FUF: the Universal Unifier
User Manual
Version 2.0

CUCS-489-89

Michael Elhadad

Department of Computer Science
Columbia University
New York, NY 10027
Elhadad@cs.columbia.edu

February 1989

Abstract

This report is the user manual for FUF version 2.0, a natural language generator program that uses the technique of unification grammars. The program is composed of two main modules: a unifier and a linearizer. The unifier takes as input a semantic description of the text to be generated and a unification grammar, and produces as output a rich syntactic description of the text. The linearizer interprets this syntactic description and produces an English sentence. This manual includes a detailed presentation of the technique of unification grammars and a reference manual for the current implementation (FUF 2.0).

CUCS-489-89

FUF: the Universal Unifier User Manual Version 2.0

Michael Elhadad

Department of Computer Science
Columbia University
New York, NY 10027
Elhadad@cs.columbia.edu

28 February 1989 at 15:04

Abstract

This report is the user manual for FUF version 2.0, a natural language generator program that uses the technique of unification grammars. The program is composed of two main modules: a unifier and a linearizer. The unifier takes as input a semantic description of the text to be generated and a unification grammar, and produces as output a rich syntactic description of the text. The linearizer interprets this syntactic description and produces an English sentence. This manual includes a detailed presentation of the technique of unification grammars and a reference manual for the current implementation (FUF 2.0).

Table of Contents

1. Introduction

1.1. How to read this manual

This manual is designed to help you use the FUF package and to describe and explain the technique of unification grammars.

The FUF package is made available to people interested in text generation and/or functional unification. It can be used:

- as a front-end to a text generation system, providing a surface realization component. A grammar of English with a reasonable syntactic coverage is included for that purpose.
- as an environment for grammar development. People interested in expressing grammatical theories or developing a practical grammar can experiment with the unifier and linearizer.
- as an environment for a study of functional unification. Functional unification is a powerful technique and can be used for non-linguistic or non-grammatical applications.

This manual contains material for people interested in any of these. It starts with an introduction to functional unification, its syntax, semantics and terminology. The next sections deal with the "grammar development" tools: tracing and indexing, a presentation of the morphology component and the dictionary. Finally the last section is a reference manual to the package. One appendix is devoted to the possible non-linguistic applications of the formalism, and compares the formalism with programming languages.

1.2. Function and Content of the Package

FUF implements a natural language surface generator using the theory of unification grammars (cf section bibliography for references). Its input is a Functional Description (fd) describing the meaning of an utterance and a grammar (also described as an fd).The Syntax of fds is fully described in section 5. The output is an English sentence expressing this meaning according to the grammatical constraints expressed by the grammar.

There are two major stages in this process: unification and linearization.

Unification consists in making the input-fd and the grammar "compatible" in the sense described in [10]. It comes down to enriching the input-fd with directives coming from the grammar and indicating word order, syntactic constructions, number agreement and other features.

The enriched input is then linearized to produce an English sentence. The linearizer includes a morphology module handling all the problems of word formation (s's, preterits, ...).

2. Getting Started

Appendix I describes how to install the package on a new machine. Contact your local system administrator to learn how to load the program on your system. You should know how to load the example grammars and corresponding inputs.

2.1. Main User Functions

Once the system is loaded, you are ready to run the program. To try the unification, the user functions are:

```
(UNI FD &optional GRAMMAR Non-Interactive)
        by default the grammar used is *u-grammar*
                   non-interactive is nil
Complete work : unification + linearization. Outputs a sentence.
                If non-interactive is nil, a line of statistics is
                also printed.

(UNI-FD FD &optional GRAMMAR Non-Interactive)
        by default the grammar used is *u-grammar*
                   non-interactive is nil.
Does only the unification. Outputs the enriched fd. This is the
function to use when trying the grammars manipulating lists of gr5.1
If non-interactive is nil, a line of statistics is also printed.

           CL> (uni ir01)
           The boy loves a girl.
           CL> (uni-fd ir02)
           (# # ...)

(UNIF FD &optional GRAMMAR Non-Interactive)
        by default the grammar used is *u-grammar*
As uni-fd but works even if FD does not contain a CAT feature.
```

If you want to change the grammar, or the input you can edit the files defining it, or the function with the same name.

There are two other useful functions for grammar developers: fd-p checks whether a Lisp expression is a syntactically correct Functional Description (FD) to be used as an input. If it is not, helpful error messages are given. grammar-p checks whether a grammar is well-formed.

NOTE: use fd-p on inputs only and grammar-p on grammars only.

```
(FD-P FD)
--> T if FD is a well-formed FD.
--> nil (and error messages) otherwise.
DO NOT USE FD-P ON GRAMMARS

(GRAMMAR-P &optional GRAMMAR print-messages print-warnings)
--> T if GRAMMAR (by default *u-grammar*) is a well-formed grammar.
--> nil (and error messages) otherwise.
- FD is *u-grammar* by default
- print-messages is t by default.
  If it is non-nil, some statistics on the grammar are printed.
  It should be nil when the function is called non-interactively.
- print-warnings is nil by default.
  If it is non-nil, warnings are generated for all paths in the
  grammar. (It is sometimes a good idea to manually check that all
  paths are valid.)

(LIST-CATS &optional GRAMMAR)
--> List of categories known by the grammar (by default *u-grammar*).
```

```
Examples:
           CL> (fd-p '((a 1) (a 2)))
           ----> error, attribute a has 2 incompatible values: 1 and 2.
                nil
           CL> (grammar-p)
           ----> t
           CL> (grammar-p '((a 1) (b 2)))
           ----> error, a grammar must be a valid FD of the form:
                 ((alt (((cat c1)...) ... ((cat cn) ...)))). nil.
           CL> (list-cats)
           ----> ((cat s) (cat np) (cat vp))
```

The functions `complexity` and `avg-complexity` measure how complex is a grammar, that is how much time unification with this grammar requires. They are documented in section 7 on indexing.

3. FDs, Unification and Linearization

In this section, we informally introduce the concepts of FDs and unification. The next section provides a complete description of the FDs as used in the package, and presents all available unification mechanisms.

3.1. What is an FD?

An FD (functional description) is a data structure representing constraints on an object. It is best viewed as a list of pairs (attribute value). Here is a simple example:

```
((article "the") (noun "cat"))
```

There is a function called `fd-p` in the package that lets you know whether a given Lisp expression is a valid FD or not and gives you helpful error messages if it is not. In FUGs, the same formalism is used for representing both the input expressions and the grammar.

3.2. A simple example of unification

We present here a minimal grammar that contains just enough to generate the simplest complete sentences. It is included in file "gr0.l" in the directory containing the examples. A little more complex grammar, handling the active/passive distinction, is available in "gr1.l", and a more interesting one in "gr2.l".

```
((alt MAIN (
   ;; a grammar always has the same form: an alternative
   ;; with one branch for each constituent category.

   ;; First branch of the alternative
   ;; Describe the category S.
   ((cat s)
    (prot ((cat np)))
    (goal ((cat np)))
    (verb ((cat vp)
           (number (prot number))))
    (pattern (prot verb goal)))

   ;; Second branch: NP
   ((cat np)
    (n ((cat noun)))
    (alt (
      ;; Proper names don't need an article
      ((proper yes)
       (pattern (n)))
      ;; Common names do
      ((proper no)
       (pattern (det n))
       (det ((cat article)
             (lex "the")))))))

   ;; Third branch: VP
   ((cat vp)
    (pattern (v dots))
    (v ((cat verb))))))
```

A few comments on the form of this grammar: the skeleton of a grammar is always the same, a big `alt` (alternation of possible branches, the unifier will pick one compatible branch to unify with). Each branch of this alternation corresponds to a single category (here, `S`, `NP` and `VP`).

The second remark is about the form of the input: as shown in the following example, an input is an FD, giving some constraints on certain constituents. The grammar decides what grammatical category corresponds to each constituent.

The next main function of the grammar is to give constraints on the ordering of the words. This is done using the `pattern` special attribute. A `pattern` is followed by a picture of how the constituents of the current FD should be ordered: `(Pattern (prot verb goal))` means that the prot constituent should come just before the verb constituent, etc.

In the first branch, the only thing to notice is how the agreement subject/verb is described: the number of the `PROT` will appear in the input as a feature of the FD appearing under `PROT`, as in:

```
(prot ((number plural) (lex "car")))
```

standing for "cars". To enforce the subject/verb agreement, the grammar picks the feature `number` from the `prot` sub-fd and requests that it be unified with the corresponding feature of the `verb` sub-fd. This is expressed by:

```
(verb ((number (prot number))))
```

which means: the value of the `number` feature of `verb` must be the same as the value of the `number` feature of `prot`.

In the second branch, describing the NPs, we have two cases, corresponding to proper and common nouns. Common nouns are preceded by an article, whereas proper nouns just consist of themselves, e.g., "the car" vs. "John". If the feature `proper` is not given in the input, the grammar will add it. By default, the current unifier will always try the first branch of an `alt` first. That means that in this grammar, proper nouns are the default.

Finally, a brief word about the general mechanism of the unification: the unifier first unifies the input FD with the grammar. In the following example, this will be the first pass through the grammar. Then, each sub-constituent of the resulting FD that is part of the `cset` (constituent-set) of the FD will be unified again with the whole grammar. This will unify the sub-constituents `prot`, `verb` and `goal` also. This is how recursion is triggered in the grammar. The next section describes how the `cset` is determined. All you need to know at this point is that if a constituent contains a feature `(cat xxx)` it will be tried for unification.

In the input FDs, the sign "===" is used as a shortcut for the notation:

```
(n === John)    <===>   (n ((lex John)))
```

The `lex` feature always contains the single string that is to be used in the English sentence.

```
When unified with the following FD, the grammar will output the
sentence "John likes Mary".

(setq ir01 '((cat s)
             (prot ((n === john)))
             (verb ((v === like)))
             (goal ((n === Mary)))))


That corresponds to the linearization of the following complete
FD (this is the result of the unification):

CLISP> (uni-fd ir01)

((cat s)
 (prot ((n ((lex "john")
            (cat noun)))
        (cat np)
        (proper yes)
        (pattern (n))))
 (verb ((v ((lex "like")
            (cat verb)))
        (cat vp)
        (number nil)
        (pattern (v dots))))
 (goal ((n ((lex "Mary")
            (cat noun)))
        (cat np)
        (proper yes)
        (pattern (n))))
 (pattern (prot verb goal)))
```

Following the trace of the program will be the easiest way to figure out what is going on:

```
CLISP> (uni ir01)
-->
Entering alt MAIN -- Branch #1 (CAT S)
-->Enriching input with (CAT NP) at level (PROT)
-->Enriching input with (CAT NP) at level (GOAL)
-->Enriching input with (CAT VP) at level (VERB)
-->Enriching input with (NUMBER (PROT NUMBER)) at level (VERB)
-->Enriching input with (PATTERN (PROT VERB GOAL)) at level NIL
-->

-->
Entering alt MAIN -- Branch #2 (CAT NP)
-->Enriching input with (CAT NOUN) at level (PROT N)
-->Enriching input with (PROPER YES) at level (PROT)
-->Enriching input with (PATTERN (N)) at level (PROT)
-->

-->
Entering alt MAIN -- Branch #3 (CAT VP)
-->Enriching input with (PATTERN (V DOTS)) at level (VERB)
-->Enriching input with (CAT VERB) at level (VERB V)
-->

-->
Entering alt MAIN -- Branch #2 (CAT NP)
-->Enriching input with (CAT NOUN) at level (GOAL N)
-->Enriching input with (PROPER YES) at level (GOAL)
-->Enriching input with (PATTERN (N)) at level (GOAL)
-->


[Used 17 backtracking points]

John likes Mary.
```

In the figure, you can identify each step of the unification: first the top level category is identified: (cat s). The input is unified with the corresponding branch of the grammar (branch #1). Then the constituents are identified. We have here 3 constituents: PROT of cat NP, VERB of cat VP and GOAL of CAT NP. Each constituent is unified in turn. Then for each constituent, the unifier identifies the sub-constituents. In this case, no constituent has a sub-constituent, and unification succeeds. Note that in general, the hierarchy of constituents is traversed breadth first.

Now, it is also important to know when unification fails. The following example tries to override the subject/verb agreement, causing the failure:

```
(setq ir02 '((cat s)
             (prot ((n === john) (number sing)))
             (verb ((v === like) (number plural)))
             (goal ((n === Mary)))))

CLISP> (uni ir02)
-->
Entering alt MAIN -- Branch #1 (CAT S)
-->Enriching input with (CAT NP) at level (PROT)
-->Enriching input with (CAT NP) at level (GOAL)
-->Enriching input with (CAT VP) at level (VERB)
-->Fail in trying PLURAL
   with SING at level (VERB NUMBER)


<fail>
```

3.3. Linearization

Once the unification has succeeded, the unified fd is sent to the linearizer. The linearizer works by following the directives included in the `pattern` . The exact way to define these features is explained in section 5.5. The linearizer works as follows:

1. Identify the `pattern` feature in the top level: for ir01, it is `(pattern (prot verb goal))`.
2. If a pattern is found:
   a. For each constituent of the pattern, recursively linearize the constituent. (That means linearize PROT, VERB and GOAL).
   b. The linearization of the fd is the concatenation of the linearizations of the constituents in the order prescribed by the pattern feature.
3. If no feature pattern is found:
   a. Find the `lex` feature of the fd, and depending on the category of the constituent, the morphological features needed. For example, if fd is of `(cat verb)`, the features needed are: `person, number, tense`.
   b. Send the lexical item and the appropriate morphological features to the morphology module . The linearization of the fd is the resulting string. For example, if lex=''give'' and the features are the default values (as it is in ir01), the result is ''gives.''

Note that when the fd does not contain a morphological feature, the morphology module provides reasonable defaults. More details on morphology are provided in section 8.

Note also that if a pattern contains a reference to a constituent and that the constituent does not exist, nothing happens: the linearization of an empty constituent is the empty string. The following example illustrates this feature:

```
Unified FD:
((cat s)
 (pattern (prot verb goal benef))
 (prot ((cat noun) (lex ''John'')))
 (verb ((cat verb) (lex ''like''))))

Linearized string (note that constituents GOAL and BENEF are missing):
John likes.
```

Finally, note that if one of the constituent sent to the morphology is not a known morphological category, the morphology module can not preform the necessary agreements. This is indicated by the following output:

```
Unified FD:
((cat s)
 (pattern (prot verb goal))
 (prot ((cat noun) (lex ''John'')))
 (verb ((cat verb) (lex ''like'')))
 (goal ((cat zozo) (lex ''trotteur''))))

Linearized string:
John likes <unknown cat ZOZO: trotteur>
```

In general, when you find that in your output, it means you have done something wrong. You should check the list of legal morphological categories (see section 8) or you should check why a high level constituent is sent to the morphology (your fd is too flat). You can use the function `morphology-help` to have on-line help on what the morphology module can do.

4. Writing and Modifying Grammars

In this section, we briefly outline what steps must be followed to develop a Functional Unification Grammar. The methodology is the following:

1. Determine the input to use. In general, input is given by an underlying application. If not, the criterion to decide what is a good input is that it should be as much "semantic" as possible, and contain the fewest syntactic features as possible.
2. Identify the types of sentences to produce.
3. For each type of sentence, identify the constituents and sub-constituents, and their function in the sentence. A constituent is a group of words that are "tied together" in a clause. A constituent in general plays a certain function with respect to the higher level constituent containing it. For example, in "John gives a book to Mary," the group "a book" forms a constituent, of category "noun-group," and it plays the role of the "object upon which action is performed" in the clause. Such role is often called the "medium" in functional grammars.
4. Determine the output (that is, the unified fds before linearization). In the output, constituents should be grouped in the same pair and the attribute should indicate what function the constituent is fulfilling. In the previous example, we want to have a pair of the form `(medium <fd describing ''a book''>)` in the output. The output must also contain all ordering constraints necessary to linearize the sentence and provide all the morphological feature needed to derive all word inflections (e.g., number, person, tense).
5. Determine the "difference" between the input and the output. All features that are in the output but not in the input must be added by the grammar.
6. For each category of constituent, write a branch of the grammar. To do that, you need to specify under

which conditions each feature of the "difference" must be added to the input.

This is of course an over-simplified description of the process. Sometimes, the mapping from the input to the output is best considered if decomposed in several stages. For example, in gr4 (cf. file `gr4.l`), the grammar first maps the roles from semantic functions (like `agent` or `medium`) to syntactic roles (like `subject` or `direct-object`), and then does the required syntactic adjustments.

In general, the important idea here is that you must first determine your input and your output and the grammar is the difference of the two.

5. Precise characterization of FDs

5.1. Generalities: features, syntax, paths

Pairs are called features. The attribute of a feature needs to be an atom. The value of a feature can be either an atom or recursively an FD. Here is an example:

```
(1) ((cat np)
     (det ((cat article)
           (definite yes)))
     (n   ((cat noun)
           (number plural))))
```

A given attribute in an FD must have at most ONE value. Therefore, the FD `((size 1) (size 2))` is illegal. In fact FDs can be viewed as a conjunction of constraints on the description of an object: for an object to be described by `((size 1) (size 2))` it would need to have its property `size` to have both the values 1 and 2. Conversely, if the attribute `size` does not appear in the FD, that means its value is not constrained and it can be anything. The FD `nil` (empty list of pairs) thus represents all the objects in the world. The pair `(att nil)` expresses the constraint that the value of `att` can be anything. It is therefore useless, and the FD `((att1 nil) (att2 val2))` is exactly equivalent to the FD `((att2 val2))`.

Any position in an FD can be unambiguously refered to by the "path" leading from the top-level of the FD to the value considered. For example, FD (1) can be described by the set of expressions:

```
(cat) = np
(det cat) = article
(det definite) = yes
(n cat) = noun
(n number) = plural
```

Paths are represented as simple lists of atoms (for example, `(det definite)`). This notation is not ambiguous because at each level there is at most one feature with a given attribute.

A path can be "absolute" or "relative." An absolute path gives the way from the top-level of the FD down to a value. A relative path starts with the symbol "^" (up-arrow). It refers to the FD embedding the current feature. You can have several "^" in a row to go up several levels. For example:

```
((cat s)
 (prot ((cat np)
        (number sing)))
 (verb ((cat vp)
        (number (^ ^ prot number)))))
                    ^
____________________|
this is refering to the absolute path (prot number)
```

The value of a pair can be a path. In that case, it means that the values of the pair pointed to by the path and the value of the current pair must always be the same. In this case, the two features are said to be unified. In the previous example, the features at the paths <`verb number`> and <`prot number`> are unified. That means they are absolutely equivalent, they are two names for the same object. This is equivalent to the systemic operation of "conflation".

The only case where a given attribute can appear in several pairs is when it is followed by paths in all but one pairs. That is:

```
((a ((a1 v1)))
 (a (b))
 (a (c)))
```

is a valid FD. It is equivalent for example to:

```
((b ((a1 v1)))
 (a (b))
 (c (b)))
```

5.2. FDs as graphs

It is often useful to represent FDs as Directed Acyclic Graphs (DAGs). Here is how the correspondance is established: an FD is a node, each pair `(attr. value)` is an arc leaving this node. The `attr` of the pair is the label of the arc, the value is the adjacent node. Internal nodes in the graph have therefore no label whereas leaves are atomic values.

```
                                         *
((cat s)                               / | \
 (prot ((cat np)                     /   |   \
        (number sing)))    <====>  prot cat  verb
 (verb ((cat vp)                    |    |    |
        (number sing))))            *    *    *
                                   / \   |   / \
                                 /    \  s  /    \
                                |     |  |       |
                              number cat cat   number
                                |     |  |       |
                                |     |  |       |
                               sing  np  vp     sing
```

When a relative path occurs somewhere in an FD, to find where it points to, just go up on the arcs, one arc for each "^". When the value of a pair is a path, e.g., `(a (b))` it means that the current arc is actually pointing to the same node as the path given.

```
                                             *
((cat s)                                   / | \
 (prot ((cat np)                         /   |   \
        (number sing)))       <====>  prot cat  verb
 (verb ((cat vp)                       |     |    |
        (number (^ ^ prot number))))   *     *    *
                                      / \    |   / \
                                    /     \  s /     \
                                  |       |  |       |
                                 cat number number  cat
                                  |       |  |       |
                                  |       +--+       |
                                  |        |         |
                                 np       sing       vp
```

The following attributes have a special unification behavior: `alt`, `opt`, `pattern`, `cset`, `test`, `control` and `cat`. The following values have a special unification behavior: `none`, `any` and `given`. These are all the "keywords" known by the unifier.

5.3. Disjunctions: The ALT keyword

`alt` stands for "alternation". The syntax for using `alt` is:

```
((att1 val1)
 (att2 val2)
  ...
 (ALT (fd1 fd2 ... fdn))
  ...
 (attn valn))
```

The meaning of a pair with an `alt` attribute is: the unifier will try to unify the total FD by replacing first the pair `alt` by the FD `fd1`, if this unification fails, then the unifier will try the following alternatives. If all branches of the `alt` fail, the unification fails.

The order in which branches are put within the `alt` does not change the result of the unification. (This is an important feature of the process of unification: the result is always order-independent.) However, since only the first successful unification is returned, order can be used to specify default values. For example, if you want to specify that a sentence should be at the active voice by default, the following order should be used:

```
(ALT (((voice active)
        ...)
       ((voice passive)
        ...)))
```

An `alt` can be embedded within another `alt` or it can be the value of a feature.

5.4. Optional features: the OPT keyword

opt is used to indicate that a set of features is optional. The syntax is

```
((att1 val1)
     ...
 (OPT fd)
     ...
 (attn valn))
```

The meaning is: if the unification of the whole FD succeeds with fd, it is returned as the result. If it fails, the unifer tried again without fd. opt is therefore a more readable equivalent to the form:

```
(ALT (fd nil))
```

opt is used exactly in the same way as alt.

5.5. Control of the ordering: the PATTERN keyword

As mentioned previously, the generation of a sentence is made of two subprocesses: the unification and the linearization. The unification produces a complex description of a sentence, made of several constituents. Each constituent is described by an FD, and can recursively contain other subconstituents.

The linearization takes such a complex non ordered description and outputs a linear, ordered string of words. This operation is constrained by directives put within the FD. These constraints on the ordering are put after the special attribute pattern.

For example, in a sentence containing the constituents prot, goal and verb, the following pattern can be used:

```
(PATTERN (PROT VERB GOAL))
```

This means that the linearizer should output a string made of the linearization of the constituent prot first, followed by the linearization of the constituent verb and finished by the linearization of the constituent goal. It also means that nothing can come before prot and after goal, and nothing can come between each pair.

The constituents correspond to features of the FD describing the sentence. That is, this FD must contain pairs with the attributes prot, verb and goal. For example:

```
((cat S)
 (PROT (...))
 (GOAL (...))
 (VERB (...))
 (PATTERN (PROT VERB GOAL)))
```

If a constituent mentioned in the pattern is not present in the FD, nothing happens: the linearization of an empty (or non existent) constituent is the empty string.

The pattern directives are generally added by the grammar, since the input to the unifier should be a semantic representation and therefore does not contain any constraint on word ordering.

A given grammar can generate several constraints, that is it can add 2 or more `pattern` pairs to the result. The unifier therefore includes a `pattern` unifier. The role of the pattern unifier is to take several constraints on the ordering and to output one ordering that subsumes all of them.

The following symbols have a special meaning for the pattern unifier: `dots` and `pound` (standing respectively for the notations '...' and '#').

A pattern `(c1 ... c2)` (noted in the program `(c1 dots c2)`) indicates that the constituent `c1` must precede the constituent `c2`, but they need not be adjacent. Zero, one or many other constituents can come in between. The pattern `(c1 ... c2)` still requires the sentence to start with constituent `c1` and to end with `c2`. The pattern `(... c1 ... c2 ...)` only forces `c1` to come before `c2`.

The `pound` (#) symbol is used to represent 0 or 1 constituent. For example, if you want to allow a sentence to start with an optional adverbial, you can specify it with the pattern `(# prot ... verb ...)`. This directive will be compatible with both `(prot verb goal)` and `(adverb prot verb goal)` for example.

As a consequence of the use of the two symbols `pound` and `dots`, the constraints described by `pattern` directives are PARTIAL orderings.

Appendix II describes some advanced uses of pattern unification.

In addition, the pattern unifier can be used to enforce the unification of constituents. The classical example is given by the `focus` constituent. There is good linguistic evidence that the focus of a sentence tends to occur first in a sentence. To represent this constraint, a grammar can include the following directive:

```
(PATTERN (FOCUS DOTS))
```

That is, a sentence should start with its focus. Now, we also know that a sentence at the active voice should start with its subject, that is its `prot` constituent. This is expressed by:

```
(PATTERN (PROT ... VERB ...))
```

If both constraints are to be satisfied, we need to say that `focus` and `prot` are actually the same constituent, otherwise, the 2 patterns are incompatible. That is, the constituents `focus` and `prot` need to be unified. This mechanism would be quite expensive to implement for all constituents, and would need to meaningless attempts most of the time. Therefore, to allow this kind of unification to occur, the current unifier requires the pattern to include a special directive, indicating that a constituent can be unified with other constituents to make two patterns compatible. The notation used is: `(* constituent)`.

```
Example:
(PATTERN ((* FOCUS) DOTS))
(PATTERN (PROT DOTS VERB DOTS))
```

are compatible, and require the unification of the constituents `focus` and `prot`. Note that `prot` needs not be "stared" to be unified with `focus`. The notation can be understood as specifying that `focus` is a kind of "meta-constituent".

NOTE: Patterns can contain full paths to specify constituents. For example, the following is a legal pattern:

```
(PATTERN ((prot n) (verb v) goal))
```

NOTE: the unification of patterns is a non-deterministic operation. It can produce several results for a given input, and there is no way to produce in which order these possible solutions will be tried. Caution should be exercised when specifying patterns: they should be specific enough to allow only acceptable word orderings (do not use too many dots) but should not be too specific to allow for as yet not supported constituents (for example, a sentence can start with an Adverbial, not necessarily an NP).

5.6. Explicit specification of sub-constituents: the CSET keyword

The unifier works recursively: it unifies first the top-level FD against a grammar (generally the top-level FD represents a sentence), and then, recursively, it unifies each of its constituents. For example, to unify a sentence, the unifer first takes the whole FD and unifies it with the grammar of the sentences `(cat s)`, then it unifies the `prot` and `goal` with the grammar of NPs `(cat np)`, then it unifies the `verb` with the grammar of VPs `(cat vp)`.

You can specify explicitly which features of an FD corresponds to constituents and therefore need to be recursively unified. To do that, add a pair:

```
(CSET (c1 ... cn))

For example:
(CSET (PROT VERB GOAL))
```

The value of a `cset` (stands for Constituent SET) is considered as a SET (in the mathematical sense). Therefore the 2 following pairs are correctly unified:

```
(CSET (PROT VERB GOAL))
(CSET (VERB GOAL PROT))
```

Actually, two `cset` pairs are unified if and only if there values are two equal sets.

The current version of the unifier does not rely exclusively of `csets` to find the constituents to be recursively unified. Here is the procedure followed to identify the constituent set of an fd:

1. If a feature `(cset (c1 ... cn))` is found in the FD, the constituent set is just `(c1 ... cn)`.
2. If no feature `cset` is found, the constituent set is the union of the following sub-fds:
    a. If a pair contains a feature `(cat xx)`, it is considered a constituent.
    b. If a sub-fd is mentioned in the pattern, it is considered a constituent.

As a consequence, `csets` are rarely necessary. They are generally used when an fd contains a sub-fd that either is mentioned in the pattern or contains a feature `cat`, but that you do NOT want to unify. In that case, you can explicitly specify the cset without including this unwanted sub-fd.

NOTE: A `cset` values can contain full paths to specify constituents. So for example, the following is a legal feature:

```
(cset ((prot n) (verb v) goal))
```

5.7. The special value NONE

There is a way to prevent an FD from ever getting a value for a given attribute. The syntax is: `(att NONE)`. It means that the FD containing that pair will NEVER have a value for `att`. Or in other words, that the object described by the FD has no attribute `att`.

5.8. The special value ANY - The Determination stage

An `any` value in a pair means that the feature must have a determined value at the end of the unification. A complete unified FD will never contain an `any`, since an `any` stands for something that must be specified. If after unifying everything, the resulting FD contains an `any`, then the unification fails.

An `any` represents a strong constraint. It means that a feature MUST be instantiated. `any` should not be understood as "the feature has a value in the input" but as "the feature WILL have a value in the result".

The idea of a "resulting final FD" coming out of the unification is important. It actually implies that the process of unification is the composition of 2 sub-processes: the unification per se and what we call here the "determination".

The determination process assures that the resulting FD is well formed. It is a necessary stage since the "resulting final" FD is more constrained than regular FDs. Here is what the determination does:

- checks that no `any` is left.
- tests all the `test` constraints.

It is important to realize that none of this can be done before the unification is finished.

Note that in practice, ANY is used VERY rarely.

5.9. The special value GIVEN

NOTE: GIVEN is a keyword specific to this implementation. Its use is not recommended. See appendix IV for a list of the non-standard features of this implementation.

A `given` value in a pair means that the feature must have a real value at the beginning of the unification. A unified fd will never contain a `given` since `given` will always be unified with a real value. `given` is useful to specify what features are necessary in an input. It is also much more efficient than `any`. It is often used in branches of an `alt`, to "test" for the presence of a feature.

The rule is: when you think of using `any`, you often want to use `given`.

5.10. The special attribute CAT: general outline of a grammar

Each constituent of an FD is generally characterized by its "category". In FD terms, that means each constituent includes a feature of the form `(CAT category-name)`, where category-name is expected to be an atom.

A grammar is expected to give directives for each possible category, for example NP, VP, or NOUN. The outline of a grammar must be:

```
((alt (
      ((cat s)
       <rest of grammar for category S>)
      ((cat np)
       <rest of grammar for category NP>)
      <other categories>
     )))
```

NOTE: The current version of the unifier makes the assumption that the grammar has such a form. The `(CAT xxx)` pairs must appear first. The function `grammar-p` checks that a grammar has the right form. The list of categories known by the grammar can be found by using the function `list-cats`. See appendix IV for a list of the non-standard features of this implementation.

6. Tracing

There are plenty of methods to trace the process of unification, generating more or less output. You want to choose the method generating only the most relevant trace.

6.1. External vs. Internal Traces: switches

For the purpose of debugging the unifier, there is a switch generating an extremely detailed output.

```
To use it, type:
(internal-trace-on)

To switch it off:
(internal-trace-off)
```

The other traces are used to follow the process of unification, and are used to debug a grammar, they don't give any information on the internals of the program. These are the external traces users generally use.

Since these traces are oriented to a grammar developper, we want the grammar developper to indicate what portions of the grammar must be traced: the grammar is traced, not the program. Therefore, to trigger tracing, one must put directives into the grammar. At the Lisp level, and for a given grammar including tracing directives, traces can be switched on or off by the functions:

```
(trace-on)  enable all trace messages to be output.

(trace-off) disable all trace messages to be output

(all-tracing-flags &optional (grammar *u-grammar*))
            return the list of all tracing flags defined in grammar.

(trace-disable flag)  disable flag.  Everything works as if flag was not
                      defined in the grammar.

(trace-enable flag)   re-enable a disabled flag.

(trace-disable-all)   disable all flags.

(trace-enable-all)    re-enable all flags.

(trace-disable-match string)
                      disable all flags whose names contain string.

(trace-enable-match  string)
                      re-enable all flags whose names contain string.
```

6.2. Tracing of alternatives and options

The most useful trace of the unification is generated by giving a name to an alternative of the grammar. It is done by adding an atomic name after the keywords `alt` or `opt` in the grammar:

```
((alt PASSIVE
  (
   ;; branch 1 of alt passive
   ((verb ((voice passive)))
    (prot none))

   ;; branch 2 of alt passive
   ((verb ((voice passive)))
    (prot any)
    (prot ((cat np)))
    (by-obj ((cat pp) (prep ((lex "by"))) (np (^ prot))))
    (pattern (dots verb by-obj dots)))))

 ;; body of alt passive (common to all branches)
 (verb ((cat verb-group)))
 ...)
```

Here, this fraction of the grammar has been marked by the directive: `(alt PASSIVE ...)`. (An equivalent notation is `(alt (trace with PASSIVE) ...)`.) The effect will be that all unification done subsequently will be traced, producing the following output:

```
--> Entering ALT PASSIVE
--> Trying Branch #1 in ALT PASSIVE:
--> Fail on trying (prot none) with
                   (prot ((nnp ((n ((lex boy)))))))
--> Trying Branch #2 in ALT PASSIVE:
...
```

If a traced alternative is found later in the grammar, the level of indentation will increase. If the level of indentation decreases, that means a whole `(alt ...)` has failed. It is indicated by the output:

```
          --> Fail on ALT PROT.
```

The possible messages printed when the grammar is traced are:

```
Move in the alternatives:
        ENTERING ALT f: BRANCH #i
        FAIL IN ALT f
        When the alt is indexed (cf section 7):
        ENTERING ALT f -- JUMP INDEXED TO BRANCH #i INDEX-NAME
        NO VALUE GIVEN IN INPUT FOR INDEX INDEX-NAME - NO JUMP
For options:
        TRYING WITH OPTION o
        TRYING WITHOUT OPTION o
Regular unification:
        ENRICHING INPUT WITH s AT LEVEL 1
        FAIL IN TRYING s with s AT LEVEL 1
Pattern unification:
        UNIFYING PATTERN p with p
        TRYING PATTERN p
        ADDING CONSTRAINTS c
        FAIL ON PATTERN p
Unification between pointers to constituents:
        UPDATING s WITH VALUE s AT LEVEL 1
        s BECOMES A POINTER TO s AT LEVEL 1
        UPDATING BOTH PATHS TO A BOUND
```

HINTS: You want to trace only the most relevant alternatives of your grammar to generate the less output possible. It is a good idea to trace first inner alternatives. Use `trace-disable` and `trace-enable` to control which flags you want to use.

6.3. Local tracing with boundaries

If you want a more focused tracing, you can put anywhere in the grammar a pair of atomic flags whose first character must be a "%" (value of variable `*trace-marker*`). All the unification done between the 2 flags will be traced, and will produce the same messages as usual.

```
          ;; branch 2 of alt passive
          ((verb ((voice passive)))
           (prot any)
           %by-obj%
           (prot ((cat np)))
           (by-obj ((cat pp) (prep ((lex "by"))) (np (^ prot))))
           %by-obj%
           (pattern (dots verb by-obj dots)))
           ...
```

All the unification done between the 2 flags %by-obj% will be traced. You furthermore will have a message:

```
Switching local trace flags on and off:
        TRACING FLAG f
        UNTRACING FLAG f
```

HINTS: You generally want to have only small portions of the grammar put between tracing flags.

6.4. The trace-enable and trace-disable family of functions

In general, a grammar is defined in a file, that you load in your Lisp environment. The tracing flags are defined in that file after the alts and opts or as local flags. When you develop a grammar, you want to focus on different parts of the grammar. In order to do that, you can selectively enable or disable some of the flags defined in the grammar.

The function `all-tracing-flags` returns a list of all the flags defined in the grammar. You can then choose to enable or disable all the flags, only a given flag, or all flags whose name matches a given string.

When a flag is disabled, everything happens as if the flag was not defined at all in the grammar. Note that you cannot create a new flag in the grammar by using these functions. You can simply turn on and off existing flags. It is therefore a good idea to define all the possible flags in a grammar and to adjust the list of enabled flags from within lisp.

7. Indexing and Complexity of grammars

In order to increase the efficiency of the unification, the program allows the inclusion of index declarations in the grammar. To better understand why such declarations can make things faster it is necessary to understand what makes unification slow.

7.1. Indexing

The main problem for the program is to handle non-deterministic constructs in the grammar. The non-deterministic constructs are currently: `alt`, `opt` and `pattern`. Unification of these constructs with an input can produce several results. Whenever the unifier encounters such a construct, it does not know which of the possible results to choose. For example, when unifying an `alt` there is no way to choose a branch out of the many available in the `alt`. The way the program works is to try each of the possibilities one after the other. When the unification later on fails, the program backtracks and tries the next possibility.

This method is actually a blind search through the space of all the descriptions compatible with the grammar. Indexing is a technique used to guide the search in a more efficient way when more knowledge is available.

The program allows indexing of `alt` constructs.[1]The indexing tells the unifier how to choose one branch out of the alternation based on the value of the index only, and without considering the other branches ever. The following example illustrates the technique.

[1]A `opt` construct is actually an `alt` with 2 branches, one being the trivial nil. It would not make sense to index it. A `pattern` construct is ambiguous because patterns like (...a...b...) and (...c...d...) can be combined in many ways. Actually, it is always more efficient to put patterns at the end of the grammar, because much of the ambiguity generated by these patterns would not change the unification anyway, except when the (* constituent) device is used. In any case, the equivalent of 'indexing' a pattern, that is reducing the ambiguity, is to use as few dots as possible in the patterns.

```
Example taken from gr4

((alt (trace with process) (index on process-type)
   (((process-type actions)
     ...)
    ((process-type mental)
     ...)
    ((process-type attributive)
     ...)
    ((process-type equative)
     ...)))
 ...)
```

In the example, the `(index on process-type)` declaration indicates that all the branches of the alternation can be distinguished by the value of the `process-type` feature alone. If the input contains a bound feature `process-type`, it is possible to directly choose the corresponding branch of the alternation. If however the input does not correspond such a feature, it has to go through the alt in the regular way, with no jumping around.

This is what happens in the tracing messages for each case:

```
If input is:
    ((cat clause) (process-type attributive) ...)
Trace message is:
    -->Entering alt PROCESS -- Jump indexed to branch 3 ATTRIBUTIVE

If input does not contain a feature process-type:
    ((cat clause) (prot John) ...)
Trace message is:
    -->No value given in input for index PROCESS-TYPE - No jump
    -->Entering alt PROCESS -- Branch #1
```

A grammar is always indexed at the top-level by the `cat` feature. It makes more sense to index on the features that will be bound in the input or at the moment the `alt` will get tried, but it never hurts to index an `alt`, so it is recommended to index whatever is indexable. A program will be soon released to perform this indexation.

The function `fd-sem` checks that an index declaration is valid, that is, that each branch of the alternation actually has a bound value for the index, and that all the branches have a different value for the indexed feature.

Note the syntax of an `alt` construct:

```
alt-form   : (alt {trace-decl} {index-decl} ( list-of-fds ))

trace-decl : atomic-flag | (trace {...} any-flag)

index-decl : (index {...} index-path)

index-path : atomic-feature | valid-path
```

The indexed feature can be at the top level of all the branches, as in the first example for `process-type`, but it can also be at lower levels, like in the following example:

```
Example taken from gr4:

((alt verb-trans (index on (verb transitivity-class))
   (((verb ((transitivity-class intransitive)))
     ...)
    ((verb ((transitivity-class transitive)))
     ...)
    ((verb ((transitivity-class bitransitive)))
     ...)
    ((verb ((transitivity-class neuter)))
     ...))
 ...))
```

NOTE: you CANNOT index an alternation if one of the indexed values is NONE, NIL, ANY or GIVEN.

7.2. Complexity

The complexity of a grammar can be described by the number of possible paths through it, each path corresponding to the choice of one branch for each alternation. (This measure of complexity is the number of branches the grammar would have in disjunctive normal form (cf bibliography).) Indexing the grammar actually divides this measure of complexity by a great number.

The functions `complexity` and `avg-complexity` compute different measures of the complexity of a grammar.

```
(COMPLEXITY &optional grammar with-index)
--> number of branches of grammar in disjunctive normal form.
- By default, grammar is *u-grammar*
- By default, with-index is T. When it is T, all indexed alts are
  considered as one single branch, when it is nil, they are
  considered as regular alts.

(AVG-COMPLEXITY &optional grammar with-index rough-avg)
--> "average" number of branches tried when input contains no
    constraint.
- By default, grammar is *u-grammar*
- By default, with-index is T.  When it is T, all indexed alts are
  considered as one single branch, when it is nil, they are
  considered as regular alts.
- By default, rough-avg is nil. When it is nil, the average of an
  alt is the sum of the complexity of the half first branches. When
  it is T, the average is half of the sum of the complexity of all
  branches.
```

Note that these functions do not currently measure the ambiguity of the patterns included in the grammar.

8. Morphology and Linearization

The morphology module (partially written by Jay Meyers USC/ISI) makes many assumptions on the form of the incoming functional description. If you want to use it, you must be aware of the following conventions.

8.1. Lexical categories are not unified

The categories that are handled by the morphology module can be declared to be "lexical categories". If a category is a lexical category, it is not unified by the unifier, and it is passed unchanged to the morphology module. The assumption here is that the morphology module will do all the reasoning necessary for these categories.

To declare that a category is lexical, you can simply add its name to the global variable `*lexical-categories*`. This variable is defined in file TOP.L. Its current value is:

```
(defvar *lexical-categories*
  '(verb noun adj prep conj relpro adv punctuation modal)
  "The Lexical Categories not to be unified")
```

8.2. CATegories Accepted by the morphology module

The following categories only are known by the morphology module. If a category of another type is sent to the morphology, no agreement can be performed. The output in that case is:

```
<Unknown cat CC: LEX>
```

```
MORPH accepts the following values as the value of the attribute CAT:
          ADJ, ADV, CONJ, MODAL, PREP, RELPRO, PUNCTUATION, PHRASE:
     words are sent unmodified.
NOUN:
     agreement in number is done.
     irregular plural must be put in the list *IRREG-PLURALS*
     in file LINEARIZE.L
PRONOUN:
     agreement done on pronoun-type, case, gender, number,
     distance, person.
     irregular pronouns are defined in file LINEARIZE.L
VERB:
     agreement is done on number, person, tense and ending.
     irregular verbs must be put in the list *IRREG-VERBS*
     in file LINEARIZE.L
DET :
     agreement is done on number, definite and first letter of
     following word for "a"/"an" or feature a-an of following word.
```

The function `morphology-help` will given you this information on-line if you need it.

8.3. Accepted features for VERB, NOUN, PRONOUN, DET and PUNCTUATION:

```
VERB:
     ENDING: {ROOT, INFINITIVE, PAST-PARTICIPLE, PRESENT-PARTICIPLE}
     NUMBER: {SINGULAR, PLURAL}
     PERSON: {FIRST, SECOND, THIRD}
     TENSE : {PRESENT, PAST}

NOUN:
     NUMBER: {SINGULAR, PLURAL}
     FEATURE:{POSSESSIVE}
     A-AN:   {AN, CONSONANT}

PRONOUN:
     PRONOUN-TYPE: {PERSONAL, DEMONSTRATIVE, QUESTION, QUANTIFIED}
     CASE:         {SUBJECTIVE, POSSESSIVE, OBJECTIVE, REFLEXIVE}
     GENDER:       {MASCULINE, FEMININE, NEUTER}
     PERSON:       {FIRST, SECOND, THIRD}
     NUMBER:       {SINGULAR, PLURAL}
     DISTANCE:     {NEAR, FAR}

DET :
     NUMBER: {SINGULAR, PLURAL}

PUNCTUATION:
     BEFORE: {";", ".", ":", "(", ")", ...}
     AFTER : {";", ".", ":", "(", ")", ...}
```

The feature A-AN is used to indicate exceptions to the rule: normally, a noun starting with a consonant is preceded by the indefinite article ''a'' and if the noun starts with a vowel, it is preceded by ''an.'' Some nouns start with a consonant but must still be preceded by ''an'' (for example, ''honor'' or acronyms ''an RST''). In that case, the feature (a-an an) must be added to the corresponding noun.

8.4. Possible values for features NUMBER, PERSON, TENSE, ENDING, BEFORE, AFTER, CASE, GENDER, PERSON, DISTANCE, PRONOUN-TYPE, A-AN

```
NUMBER: {SINGULAR, PLURAL}
        Default is SINGULAR.

ENDING: {ROOT, INFINITIVE, PAST-PARTICIPLE, PRESENT-PARTICIPLE}
        Default is none.

PERSON: {FIRST, SECOND, THIRD}
        Default is THIRD.

TENSE : {PRESENT, PAST}
        Default is PRESENT.

BEFORE: (";", ",", ":", "(", ")", ...) (any punctuation sign)
        Default is none.

AFTER : (";", ",", ":", "(", ")", ...)
        Default is none.

CASE:   {SUBJECTIVE, OBJECTIVE, POSSESSIVE, REFLEXIVE}
        Default is SUBJECTIVE.

GENDER: {MASCULINE, FEMININE, NEUTER}
        Default is MASCULINE.

PERSON: {FIRST, SECOND, THIRD}
        Default is THIRD.

DISTANCE: {FAR, NEAR}
        Default is NEAR.

PRONOUN-TYPE: {PERSONAL, DEMONSTRATIVE, QUESTION, QUANTIFIED}
        Default is none.

A-AN:   {AN, CONSONANT}
        Default is CONSONANT.
```

9. The Dictionary

The package includes a dictionary to handle the irregularities of the morphology only: verbs with irregular past forms and nouns with irregular plural only need to be added to the dictionary.

There is no semantic information within this dictionary. In fact, a more sophisticated form of lexicon should have the form of an FD. This dictionary is a part of the morphological module only.

The way to add information to the lexicon is to edit the values of the special variables *irreg-plurals* and *irreg-verbs*. These variables are defined in the file LEXICON.L. After the modification, you need to execute the function (initialiaze-lexicon). The best way to do that is to edit a copy of the file LEXICON.L and to load it back. After loading it, the new lexicon will be ready to use.

The variable *irreg-plurals* is a list of pairs of the form (key plural). The default list starts like this:

```
'(("calf" "calves")
  ("child" "children")
  ("clothes" "clothes")
  ("data" "data")
  ...)
```

The variable *irreg-verbs* is a list of 5-uples of the form: (root present-third-person-singular past present-participle past-participle)

The default value starts like that:

```
'(("become" "becomes" "became" "becoming" "become")
  ("buy" "buys" "bought" "buying" "bought")
  ("come" "comes" "came" "coming" "come")
  ("do" "does" "did" "doing" "done")
  ...)
```

10. Reference Manual

For the sake of completeness, this section includes a list of all the functions, variables and switches that a user of FUF can manipulate. They are grouped under 6 categories. In each category, the list is sorted alphabetically.

10.1. Unification functions

10.1.1. *lexical-categories*

Type: variable

Description: The `*lexical-categories*` variable is a list of category names. These categories are those that are sent to the morphology component without being unified.

Standard Value: `(verb noun adj prep conj relpro adv punctuation modal)`

10.1.2. *u-grammar*

Type: variable

Description: The `*u-grammar*` variable contains a Functional Unification Grammar. It is the default value to all the functions expecting a grammar as argument. It is a valid form if `grammar-p` accepts it.

10.1.3. u

Type: function

Calling form: (u *fd1 fd2*)

Arguments:

- `fd1` and `fd2` are arbitrary FDs. `fd1` cannot contain non-deterministic constructs, `fd2` can.

Description: u unifies *fd1* with *fd2* and returns 3 values: a resulting fd, a continuation to call if more results are needed and a "stack-frame" containing information needed to run the continuation. u is a low-level function.

10.1.4. uni

Type: function

Calling form: (`uni` *input-fd* `&optional` *grammar non-interactive*)

Arguments:

- `input-fd` is an input fd. It must be recognized by `fd-p`.
- `grammar` is a FUG. It must be recognized by `grammar-p`. By default, it is `*u-grammar*`.
- `non-interactive` is a flag. It is `nil` by default.

Description: `uni` unifies *input-fd* with *grammar* and linearizes the resulting fd. It prints the result and some statistics if *non-interactive* is `nil`. It returns no value. *grammar* is always considered as indexed on the feature `cat`. If *input-fd* contains no feature `cat` the unification fails. (cf. `unif` if this is the case.)

10.1.5. uni-fd

Type: function

Calling form: (`uni-fd` *input-fd* `&optional` *grammar non-interactive*)

Arguments:

- `input-fd` is an input fd. It must be recognized by `fd-p`.
- `grammar` is a FUG. It must be recognized by `grammar-p`. By default, it is `*u-grammar*`.
- `non-interactive` is a flag. It is `nil` by default.

Description: `uni-fd` unifies *input-fd* with *grammar* and returns the resulting total fd. The result is determined. `uni-fd` prints the same statistics as `uni` if *non-interactive* is `nil`. *grammar* is always considered as indexed on the feature `cat`. If *input-fd* contains no feature `cat` the unification fails. (cf. `unif` if this is the case.)

10.1.6. unif

Type: function

Calling form: (`unif` *input-fd* `&optional` *grammar non-interactive*)

Arguments:

- `input-fd` is an input fd. It must be recognized by `fd-p`.
- `grammar` is a FUG. It must be recognized by `grammar-p`. By default, it is `*u-grammar*`.
- `non-interactive` is a flag. It is `nil` by default.

Description: `unif` unifies *input-fd* with *grammar* and returns the resulting total fd. The result is determined.

If *input-fd* contains no feature `cat`, `unif` tries all the categories returned by `list-cats` until one returns a successful unification.

`unif` checks *input-fd* with `fd-p` and it checks *grammar* with `grammar-p`. `unif` prints the same statistics as `uni` if *non-interactive* is `nil`.

10.2. Checking

10.2.1. fd-syntax

Type: function

Calling form: (`fd-syntax &optional` *fd print-warnings*)

Arguments:

- `fd` is a list of pairs. It is `*u-grammar*` by default.
- `print-warnings` is a flag. It is nil by default.

Description: `fd-syntax` verifies that *fd* is a valid fd. If it is, it returns `T`. Otherwise, it prints helpful messages and returns `nil`. If *print-warnings* is non-`nil` it also print warnings for all the paths it encounters in the grammar. This is useful when you suspect that one path is invalid or pointing to a bad location.

Diagnostics detected by `fd-syntax`	
message	condition
FD should be a list of attr-value pairs.	One of the element of the list of pairs is not a pair and not a valid tracing flag.
--- WARNING: ~A is used as an attribute not as a flag.	One of the attributes of the pairs is a valid tracing flag, but is not considered as a tracing flag but as a regular attribute.
Too many values given.	One of the pairs contain more than 2 valid elements.
Too few values given.	One of the pairs contain less than 2 valid elements.
Illegal use of flag or too many values given.	A tracing flag is in a bad position.
Illegal value for the attribute OPT.	OPT expects a valid FD as a value.
Value of special attribute ALT should be a list of FD's.	The syntax of `alt` is `(alt (fd1 ... fdn))`. The value of `alt` is not a list of valid fds.
Value of special attribute OPT should be an FD.	Syntax of opt is `(opt fd)`.
Value of special attribute CSET must be a list of paths.	`cset` accepts a flat list of atoms or paths only as constituents.
Value of special attribute PATTERN should be a list of paths or mergeable atoms.	`pattern` accepts a flat list of atoms, paths or mergeable constituents. A mergeable constituent is marked (* c).
A value **should be either** a symbol, a valid path or an FD.	A pair is an `(attribute value)` list and value can only be a symbol, or a valid path (that is, a flat list of constituent names starting with 0 or more '^) or recursively an fd. None of these 3 categories has been recognized in this case.
--- WARNING: ~s is assumed to be a valid path.	When *print-warnings* is non-`nil`, this message is printed for all paths occuring in *fd*.

10.2.2. fd-sem

Type: function

Calling form: (`fd-sem &optional` *fd grammar-p*)

Arguments:

- `fd` is a syntactically valid fd. It must be recognized by `fd-p`. It is `*u-grammar*` by default.

- `grammar-p` is a flag. It is `T` by default.

Description: `fd-sem` verifies that *fd* is a semantically valid fd. If it is, it returns `T`. Otherwise, it prints helpful messages and returns `nil`. If *grammar-p* is non-`nil` `fd-sem` expects *fd* to be a grammar. It allows disjunctions in *fd*. In this case, `fd-sem` returns 3 values if *fd* is a valid grammar: `T`, the number of traced alternatives in the grammar, and the number of indexed alternatives.

If *grammar-p* is `nil`, *fd* is considered as an input fd. Disjunctions are not allowed. In any case, only one value is returned (`T` or `nil`).

Diagnostics detected by `fd-sem`	
message	condition
Disjunctions are not allowed in input fds.	*grammar-p* is `nil` and a disjunction has been found in *fd*.
--- Warning: PATTERN or CSET should not be placed in input.	*grammar-p* is `nil` and a `pattern` or `cset` has been found in *fd*.
Contradicting values for attribute ~s.	An attribute has been found with 2 different atomic values in the same branch of a disjunction. (for example, ((a 1) (a 2))).

10.2.3. fd-p

Type: function

Calling form: (`fd-p` *input-fd*)

Arguments:

- *input-fd* is an fd with no disjunctions.

Description: checks that *input-fd* is both syntactically and semantically a valid fd.

NOTE: Do not use fd-p on grammars.

10.2.4. grammar-p

Type: function

Calling form: (`grammar-p &optional` *fd print-messages print-warnings*)

Arguments:

- `fd` is a FUG. It is `*u-grammar*` by default.
- `print-messages` is a flag. It is `T` by default.
- `print-warnings` is a flag. It is `nil` by default.

Description: `grammar-p` verifies that *fd* is a valid grammar, both syntactically and semantically. If it is, it prints some statistics and returns `T`. Otherwise, it prints helpful messages and returns `nil`.

If *print-messages* is `nil` no statistics are printed.

If *print-warnings* is non-`nil` warnings are printed for all the paths encountered in the grammar. This is useful when you suspect that one path is invalid or pointing to a bad location.

NOTE: do not use `grammar-p` on input fds.

10.3. Tracing

10.3.1. *all-trace-off*

Type: variable.

Description: The `*all-trace-off*` variable contains a flag that is recognized by the unifier and terminates the printing of all tracing messages. It must be placed in a valid position for a tracing flag.

Standard Value: %TRACE-OFF%

10.3.2. *all-trace-on*

Type: variable.

Description: The `*all-trace-on*` variable contains a flag that is recognized by the unifier and undoes the effect of the *all-trace-off* flag, that is, it reenables all tracing messages. It must be placed in a valid position for a tracing flag.

Standard Value: %TRACE-ON%

10.3.3. *trace-determine*

Type: variable.

Description: The `*trace-determine*` is a switch enabling the printing of tracing messages on the determination stage. It indicates which `TEST` expressions are evaluated.

Standard Value: `nil`

10.3.4. *trace-marker*

Type: variable.

Description: The `*trace-marker*` variable contains a character. It is used to determine valid tracing flags: if the first character of the name of a symbol is *trace-marker*, the symbol is a valid tracing-flag.

Standard Value: `#\%`

10.3.5. *top*

Type: variable.

Description: The `*top*` variable is a switch enabling the printing of extensive debugging messages on the backtracking behavior of the unifier. Should be used for development only.

Standard Value: `nil`

10.3.6. all-tracing-flags

Type: function

Calling form: `(all-tracing-flags &optional` *grammar*`)`

Arguments:

- *grammar* is a FUG. It must be recognized by `grammar-p`. By default, it is `*u-grammar*`.

Description: `all-tracing-flags` returns a list of all the tracing flags defined in *grammar*, in the order where they are defined in the grammar.

10.3.7. internal-trace-off

Type: function

Calling form: (`internal-trace-off`)

Description: `internal-trace-off` turns off the tracing of internal debugging information. Initially, no debugging information is printed.

10.3.8. internal-trace-on

Type: function

Calling form: (`internal-trace-on`)

Description: `internal-trace-on` turns on the tracing of internal debugging information. Initially, no debugging information is printed. Should be used for development only.

10.3.9. trace-disable

Type: function

Calling form: (`trace-disable` *flag*)

Arguments:

- `flag` is a tracing flag. A tracing flag must be an element of the result of `all-tracing-flags`.

Description: `trace-disable` disables the tracing flag *flag*. Initially, all tracing flags are enabled.

10.3.10. trace-disable-all

Type: function

Calling form: (`trace-disable-all`)

Description: `trace-disable-all` disables all tracing flags. Initially, all tracing flags are enabled.

10.3.11. trace-disable-match

Type: function

Calling form: (`trace-disable-match` *string*)

Arguments:

- *string* is a string.

Description: `trace-disable-match` disables all tracing flags whose names contain *string* as a substring. Initially, all tracing flags are enabled.

10.3.12. trace-enable

Type: function

Calling form: (`trace-enable` *flag*)

Arguments:

- `flag` is a tracing flag. A tracing flag must be an element of the result of `all-tracing-flags`.

Description: `trace-enable` enables the tracing flag *flag*. Initially, all tracing flags are enabled.

10.3.13. trace-enable-all

Type: function

Calling form: (`trace-enable-all`)

Description: `trace-enable-all` enables all tracing flags. Initially, all tracing flags are enabled.

10.3.14. trace-enable-match

Type: function

Calling form: (`trace-enable-match` *string*)

Arguments:

- *string* is a string.

Description: `trace-enable-match` enables all tracing flags whose names contain *string* as a substring. Initially, all tracing flags are enabled.

10.3.15. trace-off

Type: function

Calling form: (`trace-off`)

Description: `trace-off` turns off tracing. If no argument is provided, all tracing is turned off. Initially, tracing is off.

10.3.16. trace-on

Type: function

Calling form: (`trace-on`)

Description: `trace-on` turns on tracing.

Initially, tracing is off.

10.4. Complexity

10.4.1. avg-complexity

Type: function

Calling form: (`avg-complexity &optional` *grammar with-index rough-avg*)

Arguments:

- `grammar` is a grammar. It must be recognized by `grammar-p`. It is `*u-grammar*` by default.
- `with-index` is a flag. It is `T` by default.
- `rough-avg` is a flag. It is `nil` by default.

Description: `avg-complexity` computes a measure of the average complexity of a grammar. It tries to compute an "average" number of branches tried when the input to unification contains no constraint.

When *with-index* is `T`, all indexed alts are considered as single branches, when it is `nil`, they are considered as regular `alts`.

When *rough-avg* is `nil`, the average of an `alt` is the sum of the complexity of the first half of the branches. When it is `T`, the average is half the sum of the complexity of all branches.

10.4.2. complexity

Type: function

Calling form: (`complexity &optional` *grammar with-index*)

Arguments:

- `grammar` is a grammar. It must be recognized by `grammar-p`. It is `*u-grammar*` by default.
- `with-index` is a flag. It is `T` by default.

Description: `complexity` computes a measure of the complexity of a grammar. It tries to compute the worst case number of branches tried when the input to unification contains no constraint. The number it returns is equivalent to the number of branches the grammar would have in disjunctive normal form.

When *with-index* is `T`, all indexed alts are considered as single branches, when it is `nil`, they are considered as regular `alts`.

10.5. Manipulation of the dictionary

10.5.1. *dictionary*

Type: variable

Description: The `*dictionary*` variable is a hash-table containing different types of entries. Each entry contains information on irregular morphological words.

The current dictionary contains entries for verbs, nouns and pronouns. It is defined in file LEXICON.L

The entries contain the following properties:

- verb : present-third-person-singular past present-participle past-participle
- noun : plural
- pronoun : subjective objective possessive reflexive.

10.5.2. lexfetch

Type: function

Calling form: (`lexfetch` *key property*)

Arguments:

- *key* is a non-inflected "root" form of a word. It must be a string.
- *property* is one of the properties defined in *dictionary* for the part-of-speech of the word.

Description: `lexfetch` fetches the inflected form of the word *key* from the hash-table `*dictionary*`. The properties accessible are those defined in `*dictionary*`.

10.5.3. lexstore

Type: function

Calling form: (`lexstore` *key property value*)

Arguments:

- *key* is a non-inflected "root" form of a word. It must be a string.
- *property* is one of the properties defined in *dictionary* for the part-of-speech of the word.
- *value* is the inflected form of *key* for *property*. It must be a string.

Description: `lexstore` stores the inflected form *value* of the word *key* in the hash-table `*dictionary*`. The properties accessible are those defined in `*dictionary*`.

10.6. Linearization and Morphology

10.6.1. call-linearizer

Type: function

Calling form: (`call-linearizer` *fd*)

Arguments:

- *fd* is a unified determined total fd. It must be accepted by `fd-p`.

Description: `call-linearizer` takes a complete determined fd in input and returns a string corresponding to the linearization of the fd.

10.6.2. gap

Type: feature.

Description: if a constituent contains the feature gap, it is not realized in the surface (it is a gap, still holding the place of an invisible constituent in the structure). It is used for implementing long-distance dependencies.

10.6.3. morphology-help

Type: function.

Calling form: `(morphology-help)`

Description: gives on-line help on what the morphology component can do.

10.7. Manipulation of FDs as data-structures

10.7.1. FD-intersection

Type: function

Calling form: (`fd-intersection` *fd1 fd2*)

Arguments:

- *fd1* and *fd2* are valid fds (recognized by `fd-p`). They represent lists as fds, using constituents `car` and `cdr`, and are terminated by a `(cdr none)`.

Description: `fd-intersection` computes the intersection of two lists represented as FDs, and returns the result as a regular Lisp list.

10.7.2. FD-member

Type: function

Calling form: (`fd-member` *elt fdlist*)

Arguments:

- *elt* is any value acceptable as a value to an (attribute value) pair.
- *fdlist* is a valid fd (recognized by `fd-p`). It represents a list as an fd, using constituents `car` and `cdr`, and is terminated by a `(cdr none)`.

Description: `fd-member` works as the lisp function `member` but on a list represented by an fd. It returns a list represented by an fd.

10.7.3. FD-to-list

Type: function

Calling form: (`fd-to-list` *fdlist*)

Arguments:

- *fdlist* is a valid fd (recognized by `fd-p`). It represents a list as an fd, using constituents `car` and `cdr`, and is terminated by a `(cdr none)`.

Description: `fd-to-list` converts a list from an fd representation to a lisp representation.

10.7.4. gdp

Type: function

Calling form: (`gdp` *fd path*)

Arguments:

- *fd* is a valid fd (recognized by `fd-p`).
- *path* is a valid path (that is a flat list of constituent names, starting with 0 or more ^)

Description: `gdp` goes down the path *path* (hence its name: GoDownPath) and returns the fd found at the end of *path*. It is the only function that should be used to access sub-parts of an fd. `gdp` always returns a valid fd.

`gdp` works only if the special variable *input* is accessible and bound to the total fd containing *fd*.

If *path* leads to a non-existent sub-fd, `gdp` returns:

- `NONE`: if the fd cannot be extended to include such a sub-fd (that's when we meet an atom on the way down)
- `ANY` : if the fd MUST be extended to include such a sub-fd (and exactly this sub-fd, that is only when the value is ANY)
- `NIL` : otherwise (that is, an UNRESTRICTED fd).

10.7.5. gdpp

Type: function

Calling form: (`gdpp` *fd path frame*)

Arguments:

- *fd* is a valid fd (recognized by `fd-p`).
- *path* is a valid path (that is a flat list of constituent names, starting with 0 or more ^)
- *frame* is a structure of type `frame`. By default it is `dummy-frame`, an empty frame.

Description: `gdpp` goes down the path *path* (hence its name: GoDownPathPair) and returns the pair whose value is the fd found at the end of *path*. It is the function that should be used to work as the basis to the `setf` of `gdp`, to set values to parts of an fd. `gdpp` always returns a pair whose second is a valid fd, and is never a path or `none` if *fd* cannot e extended to include *path*. `(gdpp *input* nil)` returns the pair `(*top* *input*)` (where *input* refers to the total fd).

`gdpp` works only if the special variable *input* is accessible and bound to the total fd containing *fd*.

If *path* leads to a non-existent sub-fd, `gdpp` extends (by physical modification) *fd* to include a path down to the required *path* if possible, or the function returns `none`. When the fd is modified physically, *frame* is updated (the field *undo*) to keep track of the modification.

10.7.6. list-to-FD

Type: function

Calling form: (`list-to-fd` *list*)

Arguments:

- *list* is a regular lisp list.

Description: `list-to-fd` converts a list from a a lisp representation to an FD representation.

10.8. Fine tuning of the unifier

10.8.1. *any-at-unification*

Type: variable

Description: If `*any-at-unification*` is `nil`, and the unifier encounters a pair (attribute `any`) in the grammar, and no feature `attribute` exists in the input, the unification succeeds and the input is enriched with the pair (attribute `any`). Only at the determination stage, it is checked whether `any`s remain in the total fd. If it is the case, the unification fails, and the unifier backtracks.

If `*any-at-unification*` is non-`nil`, the test to decide whether the feature attribute exists or not is performed immediately on the non-determined fd. The result may be incorrect, but it is much faster. The result is assured to be correct if the feature tested is one that is never instantiated by the grammar, and is expected to be provided in the input.

Standard Value: T

10.8.2. *keep-cset*

Type: variable

Description: If `*keep-cset*` is `nil`, the determination stage removes all the `cset` features from the total fd. If it is `T` it keeps them.

Standard Value: `nil`

10.8.3. *keep-none*

Type: variable

Description: If `*keep-none*` is `nil`, the determination stage removes all the pairs whose value is `none` from the total fd. If it is `T` it keeps them.

Standard Value: `T`

Appendix I
Installation of the Package

I.1. Finding the files

You need to find out on which machine and under which directory the system is available. You also need to know how to run Common Lisp on that machine.

```
Language : Common Lisp
System   : At Columbia, available
           on Lisp-A (Symbolics), in directory >elhadad>fuf>
           on the HP workstations (HP-UX), in /u/cs/elhadad/Fug/work/
           (define environment variable "fug2" to this value:
            under csh: setenv fug2 /u/cs/elhadad/Fug/work
            under ksh: fug2=/u/cs/elhadad/Fug/work; export fug2)
          Examples are in the subdirectory named "examples".

Start    : on Lisp-A:   (load ">elhadad>fuf>fug")
           on the HPs: % cl    #need to have /lisp/bin is path
                       CL> (load "$fug2/fug2")
```

The file FUG2.L will load all the required modules. Examples are in the files GR0, GR1 and up for the grammars, and in files IR0, IR1, ... and up for the inputs. The examples are of increasing complexity.

```
To try the examples, type:

           CL> (load "gr0")
           t
           CL> (load "ir0")
           t
```

I.2. Porting to a new machine

The program is contained in 16 files of source and 10 files of examples. All the source files should be grouped in a directory, that we will call here `$fug2`, and the example files in a subdirectory of `$fug2` called `examples`.

Once this is done, you probably need to edit the file FUG2.L. This file loads all the required modules and defines a few functions useful for compiling or loading the package. In the file FUG2.L, the function `require` is used to load all submodules. `require` takes as first argument the name of a module, and accepts a second optional argument, the name of the file containing that module.

You must change the second arguments of all the `require` statements in file FUG2.L and update there the name of the directory, from `$fug2` to the name of your directory.

You also need to edit the first line of the functions `compile-fug` and `reload-fug` and change there the name of the directory from `$fug` to the new name.

When the file FUG2.L is updated, load it in your common-lisp environment and follow these 4 steps:

```
(load "$fug2/fug2")

(in-package "FUG2")

(compile-fug2)

(reload-fug2)
```

NOTE TO UNIX USERS: if you run CommonLisp under Unix, and your version of Lisp can read environment variables and expands such variables in file names (for example, `(load "~userx/file1")` is a valid statement, or `(load "$var/file2")`), then you don't need to edit the file FUG2.L. All you need to do is to define the environment variable `"fug2"` to the complete pathname of the directory containing the source files.

Once this installation is done, all you need to do to load the package is `(load "$fug2/fug2")` (with `$fug2/` replaced by the name of your directory if you are not under Unix).

I.3. Packages

The whole package is loaded in package `'FUG2`. The easiest way to access it is to type:

```
(in-package "FUG2")    ;; note the upper-case

or

(use-package "FUG2")
```

The following symbols are exported from package 'FUG2 (they are the external symbols of the package, cf [Steele-84, chapter 11, p171-192]):

External Symbols of package FUG	
File	Symbols
checker.l	fd-p fd-syntax fd-sem grammar-p
complexity.l	complexity avg-complexity
determine.l	*keep-cset* *keep-none*
graph.l	*any-at-unification*
lexicon.l	*dictionary* lexfetch lexstore
linearize.l	call-linearizer morphology-help
path.l	gdp gdpp
top.l	*u-grammar* *lexical-categories* uni uni-fd unif list-cats
trace.l	trace-on trace-off internal-trace-on internal-trace-off trace-enable trace-disable trace-enable-all trace-disable-all trace-enable-match trace-disable-match all-tracing-flags *trace-marker* *all-trace-off* *all-trace-on* *trace-determine* *top*

In addition, the following symbols are external. These are the keywords used as names in the code:

External Symbols of package FUG (keywords)	
File	Symbols
	`===`
	`* already exists in LISP`
	`trace already exists in USER`
	`@`
	`^`
	`alt`
	`any`
	`cat`
	`control`
	`cset`
	`dots`
	`*done*`
	`gap`
	`given`
	`index`
	`lex`
	`mergeable`
	`none`
	`opt`
	`pattern`
	`pound`
	`punctuation`
top.l	`test`

All these symbols are documented for reference in section 10. If you use the package FUG2 in another package, only these symbols will be imported.

Appendix II
Advanced Features

II.1. Advanced Uses of Patterns

In addition to constrain the ordering of constituents, the pattern unifier can be used to enforce the unification of constituents. The classical example is given by the `focus` constituent. There is good linguistic evidence that the focus of a sentence tends to occur first in a sentence. To represent this constraint, a grammar can include the following directive:

```
(PATTERN (FOCUS DOTS))
```

That is, a sentence should start with its focus. Now, we also know that a sentence at the active voice should start with its subject, that is its `prot` constituent. This is expressed by:

```
(PATTERN (PROT ... VERB ...))
```

If both constraints are to be satisfied, we need to say that `focus` and `prot` are actually the same constituent, otherwise, the 2 patterns are incompatible. That is, the constituents `focus` and `prot` need to be unified. This mechanism would be quite expensive to implement for all constituents, and would need to meaningless attempts most of the time. Therefore, to allow this kind of unification to occur, the current unifier requires the pattern to include a special directive, indicating that a constituent can be unified with other constituents to make two patterns compatible. The notation used is: `(* constituent)`.

```
Example:
(PATTERN ((* FOCUS) DOTS))
(PATTERN (PROT DOTS VERB DOTS))
```

are compatible, and require the unification of the constituents `focus` and `prot`. Note that `prot` needs not be "stared" to be unified with `focus`. The notation can be understood as specifying that `focus` is a kind of "meta-constituent".

II.2. Advanced uses of CSET

Note that CSET is rarely used, and most often used when you DO NOT want a sub-fd to be unified as a constituent, even though it is mentioned in a pattern or it contains a feature (cat xx).

When a CSET feature is specified, the order of the constituents can be important to make unification more efficient. The unifier traverses the input fd breadth-first identifying constituents at each level. Within the same level, the CSET feature when present specifies in which order the constituents must be unified. Therefore, if there is a constituent known to be easy to unify, and whose value condition the unification of the brother constituents, it should be unified first, and placed first in the CSET. This way, the CSET feature can be used to optimize the work of the unifier.

```
((cat hard)
 (a #)      ; is hard to unify
 (b #)      ; is hard to unify
 (c #)      ; is easy to unify and constrains the unification of a and b
 (cset (c a b)))      ; unify c first, then a and b.
```

II.3. Long Distance Dependencies and the GAP feature

The special feature gap is used to indicate that a constituent must not be realized in the surface text. If a constituent contains an attribute gap with any non-NONE value, the linearizer will skip it.

This device is used to implement long-distance dependencies in grammars. For example, in a relative clause, the relative pronoun can be viewed as the marker of the relativization, and the relative clause as a complete clause, with one constituent elided. Thus, in *The man whom I know*, the relative clause would have the structure *I know the man* and the constituent *the man* would be a gap, whereas the relative pronoun *whom* would inherit its properties.

II.4. Specifying complex constraints: the TEST and CONTROL keywords

NOTE: These two keywords are specific to this implementation. Their use is not recommended. See appendix IV for a list of the non-standard features of this implementation.

test and control are two "impure" specifications: they do not rely on the principle of unification to prevent a successful unification of 2 FDs. control should not be used except under extremely special circumstances. For the time being, it can be considered a synonym of test.

test is used to add a complex constraint on the result of a unification. A complex constraint refers to any Lisp predicate. If at the end of the unification the predicate is satisfied when applied to the resulting fd, the unification succeeds, otherwise it fails, and the unifier backtracks to find another solution.

The special character '@' is used to refer to parts of the FD in the expression of the constraints. A ' must be followed by a valid path (either absolute or relative). The expression @(^ ^ a b) is replaced by the value of the feature refered to by that path before the predicate is evaluated.

The order in which the test predicates will be evaluated is obviously not determined. Side effects are therefore STRONGLY discouraged within the body of the test constraints.

```
Examples:
((a 1)
 (test (equal @(a) @(b))))

is equivalent to the nicer:

((a 1)
 (b (a)))

((a 1)
 (test (numberp @(a))))
```

There is conceptually the same difference between TEST and CONTROL as there is between ANY and

GIVEN: TEST constraints are tested at determination time, whereas CONTROL constraints are tested as soon as the unifier meets them. CONTROL is therefore in general much more efficient than TEST, but the results it provides are unpredictable in certain cases (if the features tested are given a different value later on during the unification, the result of the test could be different).

Appendix III
Non linguistic applications of the unifier: dealing with lists

Unification as used in the theory of functional unification grammars is a powerful mechanism that is not restricted to linguistic domains. It can be viewed as a "programming language" of its own. Actually, it is similar by many aspects to PROLOG. There are however some very specific features that make working with this version of unifcation well adapted to grammars, and not so well to more classic programming tasks.

III.1. The member/append example

To make things clear, this implementation includes a "grammar" doing some list processing. The only operations presented are *member* and *append*. This grammar is in the directorey *examples* in file GR5.L. It is printed here for easy reference for the discussion.

```
'((alt
   (((cat append)
     (alt append
      ;; First branch: append([],Y,Y).
      (((x none)
        (z (^ y))
        ;; This is to normalize the result of a (cat append):
        ;; it must contain the CAR and CDR of the result.
        (car (^ z car))
        (cdr (^ z cdr)))

      ;; Second branch: append([X/Xs],Y,[X/Z]):-append(Xs,Y,Z).
       ((alt (((x ((car any))))    ; this alt allows for partially
              ((x ((cdr any)))))) ; defined lists X in input.
        ;; recursive call to append
        ;; with new arguments x, y and z.
        (cset (z))
        (z ((car (^ ^ x car))
            (cdr ((cat append)
                  (x (^ ^ ^ x cdr))
                  (y (^ ^ ^ y))))))
        (car (^ z car))
        (cdr (^ z cdr))))))
    (((cat member)
      (alt member
       (((x (^ y car)))
        ((y ((cdr any)))
         (m ((cat member)
             (x (^ ^ x))
             (y (^ ^ y cdr)))))))))))
```

This grammar is actually almost equivalent to the following PROLOG program:

```
member(X, [X|_]).
member(X, [_|Y]) :- member(X,Y)

append([],Ys,Ys).
append([X|Xs],Ys, [X|Zs]) :- append(Xs,Ys,Zs).
```

Note that the PROLOG form is much nicer! But there are reasons to look at the FUG version anyway. Here is how it works.

III.2. Representing lists as FDs

The first problem to handle lists with FUGs, is to represent lists as FDs, since FUGs can handle only FDs.

Quite simply, lists are represented as an FD with two features, CAR and CDR (with names ala Lisp).

```
The list (a b c) is represented by the FD:

((car a)
 (cdr ((car b)
       (cdr ((car c)
             (cdr none))))))

The list (a (b c)) is represented by the FD:

((car a)
 (cdr ((car ((car b)
             (cdr ((car c)
                   (cdr none)))))
       (cdr none))))
```

III.2.1. NIL and variables

Note in the previous example that the equivalent of the lisp atom NIL is NONE in the FD. NIL in an FD means "anything can come here" whereas NONE means "nothing can come here". NIL therefore plays a role similar to uninstantiated variables in PROLOG.

```
The PROLOG expression [a X c] can be represented by the FD:

((car a)                                  ((car a)
 (cdr ((car nil)                           (cdr ((cdr ((car c)
       (cdr ((car c)          <==>                     (cdr none))))))
             (cdr none))))))

The PROLOG expression [a b | Xs] can be represented by the FD:

((car a)
 (cdr ((car b))))
```

III.2.2. The "~" notation

The car/cdr notation for lists is very awkward to use. The file FDLIST.L includes a mechanism to translate between the regular Lisp notation and the FD notation. It defines the macro-character "~" to indicate list values.

```
((cat member)           ((cat member)
 (x a)             <==>  (x a)
 (y ~(c b a)))           (y ((car c)
                             (cdr ((car b)
                                   (cdr ((car a)
                                         (cdr none))))))))
```

Note that the "~" notation can be used only for completely specified lists. If some elements are uninstantiated, you must describe the list with the car/cdr notation.

III.3. Environment and variable names vs. FD and path

The notions of environment and variable in PROLOG or LISP correspond to the notion of "total FD" and path in Functional Unification. What we call a "total FD" is the highest level FD, the one corresponding to the path (). It is the FD corresponding to the input to the unifier, and that will be "determined" at the end of unification. This FD contains all the environment of a computation.

Variables are then just places or positions within this total FD.

```
If the total FD is the FD corresponding to [a X c]

((car a)
 (cdr ((cdr ((car c)
             (cdr none))))))

The variable X can be refered to by using the path (cdr car)
```

III.4. Procedures vs. Categories, Arguments vs. Constituents

A program in FUG can be viewed as a collection of procedures, each procedure being represented by a category. In the member example of section 10.1, an input containing the feature *(cat member)* will be sent to the *member* procedure.

Procedures expect arguments and return results. There is no notion of input and output in unification, as far as arguments are concerned. So we just consider arguments in general. For example, the *member* procedure has two arguments, called X and Y and represented in FUG notation by the constituents X and Y of the *(cat member)*.

The procedure *append* has three arguments, X, Y and Z. Z can be seen as the "result" of the procedure, or in functional notation: Z = append(X,Y).

Note that, as in the corresponding PROLOG program, the FUG implementation of *member* and *append* is non-directional. All of the arguments can be partially specified, and the unification enforces the relation existing between them.

III.5. The total FD includes the stack of all computation

One problem with the way FUG work is that there is no notion of "environment" besides the total FD. Therefore, when a program works recursively, all the local variables that are normally stacked in an external environment are stacked within the total FD. At the end, the total FD contains the whole stack of the computation, and is pretty heavy to manipulate.

As an example here is the result of the simple call append([a,b],[c,d],Z):

```
((CAT APPEND)
 (X ((CAR A) (CDR ((CAR B) (CDR NONE)))))
 (Y ((CAR C) (CDR ((CAR D) (CDR NONE)))))
 (Z
  ((CAR A)
   (CDR
     ((CAT APPEND)
      (X ((CAR B) (CDR NONE)))
      (Y ((CAR C) (CDR ((CAR D) (CDR NONE)))))
      (Z
       ((CAR B)
        (CDR
          ((CAT APPEND)
           (X NONE)
           (Y ((CAR C) (CDR ((CAR D) (CDR NONE))))
           (Z ((CAR C) (CDR ((CAR D) (CDR NONE)))))
           (CAR C)
           (CDR ((CAR D) (CDR NONE)))))))
      (CAR B)
      (CDR
        ((CAT APPEND)
         (X NONE)
         (Y ((CAR C) (CDR ((CAR D) (CDR NONE)))))
         (Z ((CAR C) (CDR ((CAR D) (CDR NONE)))))
         (CAR C)
         (CDR ((CAR D) (CDR NONE))))))))))
 (CAR A)
 (CDR
   ((CAT APPEND)
    (X ((CAR B) (CDR NONE)))
    (Y ((CAR C) (CDR ((CAR D) (CDR NONE)))))
    (Z
     ((CAR B)
      (CDR
        ((CAT APPEND)
         (X NONE)
         (Y ((CAR C) (CDR ((CAR D) (CDR NONE)))))
         (Z((CAR C) (CDR ((CAR D) (CDR NONE)))))
         (CAR C)
         (CDR ((CAR D) (CDR NONE)))))))
    (CAR B)
    (CDR
      ((CAT APPEND)
       (X NONE)
       (Y ((CAR C) (CDR ((CAR D) (CDR NONE)))))
       (Z ((CAR C) (CDR ((CAR D) (CDR NONE)))))
       (CAR C)
       (CDR ((CAR D) (CDR NONE))))))))
```

Fortunately, the only thing of interest in this FD is probably the value of the constituents CAR and CDR of Z.

III.6. Analogy with PROLOG programs

We have seen so far what aspects of FUGs are specific and different from other programming languages.

A program written using a FUG is very similar to a PROLOG program:

- The notion of success and failure in unification are equivalent to the "yes" and "no" of PROLOG programs.
- Simple statement can be combined using the connectives AND and OR: both FDs and PROLOG statements make use of conjunction and disjunction.
- Both notations rely heavily on unification, and refinement of partial descriptions to perform computations.

III.7. Use of Set values in linguistic applications

This discussion of FUGs as programming languages can appear frivolous. It is actually motivated by the desire to integrate more expressive features in linguistic grammars.

There are many different reasons to use set values in grammatical descriptions. For example, to describe a conjunction like "John, Mary and Frank" the set {John, Mary, Frank} appears as a good candidate. Many other applications for the category of set appear quite naturally when writing a grammar.

We want to be able to express grammatical constraints on such constructs within the framework of FUGs. We have found the procedures *member* and *append* to be quite useful in this attempt.

Appendix IV
Non standard features of the implementation and restrictions

The current implementation includes features not available in other systems working with functional unification, and imposes restrictions. This section lists these non-standard aspects of the implementation. For each of the restriction, it is precised whether the checking functions (`fd-p`, `fd-sem` and `grammar-p`) detect the limitation or not.

IV.1. No disjunction in input

The input must be a simple FD, containing no disjunction (`alt` or `opt`). It can contain patterns. `tests` and `controls` are not allowed in input.

It is advised not to put `patterns`, `csets` or `anys` in the input fd. These constructs are indeed best viewed as devices used by the grammar to realize or enforce some constraints. The input should be left as "declarative" as possible, and therefore should not contain such constructs.

If disjunction are found in an FD given to `fd-sem`, an error message is printed. `fd-sem` also issues warnings if its argument contains `patterns` or `csets`.

IV.2. Mergeable constituents in patterns

An extension to the standard pattern unification mechanism is the use of "mergeable constituents". A mergeable constituent in a pattern is noted `(* constituent-name)`. This notation indicates that when unifying the pattern containing it, this constituent can be "merged" or unified with another constituent that would need to be placed at the same position in the pattern.

For example, patterns `(a ... b)` and `(c ... b)` cannot be unified, because the first position of the unifying pattern would need to be both `a` and `c`. But patterns `((* a) ... b)` and `(c ... b)` can be unified, under the constraint that constituents `a` and `c` be unified (or "merged"). See also section 5.5 for a description of pattern unification.

IV.3. Indexing of alternation

This implementation allows indexing of `alts`, as described in section 7. The notation used is:

```
(alt {trace-flag} {(index {...} indexed-path)} (branches+))
```

where **each branch** is a regular fd. The validity of the `indexed-path` is checked by the function `grammar-p`.

IV.4. Test and Control

It is possible to specify arbitrary constraints on the result of an unification within the grammar by using the constructs `test` and `control` described in section 4.7. The notation is:

```
(TEST <lisp-expression>)
```

where *<lisp-expression>* is an arbitrary lisp expression, where certain variables can be `@(path)`, and refer to the value of `(path)` in the determined result of the unification (see section 5.8 for a definition of the determination stage of unification).

Unification succeeds if the evaluation of *<lisp-expression>* in the environment of the determined result is non-`nil`. If it is `nil`, the unifier backtracks.

`control` works in a similar way, except that the *<lisp-expression>* is evaluated immediately when the unifier encounters the `control`, and therefore is evaluated in a non-determined fd.

Note that both `test` and `control` can be used only to enforce complex constraints but not to compute complex results to be added in the unification.

The function `grammar-p` does not check that the value of `test` and `control` is a valid lisp-expression.

IV.5. GIVEN

The special value `given` is defined in this implementation. A feature `(att given)` is unified with an input fd, if the input contains a real value for attribute `att` at the beginning of the unification.

`given` is useful to check the presence of required features in inputs.

References

[1] Grosz, B.J., Sparck Jones, K. and Webber, B.L.
Readings in Natural Language Processing.
Morgan Kaufmann, Los Altos, 1986.

[2] Karttunen, L.
Features and Values.
In *Proceedings of the 10th International Conference on Computational Linguistics (COLING 84)*, pages 28-33. ACL, Stanford, California, July, 1984.

[3] Karttunen, L.
Structure Sharing with Binary Trees.
In *Proceedings of the 23rd annual meeting of the ACL*, pages 133-137. ACL, Chicago, 1985.

[4] Karttunen, L.
D-PATR: A development Environment for Unification-Based Grammars.
In *Proceedings of the 11th International Conference on Computational Linguistics (COLING 86)*, pages 74-79. ACL, Bonn, 1986.

[5] Karttunen, L.
D-PATR: A Development Environment for Unification-Based Grammars.
Technical Report CSLI-86-61, CSLI, August, 1986.

[6] Karttunen, L.
Parsing in a Free Word Order Language.
Natural Language Parsing.
Cambridge University Press, Cambridge, England, 1985, pages 279-306.

[7] Kasper, R.
Systemic Grammar and Functional Unification Grammar.
Systemic Functional Perspectives on discourse: selected papers from the 12th International Systemic Workshop.
Ablex, Norwood, NJ, 1987.

[8] Kasper, R.
A Unification Method for Disjunctive Feature Descriptions.
In *Proceedings of the 25th meeting of the ACL*, pages 235-242. ACL, Stanford University, June, 1987.

[9] Kasper, R. and W. Rounds.
A Logical Semantics for Feature Structures.
In *Proceedings of the 24th meeting of the ACL.* ACL, Columbia University, New York, NY, June, 1986.

[10] Kay, M.
Functional Grammar.
In *Proceedings of the 5th meeting of the Berkely Linguistics Society.* Berkeley Linguistics Society, 1979.

[11] Kay, M.
Algorithm Schemata and Data Structures in Syntactic Processing.
Technical Report CSL-80-12, Xerox Parc, October, 1980.
Also in Readings in NLP, p35-70.

[12] Kay, M.
Functional Unification Grammars: a Formalism for Machine Translation.
In *Proceedings of the 10th International Conference on Computational Linguistics (COLING 84)*, pages 75-78. ACL, Stanford University, 1984.

[13] Kay, M.
Parsing in Functional Unification Grammar.
Natural Language Parsing.
Cambridge University Press, Cambridge, England, 1985, pages 152-178.
Also in Reading in NLP p125-138.

[14] Pereira, F.C.N.
A Structure-Sharing Representation for Unification-Based Grammar Formalisms.
In *Proceedings of the 23rd annual meeting of the ACL*, pages 137-144. ACL, Chicago, 1985.

[15] Pereira, F. and S. Shieber.
The Semantics of Grammar Formalisms Seen as Computer Languages.
In *Proceedings of the Tenth International Conference on Computational Linguistics (COLING 84)*, pages 123-129. ACL, Stanford University, Stanford, Ca, July, 1984.

[16] Ritchie, G.D.
Simulating a Turing Machine using Functional Unification Grammar.
In *Proceedings of the Europeean Conference on AI (ECAI 84)*, pages 127-136. 1984.

[17] Ritchie, G.D.
The Computational Complexity of Sentence Derivation in Functional Unification Grammar.
In *Proceedings of the 11th International Conference on Computational Linguistics (COLING 86)*, pages 584-586. ACL, Bonn, 1986.

[18] Rounds, W.C. and A. Manaster-Ramer.
A Logical Version of Functional Grammar.
In *Proceedings of the 25th meeting of the ACL*, pages 89-96. ACL, Stanford University, June, 1987.

[19] Shieber, S.M.
The Design of a Computer Language for Linguistic Information.
In *Proceedings of the 10th International Conference on Computational Linguistics (COLING 84)*, pages 362-366. ACL, Stanford University, 1984.

[20] Shieber, S.M.
Using Restriction to Extend Parsing Algorithms for Complex Feature-Based Formalisms.
In *Proceedings of the 23rd annual meeting of the ACL*, pages 145-152. ACL, Chicago, 1985.

[21] Shieber, S.M.
A Compilation of Papers on Unification-Based Grammar Formalisms, Parts I & II.
Technical Report CSLI-85-48, CSLI, 1985.
3 papers COLING 84 + 3 ACL 85.

[22] Shieber, S.
CSLI Lecture Notes. Volume 4: *An introduction to Unification-Based Approaches to Grammar.*
University of Chicago Press, Chicago, Il, 1986.

[23] Wittenburg, K.B.
Natural Language Parsing with Combinatory Categorial Grammar in Graph Unification-Based Formalism.
PhD thesis, Austin University, 1986.

[24] Wroblewski, D.A.
Non Destructive Graph Unification.
In *Proceedings of the Sixth National Conference on AI (AAAI 87)*, pages 582-587. AAAI, Seattle, 1987.

Index