Research Report AI–1994–06

GULP 3.1: An Extension of Prolog for Unification–Based Grammar

Michael A. Covington

Artificial Intelligence Center
The University of Georgia
Athens, Georgia 30602–7415 U.S.A.

GULP 3.1: An Extension of Prolog for Unification–Based Grammar

Michael A. Covington
Artificial Intelligence Center
The University of Georgia
Athens, Georgia 30602–7415 U.S.A.
mcovingt@ai.uga.edu

July 1994

Abstract

This paper documents GULP 3.1, a simple extension to Prolog that facilitates implementation of unification–based grammars (UBGs) by adding a notation for feature structures. For example, `a:b..c:d` denotes a feature structure in which `a` has the value `b`, `c` has the value `d`, and the values of all other features are unspecified. A modified Prolog interpreter translates feature structures into Prolog terms that unify in the desired way. Thus, the extension is purely syntactic, analogous to the automatic translation of `"abc"` to `[97,98,99]` in Edinburgh Prolog.

This is a revision of the GULP 2.0 report (1989) and includes a tutorial introduction to unification–based grammar.

Contents

1 Introduction **2**

2 What is unification–based grammar? **3**
2.1 Unification–based theories 3
2.2 Grammatical features 3
2.3 Unification–based grammar 4
2.4 A sample grammar 7
2.5 Unification 10
2.6 Declarativeness 11
2.7 Building structures and moving data 12

3 The GULP translator **13**
3.1 Feature structures in GULP 13
3.2 GULP syntax 14
3.3 Automatic translation 15
3.4 The GULP environment 15
3.5 Internal representation 17
3.6 How translation is done 18

4 GULP in practical use **19**
4.1 A simple definite clause grammar 19
4.2 A hold mechanism for unbounded movements 23
4.3 Building discourse representation structures 24
4.4 Left–corner parsing 29

5 Future Prospects **32**
5.1 Possible improvements 32
5.2 Keyword parameters via GULP 32

6 References **33**

1 Introduction

A number of software tools have been developed for implementing unification–based grammars, among them PATR–II (Shieber 1986a,b), D–PATR (Karttunen 1986a), PrAtt (Johnson and Klein 1986), and AVAG (Sedogbo 1986). This paper describes a simple extension to the syntax of Prolog that serves the same purpose while making a much less radical change to the language. Unlike PATR–II and similar systems, this system treats feature structures as first–class objects that appear in any context, not just in equations. Further, feature structures can be used not only in natural language processing, but also to pass keyword arguments to any procedure.[1]

The extension is known as GULP (Graph Unification Logic Programming). It allows the programmer to write `a:b..c:d` to stand for a feature structure in which feature `a` has the value `b`, feature `c` has the value `d`, and all other features are uninstantiated. The interpreter translates feature structures written in this notation into ordinary Prolog terms that unify in the desired way. Thus, this extension is similar in spirit to syntactic devices

[1]This document supersedes the GULP 2 report (Covington 1989), but GULP 2 continues to be the latest version available for some Prolog compilers. The first version of GULP (Covington 1987) was developed with support from National Science Foundation Grant IST–85–02477.

already in the language, such as writing `"abc"` for `[97,98,99]` or writing `[a,b,c]` for `.(a,.(b,.(c,nil)))`.

GULP can be used with grammar rule notation (definite clause grammars, DCGs) or with any parser that the programmer cares to implement in Prolog. GULP's use of the colon does conflict with the use of the colon to designate modules, but so far, this has not caused problems.

2 What is unification–based grammar?

2.1 Unification–based theories

Unification–based grammar (UBG) comprises all theories of grammar in which unification (merging) of feature structures plays a prominent role. As such, UBG is not a theory of grammar but rather a formalism in which theories of grammar can be expressed. Such theories include functional unification grammar, lexical–functional grammar (Kaplan and Bresnan 1982), generalized phrase structure grammar (Gazdar et al. 1986), head–driven phrase structure grammar (Pollard and Sag 1987, 1994), and others.

UBGs use context–free grammar rules in which the nonterminal symbols are accompanied by sets of features. The addition of features increases the power of the grammar so that it is no longer context–free; indeed, in the worst case, parsing with such a grammar can be NP–complete (Barton, Berwick, and Ristad 1987:93–96).

However, in practice, these intractable cases are rare. Theorists restrain their use of features so that the grammars, if not actually context–free, are close to it, and context–free parsing techniques are successful and efficient. Joshi (1986) has described this class of grammars as "mildly context–sensitive."

2.2 Grammatical features

Grammarians have observed since ancient times that each word in a sentence has a set of attributes, or features, that determine its function and restrict its usage. Thus:

$$\begin{array}{ccc} \textit{The} & \textit{dog} & \textit{barks.} \\ \begin{bmatrix} \textit{category:determiner} \end{bmatrix} & \begin{bmatrix} \textit{category:noun} \\ \textit{number:singular} \end{bmatrix} & \begin{bmatrix} \textit{category:verb} \\ \textit{number:singular} \\ \textit{person:3rd} \\ \textit{tense:present} \end{bmatrix} \end{array}$$

The earliest generative grammars of Chomsky (1957) and others ignored all of these features except category, generating sentences with context–free phrase– structure rules such as

sentence $\rightarrow$ *noun phrase* + *verb phrase*

noun phrase $\rightarrow$ *determiner* + *noun*

plus transformational rules that rearranged syntactic structure. Syntactic structure was described by tree diagrams. Number and tense markers were treated as separate elements of the string (e.g., *boys* = *boy* + *s*). "Subcategorization" distinctions, such as the fact that some verbs take objects and other verbs do not, were handled by splitting a single category, such as verb, into two categories ($\text{verb}_{\text{transitive}}$ and $\text{verb}_{\text{intransitive}}$).

But complex, cross–cutting combinations of features cannot be handled in this way, and Chomsky (1965) eventually attached feature bundles to all the nodes in the tree (cf. Figure 1). His contemporaries accounted for grammatical agreement (e.g., the agreement of the number features of subject and verb) by means of transformations that copied features from one node to another. This remained the standard account of grammatical agreement for many years.

Feature copying is unnecessarily procedural. It presumes, unjustifiably, that whenever two nodes agree, one of them is the source and the other is the destination of a copied feature. In practice, the source and destination are hard to distinguish. Do singular subjects require singular verbs, or do singular verbs require singular subjects? This is an empirically meaningless question. Moreover, when agreement processes interact to combine features from a number of nodes, the need to distinguish source from destination introduces unnecessary clumsiness.

2.3 Unification–based grammar

Unification–based grammar attacks the same problem non–procedurally, by stating constraints on feature values. For example, the rule

(1) $$PP \rightarrow P \quad \underset{[\,\mathit{case{:}acc}\,]}{NP}$$

says that the object of the preposition is in the accusative case, and

(2) $$\underset{[\,\mathit{number{:}X}\,]}{NP} \rightarrow \underset{[\,\mathit{number{:}X}\,]}{D} \quad \underset{[\,\mathit{number{:}X}\,]}{N}$$

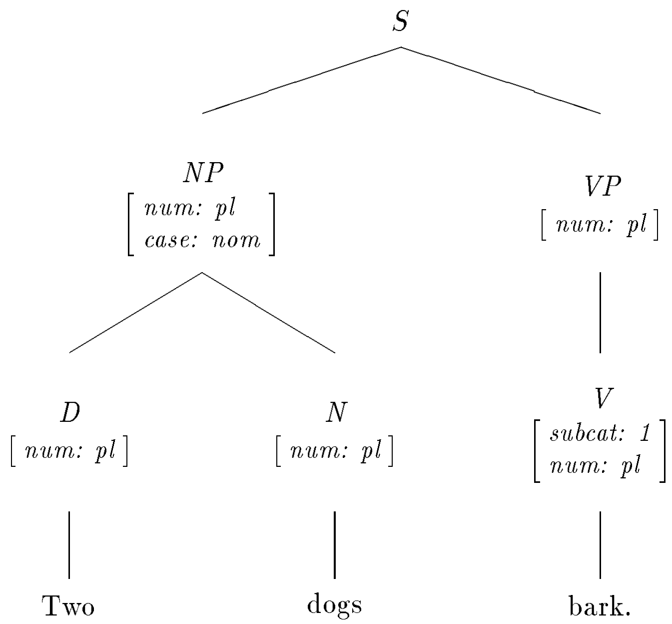

Figure 1: A tree with features on every node.

says that in a noun phrase, the NP, determiner, and noun all have the same number (singular or plural).

More precisely, rule 1 says the feature structure $[case : acc]$ must be UNIFIED (merged) with whatever features the N already has, and rule 2 says that $[number : X]$ must be unififed with whatever features the NP, D, and N already have. Here X is a variable that takes on the same value on all three nodes. Thus, without specifying which features get copied from where to where, the latter rule simply equates the values of *number* on all three nodes.

Strictly speaking, the category label (S, NP, VP, etc.) is part of the feature structure. Thus,

$$\begin{array}{c} NP \\ [\, case{:}acc \,] \end{array}$$

is short for:

$$\left[\begin{array}{l} category{:}NP \\ case{:}acc \end{array} \right]$$

In practice, however, the category label usually plays a primary role in parsing, and it is convenient to give it a special status.

Grammar rules can alternatively be written in terms of equations that the feature values must satisfy. In equational notation, rules 1 and 2 become:

(3) $PP \rightarrow P\ NP$
NP case = acc

(4) $S \rightarrow NP\ VP$
NP person = VP person
NP number = VP number

or even, if the category label is to be treated as a feature,

(5) $X \rightarrow Y\ Z$
X category = PP
Y category = P
Z category = NP
Z case = acc

(6) $X \rightarrow Y\ Z$
X category = S

Y category = NP
Z category = VP
Y person = Z person
Y number = Z number

where X, Y, and Z are variables. Equations are used in PATR–II, PrAtt, and other implementation tools, but are not essential to GULP.

The value of a feature can itself be a feature structure. This makes it possible to group features together to express generalizations. For instance, one can group syntactic and semantic features together, creating structures such as:

$$\begin{bmatrix} \textit{syn:} & \begin{bmatrix} \textit{case:acc} \\ \textit{gender:masc} \end{bmatrix} \\ \textit{sem:} & \begin{bmatrix} \textit{pred:MAN} \\ \textit{countable:yes} \\ \textit{animate:yes} \end{bmatrix} \end{bmatrix}$$

Then a rule can copy the syntactic or semantic features en masse to another node, without enumerating them.

2.4 A sample grammar

Features provide a powerful way to pass information from one place to another in a grammatical description. The grammar in Figure 2 is an example. It uses features not only to ensure the grammaticality of the sentences generated, but also to build a representation of the meaning of the sentence. Every constituent has a *sem* feature representing its meaning. The rules combine the meanings of the individual words into predicate–argument structures representing the meanings of all of the constituents. The meaning of the sentence is represented by the *sem* feature of the topmost S node.

Like all the examples given here, this grammar is intended only as a demonstration of the power of unification–based grammar, not as a viable linguistic analysis. Thus, for simplicity, the proposal to group syntactic features together is abandoned.

Now look back at Figure 1, which shows a complete sentence generated by this grammar. Note that all the constraints specified in the rules are obeyed. If they weren't, the unification would fail somewhere. Because the grammar treats feature copying equationally rather than procedurally, the same grammar can be used to parse bottom–up, top–down, or in other ways.

$$(1) \quad S \rightarrow \begin{array}{c} NP \\ \left[\begin{array}{l} case\text{: } nom \\ num\text{: } X \end{array}\right] \end{array} \quad \begin{array}{c} VP \\ \left[\, num\text{: } X \,\right] \end{array}$$

$$(2) \quad \begin{array}{c} VP \\ \left[\, num\text{: } X \,\right] \end{array} \rightarrow \begin{array}{c} V \\ \left[\begin{array}{l} subcat\text{: } 1 \\ num\text{: } X \end{array}\right] \end{array} \qquad \text{(for verbs without objects)}$$

$$(3) \quad \begin{array}{c} VP \\ \left[\, num\text{: } X \,\right] \end{array} \rightarrow \begin{array}{c} V \\ \left[\begin{array}{l} subcat\text{: } 2 \\ num\text{: } X \end{array}\right] \end{array} \quad \begin{array}{c} NP \\ \left[\, case\text{: } acc \,\right] \end{array} \qquad \text{(for verbs with objects)}$$

$$(4) \quad \begin{array}{c} NP \\ \left[\, num\text{: } X \,\right] \end{array} \rightarrow \begin{array}{c} D \\ \left[\, num\text{: } X \,\right] \end{array} \quad \begin{array}{c} N \\ \left[\, num\text{: } X \,\right] \end{array} \qquad \text{(number agreement)}$$

$$(5) \quad \begin{array}{c} NP \\ \left[\begin{array}{l} case\text{: } C \\ num\text{: } X \end{array}\right] \end{array} \rightarrow \begin{array}{c} Pronoun \\ \left[\begin{array}{l} case\text{: } C \\ num\text{: } X \end{array}\right] \end{array}$$

$$(6) \quad \begin{array}{c} Pronoun \\ \left[\, num\text{: } sg \,\right] \end{array} \rightarrow \text{it} \qquad \text{(both nom. and acc.)}$$

$$(7) \quad \begin{array}{c} Pronoun \\ \left[\begin{array}{l} case\text{: } nom \\ num\text{: } pl \end{array}\right] \end{array} \rightarrow \text{they}$$

$$(8) \quad \begin{array}{c} Pronoun \\ \left[\begin{array}{l} case\text{: } acc \\ num\text{: } pl \end{array}\right] \end{array} \rightarrow \text{them}$$

Figure 2: A unification–based grammar that parses and generates a small subset of English. (Continued on next page.)

(9) $\underset{\left[\,\textit{num: sg}\,\right]}{N} \rightarrow$ dog

(10) $\underset{\left[\,\textit{num: pl}\,\right]}{N} \rightarrow$ dogs

(11) $\underset{\left[\,\textit{num: sg}\,\right]}{D} \rightarrow$ a

(12) $\underset{\left[\,\textit{num: pl}\,\right]}{D} \rightarrow$ two

(13) $D \rightarrow$ the (both singular and plural)

(14) $\underset{\begin{bmatrix}\textit{num: sg}\\ \textit{subcat: 1}\end{bmatrix}}{V} \rightarrow$ barks

(15) $\underset{\begin{bmatrix}\textit{num: pl}\\ \textit{subcat: 1}\end{bmatrix}}{V} \rightarrow$ bark

(16) $\underset{\begin{bmatrix}\textit{num: sg}\\ \textit{subcat: 2}\end{bmatrix}}{V} \rightarrow$ scares

(17) $\underset{\begin{bmatrix}\textit{num: pl}\\ \textit{subcat: 2}\end{bmatrix}}{V} \rightarrow$ scare

2.5 Unification

Our sample grammar relies on the merging of partially specified feature structures. Thus, the subject of the sentence gets case from one rule and person and number from another. This merging can be formalized as UNIFICATION. The unifier of two feature structures A and B is the smallest feature structure C that contains all the information in both A and B.

Feature structure unification is equivalent to GRAPH UNIFICATION, i.e., merging of directed acyclic graphs, as defined in graph theory. The unifier of two graphs is the smallest graph that contains all the nodes and arcs in the graphs being unified. This is similar but not identical to Prolog TERM UNIFICATION; crucially, elements of the structure are identified only by name, not (as in Prolog) by position.

Formally, the unification of feature structures A and B (giving C) is defined as follows:

1. Any feature that occurs in A but not B, or in B but not A, also occurs in C with the same value.

2. Any feature that occurs in both A and B also occurs in C, and its value in C is the unifier of its values in A and B.

Feature values, in turn, are unified as follows:

1. If both values are atomic symbols, they must be the same atomic symbol, or else the unification fails (the unifier does not exist).

2. A variable unifies with any object by becoming that object. All occurrences of that variable henceforth represent the object with which the variable has unified. Two variables can unify with each other, in which case they become the same variable (just as in Prolog).

3. If both values are feature structures, they unify by applying this process recursively.

Thus the two feature structures

$$\begin{bmatrix} a{:}b \\ c{:}d \end{bmatrix} \qquad \begin{bmatrix} c{:}d \\ e{:}f \end{bmatrix}$$

unify giving:

$$\begin{bmatrix} a{:}b \\ c{:}d \\ it\ e{:}f \end{bmatrix}$$

Likewise, $[a : X]$ and $[a : b]$ unify, instantiating X to the value b; and the structures

$$\begin{bmatrix} a{:}X \\ b{:}c \end{bmatrix} \qquad \begin{bmatrix} a{:}c \\ b{:}Y \end{bmatrix}$$

unify by instantiating both X and Y to c.

As in Prolog, unification is not always possible. Specifically, if A and B have different (non–unifiable) values for some feature, unification fails. A grammar rule requiring A to unify with B cannot apply if A and B are not unifiable.

Unification–based grammars rely on failure of unification to rule out ungrammatical sentences. Consider, for example, why our sample grammar generates *It scares them* but not *Them scare it.* In *Them scare it,* rule 8 specifies that the pronoun *them* must have $case : acc$, but rule 1 specifies $case : nom$ on the NP node above it, and rule 5 specifies that the case of the NP and of the pronoun is the same. Because of the conflict, the necessary unifications are not possible and the ungrammatical sentence is not generated or parsed.

2.6 Declarativeness

Unification–based grammars are declarative, not procedural. That is, they are statements of well–formedness conditions, not procedures for generating or parsing sentences. That is why, for example, sentences generated by our sample grammar can be parsed either bottom–up or top–down.

This declarativeness comes from the fact that unification is an order–independent operation. The unifier of A, B, and C is the same regardless of the order in which the three structures are combined. This is true of both graph unification and Prolog term unification.

The declarative nature of UBGs is subject to two caveats. First, although unification is order–independent, particular parsing algorithms are not. Recall that grammar rules of the form

$$A \rightarrow A\ B$$

cannot be parsed top–down, because they lead to infinite loops ("To parse an A, parse an A and then..."). Now consider a rule of the form

$$\underset{[\,f{:}X\,]}{A} \rightarrow \underset{[\,f{:}X\,]}{A} \quad B$$

If X and Y have different values, then top–down parsing works fine; if either X or Y does not have a value at the time the rule is invoked, top–down parsing will lead to a loop. This shows that one cannot simply give an arbitrary UBG to an arbitrary parser and expect useful results; the order of instantiation must be kept in mind.

Second, many common Prolog operations are not order–independent, and this must be recognized in any implementation that allows Prolog goals to be inserted into grammar rules. Obviously, the cut (`!`) interferes with order–independence by blocking alternatives that would otherwise succeed. More commonplace predicates such as `write`, `is`, and `==` lack order–independence because they behave differently depending on whether their arguments are instantiated at the time of execution. Colmerauer's Prolog II (Giannesini et al. 1986) avoids some of these difficulties by allowing the programmer to postpone tests until a variable becomes instantiated, whenever that may be.

2.7 Building structures and moving data

Declarative unification–based rules do more than just pass information up and down the tree. They can build structure as they go. For example, the rule

$$\overset{VP}{\left[\mathit{sem}: \begin{bmatrix} \mathit{pred}{:}X \\ \mathit{arg}{:}Y \end{bmatrix} \right]} \rightarrow \overset{V}{\left[\mathit{sem}{:}X \right]} \quad \overset{NP}{\left[\mathit{sem}{:}Y \right]}$$

builds a semantic representation on the VP node from those of the V and NP nodes.

Unification can pass information around in directions other than along the lines of the tree diagram. This is done by splitting a feature into two sub–features, one for input and the other for output. The inputs and outputs can then be strung together in any manner. Consider for example the rule:

$$\overset{S}{\left[\mathit{sem}: \begin{bmatrix} \mathit{in}{:}X1 \\ \mathit{out}{:}X3 \end{bmatrix} \right]} \rightarrow \overset{NP}{\left[\mathit{sem}: \begin{bmatrix} \mathit{in}{:}X1 \\ \mathit{out}{:}X2 \end{bmatrix} \right]} \quad \overset{VP}{\left[\mathit{sem}: \begin{bmatrix} \mathit{in}{:}X2 \\ \mathit{out}{:}X3 \end{bmatrix} \right]}$$

This rule assumes that sem of the S has some initial value (perhaps an empty list) which is passed into $X1$ from outside. $X1$ is then passed to the NP, which modifies it in some way, giving $X2$, which is passed to the VP for further modification. The output of the VP is $X3$, which becomes the output of the S.

Such a rule is still declarative and can work either forward or backward; that is, parsing can still take place top–down or bottom–up. Further, any node in the tree can communicate with any other node via a string of input and output features, some of which simply pass information along unchanged. The example in section 4.2 below uses input and output features to undo unbounded movements of words. Johnson and Klein (1985, 1986) use *in* and *out* features to perform complex manipulations of semantic structure; see section 4.3 for a GULP reconstruction of part of one of their programs.

3 The GULP translator

3.1 Feature structures in GULP

The key idea of GULP is that feature structures can be included in Prolog programs as ordinary data items. For instance, the feature structure

$$\begin{bmatrix} a{:}b \\ c{:}d \end{bmatrix}$$

is written:

```
a:b..c:d
```

and GULP translates `a:b..c:d` into an internal representation (called a VALUE LIST) in which the a position is occupied by b, the c position is occupied by d, and all other positions, if any, are uninstantiated.

This is analogous to the way ordinary Prolog translates strings such as `"abc"` into lists of ASCII codes.The GULP programmer always uses feature structure notation and never deals directly with value lists. Feature structures are order–independent; the translations of `a:b..c:d` and of `c:d..a:b` are the same.

Nesting and paths are permitted. Thus, the structure

$$\begin{bmatrix} a{:}b & \\ c{:} & \begin{bmatrix} d{:}e \\ f{:}g \end{bmatrix} \end{bmatrix}$$

is written `a:b..c:(d:e..f:g)`. The same structure can be written as

$$\begin{bmatrix} a{:}b \\ c{:}d{:}e \\ c{:}f{:}g \end{bmatrix}$$

or, in GULP notation, `a:b..c:d:e..c:f:g`.

GULP feature structures are data items — complex terms — not statements or operations. They are most commonly used as arguments of DCG rules. Thus, the rule

$$\begin{array}{ccccc} S & & NP & & VP \\ \begin{bmatrix} \mathit{person{:}X} \\ \mathit{number{:}Y} \end{bmatrix} & \rightarrow & \begin{bmatrix} \mathit{person{:}X} \\ \mathit{number{:}Y} \end{bmatrix} & & \begin{bmatrix} \mathit{person{:}X} \\ \mathit{number{:}Y} \end{bmatrix} \end{array}$$

can be written in DCG notation, using GULP, as:

```
s(person:X..number:Y) -->
            np(person:X..number:Y),
            vp(person:X..number:Y).
```

GULP feature structures can also be processed by ordinary Prolog predicates. For example, the predicate

```
nonplural(number:X) :- nonvar(X), X \= plural.
```

succeeds if and only if its argument is a feature structure whose number feature is instantiated to some value other than plural.

Any feature structure unifies with any other feature structure unless prevented by conflicting values. Thus, the internal representations of `a:b..c:d` and `c:d..e:f` are unifiable, giving `a:b..c:d..e:f`. But `a:b` does not unify with `a:d` because `b` and `d` do not unify with each other.

3.2 GULP syntax

Formally, GULP adds to Prolog the operators '`:`' and '`..`' and a number of built–in predicates. The operator '`:`' joins a feature to its value, which itself can be another feature structure. Thus in `c:d:e`, the value of `c` is `d:e`. A feature–value pair is the simplest kind of feature structure.

The operator '`..`' combines feature–value pairs to build more complex feature structures. This is done by simply unifying them. For example, the internal representation of `a:b..c:d` is built by unifying the internal representations of `a:b` and `c:d`.

This fact can be exploited to write "improperly nested" feature structures. For example,

```
a:b..c:X..c:d:Y..Z
```

denotes a feature structure in which:

- the value of a is b,
- the value of c unifies with X,
- the value of c also unifies with $d : Y$, and
- the whole structure unifies with Z.

Both operators, '`:`' and '`..`', are right–associative; that is, `a:b:c` = `a:(b:c)`. For compatibility with GULP 1.0 and 1.1, '`..`' can be written '`::`'.

3.3 Automatic translation

GULP feature structures occurring in Prolog programs are automatically translated to their internal representations when the program is `consulted` or `compiled`. They are translated back into GULP notation when output by `print` and when displayed by the debugger. Thus, to a considerable extent, translation of feature structures into value lists is transparent to the GULP user.

Automatic translation is a new feature of GULP 3. Note that you cannot use feature structure notation in queries because queries are *not* translated.[2]

3.4 The GULP environment

GULP is an ordinary Prolog environment with some built–in predicates added, and with modifications to the behavior of `consult` (and all other predicates that load or compile programs), `print`, and the debugger.

All programs are run through the GULP translator when they are loaded; thus, all feature structures get translated into their internal representation (value lists). Although not required, it is desirable, for efficiency reasons, to begin the program with a declaration of the form

```
g_features([gender,number,case,person,tense]).
```

declaring all feature names before they are used. Further, parsing is more efficient if the morphosyntactic features (case, number, person, etc.) precede features that are purely semantic.

The GULP built–in predicate `g_translate/2` interconverts feature structures and their internal representations. This makes it possible to process, at

[2]A Prolog top level with automatic translation of input and output is being contemplated.

runtime, feature structures in GULP notation rather than translated form. For instance, if `X` is the internal representation of a feature structure, then `g_translate(Y,X), write(Y)` will display it in GULP notation. In GULP 3, `print(Y)` will of course do the same thing more conveniently.

The predicate `display_feature_structure/1` outputs a feature structure, not in GULP notation, but in a convenient tabular format, thus:

```
syn: case: acc
     gender: masc
sem: pred: MAN
     countable: yes
     animate: yes
```

This is similar to traditional feature structure notation, but without brackets.

The other built–in predicates provided by the GULP system are:

- `g_display/1`, which takes a feature structure (in internal form) and displays it in an indented tabular format;

- `display_feature_structure/1`, equivalent to `g_display/1`;

- `g_fs/1`, which succeeds if its argument is a feature structure (in *external* form, i.e., a structure held together by colons and double dots);

- `g_not_fs/1`, the opposite of `g_fs`;

- `g_vl/1`, which succeeds if its argument is a value list (a feature structure in internal form);

- `g_printlength(Atom,N)`, which instantiates `N` to the length of the atom;

- `writeln(Arg)`, which outputs `Arg`, translating all value lists into external form, and then starting a new line (if `Arg` is a list, each of its elements is output on a fresh line);

- `append/3` and `member/2` with their usual meanings;

- `remove_duplicates/2`, which takes a list and produces, from it, a list with no duplicate members;

- `call_if_possible(Goal)`, which attempts to execute the goal, but fails quietly (without raising an error condition) if there are no clauses for it;
- `g_herald/0`, which writes out an announcement of the version of GULP being used.

3.5 Internal representation

The internal representation of feature structures has changed considerably since GULP version 2. In GULP 2, a value list was a list–like structure. In GULP 3, value lists are structures with a fixed number of arguments (although they are still called value lists in the documentation).

The nearest Prolog equivalent to a feature structure is a complex term with one position reserved for the value of every feature. Thus

$$\begin{bmatrix} \textit{number:plural} \\ \textit{person:third} \\ \textit{gender:fem} \end{bmatrix}$$

could be represented as `g_(plural,third,fem)` or the like. It is necessary to decide in advance which argument position corresponds to each feature.

A feature structure that does not use all of the available features is equivalent to a term with anonymous variables; thus [*person:third*] would be represented as `g_(_,third,_)`.

Structures of this type simulate graph unification in the desired way. They can be recursively embedded. Further, structures built by instantiating Prolog variables are inherently re–entrant in the sense of Shieber (1986), since an instantiated Prolog variable is actually a pointer to the memory representation of its value. Most importantly, Schöter (1993) has shown that this is the most efficient structure for the purpose.

However, such a representation assumes that the entire list of features is known before any translation is done — an assumption that may not hold up. GULP therefore leaves room for additional features that were not declared in `g_features`. The actual structure of a value list is `g_(_,F1,F2,F3,F4,F5,F6)` with the first argument left open (uninstantiated). It provides a place to put additional features that were not declared in `g_features`, by expanding the uninstantiated first element from `_` to `[F7|_]`, then `[F7|[F8|_]]` and so on, by further instantiating the uninstantiated tail. Thus, positions for more features can be created at any time without disrupting the positions already created.

One more refinement (absent before GULP version 2.0) is needed. We want to be able to translate value lists back into feature structure notation. For this purpose we must distinguish features that are unmentioned from features that are merely uninstantiated. That is, we do not want `tense:X` to turn into an empty feature structure just because `X` is uninstantiated. It may be useful to know, during program testing, that `X` has unified with some other variable even if it has not acquired a value. Thus, we want to record, somehow, that the variable `X` was mentioned in the original feature structure whereas the values of other features (person, number, etc.) were not.

Accordingly, `g_/1` (distinct from `g_/2`) is used to mark all features that were mentioned in the original structure. If person is second in the canonical order, and tense is fifth in the canonical order (as before), then

```
tense:present..person:X    =>    g_(_,g_(X),_,_,g_(present))
```

And this is the representation actually used by GULP. Note that the use of`g_/1` does not interfere with unification, because `g_(present)` will unify both with `g_(Y)` (an explicitly mentioned variable) and with an empty position.

3.6 How translation is done

In both LPA Prolog and Quintus Prolog, the program loader (consulter) calls `term_expansion/2` to preprocess every term that it reads. If `term_expansion` succeeds, the Prolog system uses its result, rather than the original term.

Accordingly, in GULP, `term_expansion` runs every term through the GULP translator (`g_translate`). Also, GULP contains its own DCG rule expander (slightly modified from one distributed by R. A. O'Keefe), for two reasons: by using `term_expansion` we bypass the built–in Quintus or LPA DCG expander, and by using our own, we ensure that the translations are the same in all versions of Prolog to which GULP is ported.

To make translation possible, GULP maintains a stored set of forward translation schemas, plus one backward schema. For example, a program that uses the features `a`, `b`, and `c` (declared in that order) will result in the creation of the schemas:

```
g_forward_schema(a,X,g_(_,g_(X),_,_)).
g_forward_schema(b,X,g_(_,_,g_(X),_)).
g_forward_schema(c,X,g_(_,_,_,g_(X))).
```

```
g_backward_schema(a:X..b:Y..c:Z,g_(_,X,Y,Z)).
```

Each forward schema contains a feature name, a variable for the feature value, and the minimal corresponding value list. To translate the feature structure `a:xx..b:yy..c:zz`, GULP will mark each of the feature values with `g_()`, and then call, in succession,

```
g_forward_schema(a,g_(xx), ... ),
g_forward_schema(b,g_(yy), ... ),
g_forward_schema(c,g_(zz), ... ) ...
```

and unify the resulting value lists. The result will be the same regardless of the order in which the calls are made. To translate a complex Prolog term, GULP first converts it into a list using '`=..`', then recursively translates all the elements of the list except the first, then converts the result back into a term.

Backward translation is easier; GULP simply unifies the value list with the second argument of `g_backward_schema`, and the first argument immediately yields a rough translation. It is rough in two ways: it mentions all the features in the grammar, and it contains `g_(...)` marking all the feature values that were mentioned in the original feature structure. The finished translation is obtained by discarding all features whose values are not marked by `g_(...)`, and removing the `g_(...)` from values that contain it.

The translation schemas are built automatically. Whenever a new feature is encountered, a forward schema is built for it, and the pre–existing backward schema, if any, is replaced by a new one. A `g_features` declaration causes the immediate generation of schemas for all the features in it, in the order given. In addition, GULP maintains a current `g_features` clause at all times that lists all the features actually encountered, whether or not they were originally declared.

4 GULP in practical use

4.1 A simple definite clause grammar

Figure 3 shows a simple unification–based grammar implemented with the definite clause grammar (DCG) parser that is built into Prolog. Each nonterminal symbol has a GULP feature structure as its only argument.

```
% A grammar in DCG notation, with GULP feature structures.

s(sem: (pred:X .. arg1:Y .. arg2:Z)) -->  np(sem:Y .. case:nom),
                                          vp(sem: (pred:X .. arg2:Z)).

vp(sem: (pred:X1 .. arg2:Y1)) -->  v(sem:X1),
                                   np(sem:Y1).

v(sem:'SEES')  --> [sees].

np(sem:'MAX')  --> [max].

np(sem:'BILL') --> [bill].

np(sem:'ME' .. case:acc) --> [me].


% Procedure to parse a sentence and display its features

try(String) :- writeln([String]),
               phrase(s(Features),String),
               display_feature_structure(Features).


% Example sentences

test1 :- try([max,sees,bill]).
test2 :- try([max,sees,me]).
test3 :- try([me,sees,max]).  /* should fail */
```

Figure 3: Example of a grammar in GULP notation.

Parsing is done top–down. The output of the program reflects the feature structures built during parsing. For example:

```
?- test1.
[max,sees,bill]        (String being parsed)
sem: pred: SEES        (Displayed feature structure)
     arg1: BILL
     arg2: MAX
```

Figure 4 shows the same grammar written in a more PATR–like style. Instead of using feature structures in argument positions, this program uses variables for arguments, then unifies each variable with appropriate feature structures as a separate operation. This is slightly less efficient but can be easier to read, particularly when the unifications to be performed are complex.

In this program, the features of `np` and `vp` are called `NPfeatures` and `VPfeatures` respectively. More commonly, the features of `np`, `vp`, and so on are in variables called `NP`, `VP`, and the like. Be careful not to confuse upper– and lower–case symbols.

The rules in Figure 4 could equally well have been written with the unifications before the constituents to be parsed. That is, we can write either

```
s(Sfeatures) --> np(NPfeatures), vp(VPfeatures),
                                  { Sfeatures = ... }.
```

or

```
s(Sfeatures) --> { Sfeatures = ... },
                            np(NPfeatures), vp(VPfeatures).
```

Because unification is order–independent, the choice affects efficiency but not correctness. The only exception is that some rules can loop when written one way but not the other. Thus

```
s(S1) --> s(S2), { S1 = x:a, S2 = x:b }.
```

loops, whereas

```
s(S1) --> { S1 = x:a, S2 = x:b }, s(S2).
```

does not, because in the latter case `S2` is instantiated to a value that must be distinct from `S1` before `s(S2)` is parsed.

```
% Same as first GULP example, but written in a much more PATR-like style,
% treating the unifications as separate operations.

s(Sfeatures) --> np(NPfeatures), vp(VPfeatures),
                 { Sfeatures = sem: (pred:X .. arg1:Y .. arg2:Z),
                   NPfeatures = sem:Y .. case:nom,
                   VPfeatures = sem: (pred:X .. arg2:Z) }.

vp(VPfeatures) -->  v(Vfeatures), np(NPfeatures),
                    { VPfeatures = sem: (pred:X1 .. arg2:Y1),
                      Vfeatures  = sem:X1,
                      NPfeatures = sem:Y1 }.

v(Features)   --> [sees], { Features = sem:'SEES' }.

np(Features)  --> [max],  { Features = sem:'MAX' }.

np(Features)  --> [bill], { Features = sem:'BILL' }.

np(Features)  --> [me],   { Features = sem:'ME' .. case:acc }.


% Procedure to parse a sentence and display its features

try(String) :- writeln([String]),
               s(Features,String,[]),
               display_feature_structure(Features).


% Example sentences

test1 :- try([max,sees,bill]).
test2 :- try([max,sees,me]).
test3 :- try([me,sees,max]).  /* should fail */
```

Figure 4: The same grammar in more PATR–like notation.

4.2 A hold mechanism for unbounded movements

Unlike a phrase–structure grammar, a unification–based grammar can handle unbounded movements. That is, it can parse sentences in which some element appears to have been moved from its normal position across an arbitrary amount of structure.

Such a movement occurs in English questions. The question–word (*who, what,* or the like) always appears at the beginning of the sentence. Within the sentence, one of the places where a noun phrase could have appeared is empty:

The boy said the dog chased the cat. What did the boy say ␣ chased the cat? (The dog.) What did the boy say the dog chased ␣ ? (The cat.)

Ordinary phrase–structure rules cannot express the fact that only one noun phrase is missing. Constituents introduced by phrase–structure rules are either optional or obligatory. If noun phrases are obligatory, they can't be missing at all, and if they are optional, any number of them can be missing at the same time.

Chomsky (1957) analyzed such sentences by generating what in the position of the missing noun phrase, then moving it to the beginning of the sentence by means of a transformation. This is the generally accepted analysis.

To parse such sentences, one must undo the movement. This is achieved through a hold stack. On encountering what, the parser does not parse it, but rather puts it on the stack and carries it along until it is needed. Later, when a noun phrase is expected but not found, the parser can pop what off the stack and use it.

The hold stack is a list to which elements can be added at the beginning. Initially, its value is `[]` (the empty list). To parse a sentence, the parser must:

1. Pass the hold stack to the NP, which may add or remove items.

2. Pass the possibly modified stack to the VP, which may modify it further.

In traditional notation, the rule we need is:

$$\overset{S}{\left[\ \textit{hold:}\ \begin{bmatrix} \textit{in:H1} \\ \textit{out:H3} \end{bmatrix}\ \right]} \ \rightarrow\ \overset{NP}{\left[\ \textit{hold:}\ \begin{bmatrix} \textit{in:H1} \\ \textit{out:H2} \end{bmatrix}\ \right]} \quad \overset{VP}{\left[\ \textit{hold:}\ \begin{bmatrix} \textit{in:H2} \\ \textit{out:H3} \end{bmatrix}\ \right]}$$

Here *hold* : *in* is the stack before parsing a given constituent, and *hold* : *out* is the stack after parsing that same constituent. Notice that three different states of the stack — `H1`, `H2`, and `H3` — are allowed for.

Figure 5 shows a complete grammar built with rules of this type. There are two rules expanding S. One is the one above ($S \rightarrow NP\ VP$). The other one accepts *what did* at the beginning of the sentence, places *what* on the stack, and proceeds to parse an NP and VP. Somewhere in the NP or VP — or in a subordinate S embedded therein — the parser will use the rule

```
np(NP) --> [], { NP = hold: (in:[what|H1]..out:H1) }.
```

thereby removing *what* from the stack.

4.3 Building discourse representation structures

Figure 6 shows a GULP reimplementation of a program by Johnson and Klein (1986) that makes extensive use of in and out features to pass information around the parse tree. Johnson and Klein's key insight is that the logical structure of a sentence is largely specified by the determiners. For instance, *A man saw a donkey* expresses a simple proposition with universally quantified variables, but *Every man saw a donkey* expresses an "if–then" relationship (If X is a man then X saw a donkey). On the syntactic level, *every* modifies *man*, but semantically, *every* gives the entire sentence a different structure.

Accordingly, Johnson and Klein construct their grammar so that almost all the semantic structure is built by the determiners. Each determiner must receive, from elsewhere in the sentence, semantic representations for its scope and its restrictor. The scope of a determiner is the main predicate of the clause, and the restrictor is an additional condition imposed by the NP to which the determiner belongs. For instance, in *Every man saw a donkey,* the determiner *every* has scope *saw a donkey* and restrictor *man.*

Figure 6 is a reimplementation, in GULP, of a sample program Johnson and Klein wrote in PrAtt (a different extension of Prolog). The semantic representations built by this program are those used in Discourse Representation Theory (Kamp, 1981; Spencer–Smith, 1987). The meaning of a sentence or discourse is represented by a discourse representation structure (DRS) such as:

```
[1,2,man(1),donkey(2),saw(1,2)]
```

```
% S may or may not begin with 'what did'.
% In the latter case 'what' is added to the stack
% before the NP and VP are parsed.

s(S) --> np(NP), vp(VP),
               { S  = hold: (in:H1..out:H3),
                 NP = hold: (in:H1..out:H2),
                 VP = hold: (in:H2..out:H3) }.

s(S) --> [what,did], np(NP), vp(VP),
               { S  = hold: (in:H1..out:H3),
                 NP = hold: (in:[what|H1]..out:H2),
                 VP = hold: (in:H2..out:H3) }.

% NP is parsed by either accepting det and n,
% leaving the hold stack unchanged, or else
% by extracting 'what' from the stack without
% accepting anything from the input string.

np(NP) --> det, n, { NP = hold: (in:H..out:H) }.

np(NP) --> [], { NP = hold: (in:[what|H1]..out:H1) }.


% VP consists of V followed by NP or S.
% Both hold:in and hold:out are the same
% on the VP as on the S or NP, since the
% hold stack can only be altered while
% processing the S or NP, not the verb.

vp(VP) --> v, np(NP), { VP = hold:H,
                        NP = hold:H }.

vp(VP) --> v, s(S), { VP = hold:H,
                      S  = hold:H }.

% Lexicon

det --> [the];[a];[an].
n   --> [dog];[cat];[boy].
v   --> [said];[say];[chase];[chased].

try(X) :- writeln([X]),
          S = hold: (in:[]..out:[]),
          phrase(s(S),X,[]).

test1 :- try([the,boy,said,the,dog,chased,the,cat]).
test2 :- try([what,did,the,boy,say,chased,the,cat]).
test3 :- try([what,did,the,boy,say,the,cat,chased]).
test4 :- try([what,did,the,boy,say,the,dog,chased,the,cat]).
           /* test4 should fail */
```

Figure 5: Grammar with a holding stack.

```
% Discourse Representation Theory  (adapted from Johnson & Klein 1986)

% unique_integer(N)
%          instantiates N to a different integer each time called.

unique_integer(N) :-
            retract(unique_aux(N)),
            !,
            NewN is N+1,
            asserta(unique_aux(NewN)).

:- dynamic(unique_aux/1).
unique_aux(0).

% Nouns
%      Each noun generates a unique index and inserts it, along with
%      a condition, into the DRS that is passed to it.

n(N) --> [man],
              { unique_integer(C),
                N = syn:index:C ..
                sem: (in:  [Current|Super] ..
                      out: [[C,man(C)|Current]|Super]) }.

n(N) --> [donkey],
              { unique_integer(C),
                N = syn:index:C ..
                sem: (in:  [Current|Super] ..
                      out: [[C,donkey(C)|Current]|Super]) }.

% Verbs
%      Each verb is linked to indices of its arguments through syntactic
%      features. Using these indices, it adds appropriate predicate to semantics.

v(V) --> [saw],
          { V = syn: (arg1:Arg1 .. arg2:Arg2) ..
                sem: (in:  [Current|Super] ..
                      out: [[saw(Arg1,Arg2)|Current]|Super]) }.
```

Figure 6: Partial implementation of Discourse Representation Theory (continued on next page).

```
% Determiners
%      Determiners tie together the semantics of their scope and restrictor.
%      The simplest determiner, 'a', simply passes semantic material to its
%      restrictor and then to its scope. A more complex determiner such as
%      'every' passes an empty list to its scope and restrictor, collects whatever
%      semantic material they add, and then arranges it into an if-then structure.

det(Det) --> [a],
          { Det = sem:res:in:A,   Det = sem:in:A,
            Det = sem:scope:in:B, Det = sem:res:out:B,
            Det = sem:out:C,      Det = sem:scope:out:C }.

det(Det) --> [every],
          { Det = sem:res:in:[[]|A],    Det = sem:in:A,
            Det = sem:scope:in:[[]|B],  Det = sem:res:out:B,
            Det = sem:scope:out:[Scope,Res|[Current|Super]],
            Det = sem:out:[[Res>Scope|Current]|Super] }.

% Noun phrase
%      Pass semantic material to determiner, which will specify logical structure.

np(NP) --> { NP=sem:A,      Det=sem:A,
             Det=sem:res:B, N=sem:B,
             NP=syn:C,      N=syn:C },  det(Det),n(N).

% Verb phrase
%      Pass semantic material to the embedded NP (the direct object).

vp(VP) --> { VP = sem:A,          NP = sem:A,
             NP = sem:scope:B,    V = sem:B,
             VP = syn:arg2:C,     NP = syn:index:C,
             VP = syn:D,          V = syn:D },    v(V), np(NP).

% Sentence
%      Pass semantic material to the subject NP.
%      Pass VP semantics to the subject NP as its scope.

s(S) --> { S = sem:A,         NP = sem:A,
           S = syn:B,         VP = syn:B,
           NP = sem:scope:C,  VP = sem:C,
           VP = syn:arg1:D,   NP = syn:index:D }, np(NP), vp(VP).

% Procedure to parse and display a sentence

try(String) :-  write(String),nl,
                Features = sem:in:[[]],   /* start w. empty structure */
                phrase(s(Features),String),
                Features = sem:out:SemOut,  /* extract what was built */
                display_feature_structure(SemOut).
```



```
% Example sentences
test1 :- try([a,man,saw,a,donkey]).
test2 :- try([a,donkey,saw,a,man]).
test3 :- try([every,man,saw,a,donkey]).
test4 :- try([every,man,saw,every,donkey]).
```

Here 1 and 2 stand for entities (people or things), and `man(1)`, `donkey(2)`, and `saw(1,2)` are conditions that these entities must meet. The discourse is true if there are two entities such that 1 is a man, 2 is a donkey, and 1 saw 2. The order of the list elements is insignificant, and the program builds the list backward, with indices and conditions mixed together.

A DRS can contain other DRSes embedded in a variety of ways. In particular, one of the conditions within a DRS can have the form

```
DRS1 > DRS2
```

which means: "This condition is satisfied if for each set of entities that satisfy DRS1, it is also possible to satisfy DRS2." For example:

```
[1,man(1), [2,donkey(2)] > [saw(1,2)] ]
```

"There is an entity 1 such that 1 is a man, and for every entity 2 that is a donkey, 1 saw 2." That is, "Some man saw every donkey." Again,

```
[ [1,man(1)] > [2,donkey(2)] ]
```

means "every man saw a donkey" — that is, "for every entity 1 such that 1 is a man, there is an entity 2 which is a donkey."

Parsing a sentence begins with the rule:

```
s(S) --> { S = sem:A,         NP = sem:A,
           S = syn:B,         VP = syn:B,
           NP = sem:scope:C,  VP = sem:C,
           VP = syn:arg1:D,   NP = syn:index:D }, np(NP), vp(VP).
```

This rule stipulates the following things:

1. An S consists of an NP and a VP.
2. The semantic representation of the S is the same as that of the NP, i.e., is built by the rules that parse the NP.
3. The syntactic feature structure (*syn*) of the S is that of the NP. Crucially, this contains the indices of the subject (*arg1*) and object (*arg2*).
4. The scope of the NP (and hence of its determiner) is the semantic representation of the VP.
5. The index of the verb's subject (*arg1*) is that of the NP mentioned in this rule.

Other rules do comparable amounts of work, and space precludes explaining them in detail here. (See Johnson and Klein 1985, 1986 for further explanation.) By unifying appropriate in and out features, the rules perform a complex computation in an order–independent way.

4.4 Left–corner parsing

GULP is not tied to Prolog's built–in DCG parser. It can be used with any other parser implemented in Prolog. Figure 7 shows how GULP can be used with the BUP left–corner parser developed by Matsumoto et al. (1986) (called a bottom–up parser in the literature because its operation is partly bottom–up).

In bottom–up parsing, the typical question is not "How do I parse an NP?" but rather, "Now that I've parsed an NP, what do I do with it?" BUP puts the Prolog search mechanism to good use in answering questions like this. During a BUP parse, two kinds of goals occur. A goal such as

```
?- np(s,NPf,Sf,[chased,the,cat],[]).
```

means: "An NP has just been accepted; its features are contained in `NPf`. This occurred while looking for an S with features `Sf`. Immediately after parsing the NP, the input string was `[chased,the,cat]`. After parsing the S, it will be `[]`." The other type of goal is

```
?- goal(vp,VPf,[chased,the,cat],[]).
```

This means, "Parse a VP with features `VPf`, starting with the input string `[chased,the,cat]` and ending up with `[]`." This is like the DCG goal

```
?- vp(VPf,[chased,the,cat],[]).
```

except that the parsing is to be done bottom–up.

To see how these goals are constructed, imagine replacing the top–down parsing rule

```
s --> np, vp.
```

with the bottom–up rule

```
np, vp --> s.
```

This rule should be used when the parser is looking for a rule that will tell it how to use an NP it has just found. So `np(...)` should be the head of the Prolog clause. Ignoring feature unifications, the clause will be:

```
% BUP in GULP:
% Left-corner parsing algorithm of Matsumoto et al. (1986).

% Goal-forming clause

goal(G,Gf,S1,S3) :-
        word(W,Wf,S1,S2),
        NewGoal =.. [W,G,Wf,Gf,S2,S3],
        call(NewGoal).

% Terminal clauses for nonterminal symbols

s(s,F,F,X,X).
vp(vp,F,F,X,X).
np(np,F,F,X,X).

% Phrase-structure rules

% np vp --> s

np(G,NPf,Gf,S1,S3) :-  goal(vp,VPf,S1,S2),
                       s(G,Sf,Gf,S2,S3),
                       NPf = sem:Y..case:nom,
                       VPf = sem: (pred:X..arg2:Z),
                       Sf  = sem: (pred:X..arg1:Y..arg2:Z).

% v np --> vp

v(G,Vf,Gf,S1,S3)  :- goal(np,NPf,S1,S2),
                     vp(G,VPf,Gf,S2,S3),
                     Vf  = sem:X1,
                     NPf = sem:Y1..case:acc,
                     VPf = sem: (pred:X1..arg2:Y1).

% Terminal symbols

word(v,sem:'SEES',[sees|X],X).
word(np,sem:'MAX',[max|X],X).
word(np,sem:'BILL',[bill|X],X).
word(np,sem:'ME'..case:acc,[me|X],X).

% Procedure to parse a sentence and display its features

try(String) :- writeln([String]),
               goal(s,Features,String,[]),
               display_feature_structure(Features).

% Example sentences
test1 :- try([max,sees,bill]).
test2 :- try([max,sees,me]).
test3 :- try([me,sees,max]).  /* should fail */
```

Figure 7: Implementation of Matsumoto's BUP.

```
np(G,NPf,Gf,S1,S3) :- goal(vp,VPf,S1,S2),
                      s(G,Sf,Gf,S2,S3).
```

That is: "Having just found an NP with features `NPf`, parse a VP with features `VPf`. You will then have completed an S, so look for a clause that tells you what to do with it."

Here `S1`, `S2`, and `S3` represent the input string initially, after parsing theVP, and after completing the S. `G` is the higher constituent that was being sought when the NP was found, and `Gf` contains its features. If, when the S is completed, it turns out that an S was being sought (the usual case), then execution can finish with the terminal rule

```
s(s,F,F,X,X).
```

Otherwise another clause for `s(...)` must be searched for.

Much of the work of BUP is done by the goal–forming predicate `goal`, defined thus:

```
goal(G,Gf,S1,S3) :-
  word(W,Wf,S1,S2),
  NewGoal =.. [W,G,Wf,Gf,S2,S3],
  call(NewGoal).
```

That is (ignoring features): "To parse a `G` in input string `S1` leaving the remaining input in `S3`, first accept a word, then construct a new goal depending on its category (`W`)." For example, the query

```
?- goal(s,Sf,[the,dog,barked],S3).
```

will first call

```
?- word(W,Wf,[the,dog,barked],[dog,barked]).
```

thereby instantiating `W` to `det` and `Wf` to the word's features, and then construct and call the goal

```
?- det(s,Wf,Sf,[dog,barked],S3).
```

That is: "I've just completed a `det` and am trying to parse an `s`. What do I do next?" A rule such as

```
det, n --> np
```

(or rather its BUP equivalent) can be invoked next, to accept another word (a noun) and complete an NP.

5 Future Prospects

5.1 Possible improvements

One disadvantage of GULP is that every feature structure must contain a position for every feature in the grammar. This makes feature structures larger and slower to process than they need be. By design, unused features often fall in the uninstantiated tail of the value list, and hence take up neither time nor space. But not all unused features have this good fortune. In practice, almost every value list contains gaps, i.e., positions that will never be instantiated, but must be passed over in every unification.

To reduce the number of gaps, GULP could be modified to distinguish different types of value lists. The feature structure for a verb needs a feature for tense; the feature structure for a noun does not. Value lists of different types would reserve the same positions for different features, skipping features that would never be used. Some kind of type marker, such as a unique functor, would be needed so that value lists of different types would not unify with each other.

Types of feature structures could be distinguished by the programmer — e.g., by giving alternative `g_features` declarations — or by modifying the GULP translator itself to look for patterns in the use of features. Some grammatical formalisms, such as that of Pollard and Sag (1994), explicitly specify types (sorts) for all feature structures.

5.2 Keyword parameters via GULP

Unification–based grammar is not the only use for GULP. Feature structures are a good formalization of keyword–value argument lists.

Imagine a complicated graphics procedure that takes arguments indicating desired window size, maximum and minimum coordinates, and colors, all of which have default values. In Pascal, the procedure can only be called with explicit values for all the parameters:

```
OpenGraphics(480,640,-240,240,-320,320,green,black);
```

There could, however, be a convention that 0 means "take the default:"

```
OpenGraphics(0,0,0,0,0,0,red,blue);
```

Prolog can do slightly better by using uninstantiated arguments where defaults are wanted, and thereby distinguishing “default” from “zero:”

```
?- open_graphics(_,_,_,_,_,_,red,blue).
```

In GULP, however, the argument of `open_graphics` can be a feature structure in which the programmer mentions only the non–default items:

```
?- open_graphics( foreground:red..background:blue ).
```

In this feature structure, the values for `x_resolution`, `y_resolution`, `x_maximum`, `x_minimum`, `y_maximum`, and `y_minimum` (or whatever they are called) are left uninstantiated because they are not mentioned. So in addition to facilitating the implementation of unification–based grammars, GULP provides Prolog with a keyword argument system.

6 References

Barton, G. Edward; Berwick, Robert C.; and Ristad, Eric Sven. 1987. Computational complexity and natural language. Cambridge, Massachusetts: MIT Press.

Bouma, Gosse; König, Esther; and Uszkoreit, Hans. 1988. A flexible graph–unification formalism and its application to natural–language processing. IBM Journal of Research and Development 32:170–184.

Bresnan, Joan, ed. 1982. The mental representation of grammatical relations. Cambridge, Massachusetts: MIT Press.

Chomsky, Noam. 1957. Syntactic structures. (Janua linguarum, 4.) The Hague: Mouton.

Chomsky, Noam. 1965. Aspects of the theory of syntax. Cambridge, Massachusetts: MIT Press.

Covington, Michael A. 1987. GULP 1.1: an extension of Prolog for unification–based grammar. ACMC Research Report 01–0021. Advanced Computational Methods Center, University of Georgia.

Covington, Michael A. 1989. GULP 2.0: an extension of Prolog for unification–based grammar. Research Report AI–1989–01, Artificial Intelligence Programs, University of Georgia.

Covington, Michael A. 1994. Natural language processing for Prolog programmers. Englewood Cliffs, N.J.: Prentice–Hall.

Covington, Michael A.; Nute, Donald; and Vellino, André. 1988. Prolog programming in depth. Glenview, Ill.: Scott, Foresman.

Gazdar, Gerald; Klein, Ewan; Pullum, Geoffrey; and Sag, Ivan. 1985. Generalized phrase structure grammar. Cambridge, Massachusetts: Harvard University Press.

Giannesini, Francis; Kanoui, Henry; Pasero, Robert; and van Caneghem, Michel. 1986. Prolog. Wokingham, England: Addison–Wesley.

Johnson, Mark, and Klein, Ewan. 1985. A declarative formulation of Discourse Representation Theory. Paper presented at the summer meeting of the Association for Symbolic Logic, July 15–20, 1985, Stanford University.

Johnson, Mark, and Klein, Ewan. 1986. Discourse, anaphora, and parsing. Report No. CSLI–86–63. Center for the Study of Language and Information, Stanford University. Also in Proceedings of Coling86 669–675.

Joshi, Aravind K. 1986. The convergence of mildly context–sensitive grammar formalisms. Draft distributed at Stanford University, 1987.

Kamp, Hans. 1981. A theory of truth and semantic representation. Reprinted in Groenendijk, J.; Janssen, T. M. V.; and Stokhof, M., eds., Truth, interpretation, and information. Dordrecht: Foris, 1984.

Kaplan, Ronald M., and Bresnan, Joan. 1982. Lexical–Functional Grammar: a formal system for grammatical representation. Bresnan 1982:173–281.

Karttunen, Lauri. 1986a. D–PATR: a development environment for unification–based grammars. Report No. CSLI–86–61. Center for the Study of Language and Information, Stanford University. Shortened version in Proceedings of Coling86 74–80.

Karttunen, Lauri. 1986b. Features and values. Shieber et al. 1986 (vol. 1), 17–36. Also in Proceedings of Coling84 28–33.

Matsumoto, Yuji; Tanaka, Hozumi; and Kiyono, Masaki. 1986. BUP: a bottom–up parsing system for natural languages. Michel van Caneghem and David Warren, eds., Logic programming and its applications 262–275. Norwood, N.J.: Ablex.

Pollard, Carl, and Sag, Ivan A. 1987. Information–based syntax and semantics, vol. 1: Fundamentals. (CSLI Lecture Notes, 13.) Center for the Study of Language and Information, Stanford University.

Pollard, Carl, and Sag, Ivan A. 1994. Head–driven phrase–structure grammar. Chicago: University of Chicago Press.

Schöter, Andreas (1993) Compiling feature structures into terms: an empirical study in Prolog. Thesis, M.Sc., Centre for Cognitive Science, University of Edinburgh.

Sedogbo, Celestin. 1986. AVAG: an attribute/value grammar tool. FNS–Bericht 86–10. Seminar für natürlich–sprachliche Systeme, Universität Tübingen.

Shieber, Stuart M. 1986a. An introduction to unification–based approaches to grammar. (CSLI Lecture Notes, 4.) Center for the Study of Language and Information, Stanford University.

Shieber, Stuart M. 1986b. The design of a computer language for linguistic information. Shieber et al. (eds.) 1986 (vol. 1) 4–26.

Shieber, Stuart M.; Pereira, Fernando C. N.; Karttunen, Lauri; and Kay, Martin, eds. A compilation of papers on unification–based grammar formalisms. 2 vols. bound as one. Report No. CSLI–86–48. Center for the Study of Language and Information, Stanford University.

Spencer–Smith, Richard. 1987. Semantics and discourse representation. Mind and Language 2.1: 1–26.