Research Report AI-1989-01
GULP 2.0: An Extension of Prolog for Unification-Based Grammar

Michael A. Covington

Advanced Computational Methods Center
University of Georgia
Athens, Georgia 30602

January 1989

Abstract

ABSTRACT: A simple extension to Prolog facilitates implementation of unification-based grammars (UBGs) by adding a new notational device, the *feature structure*, whose behavior emulates graph unification. 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. The extended language is known as GULP (Graph Unification Logic Programming); it is as powerful and concise as PATR-II (Shieber 1986a,b) and other grammar development tools, while retaining all the versatility of Prolog. GULP can be used with grammar rule notation (DCGs) or any other parser that the programmer cares to implement. Besides its uses in natural language processing, GULP provides a way to supply keyword arguments to any procedure.

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.[1] Further, feature structures can be used not only in natural language processing, but also to pass keyword arguments to any procedure.

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.[2] 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 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.

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

[1]The first version of GULP (Covington 1987) was developed with support from National Science Foundation Grant IST-85-02477. I want to thank Franz Guenthner, Rainer Bäuerle, and the other researchers at the Seminar für natürlich-sprachliche Systeme, University of Tübingen, for their hospitality and for helpful discussions. The opinions and conclusions expressed here are solely those of the author.

[2]The use of the colon makes the Quintus Prolog and Arity Prolog module systems unavailable; so far, this has not caused problems.

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), 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:

The	dog	barks.
category:determiner	category:noun number:singular	category:verb number:singular person:3rd tense:present

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 --> noun phrase + verb phrase

noun phrase --> determiner + noun
```

plus transformational rules that rearranged syntactic structure. Syntactic structure was described by tree diagrams (Figure 1).[3] Number and tense

[3]Figures are printed at the end of this document

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 ($verb_{transitive}$ and $verb_{intransitive}$).

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 (Figure 2). 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.

But 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

```
[2.3a]      PP          -->         P          NP
                                                [case:acc]
```

says that in a prepositional phrase, the NP must be in the accusative case.

More precisely, rule [2.3a] says the feature structure

```
[case:acc]
```

must be *unified* (merged) with whatever features the NP already has. If the NP already has `case:acc`, all is well. If the NP has no value for `case`, it acquires `case:acc`. But if the NP has `case` with some value other than acc, the unification fails and the rule cannot apply.

Agreement is handled with variables, as in the rule

$$[2.3b]\ S \rightarrow \begin{matrix} NP \\ \left[\begin{matrix} person{:}X \\ number{:}Y \end{matrix}\right] \end{matrix} \quad \begin{matrix} VP \\ \left[\begin{matrix} person{:}X \\ number{:}Y \end{matrix}\right] \end{matrix}$$

which requires the `NP` and `VP` to agree in person and number. Here `X` and `Y` are variables; `person:X` merges with the `person` feature of both the `NP` and the `VP`, thereby ensuring that the same value is present in both places. The same thing happens with `number`.

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

```
    NP
[case:acc]
```

is short for:

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

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 [2.3a] and [2.3b] become:

```
[2.3c]      PP --> P NP       NP case = acc

[2.3d]      S --> NP VP       NP person = VP person
                              NP number = VP number
```

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

```
[2.3e]      X --> Y Z         X category = PP
                              Y category = P
                              Z category = NP
                              Z case = acc

[2.3f]      X --> 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 not in the system described here.

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 3 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 internal structure of the NP is ignored and the proposal to group syntactic features together is abandoned.

To see how the grammar works, consider how the sentence *Max sees Bill* would be parsed bottom-up. The process is shown in Figure 4. First rules [c], [d], and [e] supply the features of the individual words (Figure 4a). Next the bottom-up parser attempts to build constituents.

By rule [b], sees and Bill constitute a VP (Figure 4b). At this step, construction of a semantic representation begins. The sem feature of the VP has as its value another feature structure which contains two features: `pred`, the semantics of the verb, and `arg2`, the semantics of the direct object.

Rule [b] also assigns the feature `case:acc` to *Bill*; this has no effect on the form of the noun but would be important if a pronoun had been used instead.

Finally, rule [a] allows the resulting NP and VP to be grouped together into an S (Figure 4c). This rule assigns nominative case to *Max* and combines the semantics of the NP and VP to construct the `sem` feature of the S node, thereby accounting for the meaning of the complete sentence.

It would be equally feasible to parse top-down. Parsing would then begin with an S node, expanded to NP and VP by rule [a]. The NP would then expand to *Max* using rule [d], thereby supplying a value for `sem` of NP, and hence also for `sem:arg1` of S. Similarly, expansion of the VP would supply values for the remaining features of S.

Crucially, it is possible (and necessary) to match variables with each other before giving them values. In a top-down parse, we know that `sem:arg2` of S will have the same value as `sem:arg2` of VP long before we know what this value is to be.

2.5 Functions, paths, re-entrancy, and graphs

A feature can be viewed as a partial function which, given a feature structure, may or may not yield a value. For instance, given the structure

$$\begin{bmatrix} \textit{syn:} \begin{bmatrix} \textit{case:acc} \\ \textit{gender:masc} \end{bmatrix} \\ \textit{sem:MAN} \end{bmatrix}$$

the feature `sem` yields the value `MAN`, the feature `syn` yields another feature structure, and the feature `tense` yields no value (it is a case in which the partial function is undefined).

A *path* is a series of features that pick out an element of a nested feature structure. Formally, a path is the composition of the functions just mentioned. For example, the path `syn:case` is what you get by applying the function `case` to the value of the function `syn`; applied to the structure

above, `syn:case` yields the value `acc`. Path notation provides a way to refer to a single feature deep in a nested structure without writing nested brackets. Thus one can write rules such as

```
P  -->    P        NP
                [syn:case:acc]
```

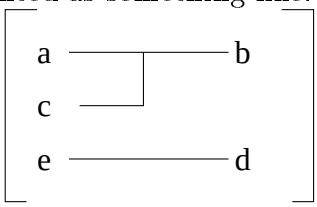
or, in equational form,

```
P --> P NP              NP syn case = acc
```

Feature structures are *re-entrant*. This means that features are like pointers; if two of them have the same value, then they point to the *same object*, not to two similar objects. If this object is subsequently modified (e. g., by giving values to variables), the change will show up in both places. Thus the structure

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

is more accurately represented as something like:



There is only one `b`, and `a` and c both point to it.

Re-entrant feature structures can be formelized as directed acyclic graphs (DAGs) as shown in Figure 5. Features are arcs and feature values are the vertices or subgraphs found at the ends of the arcs. A path is a series of arcs chained together.

Re-entrancy follows from the way variables behave in a grammar. All occurrences of the same variable take on the same value at the same time. (As in Prolog, like-named variables in separate rules are not considered to be the same variable.) The value may itself contain a variable that will later get a value from somewhere else. This is why bottom-up and top-down parsing work equally well.

2.6 Unification

Our sample grammar relies on the merging of partially specified feature structures. Thus, the subject of the sentence gets case from rule [a] and semantics from rule [d] or [e]. 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:

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

b 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.

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

Thus

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

unify giving:

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

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

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

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 *Max sees me* but not *Me sees Max*. In *Max sees me*, both rule [b] and rule [f] specify that *me* has the feature `case:acc`, giving the structure shown in Figure 6.

However, in *Me sees Max*, the case of *me* raises a conflict. Rule [a] specifies *case:nom* and rule [f] specifies `case:acc`. These values are not unifiable; hence the specified merging of feature structures cannot go through, and the sentence is not generated by the grammar.

2.7 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. Re-

call that grammar rules of the form

```
A --> 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

```
  A     -->      A      B
[f:X]          [f:Y]
```

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.

2.8 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

$$\begin{bmatrix} & \overset{VP}{} \\ sem: & \begin{bmatrix} pred{:}X \\ arg{:}Y \end{bmatrix} \end{bmatrix} \rightarrow \overset{V}{\begin{bmatrix} sem{:}X \end{bmatrix}} \quad \overset{NP}{\begin{bmatrix} sem{:}Y \end{bmatrix}}$$

builds on the VP node a *pred-arg* structure that is absent on the V and NP.

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:

$$\begin{array}{ccccc} S & & NP & & VP \\ \left[sem: \left[\begin{array}{l} in{:}X1 \\ out{:}X3 \end{array} \right] \right] & \rightarrow & \left[sem: \left[\begin{array}{l} in{:}X1 \\ out{:}X2 \end{array} \right] \right] & & \left[sem: \left[\begin{array}{l} in{:}X2 \\ out{:}X3 \end{array} \right] \right] \end{array}$$

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 (below) 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

$$\left[\begin{array}{l} a{:}b \\ c{:}d \end{array} \right]$$

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)`.[4] The same structure can be written as

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

which GULP renders as `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. Thus, the rule

$$\begin{matrix} S \\ \begin{bmatrix} person{:}X \\ number{:}Y \end{bmatrix} \end{matrix} \rightarrow \begin{matrix} NP \\ \begin{bmatrix} person{:}X \\ number{:}Y \end{bmatrix} \end{matrix} \quad lfs\,VP person{:}X number{:}Y$$

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

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

They 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`.

[4] Arity Prolog 4.0 requires a space before the '('.

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` unify, 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 wide range 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.[5] 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)` and `A..B..C = A..(B..C)`. Arity Prolog 4.0 requires an intervening space when ':' or '..' occurs adjacent to a left parenthesis; other Prologs lack this restriction.

3.3 Built-in predicates

GULP 2.0 is an ordinary Prolog environment with some built-in predicates added. The most important of these is `load`, which loads clauses into memory

[5] For compatibility with earlier versions, '..' can also be written '::'.

through the GULP translator. (A `consult` or `reconsult` would not translate feature structures into their internal representations.) Thus,

```
?- load myprog.
```

loads clauses from the file MYPROG.GLP.

Like `reconsult`, `load` clears away any pre-existing clauses for a predicate when new clauses for that predicate (with the same arity) are first encountered in a file. However, `load` does not require the clauses for a predicate to be contiguous, so long as they all occur in the same file. A program can consist of several files that are loaded into memory together.

Another predicate, `ed`, calls a full-screen editor and then loads the file. Without an argument, `ed` or `load` uses the same file name as on the most recent invocation of either `ed` or `load`.

Other special predicates are used within the program. GULP 1.1 required a declaration such as

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

declaring all feature names before any were used. This declaration is optional in GULP 2.0. If present, it establishes the order in which features will appear whenever a feature structure is output, and it can be used to optimize the program by putting frequently used features at the beginning. Further, whether or not the programmer includes a `g_features` declaration, GULP 2.0 maintains in memory an up-to-date `g_features` clause with a list of all the features actually used, in the order in which they were encountered.

The predicate `g_translate/2` interconverts feature structures and their internal representations. This makes it possible to process, at runtime, feature structures in GULP notation rather than translated form. For instance, if `X` is a feature structure, then `g_translate(Y,X)`, `write(Y)` will display it in GULP notation.

The predicate `display_feature_structure` 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.

3.4 Internal representation

The nature of value lists, which represent feature structures internally, is best approached by a series of approximations. The nearest Prolog equivalent to a feature structure is a complex term with one position reserved for the value of every feature. Thus

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

could be represented as `x(plural,third,fem)` or `[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

$$\begin{bmatrix} person{:}third \end{bmatrix}$$

would be represented as `x(_,third,_)` or `[_,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, since an instantiated Prolog variable is actually a pointer to the memory representation of its value.

All the feature structures in a program must be unifiable unless they contain conflicting values. Accordingly, if fifteen features are used in the program, every value list must reserve positions for all fifteen. One option would be to represent value lists as 15-argument structures:

```
tense:present  =>      x(_,_,_,_,present,_,_,_,_,_,_,_,_,_,_)
```

This obviously wastes memory. A better solution would be to use lists; a list with an uninstantiated tail unifies with any longer list. The improved representation is:

```
tense:present  =>      [_,_,_,_,present|_]
```

By putting frequently used features near the beginning, this representation can save a considerable amount of memory as well as reducing the time needed to do unifications. Further, lists with uninstantiated tails gain length

automatically as further elements are filled in; unifying `[a,b,c|_]` with `[_,_,_,_,e|_]` gives `[a,b,c,_,e|_]`.

If most of the lists in the program have uninstantiated tails, the program can be simplified by requiring all lists to have uninstantiated tails. Any process that searches through a list will then need to check for only one terminating condition (remainder of list uninstantiated) rather than two (remainder of list uninstantiated or empty).

But the GULP internal value list structure is not an ordinary list. If it were, translated feature structures would be confused with ordinary Prolog lists, and programmers would fall victim to unforeseen unifications. It would also be impossible to test whether a term is a value list.

Recall that Prolog lists are held together by the functor '.'. That is,

```
[a,b,c|X]    =    .(a,.(b,.(c,X)))
```

To get a distinct type of list, all we need to do is substitute another functor for the dot. GULP uses `g_/2`. (In fact, all functors beginning with `g_` are reserved by GULP for internal use.) So if tense is the fifth feature in the canonical order, then

```
tense:present    =>    g_(_,g_(_,g_(_,g_(_,g_(present,_)))))
```

It doesn't matter that this looks ugly; the GULP programmer never sees it.

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_(g_(X),g_(_,g_(_,g_(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.

3.5 How translation is done

GULP loads a program by reading it, one term a a time, from the input file, and translating all the feature structures in each term into value lists. The term is then passed to the built-in predicate `expand_term`, which translates grammar rule (DCG) notation into plain Prolog. The result is then `asserted` into the knowledge base. There are two exceptions: a term that begins with ':-' is executed immediately, just as in ordinary Prolog, and a `g_features` declaration is given special treatment to be described below.

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` (encountered in that order) will result in the creation of the schemas:

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

g_backward_schema(a:X..b:Y..c:Z,g_(X,g_(Y,g_(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 7 shows the grammar from Figure 3 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.

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 8 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 8 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.

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:

$$\underset{S}{\left[\, hold\text{:}\ \left[\begin{array}{l} in\text{:}\ H1 \\ out\text{:}\ H3 \end{array}\right] \,\right]} \rightarrow \underset{NP}{\left[\, hold\text{:}\ \left[\begin{array}{l} in\text{:}\ H1 \\ out\text{:}\ H2 \end{array}\right] \,\right]} \quad \underset{VP}{\left[\, hold\text{:}\ \left[\begin{array}{l} in\text{:}\ H2 \\ out\text{:}\ H3 \end{array}\right] \,\right]}$$

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 9 shows a complete grammar built with rules of this type. There are two rules expanding S. One is the one above (S –¿ 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 complex semantic structures

Figure 10 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 only man, but semantically, it 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 10 shows 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)]
```

Here `1` and `2` stand for entities (people or things), end `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`. In other words, "A man saw a donkey." 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 Bottom-up parsing

GULP is not tied to Prolog's built-in DCG parser. It can be used with any other parser implemented in Prolog. Figure 11 shows how GULP can be used with the BUP bottom-up parser developed by Matsumoto et al. (1986).[6]

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:

[6]It has been suggested that a combination of GULP and BUP should be known as BURP. This suggestion has not been acted upon.

```
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 the VP, 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 em 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 Comparison with other systems

5.1 GULP versus PATR-II

PATR-II (Shieber 1986a, b) is the most widely used software tool for implementing unification-based grammars, as well as the most mature and sophisticated. It differs from GULP in three main ways:

(1) Whereas GULP is an extension of Prolog, PATR-II is a new self-contained programming language.

(2) Whereas GULP allows the use of any parsing algorithm, PATR-II provides one specific parser (left-corner, Earley, or Cocke-Kasami-Younger, depending on the version).

(3) Whereas GULP grammar rules treat feature structures as data items, PATR-II grammar rules state equations on feature values.

Of these, (3) makes the biggest practical difference. The rule which GULP writes as

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

(assuming use of the DCG parser) is rendered in PATR-II as:

```
Rule S --> NP VP:
     <S number>  =  <NP number>
     <S number>  =  <VP number>
     <S person>  =  <NP person>
     <S person>  =  <VP person>.
```

or the like. Paths are permitted, of course; one could write `<NP syn agr number>` to build a more complex structure.

Here S, NP, and VP are not pure variables; the equations

```
<S cat>  = s
<NP cat> = np
<VP cat> = vp
```

(or the equivalent) are implicit. Further abbreviatory power comes from *templates*, which are predefined sets of features and values. Thus, instead of writing the lexical entry

```
Word sleeps:  <cat> = v
              <person> = third
              <number> = singular
              <subcat> = intransitive.
```

the PATR-II programmer can define the template

```
Let ThirdSingVerb be  <cat> = v
                      <person> = third
                      <number> = singular.
```

and then write:

```
Word sleeps:  ThirdSingVerb
              <subcat> = intransitive.

Word chases:  ThirdSingVerb
              <subcat> = transitive.
```

The GULP equivalent of a template is a Prolog fact such as:

```
thirdsingverb(person:third..number:singular).
```

Lexical entries can then use this as an abbreviatory device:

```
v(Vf) --> [sleeps], { thirdsingverb(Vf),
                      Vf = subcat:intransitive }.

v(Vf) --> [chases], { thirdsingverb(Vf),
                      Vf = subcat:transitive }.
```

(There is no `cat:v` here because in the DCG parser, categories are functors rather than feature values.)

Unlike GULP, PATR-II provides for *default inheritance*. That is, the programmer can invoke a template and then change some of the values that it supplies, thus:

```
Word does:  ThirdSingVerb
            <cat> = auxverb.
```

This means: "Does is a ThirdSingVerb except that its category is not `v` but rather `auxverb`." PATR-II+ also provides for *lexical redundancy rules* that transform one lexical entry into another, e. g., building a passive verb from every active verb.

Both of these capabilities are absent from GULP per se, but they could be built into a parser written in GULP. Indeed, many contrasts between GULP and PATR-II reflect the fact that PATR-II is a custom-built environment for implementing grammars that fit a particular mold, while GULP is a minimal extension to a much more general-purpose programming language.

One advantage of GULP is that the full range of Prolog data structures is available. Shieber (1986a:28-32) equips each verb with an ordered list of NPs that are its syntactic arguments (subject, object, etc.). But there are no lists in PATR-II, so Shieber has to construct them as nested feature structures:

$$\left[\begin{array}{ll} \textit{first: } \ldots \textit{first element} \ldots & \\ \textit{rest:} & \left[\begin{array}{ll} \textit{first: } \ldots \textit{second element} \ldots & \\ \textit{rest:} & \left[\begin{array}{ll} \textit{first: } \ldots \textit{third element} \ldots & \\ \textit{rest:} & \left[\begin{array}{l} \ldots \textit{fourth element} \ldots \\ \textit{rest: end} \end{array}\right] \end{array}\right] \end{array}\right] \end{array}\right]$$

This may be desirable on grounds of theoretical parsimony, but it is notationally awkward. In GULP, one can simply write `[X1,X2,X3,X4]`, where `X1`, `X2`, `X3`, and `X4` are variables, constants, feature structures, or terms of any other kind.

5.2 GULP versus PrAtt

PrAtt (**Pr**olog with **Att**ributes), described briefly by Johnson and Klein (1986), is a PATR-like extension of Prolog. In PrAtt, feature structure equations are treated as Prolog goals. An example is the DCG rule:

```
s(Sf) --> np(NPf), vp(VPf),
          { Sf:number = NPf:number,
            Sf:number = VPf:number,
            Sf:person = NPf:person,
            Sf:person = VPf:person }.
```

This looks almost like GULP syntax, but the meaning is different. `NPf:number` is not a Prolog term, but rather an evaluable expression; at execution time, it is replaced by the `number` element of structure `NPf`.

Compared to GULP, PrAtt makes a much bigger change to the semantics of Prolog. GULP merely translates data into data (changing the format from feature structures to value lists), but PrAtt translates data into extra operations.

An example will make this clearer. In order to execute `Sf:number = NPf:number`, PrAtt must extract the `number` features of `Sf` and `NPf`, then unify them. In Johnson and Klein's implementation, this extraction is done at run time; that is, on find the expression `Sf:number`, the PrAtt interpreter looks at the contents of `Sf`, and then replaces `Sf:number` with the value of `Sf`'s `number` feature.

This implies that the value of `Sf` is known. If it is not — for example, if the PrAtt-to-Prolog translation is being performed before running the program - - then extra goals must be inserted into the program to extract the appropriate feature values. The single PrAtt goal `Sf:number = NPf:number` becomes at least three goals:

(1) Unify `Sf` with something that will put the number value into a unique variable (call it `X`).

(2) Unify `NPf` with something that will put its number value into a unique variable (call it `Y`).

(3) Unify `X` and `Y`.

To put this another way, whereas GULP modifies the syntax for Prolog terms, PrAtt modifies the unification algorithm, using three calls to the existing Prolog unification algorithm to perform one PrAtt unification.

5.3 GULP versus AVAG

AVAG (**A**ttribute-**V**alue **G**rammar, Sedogbo 1986) is an implementation of generalized phrase structure grammar (GPSG), a framework for expressing linguistic analyses. A three-pass compiler translates AVAG notation into Prolog II. As such, AVAG is far more complex than GULP or PrAtt, and

there is little point in making a direct comparison. Comparing AVAG to PATR-II would be instructive but is outside the scope of this paper.

AVAG is interesting because it uses the Prolog II built-in predicates `dif` (which means "this variable must never be instantiated to this value") and `freeze` ("wait until these variables are instantiated, then test them") to implement negative-valued and set-valued features respectively. For example, the rule

```
voit:
      <cat> = verb
      <person> /= 2
      <number> = sing.
```

uses `dif` to ensure that the `person` feature never equals 2, and

```
chaque:
      <cat> = art
      <gender> = [mas,fem]
      <number> = sing.
```

uses `freeze` to ensure that when the `gender` feature becomes instantiated, its value is `mas` or `fem`. There are no direct equivalents for `dif` or `freeze` in conventional (Edinburgh) Prolog; they could be implemented only by changing the inference engine.

5.4 GULP versus STUF

STUF (**Stu**ttgart **F**ormalism) is a formal language for describing unification-based grammars. It is more comprehensive than PATR-II and as yet is only partly implemented (Bouma et al. 1988).

Comparing STUF to GULP would be rather like comparing linear algebra to Fortran; the systems are not in the same category. Nonetheless, STUF introduces a number of novel ideas that could be exploited in parsers or other systems written in GULP.

The biggest of these is nondestructive unification. In Prolog, unification is a destructive operation; terms that are being unified are *replaced* by their unifier. For example, if `X = [a,_]` and `Y = [_,b]`, then after unifying `X` and `Y`, `X = Y = [a,b]`. In STUF, on the other hand, an expression such as

```
z = (x y)
```

creates a third structure `z` whose value is the unifier of `x` and `y`; `x` and `y` themselves are unaffected. Nondestructive unification can be implemented in Prolog by copying the terms before unifying them (Covington et al. 1988:204).

Further, if the unification fails, `z` gets the special value `FAIL`. If `x` and `y` are feature structures, and parts of them are unifiable but other parts are not, the non-unifiable parts will be represented by `FAIL` in the corresponding parts of `z`. This provides a way to implement negative-valued features. For example, to ensure that a verb is not third person singular, one can stipulate that when its `person` feature is unified with `person:3`, the result is `FAIL`.

In STUF, a feature can also have a set of alternatives as its value, and when two structures containing such sets are unified, the unifier is the set of all the unifiers of structures that would result from choosing different alternatives.

Finally, STUF exploits the fact that grammar rules can themselves be treated as feature structures. For example, the rule

$$S \rightarrow \begin{array}{c} NP \\ \left[\begin{array}{l} person{:}X \\ number{:}Y \end{array}\right] \end{array} \quad \begin{array}{c} VP \\ \left[\begin{array}{l} person{:}X \\ number{:}Y \end{array}\right] \end{array}$$

(or more precisely the tree structure that it sanctions) can be expressed as the feature structure

$$\left[\begin{array}{lll} mother: & \left[\begin{array}{l} category:\ s \end{array}\right] & \\ & & \\ daughter\_1: & \left[\begin{array}{l} category:\ np \\ person:\ X \\ number:\ Y \end{array}\right] & daughter\_2: \quad \left[\begin{array}{l} category:\ vp \\ person:\ X \\ number:\ Y \end{array}\right] \end{array}\right]$$

STUF therefore implements grammar rules via "graph application," an operation that treats one feature structure (directed acyclic graph) as a function to be applied to another. Graph application is an operation with four arguments: (1) a graph expressing the function; (2) a graph to be treated as the argument; (3) a path indicating what part of the argument graph is to be unified with the function graph; and (4) a path indicating what part of the argument graph should be unified (destructively) with the result of the first unification. The ordinary GULP practice of simply unifying one graph

with another is a special case of this.

6 Future Prospects

6.1 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.

6.2 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.

References

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

[2] 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.

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

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

[5] Chomsky, Noam. 1965. *Aspects of the theory of syntax.* Cambridge, Massachusetts: MIT Press.

[6] 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.

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

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

[9] Giannesini, Francis; Kanoui, Henry; Pasero, Robert; and van Caneghem, Michel. 1986. *Prolog.* Wokingham, England: Addison-Wesley.

[10] 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.

[11] 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.

[12] Joshi, Aravind K. 1986. The convergence of mildly context-sensitive grammara formalisms. Draft distributed at Stanford University, 1987.

[13] 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.

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

[15] 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.

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

[17] 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.

[18] 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.

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

[20] 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.

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

[22] 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.

[23] Spencer-Smith, Richard. 1987. Semantics and discourse representation. *Mind and Language* 2.1: 1–26.

Appendix. GULP 2.0 User's Guide

A Overview

GULP is a laboratory instrument, not a commercial product. Although reasonably easy to use, it lacks the panache and sophistication of Turbo Pascal or Arity Prolog 5.0. The emphasis is on getting the job done as simply as possible.

The final word on how GULP works is contained in the file GULP.ARI or GULP.PL, which you should consult whenever you have a question that is not answered here.

B Installation and access

On the VAX, GULP is already installed, and you reach it by the command

```
$ gulp
```

This puts you into a conventional Prolog environment (note: *not* a Quintus Prolog split-screen environment) in which the GULP built-in predicates are available.

The IBM PC version of GULP is supplied as a modified copy of Arity Prolog Interpreter 4.0. It is for use *only on machines for which Arity Prolog is licensed* and is not for distribution outside the AI Research Group.

Many of the GULP file names are the same as files used by the unmodified Arity Prolog Interpreter. It is therefore important that GULP be installed in a *different* directory.

To run GULP you also need a full-screen editor that is accessible by the command:

edit *filename*

GULP passes commands of this form to DOS when invoking the editor. We usually use AHED.COM, renamed EDIT.COM, for the purpose, but you can use any editor that produces ASCII files.

C How to run programs

GULP is simply a version of Prolog with more built-in predicates added. All the functions and features of Prolog[7] are still available and work exactly as before. GULP is used in exactly the same way as Prolog except that:

(1) Programs containing feature structures must be loaded via the built-in predicate **load**, *not* **consult** or **reconsult**. The reason is that **consult** or **reconsult** would load the feature structures into memory without converting them into value lists. Prolog will do this without complaining, but GULP programs will not work. Never use **consult** or **reconsult** to load anything that contains GULP feature structures.

(2) You must always invoke the editor with the GULP command **ed**, *not* with whatever command you would use in Prolog. This is important because your ordinary editing command would automatically invoke **reconsult** after editing; **ed** invokes **load** instead.

(3) You cannot use feature structure notation in a query because queries do not go through the translator. Write your program so that you can invoke all the necessary predicates without having to type feature structures on the command line.

D Built-in predicates usually used as commands

?- load *filename.*

Loads the program on file *filename* into memory via the GULP translator. **load** is like **reconsult** in that, whenever a predicate is encountered that is already defined, the old definition is discarded before the new definition is loaded. This means that all clauses defining a predicate must be on the same file, but they need not be contiguous. Further, you can load definitions of

[7]Except the module system, which uses the colon(':') for its own purposes and conflicts with GULP syntax.

different predicates from different files without conflict. Further, you can embed a call to **load** in a program to make it load another program.

If the file name is not in quotes, the ending .GLP is added. If the file name contains a period, it must be typed in quotes (single or double).

?- load.

Loads (again) the file that was used in the most recent call to **load** or **ed**.

?- ed *filename*.

Calls the editor to process file *filename*, then loads that file.

?- ed.

Edits and loads (again) the file that was used in the most recent call to **load** or **ed**.

?- list *P/N*.

Lists all clauses that define the predicate P with N arguments, provided these clauses were loaded with **ed** or **load**. (Note: In the case of grammar rules, the number of arguments includes the arguments automatically supplied by the Prolog grammar rule translator.)

?- list *P*.

Lists all clauses that define the predicate P with any number of arguments, provided these clauses were loaded with **ed** or **load**.

?- list.

Lists all clauses that were loaded with **ed** or **load**.

?- new.

Clears the workspace; removes from memory all clauses that were loaded

with **ed** or **load**. (Does not delete clauses that were placed into memory by **consult**, **reconsult**, or **assert**.)

E Built-in predicates usually used within the program

g_translate(FeatureStructure,ValueList)

Translates a feature structure into a value list, or vice versa. Used when you must interconvert internal and external representations at run time (e. g., to input or output them). For example, the following will accept a feature structure in GULP notation from the keyboard, translate it into a value list, and pass the value list to your predicate **test**:

```
?- read(X), g_translate(X,Y), test(Y).
```

The following translates a feature structure X from internal to external representation and prints it out:

... g_translate(Y,X), write(Y).

display_feature_structure(X)

Displays **X** in a convenient indented notation (*not* GULP syntax), where **X** is either a feature structure or a value list.

g_display(X)

Equivalent to **display_feature_structure(X)**; retained for compatibility with GULP 1.

g_printlength(A,N)

Where **A** is an atom, instantiates **N** to the number of characters needed

to print it. Useful in constructing your own indented output routines.

writeln(X)

If **X** is a list, writes each element of **X** on a separate line and then begins a new line. If **X** is not a list, writes **X** and then begins a new line. Examples:

writeln('This is a message.').

writeln(['This is a','two-line message.']).

Lists within lists are *not* processed recursively. The elements of the outermost list are printed one per line, and lists within it are printed as lists.

append(List1,List2,List3)

Concatenates **List1** and **List2** giving **List3**, or splits **List3** into **List1** and **List2**.

member(Element,List)

Succeeds if **Element** is an element of **List**. If **Element** is uninstantiated, it will be instantiated, upon backtracking, to each successive element of **List**.

remove_duplicates(List1,List2)

Removes duplicate elements from *List1* giving *List2*.

retractalAll(P)

Retracts (abolishes) all clauses whose predicate is P.

phrase(Constituent,InputString)

Provides a simplified way to call a parser written with DCG rules; for example, the goal **?- phrase(s,[the,dog,barks])** is equivalent to **?- s([the,dog,barks],[])**.

copy(X,Y)

Copies term **X**, giving term **Y**. These terms are the same except that all uninstantiated variables in **X** are replaced by new uninstantiated variables in **Y**, arranged in the same way. Variables in **Y** can then be instantiated without affecting **X**.

call_if_possible(Goal)

Executes **Goal**, or fails without an error message if there are no clauses for Goal. (In Quintus Prolog, the program crashes with an error message if there is an attempt to query a goal for which there are no clauses.)

g_fs(X)

Succeeds if **X** is an untranslated feature structure, i. e., a term whose principal functor is '**:**', '**..**', or '**::**'.

g_not_fs(X)

Succeeds if **X** is not an untranslated feature structure.

g_vl(X)

Succeeds if **X** is a value list (the internal representation of a feature structure).

F Other built-in predicates

g_ed_command(X)

Instantiates **X** to the command presently used to call the editor. To call a different editor, assertz your own clause for this predicate (e. g., '**g_ed_command(emacs)**').

g_herald(X)

Instantiates **X** to an atom identifying the current version of GULP.

G Differences between GULP 1.1 and 2.0

(1) **g_features** declarations are no longer required, but are still permitted, and, if used, need not be complete.

(2) The operator '**..**' is now preferred in place of '**::**'. However, the older form can still be used.

(3) There have been minor changes in the operator precedence of '**:**' and '**..**' relative to other operators. This is extremely unlikely to cause problems unless you have written feature structures that contain other operators such as '+' or '-'.

(4) GULP 2.0 distinguishes between features that are mentioned but uninstantiated, and features that are never mentioned. Previously, g_display never printed out any uninstantiated features.

(5) Bugs have been corrected. Translation of value lists to feature structures works correctly.

(6) Some rarely used built-in predicates have been deleted. In all cases these predicates had more common synonyms (**ed** rather than **g_ed**, list rather than **g_list**, etc.).

(7) **list** translates feature structures into GULP notation before displaying them. (A debugger with the same capability is foreseen in the future.)

(8) Nested **loads** are now supported. That is, a file being loaded can contain a directive such as '**:- load file2.**' which will be executed correctly.

Figure 1. A syntactic tree (based on Chomsky 1957).

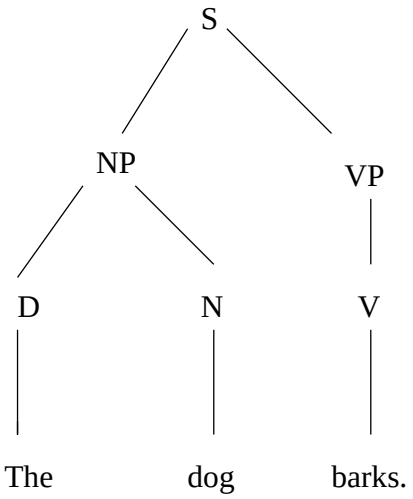


Figure 2. The same tree with features added.

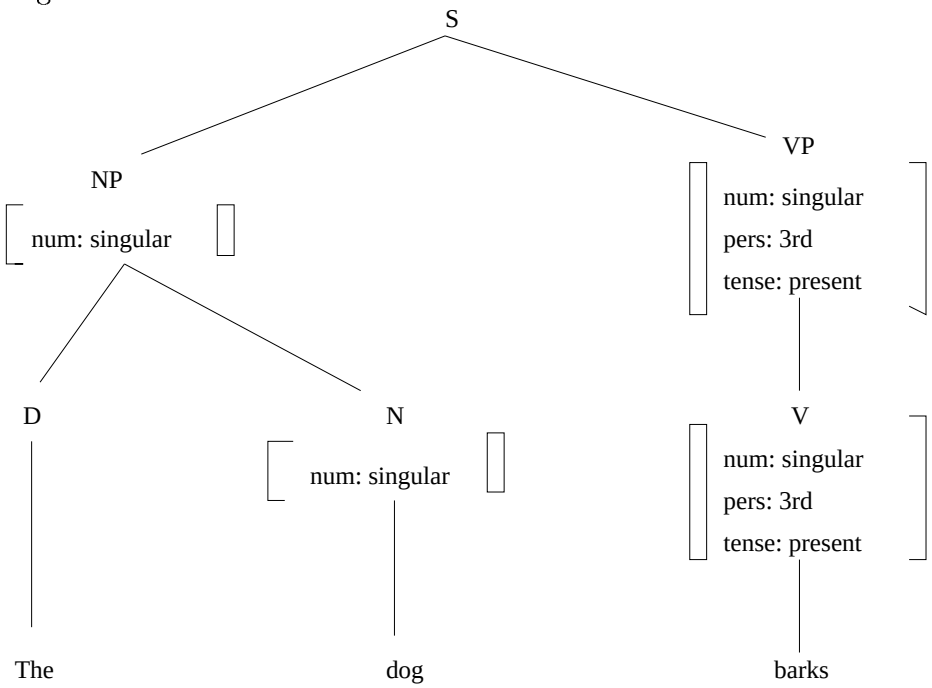


Figure 3. An example of a unification-based grammar.

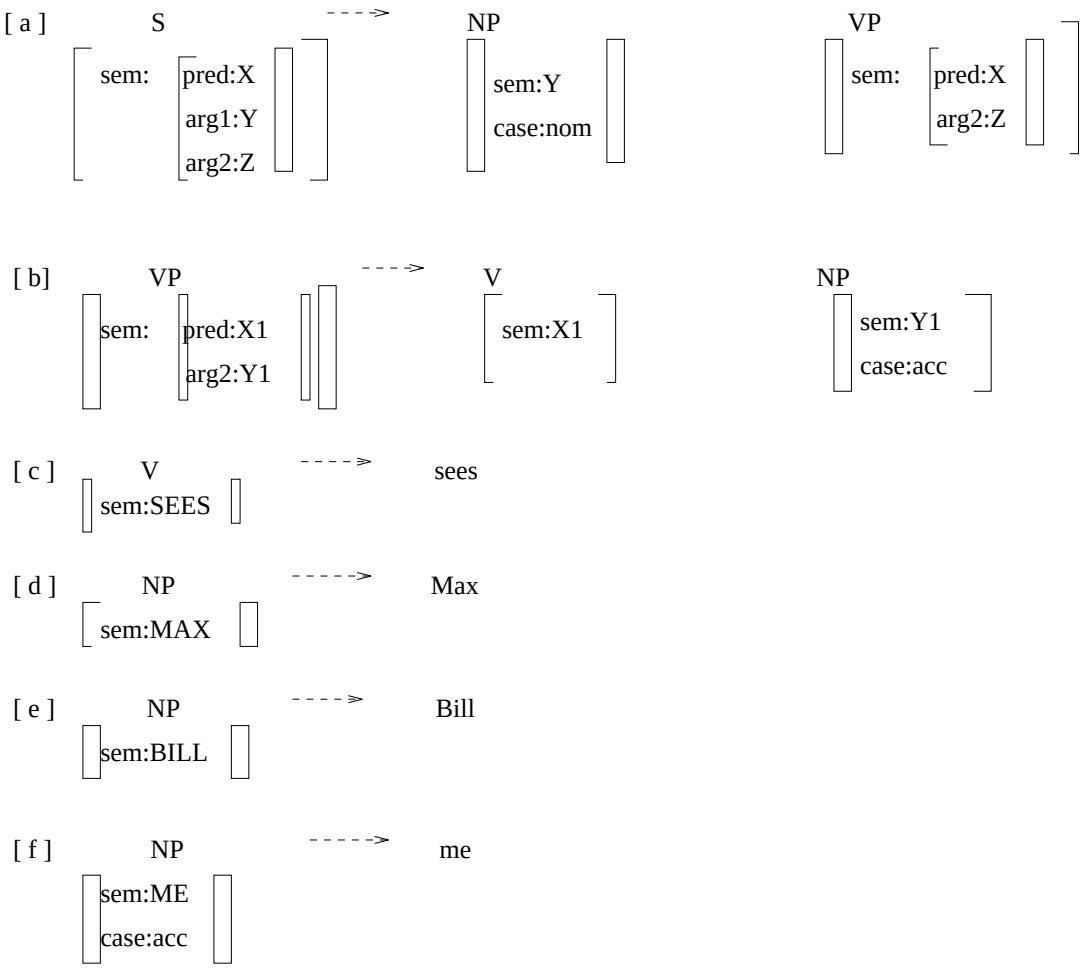
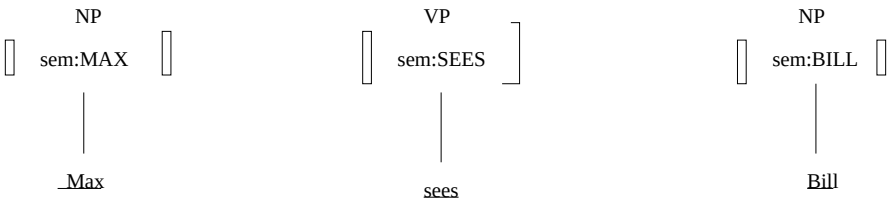
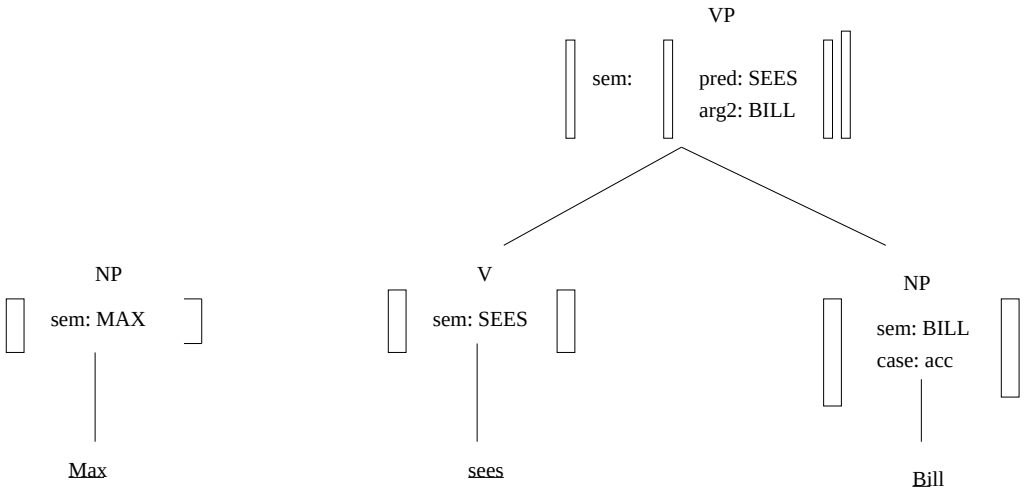


Figure 4. Bottom-up parsing of Max sees Bill. a. Rules [c], [d], and [e] supply features for individual words:



b. Rule [b] allows V and NP to be grouped into a VP:



c. Rule [a] allows NP and VP to be grouped into an S:

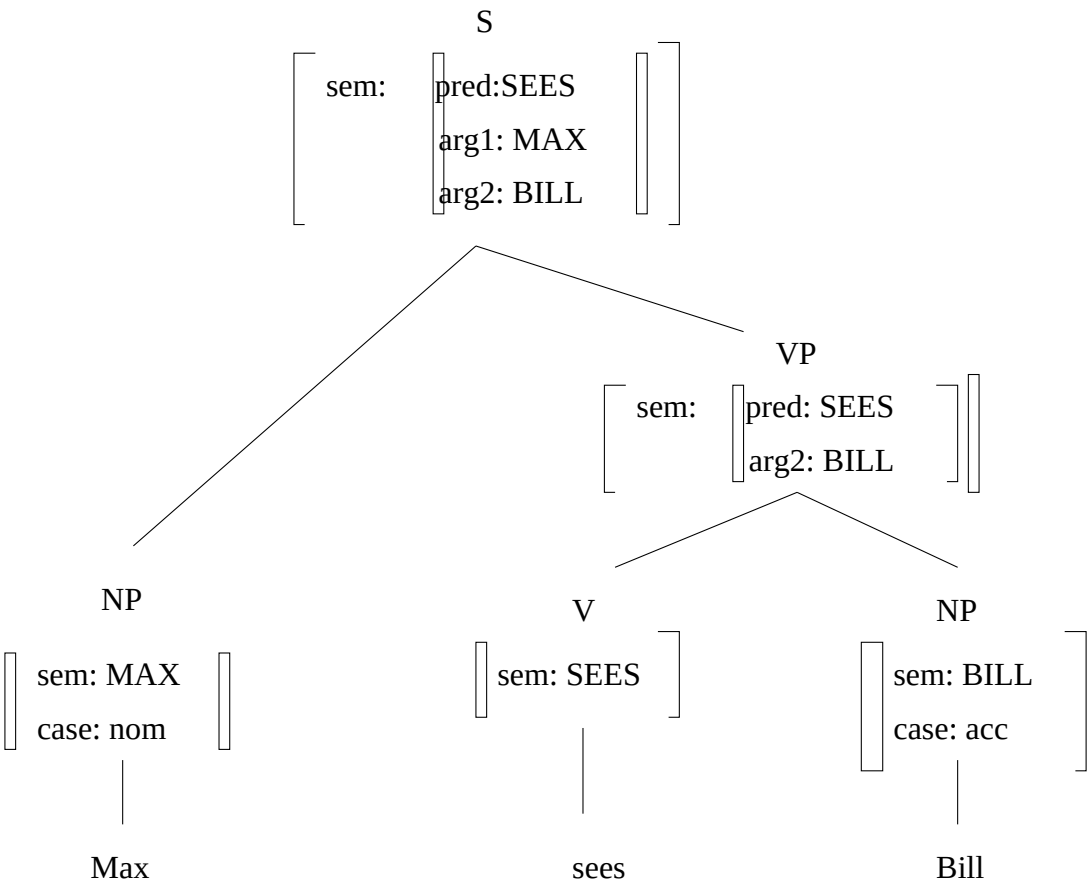


Figure 5. DAG representations of feature structures.

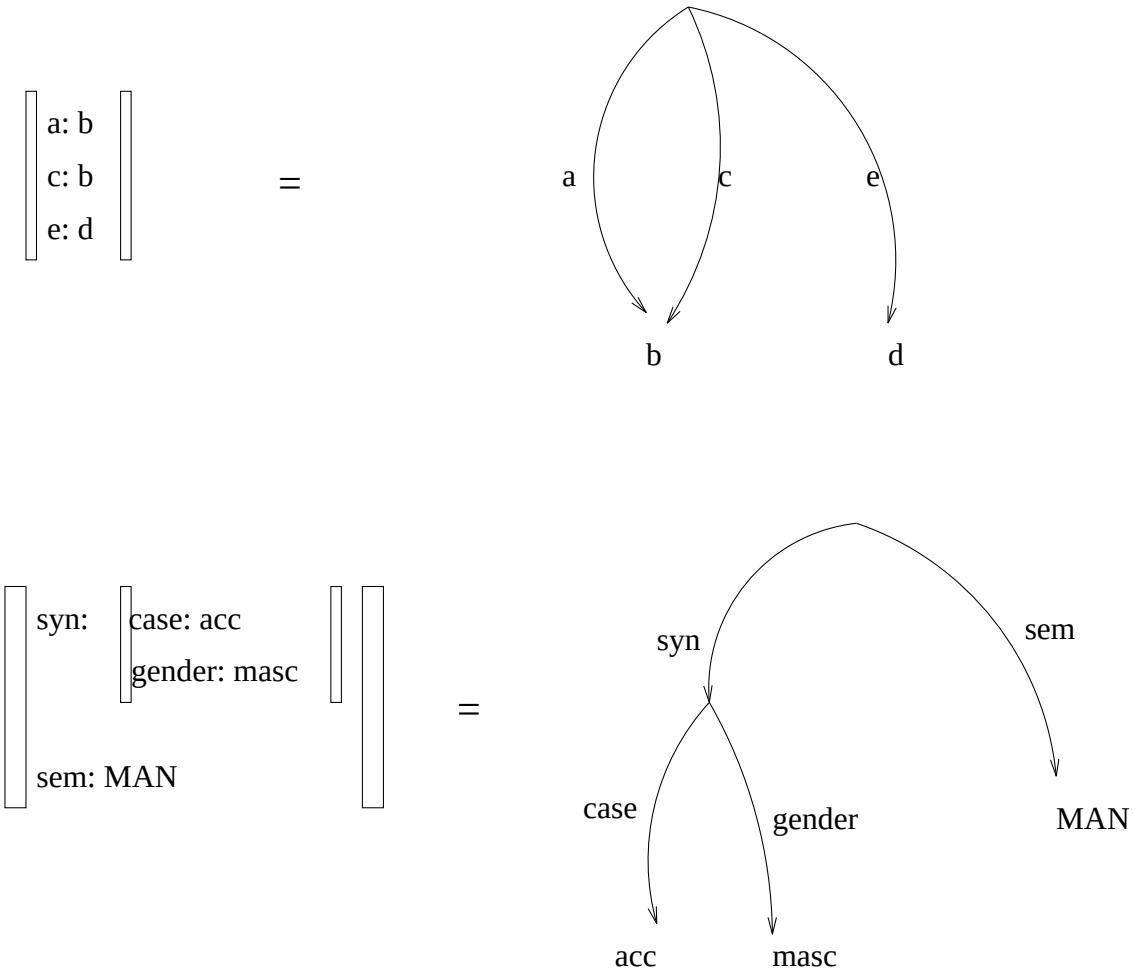


Figure 6. Parse tree for Max sees me. The ungrammatical sentence Me sees Max is ruled out by a feature conflict.

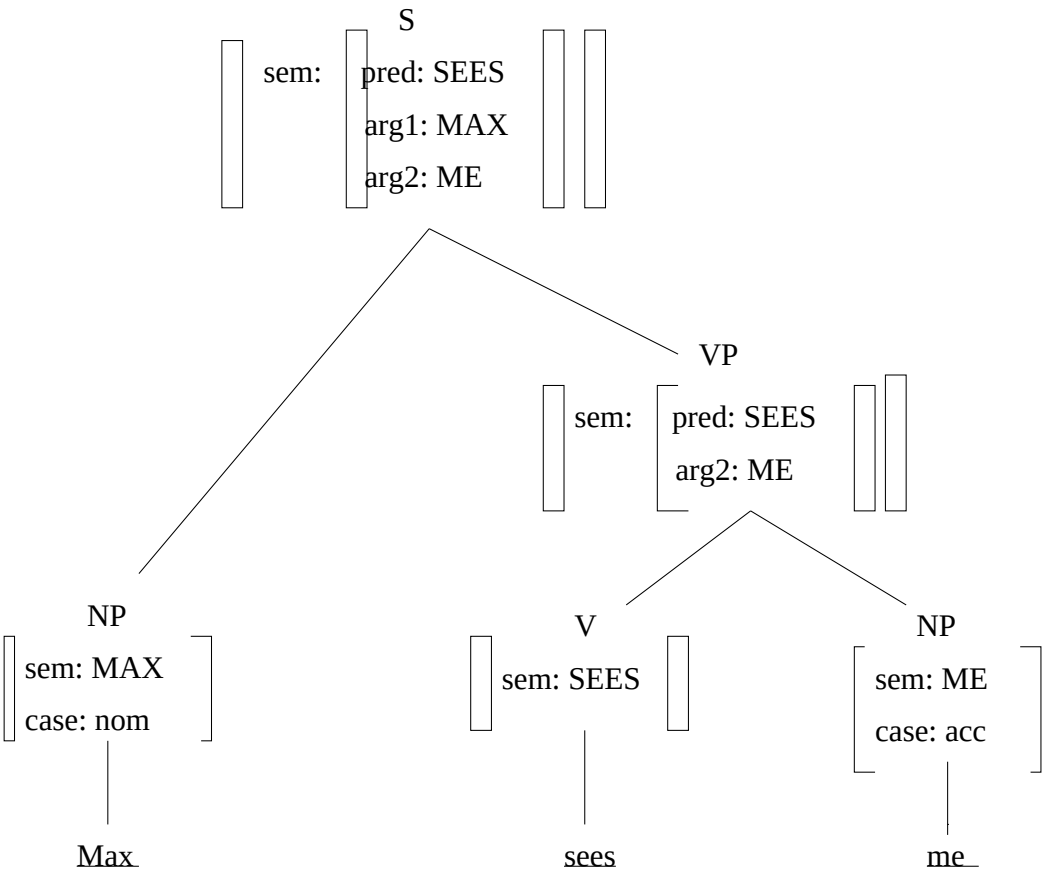


Figure 7.

```
% GULP example 1.
% Grammar from Figure 3, 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 8.

```
% Same as GULP example 1, 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 9.

```
% Demonstration of a hold stack that
% picks up the word 'what' at beginning of
% sentence, and carries it along until an
% empty NP position is found

% 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 10.

```
% Discourse Representation Theory
% (part of the program from Johnson & Klein 1986,
% translated from PrAtt into GULP).


% unique_integer(N)
%          instantiates N to a different integer
%          every time it is called, thereby generating
%          unique indices.


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

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 the indices of its arguments
%      through syntactic features. Using these indices,
%      it adds the appropriate predicate to the semantics.

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


% 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 the determiner, which
%      will specify the 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]).
```

Figure 11.

```
% BUP in GULP:
% Bottom-up parsing algorithm of Matsumoto et al. (1986)
% with the grammar from Figure 3.


% 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 */
```