A Development Environment for Large Natural Language Grammars

John Carroll, Ted Briscoe
(jac / ejb @cl.cam.ac.uk)

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

Claire Grover
(grover@cogsci.ed.ac.uk)

Centre for Cognitive Science, University of Edinburgh
2 Buccleuch Place, Edinburgh, EH8 9LW, UK

July 1991

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. (However, it is also being used to help teach students on courses in Linguistics and Computational Linguistics). This report describes the grammatical formalism employed by the GDE, and contains detailed instructions on how to use the system[1].

The GDE is implemented in Common Lisp; the source code is available as part of the 'Alvey Natural Language Tools' from the University of Edinburgh Artificial Intelligence Applications Institute.

[1]This report supersedes University of Cambridge Computer Laboratory Technical Report no. 127 which describes a previous version of the GDE.

Contents

Chapter 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 syntactic rules written in a metagrammatical formalism, developing a corresponding semantic component, 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. Chapter 3 describes the commands available, and chapters 4 to 6 go into more detail about three of the major components: the parser, the generator, and the morphological analyser.

The ability to define a semantic component for a grammar is a new addition to the GDE. It allows a compositional semantics to be built within the higher order lambda calculus. (However, other user-defined approaches are possible: constructing representations that may be interpreted procedurally to update a discourse context, for example).

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. Chapter 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 bold type).

```
Gde> read gram/example
File read

Gde> parse
17 ID rules, 1 metarules, 9 propagation rules, 5 default rules
** Warning, multiple match between VP/TAKES_TWONP and PASS
29 expanded ID rules, 30 phrase structure rules

Parse>> fido costs a pound

17 msec CPU, 33 msec elapsed, 5556 heap memory
17 edges generated
1 parse

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

Parse>> a pound is cost by fido

50 msec CPU, 9316 heap memory
31 edges generated
1 parse

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

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.

[1]Appendix A contains a listing of the grammar used in this example session, and also a listing of a second grammar containing a semantic component.

```
Parse>> view rules

             S
           .    .
          .        .
         .             .
     N2/DET/1     VP/BE_AUX1
       . .           .   .
     .   .         .       .
    .     .      .           .
    a   pound   is   VP/NOPASS(PASS/+)
                          . .
                       .    .
                    .       .
                  cost      PP
                             .
                              .
                               .
                          PP/TAKES_NP
                             . .
                           .   .
                         .     .
                        by   N2/PN
                               .
                               .
                               .
                             fido

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/TAKES_TWONP(PASS/-)
VP/TAKES_TWONP(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, a new attempt to parse the last sentence indeed fails as it should.

```
Gde> p
17 ID rules, 1 metarules, 9 propagation rules, 5 default rules
23 expanded ID rules, 24 phrase structure rules

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

33 msec CPU, 5620 heap memory
22 edges generated
No parses

Parse>> q

Gde> generate

Gen>> auto bracket 2
(fido)
(pound a)
(a pound)

Gen>> q

Gde> write gram/example
Backing up file gram/example
Writing file gram/example
```

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. Boguraev *et al.* (1988) discuss more of the features which make the GDE a powerful and easy-to-use environment for grammar development.

1.2 Background

The GDE was written to support the development of a large grammar of English (Briscoe *et al.*, 1987a; Grover *et al.*, 1989), one of the 'Alvey Natural Language Tools' projects. Briscoe *et al.* (1987b) give a summary of this project. The other collaborating projects implemented a GPSG parser (Phillips & 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 document will only very briefly touch on this capability (see Appendix B).

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. The implementations of Common Lisp in which they have been used are listed in Appendix D.

The Alvey Natural Language Tools, comprising the Common Lisp source code of the GDE, a wide-coverage grammar and lexicon for English, together with the morphological analyser and parser, are available for a nominal fee to UK universities and to commercial participants of Alvey projects from the University of Edinburgh Artificial Intelligence Applications Institute (at 80 South Bridge, Edinburgh EH1 1HN, UK). Other UK and non-UK organisations can also obtain the Tools for research purposes for a moderate one-off fee. To date, over fifty establishments have obtained a copy of the Tools.

This document describes version 1.32 of the GDE. This is the version which is distributed in the third release (July 1991) of the Tools.

Chapter 2

The Metagrammatical Formalism

This chapter 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 2.15 below) is used to compile the metagrammar into an 'object' grammar, by default a fixed-arity term 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. During metagrammar compilation, the GDE automatically checks consistency between the different types of rule in the metagrammar: the checks made are detailed in the section describing the construct involved.

The rest of this chapter 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 the Kleene operator '+', and zero or more times if followed by the '*' operator.

4. All items generated by the '+' and '*' Kleene iteration operators should in general be taken as being separated by commas although this is not explicitly expressed in the specifications; in some cases separation by just spaces changes the meaning. For example, a rule that would otherwise be an Immediate Dominance rule (section 2.7), but lacks commas between the daughters is in fact a Phrase Structure rule (section 2.8).

5. Comments (introduced by ';' and carrying on to 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`. Names of rules, features, values etc. may contain any of the characters

```
a-z A-Z 0-9 _ # $ ^ | / - + > '
```

To appear in rule names etc., each occurrence of any of the following characters must be preceded by a backslash.

```
Tab Space " ! & ( ) = ~ ` @ { [ * : } ] , < > . ? ;
```

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 forth. 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 2.14 for the definition of <category>. This category may itself contain occurrences of other aliases. As an 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 Declarations

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> : <category-feature-spec> (: <semantic-type>)+ .
<category-feature-spec> ::= <pattern-category> => <feature-set> |
(<feature-name>+) <pattern-category> => <feature-set>
<category-name> ::= <atomic-symbol>
<feature-set> ::= { <feature-name>+ } | <set-name>
<feature-name> ::= <atomic-symbol>
<set-name> ::= <atomic-symbol>
<semantic-type> ::= e | t | * | < <semantic-type> , <semantic-type> >

See section 2.14 for the definition of <pattern-category>. Category declarations without the optional list of feature names in the category feature specification ensure that top level categories which match the given pattern category contain the set of features specified. When the list of feature names is present, the names are interpreted as a 'path' (so each of the features must be category-valued) and the procedure is applied to all categories at the end of such a path. Category declarations apply both to categories in ID rules and to word definitions (section 2.13).

A category can be assigned one or more semantic types in a category declaration; the GDE checks that the semantic part of each rule (see sections 2.7 and 2.8) or word (section 2.13) which contains such a category is using it in a way that is compatible with at least one of the types declared for that category. If not, a warning message is output. Warnings are issued for arguments to functions of the wrong type, e.g.

```
*** Warning, incompatible argument of type t to function of
type <e, t> in S
```

This warning is saying that in rule S, the semantics of one of the daughters whose category had been declared as being of type <e, t> (a function from entities to truth values) is being applied to the semantics of another daughter declared as being of type t: the function demands that the argument should be of type e). A warning is also issued if the type of the whole semantics of the RHS of a rule is incompatible with the type declared for the mother category. If some latitude is needed when defining types, the type * (standing for any arbitrary type), can be used. Types are only used by the GDE for consistency checking; they can be omitted from a grammar without in any way affecting what the grammar actually does.

The name of a set may be used on the RHS of a category declaration, 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, giving in this case V2[AGR N2[PER *@per*, PLU *@plu*]].

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

2.5 Extension Declarations

Some features, such as SLASH in GPSG, 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 Top Declarations

A Top Declaration consists of a number of categories; it tells the GDE that when the parser returns the set of parses for a sentence or phrase, only those parses whose top node matches (i.e. is an extension of) one of the categories should be retained, and that the rest should be ignored.

<top-declaration> ::= <pattern-category>+ .

See section 2.14 below for the definition of <pattern-category>. For example, the declaration

```
S[FIN +, COMP NORM], N2.
```

says that only parses whose top node matches either `S[FIN +, COMP NORM]` or `N2` should be retained. If no top declaration appears in a grammar, then all complete parses are retained. (This is equivalent to defining a single top category of `[]`).

2.7 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>+ <semantic-term>* .
<idrule-name> ::= <atomic-symbol>
<rhs-term> ::= <category> |
(<category>) | (<category>)+ | (<category>)*
<semantic-term> ::= : <semantic-condition>* <semantic-form>
<semantic-condition> ::= <category-index> = <pattern-category> ,
<semantic-form> ::=
<atomic-symbol> | <category-index> | (<semantic-form>*)
<category-index> ::= <integer>

See section 2.14 for the definitions of <category> and <pattern-category>. If the RHS categories in the rule are separated by commas, then the rule will later be subject to linear precedence (LP) 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 2.8). As an example, 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].
```

A daughter category containing the feature NULL (with any value) is treated during parsing as a gap.

A semantics may optionally be associated with an ID rule. The GDE can use this to construct a semantic representation of a parsed sentence or phrase (as long as all the rules and words invloved in the parse had previously been assigned a semantics). The representation will be a composition of the semantics of the rule at each non-terminal node and of the word at each terminal node, in direct correspondence to the structure of the syntactic parse tree. When the semantic representation is built up, integers in semantic formulae are taken to be indices which refer to the semantics of syntax tree nodes dominated by the corresponding rule daughters; thus the ID rule

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

specifies that the semantics of the parse node representing the `S` mother is the result of applying the semantics of the node for the first daughter (as it occurs in the rule definition), the `NP` one, to the semantics of the node for the second daughter, the `VP` one. Whatever the final ordering in the object grammar, the numeric indices refer to daughter categories in the order in which they are written in the rule definition. In the example above, the semantics of the `NP` daughter will always be applied to that of the `VP` daughter even if the `VP` daughter were to end up ordered first. So if the semantics of the `NP` node were `(lambda (p) (p john))`, and of the `VP` node, `dance`, then the resulting semantic representation would be `((lambda (p) (p john)) dance)`. Simplifying this by *lambda-reduction* (more strictly beta-reduction), we get `(dance john)`. The GDE performs lambda-reduction at parse time and displays semantic formulae in fully reduced form, by default.

In lambda-reduction, the GDE assumes that predicate formulae over variables are of the form `(<predicate-name> (<variable>) <body>)`, where the variable is an atom. Every occurrence of this atom inside the body is taken to refer to the same variable (except of course inside the body of another embedded predicate formula which is over the same variable). Variables with the same name but 'bound' by different formulae are guaranteed to remain distinct; during beta-reduction the GDE renames them to keep them distinct when it substitutes one formula inside the body of another. For example, the formula

```
(exists (y)
   (and (dog' y)
      ((lambda (x) (exists (y) (and (cat' y) (love' x y))))
       y)))
```

reduces to

```
(exists (y)
   (and (dog' y) (exists (y1) (and (cat' y1) (love' y y1)))))
```

and not

```
(exists (y)
   (and (dog' y) (exists (y) (and (cat' y) (love' y y)))))
```

Several alternative semantic formulae (separated by colons) may be associated with any one ID rule. Alternative formulae cause several semantic representations to be constructed, one for each combination of alternatives. Conditions may be put on one or more formulae to restrict their applicability to particular syntactic contexts, the tests being made on the basis of the instantiation of feature values at parse (or generation) time. A condition is essentially a specification of the category of the mother or one of the daughters of the rule; it consists of a category index (an integer with `0` refering to the rule mother, `1` to the first daughter as it occurs in the rule definition etc.), followed by an equal sign and lastly a pattern category. Several conditions (separated by commas) may be made on a single formula. A formula is applicable if the categories in all of the conditions match the corresponding mother and daughter node categories at parse time. For example, the rule

```
NP/N1_PLU : NP[DEF -] --> H1[PLU +] :
   1 = [PRD -], (lambda (Q) (all (x) (if (1 x) (Q x)))) :
   1 = [PRD +], (lambda (x) (1 x)).
```

has two mutually-exclusive alternative semantic formulae, selected on the basis of the value of the feature `PRD` on the daughter node at the end of the parse.

2.8 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 linear precedence rules, for example in the following rule encoding 'heavy NP movement' in English.

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

If the linear precendence rules encoded the general rule that NPs precede PPs, then the rule in this example has to be a PS rule rather than an ID rule.

2.9 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 2.14 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 2.11).

```
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 NOMINALHEAD set being added with the same variable value to the N2 mother and the head daughter.

'Gap threading' regimes may be implemented quite elegantly using propagation rules with more than one value restriction. The propagation rule

```
VP --> V, NP, XP.
   GAPIN(0) = GAPIN(2), GAPOUT(2) = GAPIN(3),
   GAPOUT(3) = GAPOUT(0).
```

when applied to an ID rule would set up that rule to pass a gap value from its mother into its second daughter, one out again from the second daughter into the third daughter, and one out from the third daughter back through the mother.

2.10 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 2.14 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.
```

2.11 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>+ <semantic-term>* .
<rhs-term> ::= <category> | W | U |
(<category>) | (<category>)+ | (<category>)*
<semantic-term> ::= : <semantic-condition>* <semantic-form>
<semantic-condition> ::= <category-index> = <pattern-category> ,
<semantic-form> ::=
<atomic-symbol> | <category-index> | (<semantic-form>*)
<category-index> ::= <integer>

See section 2.14 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. (However if this happens the GDE prints a warning). 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] ).
```

In general, when a metarule is applied to an ID or PS rule, the semantics of the rule is changed. The GDE allows formulae to be associated with metarules; the semantics of each output rule is the result of first applying each metarule formula to each input ID/PS rule formula, and then removing from the resulting formulae those which either refer to daughters which have been deleted, or fail to mention all the daughters which are present. Metarule formulae may contain numeric indices; ones that appear as bound lambda variables refer to the semantics of the category at that position in the LHS (pattern) rule, and ones that are free refer to the semantics of the category at that position in the RHS (output) rule. For example, the bound 2's in the metarule

```
PASS : VP --> W, N2. ==> VP[PAS] --> W, ( P2[by] ) :
   (lambda (s)
      (lambda (y)
         ((lambda (x) ((lambda (2) (s x)) y)) 2))) :
   (lambda (s)
      (lambda (y)
         ((lambda (2) (s (some (x) (entity x)))) y))).
```

refer to the semantics of the LHS rule N2 daughter, and the free 2 in the first semantic formula refers to the semantics of the optional P2 in the output rule. When this metarule is applied to the ID rule

```
VP/TAKES_NP : VP --> H[SUBCAT NP], N2:
   (lambda (x) (1 x 2)).
```

the two semantic formulae produced would be (labelling the numeric indices with the categories whose semantics they represent):

```
((lambda (s)
    (lambda (y)
       ((lambda (x) ((lambda (2) (s x)) y)) 2)))
 (lambda (x) (1[H] x 2[N2])))
=
(lambda (y) (1[H] 2[P2] y))
```

and

```
((lambda (s)
    (lambda (y)
       ((lambda (2[N2]) (s (some (x) (entity x)))) y)))
 (lambda (x) (1[H] x 2[N2])))
=
(lambda (y) (1[H] (some (x) (entity x)) y))
```

2.12 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 2.14 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.

2.13 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> : <word-category>+ .
<word> ::= <atomic-symbol>
<word-category> ::= <category> (: <semantic-form>)*
<semantic-form> ::= <atomic-symbol> | (<semantic-form>*)

See section 2.14 for the syntax of <category>. Each syntactic category may optionally be followed by one or more semantic formulae (separated by colons). 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] : (plu cat'),
       N[+POSS, PRO -, PN -, SUBCAT NULL] : (poss cat').
```

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, category rules (section 2.4) are applied to the definition of the word to flesh out the categories in it.

2.14 Rule Patterns and Grammatical Categories

<pattern-rule> ::= <pattern-category> --> <rhs-pattern-item>+ .
<rhs-pattern-item> ::=
    <pattern-category> | W | U | (<pattern-category>) |
    (<pattern-category>)+ | (<pattern-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]`). The `@` and `~` specifications may be included in a list of possible values (e.g. `[CASE (~, @, ACC)]`, meaning the feature `CASE` is either not present, has a variable value, or has value `ACC`). Examples of ordinary categories are

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

and more examples 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.

An unfortunate ambiguity in the notation is 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 user may request the GDE to display the category after any aliases that were present have been expanded out into the constituent feature / value pairs they represent: doing this will show what course of action has been taken.

2.15 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 in figure 2.1.

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 2.11,

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

would match an ID rule introducing ditransitive verbs, such as

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

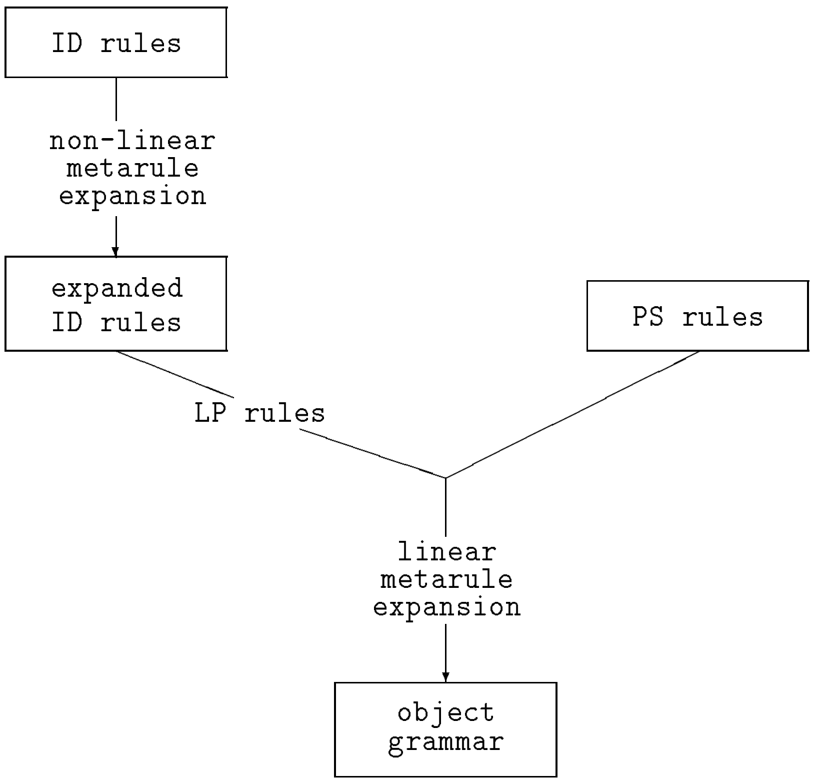

Figure 2.1: Metagrammar Compilation

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 syntactic parses), so this flag initially has the value ON. For grammars with semantics, this flag should probably be set to OFF, since the semantic representations built from the two rules would be different.

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/NP_NP` 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.

During metagrammar expansion, the features `NULL` and `H` are treated specially. A daughter which contains the feature `NULL` (with any value) is marked in the object grammar as being a gap. So for example, the ID rule

```
VP[SLASH N2] --> V, N2[NULL +, SLASH N2].
```

has a gap as its second daughter. At parse time, a gap is inserted whenever the rule is applied. The feature `H` is treated specially in that it is never carried forward into an object grammar: it may be used in GPSG-style grammars to mark the 'head' daughter in a rule.

2.16 Metagrammar Compilation and Semantics

Rules containing optional daughters will usually have alternative formulae for their semantics: one formula for when the daughter is present and another for when it is absent. For example, the rule

```
VP/TAKES_NP_PASS : VP[PAS] --> H[SUBCAT NP], ( P2[by] ) :
   (lambda (y) (1 2 y)) :
   (lambda (y) (1 (some (x) (entity x)) y)).
```

would be normalised by the GDE into the two rules

```
VP/TAKES_NP_PASS/+ : VP[PAS] --> H[SUBCAT NP], P2[by] :
   (lambda (y) (1 2 y)) :

VP/TAKES_NP_PASS/- : VP[PAS] --> H[SUBCAT NP] :
   (lambda (y) (1 (some (x) (entity x)) y)).
```

When rules with optional daughters are normalised, each resulting rule will contain all the formulae from the original rule which both refer to daughters which still exist (i.e. ones which were not optional, or were optional but have not been deleted) and also mention all the daughters present.

Semantic formulae in the grammar can be made more readable by using 'semantic operators'. These operators have to be defined directly in Lisp; for example the operators

```
(setf (get '|passive-operator-1| 'semantic-operator)
   '(lambda (s)
       (lambda (y)
          ((lambda (x) ((lambda (2) (s x)) y)) 2))))

(setf (get '|passive-operator-2| 'semantic-operator)
   '(lambda (s)
       (lambda (y)
          ((lambda (2) (s (some (x) (entity x)))) y))))
```

could be defined to make a version with semantics of the `PASS` metarule more readable. The metarule could then be defined as:

```
PASS : VP --> W, N2. ==> VP[PAS] --> W, ( P2[by] ) :
   passive-operator-1 : passive-operator-2.
```

Semantic operators must be defined before any metarule, ID rule or word which uses them is defined. The best way to manage them is to have them in a file and, at the start of each GDE session to load the file using the GDE lisp command, "!" (section 3.4.9) in conjunction with the Lisp "load" function, e.g.

```
!(load "gram/operators")
```

Chapter 3

Commands

Section 1.1 presented a typical GDE session in which a few of the more basic GDE commands were issued. This chapter 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 prompting for more input by just typing carriage return. All commands and options may be abbreviated, usually to their first letter, but in some cases to the first two or three; just enough to distinguish the command name or option from the 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", "EXtension", "Top", "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 constructs (chapter 6) "ENtry completion rule", "MUltiplication rule", and "CONsistency check".

Every declaration or rule in a grammar has a name, 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 single character may occur 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 chapter 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. Other 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. E.g.

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

If the construct is already defined, the GDE displays the existing definition and asks if it should really be redefined; if the construct is currently undefined, the GDE asks whether it should be associated with the file that the last declaration went in (or the last file that was read in if that was more recent). If the answer is ‘no’, the GDE prompts for the name of a file to which it should be written when the metagrammar is later saved to disk. When inputting a word that already has a definition, the GDE can be made to add the new definition to the already existing one by saying ‘no’ to the question asking whether the old definition should be replaced, but ‘yes’ to a further question asking whether the new definition should be added to the existing one.

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 carrying on to 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 old definition is first displayed, and the user is prompted for a new definition. On machines which allow the user to copy segments of text from one part of the screen and use them as new input, this makes for much less typing when creating a new definition.

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 using the feature, set or alias in question.

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 category statements, morphology system entry completion and multiplication rules and consistency checks (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.

On input, ID, PS and metarule names are checked to make sure that they could not be confused with the name of a rule resulting from an expansion of optional categories, a multiple metarule match or a multiple linearisation. So, for example, it is not possible to call an ID or PS rule `S/1`; otherwise confusion would be inevitable if there were also a rule in the grammar called `S` which had multiple linearisations (one of which would also be called `S/1`).

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 command 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 command loop. These commands are described in chapter 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

Invokes the generator command loop. Specialised commands (described in chapter 5) for controlling the generator are available inside this command loop.

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. A filename containing space characters may be entered by preceding each space with the backslash character.

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

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 access rules more efficiently. (The access 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 new command 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 may occur if the file had been inaccurately text-edited by the user 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 can have either the value ON or the value OFF. The flags are:

Defining messages
: Controls whether messages of the form 'Defining <construct-type>: <construct-name>' are issued when a new definition is input to the GDE. Can be set to OFF if it is felt that the "read" command, in particular, is generating excessive amounts of output.

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. If OFF, just a warning is issued.

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 system 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 to attempt a full analysis of the word.

Word structure
: On a morphology system lookup, controls whether the internal morpheme structure of the result is passed up to the GDE. Must be ON for semantics to be returned from the lexicon.

Ecrs before multiply
: Controls the order in which entry completion rules and multiplication rules are applied to word definitions

Gde word grammar
: For word lookups, allows a word grammar written in the GDE metagrammatical formalism to be used in conjunction with the existing morphology system spelling rules and morpheme retrieval facility.

TErm unification
: Whether the parser and generator match grammatical categories using fixed-arity term unification or unification with bidirectional extension. (Note that this flag has nothing to do with what type of unification the morphological analyser is using to interpret its word grammar).

LR1 parse
: If an alternative (LR1) parser is present, the flag controls whether it is to be used instead of the standard chart parser.

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 2.15 for a description of the 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 meet this requirement, 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 is therefore 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 NL Tools parser. (The contents of the resulting file are described in Appendix C). 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 that have been looked up by the morphology system in the current session. The intention is that a stand-alone version of the Tools parser without an interface to the morphology system could be run using just the definitions of these words as data. 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.

Chapter 4

The Parser

The parser is invoked by the "parse" command (section 3.1); this command starts up a new command loop with a special set of commands, described below. Control returns to the main GDE command loop on exit from the parser loop. The parser that is used by default[1] is a chart parser using a bottom-up strategy, based on the one produced by the Alvey NL Tools parser project (Phillips, 1986; Phillips & Thompson, 1987). The basic chart parsing algorithm used is unchanged, but (in addition to the port from Franz Lisp to Common Lisp) modifications include:

1. A redesign of the unification module to fix a number of bugs. (The bugs usually manifested themselves as the occasional unexpected failure of feature values to percolate down sub-trees or across from one local tree to an adjacent one; they usually meant that grammars using 'gap threading' produced parses that should have been ruled out as invalid because of a clash of values).

2. The option of either fixed-arity term unification or unification with bidirectional extension[2] of grammatical categories.

3. A more general treatment of gaps and of rules with no daughters. (In previous versions, gaps were not detected if they would have occurred within the first daughter of a rule).

4. The saving of chart edges to enable them to be displayed after a parse for grammar debugging purposes.

5. The 'packing' of similar constituents covering the same segment of the input sentence, representing all the alternative structures in a local ambiguity by a single sub-analysis, and performing subsequent processing once only, rather than once for each alternative (see Alshawi *et al.* (1988) for more details).

---

[1] The flag "LR1 parse" (section 3.4) may be used to select an alternative (LR1) parser if one is present in the system; see the Alvey NL Tools release notes for details of this.

[2] Parsing with unification with bidirectional extension is not as fast as it really could be; in the current implementation, the smaller the number of features in the grammar the better the parser performs.

6. Several other efficiency enhancements, resulting in a speedup of a factor of between four and five.

If a word in a sentence being parsed has a category which does not occur as the daughter of any rule, then the parser prints a warning of the form

```
*** Warning, lexical category <category> cannot be consumed by
any rule
```

Only those parses whose top node matches (i.e. is an extension of) one of the categories specified in the current "top" declaration are accepted as valid; if there is no "top" declaration then all parses are accepted. At the end of a 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 command loop accepts.

4.1 Commands

4.1.1 View option

The option should be one of "Bracketings", "LRules", LCategories", "Rules", "FUll", "CAtegory", "Semantics", "FOrm", "COmmon", "Parsed", "Vertices", "Edges", "ACtive", "INactive", "AI', or "IA".

The first five options display the parse tree(s) resulting from parsing the last sentence. If the "bracketings" option is specified, the output is simply an unlabelled bracketing of the words in the sentence; with the "lrules" and "lcategories" options the bracketing is labelled with rule names and category names respectively; 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).

The "category" option may be used to look at a selected category in a parse tree without 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 the options may additionally be given a number (1 meaning first, 2 meaning second, and so on) indicating that only that particular tree is to be displayed. E.g.

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

The "form" option is similar to the "category" option except that it displays the semantic form at a selected node in a parse tree. The result is normally displayed after lambda-reduction has taken place, but this may be overridden by specifying "unreduced" (e.g. `view form unreduced`). The "semantics" option displays the full semantics of the whole parse (again with the "unreduced" option available).

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 parser keeps a history of sentences parsed in the current session, and the "parsed" option may be used to print them out.

The "vertices" option prints out the words in the last sentence parsed, the words numbered to correspond with the numbering of vertices in the chart, and the number of active and inactive edges between each pair of vertices. (This can be a help in finding parts of the grammar which might be slowing the parser down). The remaining options output more 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 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 +]
```

4.1.2 Write option filename

The option should be one of "Bracketings", "LRules", "LCategories", "Rules", "Full", "Semantics", or "Parsed".

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 (see below). Using the "parsed" option makes it easy to build a corpus of sentences representing the coverage of the grammar being developed. The other options write the current set of parse trees to the given file. The "rules" and "full" options may produce several files (with names of the form `file1`, `file2` etc.) for large trees, laid out in a way that should allow large trees to be studied by pasting hardcopy printouts of the files side by side.

4.1.3 Previous

Attempts to parse the previous sentence again.

4.1.4 Fparse option input-filename output-filename

The option should be one of "Numbers", "Bracketings", "LRules", "LCategories", "Rules", "Full", "Semantics", or "Interpret".

Parses all the sentences in the given input file. Each sentence should finish with a full stop, question mark or exclamation mark. Sentences need not start on a new line so the input file can be 'running text'. 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, the output is appended to the end of it. The options produce the same forms of output as in the "view" command; however timing and chart edge statistics are suppressed, making it easier to run file difference utilities on the results of old and new runs to detect unwanted changes in grammar coverage.

4.1.5 Interpret

Calls a user-defined Lisp function `interpret-sentence` on the set of lambda-reduced semantic formulae for all the current parses. The function should take one argument, and will be passed the semantic formulae in a list.

4.1.6 Gc

Forces an immediate garbage collection by the Lisp system. This can be useful when expecting to spend some time inspecting a result from the parser, because it may avoid an automatic garbage collection at a later less predictable or convenient time.

4.1.7 Help

Displays a page of information on commands available in the parser. The commands "shell", "lisp" and "!" are also available in the parser command loop.

4.1.8 Quit

Exits from the parser command loop.

4.1.9 anything else

The input is taken to be a sentence to be parsed. Sentences prefixed with '*' followed by a space are parsed as if the asterisk were not there, but when they are saved to file with the "write parsed" command, the asterisk is preserved. (Thus a corpus may be built up of both sentences which should receive parses, and ones marked by an asterisk which should not).

Chapter 5

The Generator

The generator is invoked by the "generate" command (section 3.1). The top node of the tree to be generated is taken to be the first category in the current "top" declaration, or [] if it has not been declared. Category rules are applied to this category.

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 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 up the 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[+FIN] when the situation

```
              S
        S[-PRD, +FIN,
            AGR N2]
           .    .
        .          .
      .               .
     2          VP/BE_AUX1
N2[+NOM]      VP[-PRD, +FIN,
               +AUX, AGR N2]
                  .      .
               .             .
             .                  .
            4                   5
V[-PRD, +FIN, +AUX, AGR  VP[+PRD, AGR
    N2, SUBCAT PRED]          N2]
```

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,

```
...
 9. VP/NOPASS(PASS/+)        10. VP/NOPASS(PASS/-)
11. VP/TAKES_NP(PASS/+)      12. VP/TAKES_NP(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 clearly the overgeneration in the initial grammar. The following are the commands which may be used to control the generator.

5.1 Commands

5.1.1 View option

The option should be one of “Bracketing”, “Categories”, “Rules”, “Full” or “Semantics”.

The first four of these options display the current state of the generator tree, differing in the amount of detail they show. If the “bracketing” option is specified, an unlabelled bracketing of the tree is output, displaying words (or node numbers if the node has not been expanded that far) on each leaf node; the “categories” option is similar except that it displays the categories on unexpanded leaf nodes. 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. The “semantics” option displays the full semantics of the whole tree.

5.1.2 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.

5.1.3 Automatic 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 number) which puts a limit on the length in words of the sentences to be generated.

5.1.4 Clear

Clears the generator tree. If the tree is not cleared, it is maintained across entries to and exits from the generator.

5.1.5 Help

Displays a page of information on commands available in the generator. The commands "interpret", "shell", "lisp" and "!" are also available in the generator command loop.

5.1.6 Quit

Exits from the generator command loop.

Chapter 6

Using the Morphological Analyser with the GDE

The GDE includes version 3.0 (with the 'unrestricted unification' word grammar option) of the Alvey NL 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 overridden, even if the flag is ON.

The type of lookup that the morphological analyser performs is controlled by the flag "fast morph lookup" (section 3.4). 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

[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.

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:

```
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.

6.3 Word Grammar Semantics

The version of the morphological analyser used by the GDE allows semantics to be associated with word grammar rules. If the GDE flag "word structure" is ON, the computation of the semantics of a parsed sentence descends into the morphological structure of words and composes the constituent morpheme semantic forms (from the lexicon) using the semantic formulae assigned to the word grammar rules involved.

For example, if the morphemes `cat` and `+s` (the latter representing the plural suffix) were defined as

```
(cat cat (N (COUNT +)) cat' ())
(+s +s ((STEM (N (COUNT +))) plu ())
```

and a word grammar rule called `N-SUFFIXES` with two daughters was defined that would combine these two morphemes, then at the end of the word grammar file in a section headed `Semantics`, the declaration

```
(N-SUFFIXES (2 1))
```

would result in the morpheme sequence `cat +s` receiving the semantic analysis `(plu cat')`, i.e. the semantics of the second daughter, `+s`, applied to that of the first daughter, `cat`.

Chapter 7

Errors, Warnings and Bugs

7.1 Errors and Warnings

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

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

and abandons the current command, puts up the next command loop prompt and waits for the next command. In most cases the remedy for the error is obvious, and once corrective action has been taken the 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. This bug is not likely to be fixed in future versions of the GDE.

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. Also, the '=' pattern sometimes returns duplicate definitions.

The results obtained from the semantics for rules with kleene plus or kleene star daughters are not well-defined. Only in the cases where the kleene daughter was expanded by a single node will the behaviour be as expected. The next release of the GDE will handle the semantics of these rules properly.

During metagrammar compilation, if an ID or PS rule contains a feature with a named variable value (e.g. `@x`) and if a metarule which is intended to add another feature to the rule has, as the value to be added, the same named variable, then applying the metarule may result in the two values becoming (presumably unintentionally) bound together in the expanded rule. For example, with

```
IDRULE N2+/PRO2 :
   N2[+SPEC, -POSS, PART @x] --> H[SUBCAT NULL, +PRO, PART @x].

METARULE FOOT9 :
   [V -, BAR 2, ~WH] --> H[PRO +]. ==> [WH @x] --> H[WH @x].
```

the feature `PART` in the ID rule will become bound to the feature `WH` in the expanded ID rule:

```
N2+/PRO2(FOOT9) :
   N2[-POSS, +SPEC, +PRO, PART @15, WH @15] -->
   N[H +, SUBCAT NULL, +PRO, PART @15, WH @15].
```

Acknowledgements

We would like to thank Roger Evans (the current Tools project monitoring officer) for advice and encouragement. Thanks also to Bran Boguraev and Graham Russell for helpful comments on previous versions of this document, and to Derek Bridge, Lee Fedder, Bob Futrelle, Steve Harlow, Ben Johnson-Laird, David Plowman, Steve Pulman, Michelle Wang and Rachel Weiss for reporting bugs and suggesting enhancements to the software.

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, and by SERC / DTI-IED grant 4/1/1261 'Extensions to the Alvey Natural Language Tools'.

References

Alshawi, H., *et al.* (1988) *Interim Report on the SRI Core Language Engine,* CCSRC-005, SRI Cambridge Research Centre.

Boguraev, B., J. Carroll, E. Briscoe & C. Grover (1988) 'Software Support for Practical Grammar Development', *Proceedings of the 12th International Congress on Computational Linguistics,* Budapest, Hungary, pp. 54–58.

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

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

Grover, C., E. Briscoe, J. Carroll & B. Boguraev (1989) *The Alvey Natural Language Tools Project Grammar (Second Release): a Large Computational Grammar of English,* Technical Report No. 162, Computer Laboratory, University of Cambridge.

Phillips, J. (1986) 'A Simple, Efficient Parser for Phrase-Structure Grammars', *SSAISB Quarterly Newsletter, vol.59,* pp. 14–18.

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

Appendix A

Example Grammars

This appendix contains two small grammars as examples of the metagrammatical formalism, its syntax, and how it may be used. They are both in the style of GPSG. The first, a purely syntactic one, was the grammar read in at the beginning of the GDE session in section 1.1; the second one contains a simple semantic component.

```
; File 'gram/example'. 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 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 -].
ALIAS Pas = [VFORM EN, PRD +].

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}

TOP N2, S[+FIN].

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/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].
IDRULE N2/PP : N2 --> N2, P2.

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

PROPRULE PROP_HEAD_V : ; copy value of V from mother to head daughter
   [V (+, -)] --> [H +], U. V(1) = V(0).
PROPRULE PROP_HEAD_N : ; copy value of N from mother to head daughter
   [N (+, -)] --> [H +], U. N(1) = N(0).
PROPRULE PROP_BAR : ; non-lexical heads have same BAR level as mother
   [] --> [H +, ~SUBCAT, ~BAR], U. BAR(0) = BAR(1).
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 DEF_BAR0 : ; lexical heads are BAR 0
   [] --> [H +, SUBCAT], U. BAR(1) = 0.
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) = -.
DEFRULE N_PN : [] --> N, U. PN(1) = -.

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

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

```
; File 'gram/semantics'. A simple grammar with semantics,
; illustrating semantic types on category declarations and
; conditions on rules.

FEATURE N{+, -}
FEATURE V{+, -}
FEATURE BAR{0, 1, 2}
FEATURE MINOR{DET}
FEATURE H{+, -}
FEATURE PER{1, 2, 3}
FEATURE PLU{+, -}
FEATURE PRD{+, -}
FEATURE NTYPE{NAME, PRO, COUNT}
FEATURE DEF{+, -}
FEATURE SUBCAT{INTRANS, TRANS, DITRANS}
FEATURE PAST{+, -}
FEATURE CASE{NOM, ACC}
FEATURE VFORM{FIN, PASS, BSE}
FEATURE ADV{+, -}
FEATURE PFORM{TO, BY}
FEATURE AUX{DO, BE, -}
FEATURE INV{+, -}
FEATURE NULL{+, -}
FEATURE EMPTY{+, -}

SET VHEAD = {N, V, PER, PLU, PAST, VFORM, AUX, INV}
SET NHEAD = {N, V, PER, PLU, NTYPE, CASE, PRD}
SET PHEAD = {N, V, PFORM}
SET AHEAD = {N, V, ADV}

ALIAS S = [V +, N -, BAR 2].
ALIAS VP = [V +, N -, BAR 1].
ALIAS V = [V +, N -, BAR 0].
ALIAS NP = [N +, V -, BAR 2].
ALIAS N1 = [N +, V -, BAR 1].
ALIAS N = [N +, V -, BAR 0].
ALIAS PP = [V -, N -, BAR 2].
ALIAS P1 = [V -, N -, BAR 1].
ALIAS P = [V -, N -, BAR 0].
ALIAS AP = [V +, N +, BAR 2].
ALIAS A1 = [V +, N +, BAR 1].
ALIAS A = [V +, N +, BAR 0].
ALIAS H2 = [H +, BAR 2].
ALIAS H1 = [H +, BAR 1].
ALIAS H0 = [H +, BAR 0].
ALIAS Det = [MINOR DET].

CATEGORY S : S => {INV} : t.
```

```
CATEGORY V : [V +, N -] => {PER, PLU, PAST, VFORM, AUX, INV}.
CATEGORY VP : VP => {} : <e, t>.
CATEGORY Vi : V[SUBCAT INTRANS] => {} : <e, t>.
CATEGORY Vt : V[SUBCAT TRANS] => {} : <e, <e, t>>.
CATEGORY Vd : V[SUBCAT DITRANS] => {} : <e, <e, <e, t>>>.
CATEGORY Vaux : V[AUX (DO, BE)] => {} : <<e, t>, <e, t>>.
CATEGORY N : [N +, V -] => {PER, PLU, NTYPE, CASE, PRD}.
CATEGORY NP : NP[PRD -] => {DEF} : <<e, t>, t>.
CATEGORY NPp : NP[PRD +] => {DEF} : <e, t>.
CATEGORY N1 : N1 => {} : <e, t>.
CATEGORY Nc : N[NTYPE COUNT] => {} : <e, t>.
CATEGORY Nn : NP[NTYPE NAME] => {} : e.
CATEGORY Np : NP[NTYPE PRO] => {} : <<e, t>, t>.
CATEGORY PP : PP => {PFORM} : <<e, t>, t>.
CATEGORY P1 : P1 => {PFORM} : <<e, t>, t>.
CATEGORY P : P => {PFORM}.
CATEGORY A : [N +, V +] => {ADV}.
CATEGORY AP : AP[ADV -] => {} : <e, t>.
CATEGORY A1 : A1[ADV -] => {} : <e, t>.
CATEGORY Adj : A[ADV -] => {} : <e, t>.
CATEGORY Det : Det => {PLU, DEF} : <<e, t>, <<e, t>, t>>.

PSRULE S1 : S[VFORM FIN] --> NP[CASE NOM, PRD -] H1 : (1 2).
PSRULE S2 : S --> H0[INV +, AUX @a, EMPTY -] S[INV +, AUX @a] : 2.
PSRULE VP1 : VP --> H0[SUBCAT INTRANS] : 1.
PSRULE VP2 : VP --> H0[SUBCAT TRANS] NP[CASE ACC, PRD -] :
   (lambda (x) (2 (lambda (y) (1 x y)))).
PSRULE VP8 : VP --> H0[SUBCAT DITRANS] NP[CASE ACC, PRD -]
   NP[CASE ACC, PRD -] :
   (lambda (x) (3 (lambda (y) (2 (lambda (z) (1 x y z)))))).
PSRULE VP9 : VP --> H0[SUBCAT DITRANS] NP[CASE ACC, PRD -]
   PP[PFORM TO] :
   (lambda (x) (2 (lambda (y) (3 (lambda (z) (1 x y z)))))).
PSRULE VP18 : VP --> H0[AUX DO] VP[VFORM BSE, AUX -, INV -] : 2.
PSRULE VP22 : VP --> H0[AUX BE] VP[VFORM PASS, INV -] : 2.
PSRULE VP23 : VP --> H0[AUX BE] AP[ADV -] : 2.
PSRULE VP24 : VP --> H0[AUX BE] NP[CASE ACC, PRD +, DEF -] : 2.
PSRULE VP25 : VP --> H0[AUX BE] NP[CASE ACC, PRD -, DEF +] :
   (lambda (x) (2 (lambda (y) (Equal x y)))).
PSRULE VP28 : VP[VFORM PASS] --> H0[SUBCAT TRANS] ( PP[PFORM BY] ) :
   (lambda (x) (2 (lambda (y) (1 y x)))) :
   (lambda (x) (some (y) (1 y x))).
PSRULE VP29 :
   VP[VFORM PASS] --> H0[SUBCAT DITRANS] NP[CASE ACC, PRD -]
   ( PP[PFORM BY] ) :
   (lambda (x) (2 (lambda (y) (3 (lambda (z) (1 z y x)))))) :
   (lambda (x) (2 (lambda (y) (some (z) (1 z y x))))).
PSRULE VP30 : VP[VFORM PASS] --> H0[SUBCAT DITRANS] PP[PFORM TO]
```

```
   ( PP[PFORM BY] ) :
   (lambda (x) (2 (lambda (y) (3 (lambda (z) (1 z x y)))))) :
   (lambda (x) (2 (lambda (y) (some (z) (1 z x y))))).
PSRULE V1 : V[INV +, EMPTY +, AUX @x] --> [NULL +] :
   (lambda (P) (P)).
PSRULE NP1 : NP[DEF @d] --> Det[PLU @p, DEF @d]
   H1[PLU @p, PER 3, NTYPE COUNT] :
   2 = [PRD -], (1 2) : 2 = [PRD +], 2.
PSRULE NP2 : NP[PRD -] --> H1[PLU +] :
   (lambda (P) (All (x) (If (1 x) (P x)))).
PSRULE NP3 : N1 --> H0 : 1.
PSRULE NP8 : N1 --> AP H1 : (lambda (x) (And (1 x) (2 x))).
PSRULE PP1 : PP --> H1 : 1.
PSRULE PP2 : P1 --> H0[SUBCAT TRANS] NP[CASE ACC, PRD -] : 2.
PSRULE AP1 : AP --> H1 : 1.
PSRULE AP3 : A1 --> H0 : 1.

PROPRULE HFC_V : [V +, N -] --> [H +], U. F(0) = F(1), F in VHEAD.
PROPRULE HFC_N : [V -, N +] --> [H +], U. F(0) = F(1), F in NHEAD.
PROPRULE HFC_P : [V -, N -] --> [H +], U. F(0) = F(1), F in PHEAD.
PROPRULE HFC_A : [V +, N +] --> [H +], U. F(0) = F(1), F in AHEAD.
PROPRULE AGR1 : S --> NP, [H +], U. F(1) = F(2), F in {PER, PLU}.
PROPRULE AGR1a : S --> [H +], NP, U. F(1) = F(2), F in {PER, PLU}.
PROPRULE SAI : VP --> [H +, AUX (DO, BE)], U. INV(1) = EMPTY(1).
PROPRULE AGR3 : S[INV +] --> H0 S[INV +].
   F(1) = F(2), F in {PER, PLU}.

WORD laughs :
   V[SUBCAT INTRANS, PLU -, PER 3, PAST -, VFORM FIN, AUX -, INV -] :
   laugh1.
WORD laugh :
   V[SUBCAT INTRANS, PLU -, PER 1, PAST -, VFORM FIN, AUX -, INV -] :
   laugh1,
   V[SUBCAT INTRANS, PLU -, PER 2, PAST -, VFORM FIN, AUX -, INV -] :
   laugh1,
   V[SUBCAT INTRANS, PLU +, PAST -, VFORM FIN, AUX -, INV -] :
   laugh1,
   V[SUBCAT INTRANS, VFORM BSE, AUX -, INV -] : laugh1.
WORD chases :
   V[SUBCAT TRANS, PLU -, PER 3, PAST -, VFORM FIN, AUX -, INV -] :
   chase1.
WORD chase :
   V[SUBCAT TRANS, PLU -, PER 1, PAST -, VFORM FIN, AUX -, INV -] :
   chase1,
   V[SUBCAT TRANS, PLU -, PER 2, PAST -, VFORM FIN, AUX -, INV -] :
   chase1,
   V[SUBCAT TRANS, PLU +, PAST -, VFORM FIN, AUX -, INV -] :
   chase1,
```

```
   V[SUBCAT TRANS, VFORM BSE, AUX -, INV -] : chase1.
WORD chased : V[SUBCAT TRANS, VFORM PASS, AUX -, INV -] : chase1.
WORD gives :
   V[SUBCAT DITRANS, PLU -, PER 3, PAST -, VFORM FIN, AUX -, INV -] :
   give1.
WORD give :
   V[SUBCAT DITRANS, PLU -, PER 1, PAST -, VFORM FIN, AUX -, INV -] :
   give1,
   V[SUBCAT DITRANS, PLU -, PER 2, PAST -, VFORM FIN, AUX -, INV -] :
   give1,
   V[SUBCAT DITRANS, PLU +, PAST -, VFORM FIN, AUX -, INV -] :
   give1,
   V[SUBCAT DITRANS, VFORM BSE, AUX -, INV -] : give1.
WORD gave :
   V[SUBCAT DITRANS, PAST +, VFORM FIN, AUX -, INV -] : give1.
WORD given : V[SUBCAT DITRANS, VFORM PASS, AUX -, INV -] : give1.
WORD does : V[AUX DO, PLU -, PER 3, PAST -, VFORM FIN, EMPTY -].
WORD do : V[AUX DO, PLU -, PER 1, PAST -, VFORM FIN, EMPTY -],
   V[AUX DO, PLU -, PER 2, PAST -, VFORM FIN, EMPTY -],
   V[AUX DO, PLU +, PAST -, VFORM FIN, EMPTY -],
   V[AUX DO, INV -, VFORM BSE, EMPTY -].
WORD did : V[AUX DO, PAST +, VFORM FIN, EMPTY -].
WORD is : V[AUX BE, PLU -, PER 3, PAST -, VFORM FIN, EMPTY -].
WORD am : V[AUX BE, PLU -, PER 1, PAST -, VFORM FIN, EMPTY -].
WORD are : V[AUX BE, PLU -, PER 2, PAST -, VFORM FIN, EMPTY -],
   V[AUX BE, PLU +, PAST -, VFORM FIN, EMPTY -].
WORD be : V[AUX BE, INV -, VFORM BSE, EMPTY -].
WORD was : V[AUX BE, PER 3, PLU -, PAST +, VFORM FIN, EMPTY -],
   V[AUX BE, PER 1, PLU -, PAST +, VFORM FIN, EMPTY -].
WORD were : V[AUX BE, PLU -, PER 2, PAST +, VFORM FIN, EMPTY -],
   V[AUX BE, PLU +, PAST +, VFORM FIN, EMPTY -].
WORD Hannah :
   NP[PLU -, PER 3, NTYPE NAME, PRD -] : (lambda (P) (P hannah1)).
WORD Sam :
   NP[PLU -, PER 3, NTYPE NAME, PRD -] : (lambda (P) (P sam1)).
WORD Felix :
   NP[PLU -, PER 3, NTYPE NAME, PRD -] : (lambda (P) (P felix1)).
WORD Tweety :
   NP[PLU -, PER 3, NTYPE NAME, PRD -] : (lambda (P) (P tweety1)).
WORD cat : N[PLU -, PER 3, NTYPE COUNT] : cat1.
WORD bird : N[PLU -, PER 3, NTYPE COUNT] : bird1.
WORD I : NP[PLU -, PER 1, CASE NOM, NTYPE PRO, PRD -] :
   (lambda (P) (Some (x) (And (Speaker1 x) (P x)))).
WORD me : NP[PLU -, PER 1, CASE ACC, NTYPE PRO, PRD -] :
   (lambda (P) (Some (x) (And (Speaker1 x) (P x)))).
WORD you : NP[PER 2, PRO +, NTYPE PRO, PRD -] :
   (lambda (P) (Some (x) (And (Hearer1 x) (P x)))).
WORD she : NP[PLU -, PER 3, CASE NOM, NTYPE PRO, PRD -] :
```

```
   (lambda (P) (Some (x) (And (Female1 x) (P x)))).
WORD her : NP[PLU -, PER 3, CASE ACC, NTYPE PRO, PRD -] :
   (lambda (P) (Some (x) (And (Female1 x) (P x)))).
WORD he : NP[PLU -, PER 3, CASE NOM, NTYPE PRO, PRD -] :
   (lambda (P) (Some (x) (And (Male1 x) (P x)))).
WORD him : NP[PLU -, PER 3, CASE ACC, NTYPE PRO, PRD -] :
   (lambda (P) (Some (x) (And (Male1 x) (P x)))).
WORD it : NP[PLU -, PER 3, NTYPE PRO, PRD -] :
   (lambda (P) (Some (x) (P x))) : ).
WORD we : NP[PLU +, PER 1, CASE NOM, NTYPE PRO, PRD -] :
   (lambda (P) (Some (x) (And (Speaker1 x) (P x)))).
WORD us : NP[PLU +, PER 1, CASE ACC, NTYPE PRO, PRD -] :
   (lambda (P) (Some (x) (And (Speaker1 x) (P x)))).
WORD by : P[PFORM BY, SUBCAT TRANS].
WORD to : P[PFORM TO, SUBCAT TRANS].
WORD the : Det[DEF +] :
   (lambda (P) (lambda (Q) (Some (x) (And (P x) (Q x))))).
WORD a : Det[DEF -, PLU -] :
   (lambda (P) (lambda (Q) (Some (x) (And (P x) (Q x))))).
WORD some : Det[DEF -, PLU -] :
   (lambda (P) (lambda (Q) (Some (x) (And (P x) (Q x))))).
WORD every : Det[PLU -, DEF +] :
   (lambda (P) (lambda (Q) (All (x) (If (P x) (Q x))))).
WORD no : Det[DEF +] :
   (lambda (P) (lambda (Q) (Not (Some (x) (And (P x) (Q x)))))).
WORD angry : A[ADV -] : angry1.
WORD happy : A[ADV -] : happy1.
```

Appendix B

Customisation and Programmatic Use

B.1 Customisation

Some aspects of the standard behaviour of the GDE may easily be customised by 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. Alternatively, single-line Lisp expressions may be evaluated from the GDE command loop by prefixing them with the character '`!`'.

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 (set at the top level and never rebound) with the value `t` representing ON, and `nil` representing OFF. The variables holding the flag values are

```
*defining-messages
*prop-before-default
*addition-checking
*multiple-expansions
*multiple-linearisations
*morph-system
*fast-morph-lookup
*word-structure
*ecrs-before-multiply
*term-unification
*lr1-parse
```

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
```

Warnings from the parser for non-consumable lexical categories are controlled by the variable `g-lexical-warn`. The warnings may be suppressed by setting this variable to `nil`.

A file may be read into the GDE by calling the Lisp function `read-grammar`. The function takes one argument, the name of the file as a Lisp string.

B.2 Programmatic Use of the Parser

The source code for the interface between the GDE and the parser is in the file 'cgde/parse.lsp'. Copying and adapting parts of this file will in most cases be sufficient to enable the parser to be invoked from a user program.

The function `invoke-parser` (defined in this file) may be called to parse a sentence or phrase. It should be given two arguments: the first, the sentence to be parsed as a list of symbols (taking care that the case of each symbol is the same as that of the word as it was defined), the second a flag indicating what information should be output as the result of the parse. A value of `nil` indicates that no statistics or parse results should be output. Representations of the trees resulting from the last parse are available as the value of the special variable `*current-parse-trees`. The functions

```
display-parse-bracketing
display-parse-rule-labelling
display-parse-cat-labelling
```

(all defined in 'cgde/parse.lsp') may be called with such a list of trees as single argument to print a set of bracketings, rule-labelled or category-labelled bracketings respectively. E.g.

```
(invoke-parser '(|fido| |costs| |a| |pound|) nil)
(display-parse-bracketing *current-parse-trees)
```

Variants of these functions can easily be defined to collect the trees up into a list, instead of printing them, and to pass them on for further processing.

Appendix C

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 NL Tools parser. The rules in this file should also be acceptable to other parsers with a little editing or pre-processing. This appendix gives a specification of the object grammar output format.

The file contains 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. Each rule is represented as a list, the mother being the first element, followed by the daughter categories in order. A 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`'). Feature / value pairs are dotted-pairs whose head is the feature name and tail the corresponding value. A value may be either a symbol or another category. 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 following might be the first part of a dumped grammar file:

```
(N V BAR SUBJ SUBCAT CONJ VFORM H T BEGAP FIN PAST PRD AUX INV
   PSVE NEG COMP SLASH NFORM PER PLU COUNT CASE PN PRO PART
   SPEC PFORM LOC GERUND AFORM QUA DEF POSS ADV NUM WH UB EVER
   MOD CONJN COORD REFL AT LAT FIX INFL STEM COMPOUND PRT CAT
   MAJ REG ADDRESS ARITY COMPAR DISTR GROUP ORDER PREMOD PREP
   SUBTYPE CN1 CN2 AND TAG AGR NOSLASH NULL CN3 QFEAT)

((((N . -) (V . +) (BAR . |2|) (SUBJ . +) (CONJ . NULL)
     (VFORM . NOT) (FIN . +) (PAST . @12) (PRD . @75)
     (AUX . @14) (INV . -) (COMP . NORM) (SLASH . @19)
     (WH . @38) (UB . @39) (EVER . @40) (COORD . @)
     (AGR (N . +) (V . -) (BAR . |2|) (NFORM . @20)
        (PER . @21) (PLU . @22) (COUNT . @23) (CASE . NOM)))
    "S1/-")
   (((N . +) (V . -) (BAR . |2|) (CONJ . NULL) (BEGAP . @)
       (PRD . @) (NEG . -) (SLASH (NOSLASH . +)) (NFORM . @20)
       (PER . @21) (PLU . @22) (COUNT . @23) (CASE . NOM)
       (PN . @) (PRO . @) (SPEC . +) (AFORM . @) (DEF . @)
       (POSS . -) (NUM . @) (WH . @38) (UB . @39) (EVER . @40)
       (COORD . @) (REFL . @)))
   (((N . -) (V . +) (BAR . |2|) (SUBJ . -) (CONJ . NULL)
       (VFORM . NOT) (FIN . +) (PAST . @12) (PRD . @75)
       (AUX . @14) (NEG . @) (SLASH . @19) (COORD . @)
       (AGR (N . +) (V . -) (BAR . |2|) (NFORM . @20)
          (PER . @21) (PLU . @22) (COUNT . @23)
          (CASE . NOM)))))
```

The "dwords" command with the "unreadable" option produces output in a similar form to that produced by the "dump" command. A sequence of word definitions follows the initial list of features. A word 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 `t`. Thus a file containing a definition for the word `kim` might look like

```
(N V BAR SUBJ SUBCAT CONJ VFORM H T BEGAP FIN PAST PRD AUX INV
   PSVE NEG COMP SLASH NFORM PER PLU COUNT CASE PN PRO PART
   SPEC PFORM LOC GERUND AFORM QUA DEF POSS ADV NUM WH UB EVER
   MOD CONJN COORD REFL AT LAT FIX INFL STEM COMPOUND PRT CAT
   MAJ REG ADDRESS ARITY COMPAR DISTR GROUP ORDER PREMOD PREP
   SUBTYPE CN1 CN2 AND TAG AGR NOSLASH NULL CN3 QFEAT)

((((N . +) (V . -) (BAR . |0|) (SUBCAT . NULL) (CONJ . NULL)
     (PRD . @13) (NFORM . NORM) (PER . |3|) (PLU . -)
     (COUNT . +) (CASE . @24) (PN . +) (PRO . -) (POSS . -)
     (NUM . -) (COORD . @43) (REFL . @44))
    . T)
   |kim|)
```

Appendix D

GDE Implementations

D.1 Lisp Implementations Supported

The morphological analyser and parser were originally written in Franz Lisp, and the GDE in Cambridge Lisp. Around 1988–89, Alan Black (University of Edinburgh Department of Artificial Intelligence) ported the morphological analyser to Common Lisp, and John Carroll translated the parser and the rest of the GDE into Common Lisp and brought the three programs together to run as one integrated system. Previous versions of the Alvey NL Tools have been customised for, and successfully run in the following implementations of Common Lisp:

1. Hewlett Packard Common Lisp I, version 1.01, on an HP 9000/350 ('Bobcat').
2. Xerox Common Lisp, Lyric release, on a Xerox 1186 ('Dove').
3. POPLOG Common Lisp, version 1.0, on a Sun 3/260.
4. Coral Common Lisp (now MCL), version 1.2, on an Apple Macintosh II.
5. Sun (Lucid) Common Lisp, version 2.1.1, on a Sun 3/260.

It is believed that the current version of the Tools will still run in these implementations. Users have themselves ported previous versions to POPLOG Common Lisp, version 2 and Symbolics Lisp Genera 8.0. The current version has been thoroughly tested in the following implementations:

1. Procyon Common Lisp, version 2.1.4, on the Apple Macintosh family.
2. Kyoto Common Lisp, version 1.25, on a DECstation 3100.
3. Austin Kyoto Common Lisp, version 1.530, on a Sun Sparc1+.
4. Franz Allegro Common Lisp, version 3.1, on a DECstation 3100.
5. Hewlett Packard (Lucid) Common Lisp II, rev A.02.16, on an HP 9000/350.

Although the morphological analyser, parser and GDE when put together are a fairly large system (over 35,000 lines of source code), they are portable: setting them up to run in 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 necessary additions.

For serious use of the GDE with large grammars on UNIX machines, it is recommended that the machine have at least 16 MBytes of memory, otherwise thrashing is likely to occur. On Apple Macintoshes running MacOS, 8 MBytes is sufficient.

D.2 Benchmarks

The following timings may be useful as a guide to the performance to be expected of the GDE running on various machines and implementations of Common Lisp. Version numbers are as quoted above. Units are seconds of CPU time, excluding garbage collection and overheads.

	Mac IIci Procyon CL	DS3100 Allegro CL	Sun Sparc 1+ AKCL	HP 9000/350 HP CL II
(1) Read	56 (*22)	14	7.9	13
(2) Compile	230	105	77	140
(3) Word lookup	3.2	1.4	0.7	2.1
(4) Parse	1.6	0.6	0.8	1.3

The first test is to read in the file of basic feature etc. definitions for a (June 1991) version of the large Alvey NL Tools grammar (starred timings are with the 'defining messages' flag OFF). The second test is to compile the whole grammar. Test three is to look up the word 'hears' in the lexicon corresponding to the grammar, and four to parse the sentence 'she dictated to him whether they would accept' (not counting the time taken in looking up words).