the morphological analyser, the GDE prints some statistics on the time it took to be looked up. The definition of the word is then saved internally by the GDE so that if its definition is subsequently needed it can be retrieved again very quickly.

Entry completion rules, multiplication rules and consistency checks may be defined to the GDE in a similar manner to other constructs. The rules are applied to words defined in the GDE lexicon when they are required by the parser, the generator, or the "view full" command. The rules are also applied during lexicon compilation (invoked by the GDE "cdictionary" command). They may be input, deleted, ordered etc. using the standard GDE commands, and may also be saved to file in the usual way. Below are a typical entry completion and multiplication rule as they might appear in a file:

---

[1] The morphological analyser assumes that the lexicon, word grammar and spelling rules (before they are compiled) are held respectively in files whose names end in ".le", ".gr" and ".sp". The files produced after the analyser has compiled them end in an additional ".ma". The 'initial part' of a morphology system file is the part before the first "." in its name - the GDE assumes that, for a given compatible set of files, the initial parts of their names will be the same.

```
ENTRY BAR_MINUS_ONE: ; Add (BAR -1) as default to entries with
                     ; FIX specifications - affixes are lower
                     ; level units than complete words
   (_ _ ((FIX _fix) ~(BAR _) _rest) _ _) =>
      (& & ((FIX _fix) (BAR -1) _rest) & &).

MULTIPLICATION PRD_MINUS: ; Add an entry with (PRD -) for each
                          ; one with (VFORM ING) and (PRD +)
   (_ _ ((VFORM ING) (PRD +) _rest) _ _)
   =>>
   (
    (& & ((VFORM ING) (PRD -) _rest) & &)
   ).
```

The sets WHEAD and WDAUGHTER control the processing of the word grammar inside the morphology system as outlined in the user manual for that system. Also, as detailed there, the set MORPHOLOGYONLY should contain the names of all features which are purely internal to the morphology system. These features are stripped off word definitions passed to the parser, the generator and the "view full" command.

When displaying a word definition, the view command puts the word in parentheses if it originally came from the morphology system. The commands "view morpheme" and "names morpheme" may be used to directly access morpheme definitions in the morphology system, regardless of whether words of the same name are defined in the GDE. These commands take a pattern as argument.

6.2 Additional GDE Commands

6.2.1 DCi

Invokes the morphological analyser command loop. (DCI stands for Dictionary Command Interpreter). Typing 'h' gives a list of the commands which are available. Note that command arguments (filenames for example) which contain special characters such as '{', '}', '>' and '<' should be enclosed in double quotes. The same goes for the filename specified after the '#include' directive in analyser lexicon and word grammar source files.

6.2.2 CDictionary

Compiles a new morphology system lexicon. Word entries are merged from the lexicon source file (its name obtained by appending '.le' to the value of the "morphology system" flag) and from the GDE lexicon. At the end of compilation, the user is given the option of having all the words defined to the GDE deleted from there and inserted into the lexicon source file.

This command assumes that entry completion and multiplication rules are defined within the GDE; therefore the lexicon source file should contain only morpheme entries. If the rule declarations are kept in ordinary text files as assumed in the documentation for the morphological analyser, lexicon compilation should be invoked from the analyser command loop, rather than directly from the GDE.

6.2.3 CSpelling

Compiles a new set of spelling rules for the morphology system. The rules are expected to be in the file whose name is the result of appending '.sp' to the value of the "morphology system" flag.

6.2.4 CWgrammar

Compiles a new word grammar for the morphology system. The grammar is expected to be in the file whose name is the result of appending '.gr' to the value of the "morphology system" flag.

6.2.5 FWords input-filename output-filename

Looks up and prints the definitions of the words contained in the given input file. The output file need not be specified. If it is, then the definitions are printed to that file, otherwise they are are just output to the screen as usual. If an output file is specified and it already exists, it is first backed up.

7. Errors, Bugs and Future Enhancements

7.1 Errors and Warnings

If the GDE detects a condition that, unless corrected, would lead either to inconsistent or to misleading results being produced, the GDE signals an error by printing

```
*** Error, <informative error message>
```

and abandons the current command, returning to the next top level prompt to wait for the next command from the user. In most cases the remedy for the error is obvious, and once this is seen to the previous command can be issued again. The GDE responds to less serious conditions by printing a warning, e.g.

```
*** Warning, <informative warning message>
```

and continuing with what it was doing.

7.2 Bugs

After reordering definitions which are split over more than one file, their new order is retained only for the remainder of the same GDE session. When the files are read in again, the definitions of each construct type in the first file will always be before those in files read in later.

It is not possible for the user to individually refer to rules resulting from multiple metarule matches, multiple linearisations, or from rules originally with optional categories being split into two or more separate versions. Only the 'base' name is recognised by commands such as "view", and the name is taken to refer to all variants of the rule. The '=' pattern sometimes returns duplicate definitions. In addition, PS rules resulting from the application of linear metarules cannot be viewed, even though they are actually present, and may be dumped to file and successfully used by the parser and generator.

The "edit" command does not work in the Common Lisp version of the GDE. Most operating systems or Lisp environments allow lines or sequences of lines displayed on the screen to be input to the Lisp process; viewing a declaration, then redefining it with the "input" command, making use of this re-inputting facility is, in most cases, an acceptable alternative to "edit".

The parser sometimes fails to successfully match feature-value pairs where the value is the atom NIL. An acceptable alternative is to use the atom NULL instead. Also in the parser the unrestricted unification option (selected by setting the "term unification" flag to OFF) does not work properly. The category which results from a successful unification will contain features from only one of the unifying categories, not both. The version of the parser used by the GDE ignores rules with no daughters, and a trace which appears as the first daughter in a rule will not appear in the parse tree.

7.3 Future Enhancements

The next Common Lisp release of the Alvey Tools software will include improvements in speed to the parser and morphological analyser. It is also planned to add the ability to specify semantic formulae for ID, PS and metarules, so that the GDE will then provide support for semantics as well as morphology and syntax.

8. References

Briscoe, E., Grover, C., Boguraev, B. and Carroll, J. (1987a) 'Feature Defaults, Propagation and Reentrancy' in Klein, E. and van Benthem, J. (Eds.), *Categories, Polymorphism and Unification*, Centre for Cognitive Science, University of Edinburgh, pp. 19–34.

Briscoe, E., Grover, C., Boguraev, B. and Carroll, J. (1987b) 'A Formalism and Environment for the Development of a Large Grammar of English', *Proceedings of 10th International Joint Conference on Artificial Intelligence*, Milan, Italy, pp. 703–708.

Gazdar, G., Klein, E., Pullum, G. and Sag, I. (1985) *Generalized Phrase Structure Grammar*, Blackwell, Oxford.

Grover, C., Briscoe, E., Carroll, J., and Boguraev, B. (1988) *The Alvey Natural Language Tools Project Grammar – a Large Computational Grammar of English*, Lancaster Papers in Linguistics, Department of Linguistics, University of Lancaster.

Phillips, J. (1986) *A Parsing Tool for the Natural Language Theme, Version 13*, Department of Artificial Intelligence, University of Edinburgh.

Phillips, J. and Thompson, H. (1987) 'A Parser and an Appropriate Computational Representation for GPSG' in Klein, E. and Haddock, N. (Eds.), *Cognitive Science Working Papers 1*, Centre for Cognitive Science, University of Edinburgh.

Ritchie, G., Black, A., Pulman, S. and Russell, G. (1987) *The Edinburgh/Cambridge Morphological Analyser and Dictionary System (Version 3.0) User Manual*, Software Paper No. 10, Department of Artificial Intelligence, University of Edinburgh.

Russell, G., Pulman, S., Ritchie, G. and Black, A. (1986) 'A Dictionary and Morphological Analyser for English', *Proceedings of 11th International Conference on Computational Linguistics*, Bonn, Germany, pp. 277–279.

Appendix 1 - Example Grammar

This appendix contains a small grammar as an example of the metagrammatical formalism, its syntax, and how it may be used. It is in the style of GPSG, and was the one read in at the beginning of the GDE session in section 1.1.

```
; File 'gram.rules'. A simple GPSG style grammar.

FEATURE H{+, -}
FEATURE N{+, -}
FEATURE V{+, -}
FEATURE AGR CAT
FEATURE PRD{+, -}
FEATURE BAR{0, 1, 2}
FEATURE VFORM{BSE, EN, TO}
FEATURE FIN{+, -}
FEATURE SUBJ{+, -}
FEATURE AUX{+, -}
FEATURE PFORM{OF, BY, TO}
FEATURE PN{+, -}
FEATURE PER{1, 2, 3}
FEATURE CASE{NOM, ACC}
FEATURE PLU{+, -}
FEATURE SUBCAT{NP, PRED, BASE_VP, NP_NP, SFIN, NULL, SR, OR,
   NOPASS, TWONP, DETN}

SET MAJOR = {N, V}
SET VERBALHEAD = {PRD, FIN, AUX, VFORM, AGR}
SET NOMINALHEAD = {PLU, CASE, PRD, PN, PER}
SET PREPHEAD = {PFORM, PRD}
SET AGRFEATS = {PLU, PER}

ALIAS +N = [N +].
ALIAS +PRD = [PRD +].
ALIAS -PRD = [PRD -].
ALIAS BSE = [VFORM BSE].
ALIAS EN = [VFORM EN].
ALIAS TO = [VFORM TO].
ALIAS to = [PFORM TO].
ALIAS +NOM = [CASE NOM].
ALIAS +ACC = [CASE ACC].
ALIAS +FIN = [FIN +].
ALIAS -FIN = [FIN -].
ALIAS V = [V +, N -, BAR 0].
ALIAS N = [N +, V -, BAR 0].
ALIAS A = [V +, N +, BAR 0].
ALIAS P = [V -, N -, BAR 0].
ALIAS P1 = [N -, V -, BAR 1].
ALIAS VP = [N -, V +, BAR 2, SUBJ -].
ALIAS N2 = [N +, V -, BAR 2].
ALIAS S = [N -, V +, BAR 2, SUBJ +].
ALIAS P2 = [N -, V -, BAR 2].
ALIAS V2 = [V +, N -, BAR 2].
ALIAS +SUBJ = [SUBJ +].
ALIAS -SUBJ = [SUBJ -].
ALIAS H = [H +, BAR 0].
ALIAS H1 = [BAR 1, H +].
ALIAS H2 = [BAR 2, H +].
ALIAS DetN = [SUBCAT DETN].
ALIAS -AUX = [AUX -].
```

```
ALIAS +AUX = [AUX +].
ALIAS +PLU = [PLU +].
ALIAS -PLU = [PLU -].

LCATEGORY W_NOUN : [N +, V -] => NOMINALHEAD
LCATEGORY W_PREP : [N -, V -] => PREPHEAD
LCATEGORY W_VERB : [N -, V +] => VERBALHEAD
LCATEGORY AGR_N2 : AGR N2 => AGRFEATS

EXTENSION {H, N, V, BAR, SUBJ}

IDRULE S : S[+FIN] --> N2[+NOM], H2[-SUBJ, AGR N2].
IDRULE N2/PN : N2 --> H[SUBCAT NULL, PN +].
IDRULE N2/DET : N2 --> DetN, H[SUBCAT NULL].
IDRULE PP : P2 --> H1.
IDRULE PP/TAKES_NP : P1 --> H[SUBCAT NP], N2.
IDRULE VP/INTR : VP --> H[SUBCAT NULL].
IDRULE VP/TAKES_NP : VP --> H[SUBCAT NP], N2[-PRD].
IDRULE VP/DITR : VP --> H[SUBCAT NP_NP], N2[-PRD], N2[-PRD].
IDRULE VP/NOPASS : VP --> H[SUBCAT NOPASS], N2[+PRD].
IDRULE VP/TAKES_TWONP :
   VP --> H[SUBCAT TWONP], N2[-PRD], N2[+PRD].
IDRULE VP/BE_COP1 : VP[+AUX] --> H[SUBCAT PRED], N2[+PRD].
IDRULE VP/BE_COP2 : VP[+AUX] --> H[SUBCAT PRED], P2[+PRD].
IDRULE VP/BE_AUX1 :
   VP[+AUX, AGR N2] --> H[SUBCAT PRED], VP[+PRD, AGR N2].
IDRULE VP/BE_AUX2 :
   VP[+AUX, AGR S] --> H[SUBCAT PRED], VP[+PRD, AGR S].
IDRULE VP/TO :
   VP[+AUX, TO, -FIN, AGR N2] -->
      H[SUBCAT BASE_VP], VP[BSE, AGR N2].
IDRULE VP/SR : VP[AGR N2] --> H[SUBCAT SR], VP[TO, AGR N2].
IDRULE VP/OR : VP --> H[SUBCAT OR], N2[-PRD], VP[TO, AGR N2].

METARULE PASS :
   VP --> W, N2. ==> VP[EN, +PRD] --> W, ( P2[PFORM BY] ).

PROPRULE HFC_VERBAL :
   [N -, V +] --> [H +], U. F(0) = F(1), F in VERBALHEAD.
PROPRULE HFC_NOMINAL :
   [N +, V -] --> [H +], U. F(0) = F(1), F in NOMINALHEAD.
PROPRULE HFC_PREP :
   [N -, V -] --> [H +], U. F(0) = F(1), F in PREPHEAD.
PROPRULE AGR/NP_VP : S --> N2, H2[-SUBJ, AGR N2].
    F(1) = F(2[AGR]), F in AGRFEATS.
PROPRULE S_CONTROL : VP[AGR N2] --> H, VP[AGR N2].
    F(0[AGR]) = F(2[AGR]), F in AGRFEATS.
PROPRULE O_CONTROL : VP --> H[SUBCAT OR], N2, VP[AGR N2].
    F(2) = F(3[AGR]), F in AGRFEATS.

DEFRULE RHS_N2_CASE : [N -] --> N2, U. CASE(1) = ACC.
DEFRULE VP/AGR : VP --> W. AGR(0) = N2[PER @x, PLU @y].
DEFRULE VP_PRD : [] --> VP, U. PRD(1) = -.

LPRULE LP1 : [SUBCAT] < [~SUBCAT].
LPRULE LP2 : [N +] < P2 < V2.
LPRULE LP3 : N2[-PRD] < N2[+PRD].

WORD a : DetN.
WORD pound : N[SUBCAT NULL].
WORD is : V[SUBCAT PRED].
WORD cost : V[+PRD, EN, AGR N2, SUBCAT NOPASS].
WORD by : P[SUBCAT NP].
```

```
WORD fido : N[SUBCAT NULL].
WORD costs : V[SUBCAT NP].
```

Appendix 2 - Customisation

This appendix gives some very basic information about how the GDE may be customised. Typical customisation might involve changing the default values of some of the flags, or automatically reading a file of basic feature, set and alias definitions on entry to the GDE. Most Lisp implementations specify that if a file with a certain name exists, then the contents of the file will be evaluated when the Lisp system is entered; this initialisation file is a natural place to put Lisp calls to customise the GDE.

There are several Lisp variables affecting the functioning of the GDE that may safely be reset. The first such group of variables are the flags. These are implemented as special variables (bound once only at the top level) with the value `t` representing ON, and `nil` representing OFF. The variables holding the flag values are

```
*prop-before-default
*addition-checking
*multiple-expansions
*multiple-linearisations
*morph-system
*fast-morph-lookup
*ecrs-before-multiply
*term-unification
*parser-top-category
```

The page width and length assumed for the printer used for the hardcopy of the files written to by the parser "write rules" and "write full" commands are held in the variables

```
*file-page-width
*file-page-depth
```

and the names of the files from which the help commands take their information are held in

```
*gde-help-file
*parser-help-file
*generator-help-file
*order-help-file
```

A file may be read into the GDE by calling the Lisp function `read-grammar`. The function takes one argument, the name of a file. Instead or in addition to using an initialisation file, arbitrary Lisp expressions may be evaluated from the GDE top loop by prefixing them with the character '`!`'.

Appendix 3 - Dumped Grammar Format

The GDE can be requested, using the "dump" command, to output the current object grammar to a file. When the "unreadable" option is specified, the PS rules in the file are in a format suitable for input to a stand-alone version of the Alvey Tools Projects parser. Any other parser should be able to accept the file after a little editing or pre-processing. This appendix gives a specification of the format to enable the grammar to be used by other parsers.

The file consists of a sequence of Lisp lists. The first list contains the names of all the features in the grammar, and is followed by each PS rule in the object grammar. A rule is represented as a list, the mother being the first element, followed by the daughter categories in order. Each category is a dotted pair whose head is a list of feature / value pairs, and whose tail is, for a mother, a string representing the name of the rule, or, for a daughter, a symbol indicating whether the category is repeated one or more times ('+'), or occurs exactly once ('NIL').

The following might be the first part of a dumped grammar file

```
(N V BAR SUBJ H T VFORM FIN PAST PRD AUX AGR INV HACK NEG COMP
NFORM PER PLU COUNT CASE PN PRO PART POSS DEF SPEC PFORM LOC
GERUND AFORM QUA ADV NUM SUBCAT WH UB EVER SLASH NULL MOD CONJ
CONJN COORD TITLE REFL AT LAT FIX INFL STEM COMPOUND PRT CAT MAJ
REG)

(((#S(FV-PAIR FEATURE N VALUE -)
   #S(FV-PAIR FEATURE V VALUE +)
   #S(FV-PAIR FEATURE BAR VALUE \2)
   #S(FV-PAIR FEATURE SUBJ VALUE +)
   #S(FV-PAIR FEATURE VFORM VALUE NOT)
   #S(FV-PAIR FEATURE FIN VALUE +)
   #S(FV-PAIR FEATURE PAST VALUE #:@13)
   #S(FV-PAIR FEATURE PRD VALUE #:@14)
   #S(FV-PAIR FEATURE AUX VALUE #:@15)
   #S(FV-PAIR FEATURE AGR
              VALUE
               (#S(FV-PAIR FEATURE N VALUE +)
                #S(FV-PAIR FEATURE V VALUE -)
                #S(FV-PAIR FEATURE BAR VALUE \2)
                #S(FV-PAIR FEATURE NFORM VALUE #:@21)
                #S(FV-PAIR FEATURE PER VALUE #:@22)
                #S(FV-PAIR FEATURE PLU VALUE #:@23)
                #S(FV-PAIR FEATURE COUNT VALUE #:@24)
                #S(FV-PAIR FEATURE CASE VALUE NOM)))
  #S(FV-PAIR FEATURE INV VALUE -)
  #S(FV-PAIR FEATURE COMP VALUE NORM)
  #S(FV-PAIR FEATURE COORD VALUE #:@61))

 "S/1/-")
((#S(FV-PAIR FEATURE N VALUE +)
  #S(FV-PAIR FEATURE V VALUE -)
  #S(FV-PAIR FEATURE BAR VALUE \2)
  #S(FV-PAIR FEATURE PRD VALUE #:@73)
  #S(FV-PAIR FEATURE NEG VALUE -)
  #S(FV-PAIR FEATURE NFORM VALUE #:@21)
  #S(FV-PAIR FEATURE PER VALUE #:@22)
  #S(FV-PAIR FEATURE PLU VALUE #:@23)
  #S(FV-PAIR FEATURE COUNT VALUE #:@24)
  #S(FV-PAIR FEATURE CASE VALUE NOM)
  #S(FV-PAIR FEATURE PN VALUE #:@26)
  #S(FV-PAIR FEATURE PRO VALUE #:@27)
```

```
   #S(FV-PAIR FEATURE PART VALUE #:@28)
   #S(FV-PAIR FEATURE POSS VALUE #:@29)
   #S(FV-PAIR FEATURE DEF VALUE #:@30)
   #S(FV-PAIR FEATURE SPEC VALUE #:@31)
   #S(FV-PAIR FEATURE AFORM VALUE #:@35)
   #S(FV-PAIR FEATURE COORD VALUE #:@68)))
  ((#S(FV-PAIR FEATURE N VALUE -)
   #S(FV-PAIR FEATURE V VALUE +)
   #S(FV-PAIR FEATURE BAR VALUE \2)
   #S(FV-PAIR FEATURE SUBJ VALUE -)
   #S(FV-PAIR FEATURE VFORM VALUE NOT)
   #S(FV-PAIR FEATURE FIN VALUE +)
   #S(FV-PAIR FEATURE PAST VALUE #:@13)
   #S(FV-PAIR FEATURE PRD VALUE #:@14)
   #S(FV-PAIR FEATURE AUX VALUE #:@15)
   #S(FV-PAIR FEATURE AGR
              VALUE
               (#S(FV-PAIR FEATURE N VALUE +)
                #S(FV-PAIR FEATURE V VALUE -)
                #S(FV-PAIR FEATURE BAR VALUE \2)
                #S(FV-PAIR FEATURE NFORM VALUE #:@21)
                #S(FV-PAIR FEATURE PER VALUE #:@22)
                #S(FV-PAIR FEATURE PLU VALUE #:@23)
                #S(FV-PAIR FEATURE COUNT VALUE #:@24)
                #S(FV-PAIR FEATURE CASE VALUE NOM)))
   #S(FV-PAIR FEATURE NEG VALUE #:@19)
   #S(FV-PAIR FEATURE COORD VALUE #:@48))))
```

As can be seen, feature / value pairs are Common Lisp structures of type `FV-PAIR`, with the two fields `FEATURE` and `VALUE`. Features in a category are guaranteed to occur in exactly the same order as they appear in the list at the top of the file.

The "dwords" command with the "unreadable" option gives output in a similar form to the above. A sequence of word definitions follows the initial list of features. A definition is a dotted pair whose head is a list of categories, one for each sense of the word, and whose tail is the word itself. A category has as its head a list of feature / value pairs, and tail the symbol `word`. Thus a file containing a definition for the word `fido` might look like

```
(N V BAR SUBJ H T VFORM FIN PAST PRD AUX AGR INV HACK NEG COMP
NFORM PER PLU COUNT CASE PN PRO PART POSS DEF SPEC PFORM LOC
GERUND AFORM QUA ADV NUM SUBCAT WH UB EVER SLASH NULL MOD CONJ
CONJN COORD TITLE REFL AT LAT FIX INFL STEM COMPOUND PRT CAT MAJ
REG)

(((#S(FV-PAIR FEATURE N VALUE +)
   #S(FV-PAIR FEATURE V VALUE -)
   #S(FV-PAIR FEATURE BAR VALUE \0)
   #S(FV-PAIR FEATURE PRD VALUE #:@189)
   #S(FV-PAIR FEATURE NFORM VALUE NORM)
   #S(FV-PAIR FEATURE PER VALUE \3)
   #S(FV-PAIR FEATURE PLU VALUE -)
   #S(FV-PAIR FEATURE COUNT VALUE #:@188)
   #S(FV-PAIR FEATURE CASE VALUE #:@190)
   #S(FV-PAIR FEATURE PN VALUE +)
   #S(FV-PAIR FEATURE PRO VALUE -)
   #S(FV-PAIR FEATURE PART VALUE -)
   #S(FV-PAIR FEATURE POSS VALUE -)
   #S(FV-PAIR FEATURE SUBCAT VALUE NULL))

  |word|)
 |fido|)
```

Appendix 4 - GDE Implementations

The morphological analyser and parser were originally written in Franz Lisp, and the GDE in Cambridge Lisp. The morphological analyser has since been ported to Common Lisp by Alan Black (University of Edinburgh Department of Artificial Intelligence), and John Carroll has translated the parser and GDE proper into Common Lisp and brought the three programs together to run as one integrated system. The system has been tested in the following implementations of Common Lisp:

(1) Hewlett Packard Common Lisp, version 1.01, on an HP 9000 Series ('Bobcat') workstation under the HP-UX operating system.

(2) Kyoto Common Lisp, version 1.25, on a Sun 3/160 under Sun UNIX 4.2.

(3) Xerox Common Lisp, Lyric release, on a Xerox 1186 ('Dove') workstation.

(4) POPLOG Common Lisp, version 0.95-A, on a GEC Series 63 under UX63.

(5) Tektronics Common Lisp, release 1, on a Tektronics 4405.

Although the morphological analyser, parser and GDE when put together are a large system (approximately 30,000 lines of source code), porting them to a new implementation of Common Lisp should be straightforward, just requiring a few operating system and implementation specific additions to a couple of source files. The distribution tape contains instructions giving details of the additions that will be necessary.

Below are timing figures for seven typical operations that might be performed during grammar development. The figures are for four of the Common Lisp implementations, and also for an implementation of Cambridge Lisp on an Acorn Cambridge Workstation (ACW). Units are seconds of CPU time excluding garbage collection.

	HP 9000 CL	Sun 3/160 KCL	Xerox CL	GEC 63 POPLOG	ACW CamLisp
(1) small read	4.5	7.6	61.3	20.0	33.8
(2) small compile	2.1	3.6	10.2	17.2	6.5
(3) small parse	<0.5	<0.5	1.3	1.5	0.7
(4) big read	21.9	81.3	265.0	110.9	188.5
(5) big compile	186.2	738.0	933.7	2516.3	762.7
(6) big word lookup	2.8	11.8	15.8	20.5	11.4
(7) big parse	4.8	17.5	96.4	102.8	17.3

The first three tests involve the small example grammar given in appendix 1. Test one measures the time taken to read the file into the GDE session, test two the time to compile the grammar, and three the time taken to parse the sentence 'fido costs a pound'. The fourth test is to read in the file of basic feature etc. definitions for a version of the large Alvey Tools Grammar Project grammar, and the fifth to compile it. Test six is to look up the word 'believed' in the lexicon corresponding to the grammar, and seven to parse the sentence 'fido eats more bones than kim does' (not including the time taken in looking up words).

Appendix 5 - Introductory Guide (version 1.22)

This introductory guide is for a version of the GDE which is later than the one described in the rest of this report. The major difference is that semantic formulae may be associated with ID, PS and metarules, and semantic types with syntactic categories in category and lexical category declarations. After a sentence has been parsed, it is thus possible to view its semantic representation (formed from the composition of the semantics associated with each rule in the parse tree).

In this later version of the GDE, the bug making it impossible to view PS rules resulting from the application of linear metarules has been fixed. The option for unrestricted unification in the parser and generator now works properly, as also do rules with no daughters, and traces which appear as first daughters.

THE GRAMMAR DEVELOPMENT ENVIRONMENT
INTRODUCTORY GUIDE

John Carroll, University of Cambridge Computer Laboratory, January 1988

The Grammar Development Environment (GDE) is a software tool designed to make it easy to experiment with and develop Natural Language grammars. This document gives a brief description of the formalism in which grammars for it should be written, and enough information about the commands it accepts for simple use of the system. A more detailed and comprehensive user manual will shortly be available.

1. An Introduction to the Grammar Formalism

The grammar formalism is feature-based and to some extent resembles Generalized Phrase Structure Grammar (GPSG). Syntactic categories, as in GPSG, consist of an unordered collection of feature-value pairs. Each feature in a category may have a 'proper' value, such as + or -, or a variable value (distinguished from proper values by starting with @) which will get instantiated via unification during parsing. A category is written with feature-value pairs separated by commas, and the whole category enclosed in square brackets, e.g.

```
[N +, V -, BAR 2, PER @per, PLU @plu]
```

A grammar contains a number of different types of rules and declarations; the most important of these are feature, alias, lexical category and word declarations, and phrase structure (PS) rules.

1.1 Feature Declarations

Feature Declarations define the feature system used by the grammar, for each feature encoding the possible values it may have. (Variable values need not be mentioned). In the following examples of feature declarations, the feature `PER` is declared to have three possible values, and `SLASH` is declared to be category-valued.

```
FEATURE PER {1, 2, 3}
FEATURE SLASH CAT
```

1.2 Alias Declarations

Aliases are a convenient abbreviatory device for naming categories and feature complexes. For example, instead of writing `[N +, V -, BAR 2]` everywhere in the grammar where a noun phrase is intended, the declaration below allows one to write just `NP`, and indeed `[N +, V -, BAR 2, PER 3, PLU -]` could be written as `NP[PER 3, PLU -]`.

```
ALIAS NP = [N +, V -, BAR 2].
```

1.3 Lexical Category Declarations

Lexical Category (LCategory) Declarations define a particular category as consisting of a given set of features. These declarations are used to expand out (by adding features with variable values) the partially specified categories which typically appear in word definitions and PS rules.

The declarations also allow categories to be annotated with their semantic type; when semantic formulae corresponding to PS rule daughters are composed, the semantic types of the daughters (as determined from LCategory declarations) are checked for consistency. The following declaration says that all noun phrases in the grammar will have the features PER, PLU and POSS with variable values unless otherwise assigned proper ones. The semantic type of NPs is expected to be a function from functions from entities to truth values to truth values.

```
LCATEGORY Nominal : NP => {PER, PLU, POSS} : <<e, t>, t>.
```

1.4 Phrase Structure Rules

Phrase Structure rules are usually the main component of a grammar. One or more semantic formulae may be associated with each rule, so in the example below the meaning of the S is derived from the application of the meaning of the NP applied to that of the VP.

```
PSRULE FiniteS : S --> NP VP : (1 2).
```

1.5 Word Declarations

Words are not part of the grammatical formalism proper, but may be defined to help test a grammar under development. A word declaration consists of the word, followed by one or more syntactic categories to be associated with it. Each category may be followed by an expression denoting the meaning of that sense of the word. E.g.

```
WORD cats : N[-POSS, PLU +, SUBCAT NULL] : (plural cat),
            N[+POSS, SUBCAT NULL] : (belong cat).
```

2. Some Basic Commands

A grammar consists of declarations and rules of the types outlined in the previous section. The grammar is normally initially created, built up and declarations in it manipulated using GDE commands such as 'input' and 'delete'; before exiting from the GDE the 'write' command is usually issued to save a copy of the grammar to a file on disk. The 'read' command may be used at the beginning of a subsequent session to retrieve the grammar from the disk file in order to continue developing it.

Most things typed to the GDE may be abbreviated, so that, for instance, instead of typing 'input', one could type just 'i'. The general rule is that commands and command options may be typed in either upper or lower case, and be abbreviated as far as possible while still remaining unambiguous.

2.1 Input

The 'input' command allows a declaration or rule to be added to the grammar, or an existing one changed. The type of declaration or rule being input must first be specified (i.e. one of Feature, Alias, LCategory, PSrule or Word), and then the name of the particular declaration. Each rule or declaration has a name which is unique to a declaration of that type (so the name of the PS rule in the example in section 1.4 is FiniteS and no other PS rule may be called FiniteS), and this name must be used in GDE commands to refer to that declaration. So, for example, when using the GDE, the feature PER in section 1.1 could be defined by typing (user input in bold):

```
Gde> input
Construct type? fe
Feature declaration? PER {1, 2, 3}
Add input item to file gram/defs? y

Defining feature: PER
```

Notice that the construct type, 'feature', has been abbreviated by the user to just 'fe'. The GDE prompts for more information when it requires it, but the more experienced user may bypass the prompts by typing commands and all their options and arguments on one line.

2.2 View

The 'view' command is used to selectively display grammar rules and declarations. As with the 'input' command, the type of the grammar construct concerned must be specified. Declaration and rule names may be "wildcarded" with the character '*', so the second command below would display all the PS rules in the grammar.

```
Gde> view fe PER
Gde> view ps *
Gde> view all altered
```

2.3 Edit

The 'edit' command enables grammar declarations and rules to be modified. It prints out the current definition of the declaration being changed, and expects the new version to be entered, using a combination of mouse double clicking and right clicking with the SHIFT key depressed to pick up parts of the old definition (see the UNIX manual entry for *hpterm* for details of how to do this), and typing at the keyboard to enter the new parts of the definition. E.g.

```
Gde> e lc Nominal
Current definition is:
Nominal : NP => {PER, PLU, POSS}
New definition?
Nominal : NP => {PER, PLU, COUNT, POSS}
```

2.4 Delete

The 'delete' command deletes one (or more than one if used in conjunction with wildcarding) definitions from the grammar. E.g.

```
Gde> delete word cats
```

2.5 Parse

The main way of testing a grammar is to attempt to parse sentences. Issuing the 'parse' command starts a new command loop (exited by 'quit') at which sentences may be typed which the GDE will try to parse with respect to the current grammar. A valid parse for a sentence typed is taken to be a complete constituent (of any type) which covers all of the input. A bracketting representing the syntactic structure is printed for each such parse.

Parse trees may be looked at in more detail using the parser 'view' command, giving it the 'full' option. This displays the trees in graphical form with rule names and categories at each node. 'View semantics' prints out the semantic translations of the parses. A semantic interpretation module can be interfaced to the GDE by defining a Lisp function 'interpret-sentence'; issuing the 'interpret' command will call this function with one argument, a list of the

formulae derived from the last sentence parsed.

2.6 Write

The 'write' command may be used, usually at the end of a GDE session, to save to disk all the grammar declarations and rules associated with a file.

2.7 Read

The 'read' command takes a file containing the text of a grammar and defines the declarations and rules in the file, adding them to those which have already been defined in the current session.

2.8 Quit

Exits from the GDE.

3. An Example Grammar

The grammar in this section is a small categorial grammar. It does not attempt to deal with semantics, and is included only to give a more coherent example of the declaration and rule types outlined in section 1, and because it is the grammar used in the example GDE session in the next section.

```
;;; File gram/cat - an example categorial grammar. The character
;;; '\' has to be escaped since it itself is the escape character.

FEATURE X CAT
FEATURE SYN{NP, S}
FEATURE DIR{/, \\}
FEATURE RES CAT
FEATURE ARG CAT

ALIAS NP = [X [SYN NP]].
ALIAS S = [X [SYN S]].
ALIAS NP/NP = [X [DIR /, ARG NP, RES NP]].
ALIAS S\\NP = [X [DIR \\, ARG NP, RES S]].
ALIAS S\\NP/NP = [X [DIR /, ARG NP, RES S\\NP]].

PSRULE LAPPL : ;; Left application - variables ensure that the
               ;; category of the 1st daughter is the same as
               ;; the argument of the 2nd, and the result of the
               ;; 2nd is the mother category.
   [X @x] --> [X [SYN @sn]]
      [X [DIR \\, ARG [X [SYN @sn]], RES [X @x]]].
PSRULE RAPPL : ;; Right application - argument of 1st daughter =
               ;; 2nd daughter, result of 1st daughter = mother.
   [X @x] --> [X [DIR /, ARG [X [SYN @sn]], RES [X @x]]]
      [X [SYN @sn]].

WORD john : NP.
WORD dances : S\\NP.
WORD mary : NP.
WORD loves : S\\NP/NP.
WORD the : NP/NP.
WORD cat : NP.
```

4. Invoking the GDE and an Example GDE Session

To start up the GDE from UNIX on one of the HP Bobcats, just type the command 'gde'. This first does a remote copy of the GDE Lisp object files (from Ocelot) onto the local machine's disk, then invokes Lisp, loads the files into Lisp, and finally starts up the GDE command loop. By default, Lisp is invoked by executing 'cl', but if the 'gde' command is followed by a set of arguments, these are assumed to start up a Lisp session and are executed instead of 'cl'. So, for example, if you want to run the GDE in a new window, you can execute

```
$ gde hpterm -e cl
```

The following is part of a GDE session, and should help to illustrate some of the commands described in section 2 and the situations in which they may usefully be employed.

```
Gde> read gram/cat
File read

Gde> parse
2 ID rules, 0 metarules, 0 propagation rules, 0 default rules
0 expanded ID rules, 2 phrase structure rules

Parse>> mary loves the cat

240 msec CPU, 6000 msec elapsed
10 edges generated
1 parse

(mary (loves (the cat)))


Parse>> view full

      LAPPL
        S
       . .
      .   .
     .     .
  mary   RAPPL
   NP    S\\NP
          . .
         .   .
        .     .
     loves   RAPPL
   S\\NP/NP    NP
                . .
               .   .
              .     .
            the     cat
          NP/NP    NP

Parse>> quit

Gde> input word dog : NP.
Add input item to file gram/cat? y

Defining word: dog

Gde> p

Parse>> mary loves the dog
```

```
40 msec CPU, 0 msec elapsed
10 edges generated
1 parse

(mary (loves (the dog)))


Parse>> quit

Gde> write gram/cat
Backing up file gram/cat
Writing file gram/cat

Gde> q
```

5. Problems

The most common problem initially when developing a grammar is likely to be errors in the syntax of declarations entered using the 'input' command. The GDE will not accept an ill-formed declaration and will print out an error message giving an indication of what part of it was incorrect. See sections 1 and 3 for examples of well-formed declarations, and 2.1 for an example of actually inputting a declaration to the GDE.

Constructing the grammar in a file using a text editor is an alternative mode of working to developing it inside the GDE using the 'input' and 'write' commands. The main drawback to the method is that syntax errors in the grammar can only be detected when the file is read into the GDE. At the first such error, the GDE abandons the reading of the file, resulting in an incomplete grammar in memory. The only thing to do if this happens is to issue the 'clear' command to remove the grammar from memory, correct the file containing the error, and try to read it in again.

The grammar must be complete and consistent before any sentences may be parsed: issuing the 'parse' command may at first result in errors such as "feature not defined". These messages should be self-explanatory, and once the faults they are referring to are all dealt with the grammar will be accepted and the parser command loop entered. Even when this point has been reached it is rare that sentences are correctly parsed first time. A couple of reasons why sentences may fail to be parsed when they really should be are:-

(1) One or more of the words in the sentence have not been defined – quit from the parser command loop and use the 'input' command to define them.

(2) PS rules have incompatible feature specifications in categories that should match – compare the top categories of complete constituents with the daughters of the rules that should dominate them. The 'view' command is useful here, since it has an option 'full' which shows PS rule categories in detail including features which have had variable values added by LCategory rules. Matching in the parser is by term unification, so to match, two categories must have exactly the same features, with compatible values of course.

If 'view semantics' in the parser prints an unexpected semantic result, the semantic formula before it was lambda-reduced may be viewed by typing the command 'view semantics unreduced'. This should help identify the source of the error.

The GDE is a powerful system, and is able to do most reasonable things to do with grammars and their development. If there is something you want to do but don't know how to ask for it, issuing the 'help' command to the main GDE command loop or the parser loop will print a brief summary of available commands and some of the options they accept. The user manual contains a full description of everything the GDE and the grammar formalism will do, and will shortly be available as a University of Cambridge Computer Laboratory Technical Report.