

**FUF: the Universal Unifier**  
**User Manual**  
**Short Version**  
**Version 5.2**

*Michael Elhadad*

Department of Computer Science  
Ben Gurion University of the Negev  
84105 Beer Sheva, Israel  
elhadad@bengus.bgu.ac.il

27 September 1995

**Abstract**

This document is the user manual for FUF version 5.2, 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 5.2). Version 5.2 includes novel techniques in the unification allowing the specification of types and the expression of complete information. It also allows for procedural unification and supports sophisticated forms of control.

Copyright © 1995 Michael Elhadad

# 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. SURGE, a grammar of English with large syntactic coverage written in FUF, 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 falling in any of these categories. It starts with an introduction to functional unification, its syntax, semantics and terminology. The following chapters deal with the ‘‘grammar development’’ tools: tracing and indexing, a presentation of the morphology component and the dictionary. The next two chapters present the novel features of FUF: typing and control facilities. A chapter is devoted to typing in FUF: type definition, user-defined unification methods and expression of complete information. One chapter is devoted to flow of control specification (indexing, dependency-directed backtracking and goal freezing). Finally the last chapter is a reference manual to the package. One appendix is devoted to possible non-linguistic applications of the formalism, and compares the formalism with programming languages, in particular with PROLOG.

Note that this manual does **not** describe or document the example grammars provided as examples with the unifier. The sample grammars contain a brief documentation on-line and are accompanied by example inputs. The SURGE grammar is documented in a separate manual.

## 1.2. Function and Content of the Package

FUF implements a natural language surface generator using the theory of unification grammars (cf. bibliography for references). It follows most closely the original FUG formalism introduced in [Kay79]. 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 [Kay79]. 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 INSTALLATION 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. If you are in a hurry to try the system, the user functions are:

```
(UNI INPUT &key GRAMMAR NON-INTERACTIVE (LIMIT 10000))
  by default the grammar used is *u-grammar*
  non-interactive is nil
  limit is 10000
Complete work : unification + linearization. Outputs a sentence.
  If non-interactive is nil, a line of statistics is
  also printed.
  In any case, stops after limit backtracking points.

(UNI-FD INPUT &key GRAMMAR NON-INTERACTIVE (LIMIT 10000))
  by default the grammar used is *u-grammar*
  non-interactive is nil.
  limit is 10000
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.
In any case, stops after limit backtracking points.

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

(UNIF FD &key (GRAMMAR *u-grammar*))
  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 &key (PRINT-MESSAGES t) (PRINT-WARNINGS t))
--> T if FD is a well-formed FD.
--> nil (and error messages) otherwise.
The error messages and warnings are only printed if PRINT-MESSAGES and
PRINT-WARNINGS are true.
DO NOT USE FD-P ON GRAMMARS
```

```
(GRAMMAR-P &optional (GRAMMAR *u-grammar*)
            &key (PRINT-MESSAGES t) (PRINT-WARNINGS t))
--> 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.)
```

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.
```

### 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`.<sup>1</sup>

```
((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))))))
```

<sup>1</sup>Note that the simplest grammars presented in the manual use the standard phrase structure approach  $S \rightarrow NP VP$ . More advanced grammars use a systemic approach to language (after `gr4`). In general, the FUG formalism is convenient to write systemic grammars, but it can also be used to implement other linguistic models (PS rules, LFG, GPSG or HPSG).

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 the input). 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`. The curly-braces notation denotes what is called a ‘path’ which is a pointer within an fd. Note that in this line of the grammar, we refer to `{prot number}` even though the `{prot number}` feature does not appear under `prot` in the rest of the grammar. This is a general feature of FUF: any attribute can appear in an FD, and its value can be given either by the grammar directly where it would appear, or by the input, or by the grammar coming from a distant place and using a path.

Note also that the agreement constraint could have been written in the ‘opposite’ direction:

```
(prot ((number {verb number})))
```

Or even:

```
({prot number} {verb number})
```

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 for all “terminal” constituents.

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))))))
```

Which 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 {prot number})
        (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:

```

LISP> (uni ir01)
-->
>STARTING CAT S AT LEVEL {}

-->Entering alt TOP -- Jump indexed to branch #1: S matches input S
-->Updating (CAT NIL) with NP at level {PROT CAT}
-->Updating (CAT NIL) with NP at level {GOAL CAT}
-->Updating (CAT NIL) with VP at level {VERB CAT}
-->Enriching input with (NUMBER {PROT NUMBER}) at level {VERB}
-->Enriching input with (PATTERN (PROT VERB GOAL)) at level {}
-->Success with branch #1 S in alt TOP

>STARTING CAT NP AT LEVEL {PROT}

-->Entering alt TOP -- Jump indexed to branch #2: NP matches input NP
-->Updating (CAT NIL) with NOUN at level {PROT N CAT}
-->Enriching input with (NUMBER {PROT NUMBER}) at level {PROT N}
-->Updating (PROPER NIL) with YES at level {PROT PROPER}
-->Enriching input with (PATTERN (N)) at level {PROT}
-->Success with branch #2 NP in alt TOP

>STARTING CAT VP AT LEVEL {VERB}

-->Entering alt TOP -- Jump indexed to branch #3: VP matches input VP
-->Enriching input with (PATTERN (V DOTS)) at level {VERB}
-->Updating (CAT NIL) with VERB at level {VERB V CAT}
-->Success with branch #3 VP in alt TOP

>STARTING CAT NP AT LEVEL {GOAL}

-->Entering alt TOP -- Jump indexed to branch #2: NP matches input NP
-->Updating (CAT NIL) with NOUN at level {GOAL N CAT}
-->Enriching input with (NUMBER {GOAL NUMBER}) at level {GOAL N}
-->Updating (PROPER NIL) with YES at level {GOAL PROPER}
-->Enriching input with (PATTERN (N)) at level {GOAL}
-->Success with branch #2 NP in alt TOP

[Used 3 backtracking points - 0 wrong branches - 0 undos]
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 tree 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)))))

LISP> (uni ir02)

>STARTING CAT S AT LEVEL {}

-->Entering alt TOP -- Jump indexed to branch #1: S matches input S
-->Updating (CAT NIL) with NP at level {PROT CAT}
-->Updating (CAT NIL) with NP at level {GOAL CAT}
-->Updating (CAT NIL) with VP at level {VERB CAT}
-->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 5.6. 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."

When the fd does not contain a morphological feature, the morphology module provides reasonable defaults. More details on morphology are provided in section 6.

If a pattern contains a reference to a constituent and that 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 BENEFA are missing):
John likes.
```

Finally, if one of the constituent sent to the morphology is not a known morphological category, the morphology module can not perform 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 that there is something wrong in the grammar. You should check the list of legal morphological categories (see section 6) 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 receive on-line help on which categories are known to the morphology module.

## 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 “medium” or “affected” 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 `gr11`, (cf. file `gr11.l`), there are three stages: first the clause grammar maps from semantic roles to a level called “oblique”, and then oblique is mapped to syntactic functions such as `subject` or `adjunct`.

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.

The process can be complicated if your grammar also includes a lexicon. In this case, a good part of the output should be provided by the lexicon. Grammar `gr11` illustrates one way of including the lexicon in your grammar.





making grammars “relocatable”. For example, the grammar for NPs can be unified with a subconstituent of the input FD at different levels (`{agent}` and `{affected}` for example). In each case, a feature like `(determiner ((number {^ ^ number})))` points to the number of the appropriate constituent. Without relative paths such a general constraint could not be expressed.

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. The two features are then said to be unified. In the previous example, the features at the paths `{verb number}` and `{prot number}` are unified. This means that they are absolutely equivalent, they are two names for the same object (structure sharing). This is equivalent to the systemic operation of “conflation”.

In general, an expression of the form  $x = y$ , where either  $x$  or  $y$  is a path or a leaf is called an equation. An fd can be viewed as a flat list of equations.

In FUF, it is possible to have paths on the left of a pair. It is therefore possible to represent an fd as a list of equations as follows:

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

This notation allows to freely mix the “fds as equations” view with the “fds as structure” one.<sup>23</sup>

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:

---

<sup>2</sup>Note that the possibility to put paths on the left increases the expressive power of the `external` construct, as it becomes possible to express at run-time constraints on constituents which are not dominated by the position of the external construct in the structure.

<sup>3</sup>When using a path on the left, note that the right hand side of the equation is always interpreted as occurring in the context pointed to by the left-hand side. So if you need to use relative paths, the relative path on the right is relative to the end position of the left-hand side. For example, to unify two features `{verb syntax number}` and `{prot number}` at level `{verb v}`, you must write:

```
((verb ((v (({^ syntax number} {^ ^ prot number}))))))
```

and not:

```
((verb ((v (({^ syntax number} {^ prot number}))))))
```

because in this second equation, the path `{^ prot number}` is relative to the level `{verb syntax number}` (not `{verb v}` as intended) and therefore would end up at level `{verb syntax prot number}` instead of `{prot number}`.

```

((b ((a1 v1)))
 (a {b})
 (c {b}))

or to:

((b ((a1 v1)))
 ({a} {b})
 (c {a}))

```

The function `normalize-fd` is convenient to put an FD into its canonical form. For example:

```

(setf fd1 '((a ((a1 v1))
              (b ((b1 w1))
                  (a ((a2 v2))
                      (b ((b2 ((w2 2))))
                          (b ((b2 ((w3 3))))))))))

LISP> (normalize fd1)

((a ((a1 v1)
      (a2 v2)))
 (b ((b1 w1)
      (b2 ((w2 2)
            (w3 3))))))

```

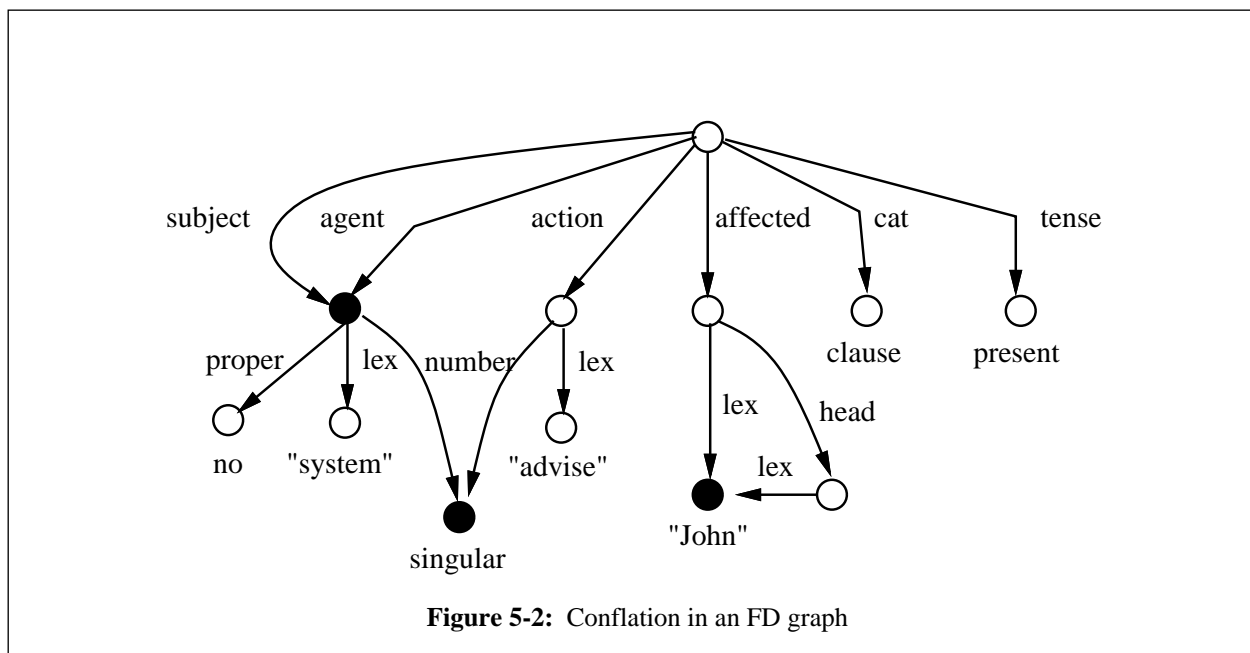
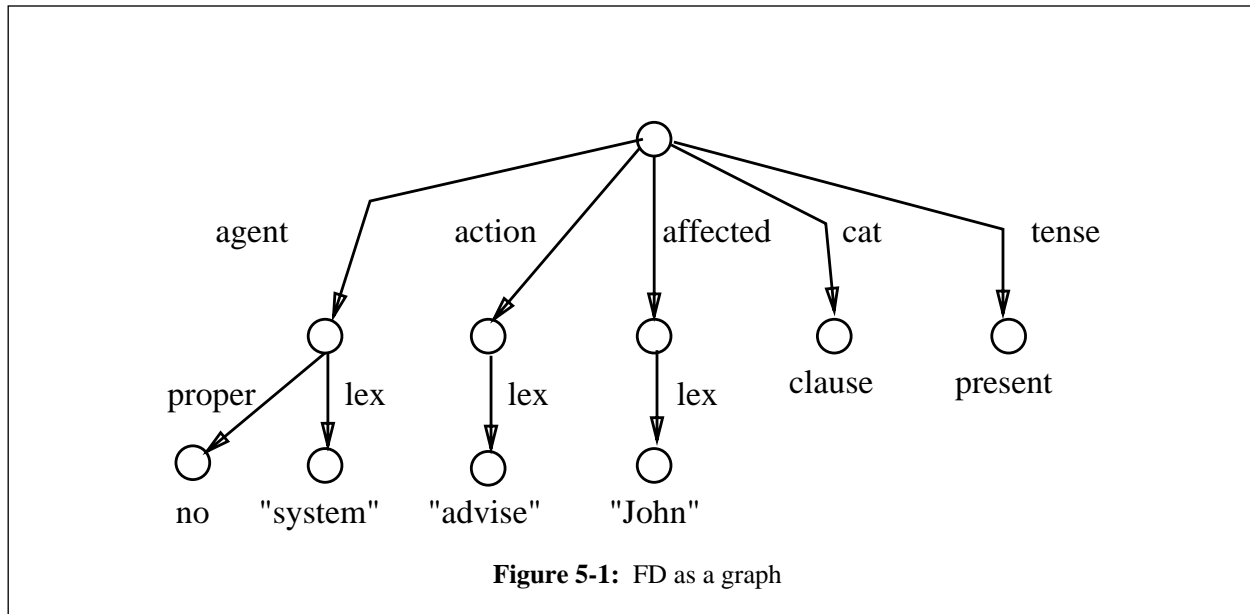
All unification functions assume that the input `fd` is given in canonical form. `normalize-fd` is particularly useful when the inputs are produced incrementally by a program. Note that `normalize-fd` will fail and return `*fail*` if the input FD is not consistent (for example `((a 1) (a 2))`).

## 5.2. FDs as Graphs

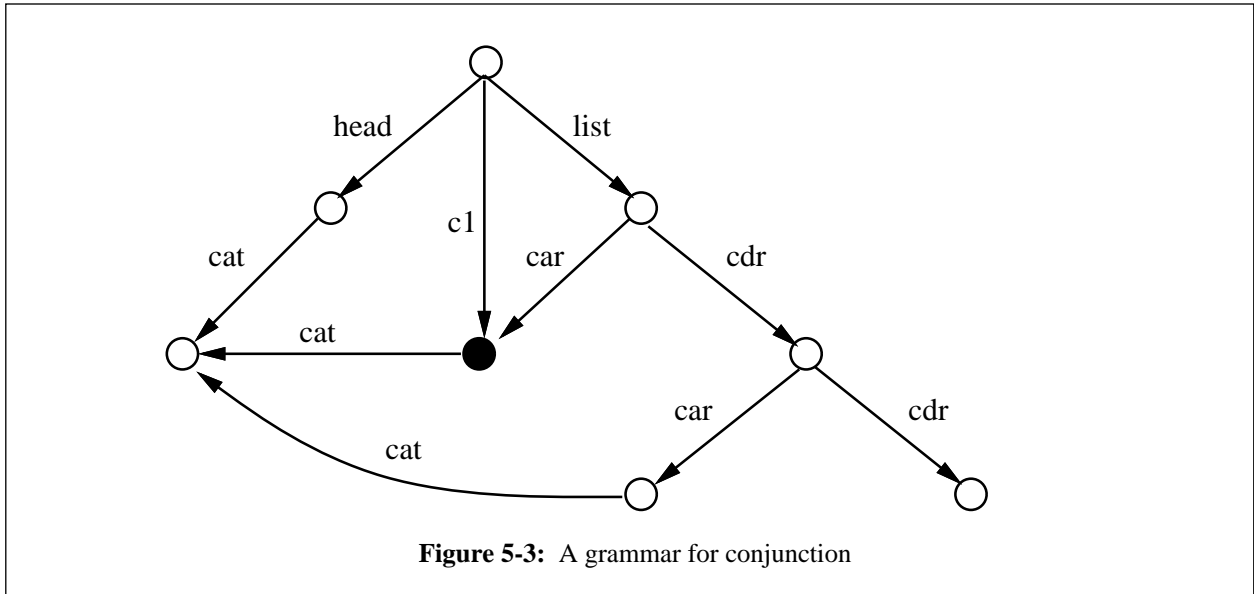
When the structure of an FD becomes complex, and more confluents with paths are introduced, a visual representation of the FD becomes extremely useful. This visual representation also provides a clear interpretation of the path mechanism and makes reading of relative path much easier. The structured format of FDs can be viewed as equivalent to a directed graph with labeled arcs as pointed out in [Karttunen-84]. The correspondence is established as follows: an FD is a node, each pair `(attr value)` is a labeled 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. The equivalence is illustrated in Fig.5-1.

The graph notation is particularly useful to interpret relative paths. When a relative path occurs somewhere in an FD, its destination can be identified by going up on the arcs, one arc for each `"^"`. When the value of a pair is a path, e.g., `(a {b})`, then the corresponding arc actually points to the same node as the given path. In this case, there is structure sharing between `a` and `b`. This configuration is illustrated in Fig.5-2, where the paths `{action number}` and `{agent number}` are conflated, as well as the paths `{affected lex}` and `{affected head lex}` and `{subject}` and `{agent}`.

The conflation of `{subject}` with `{agent}` makes all the paths that are extensions of either `agent` or `subject` equivalent. For example, `{agent lex}` and `{subject lex}` are equivalent. This equivalence is easily read in the graph notation.



The graph notation also makes it clear that the up-arrow notation can be ambiguous. Whenever a Y configuration is met in the graph, for example in the two black nodes in Fig.5-2, the up-arrow does not specify which branch of the Y must be taken. This problem is illustrated in the grammar in Fig.5-3. The FD is extracted from a grammar dealing with conjunction. The constraint enforced by the grammar is that all the conjuncts in a conjunction must have the same syntactic category. A conjunction is represented by an FD with two constituents: `head` represents the conjunction as a whole as a constituent and `list` is a list of conjuncts. The list is represented in a singly-linked list of elements, with a recursive FD containing at each level the first element of the list (feature `car`) and the rest of the list (feature `cdr`). In Fig.5-3, the path `c1` is used to point to the first constituent of the list. `c1` is therefore



defined by the equation  $\{c1\} = \{list\ car\}$ . The grammar in LISP notation is shown below along with a sample input:

```
GR = ((c1 {^ list car})
      (c1 ((cat {^ ^ head cat}))))
IN = ((head ((cat np)))
      (list ((car ((lex "cat")))
              (cdr ((car ((lex "dog")))
                      (cdr none))))))
```

The underlined line corresponds to the black dot in the graph notation shown in Fig.5-3. The problem is to interpret where the relative path  $\{^ \ ^\ head\ cat\}$  is pointing to. The notation is ambiguous between  $\{head\ cat\}$  and  $\{list\ head\ cat\}$ , depending on whether one considers the black dot as being located at address  $\{c1\}$  or  $\{list\ car\}$ . This ambiguity is solved in FUF by following the convention that up-arrows always refer to the textual location where they appear in the grammar. So in this example, the up-arrows refer to the address  $\{c1\ cat\}$  and not to the address  $\{list\ car\ cat\}$  because they are written as a pair  $(c1\ ((cat\ \{^ \ ^\ head\ cat\})))$  and not as  $(list\ ((car\ ((cat\ \{^ \ ^\ head\ cat\}))))$ .

There are special attributes and values which cannot be drawn in this graph notation because they have a special unification behavior. These are, for attributes: `alt`, `opt`, `ralt`, `pattern`, `cset`, `fset`, `test`, `control` and `cat` (or the currently specified `cat` attribute) and for values: `none`, `any` and `given`. The special constructs `\#(under x)` and `\#(external y)` have also a special meaning for the unifier. These are all the “keywords” known by the unifier. They are presented in the following sections.

### 5.3. Functional Descriptions vs. First-order Terms

To conclude the characterization of FDs as a data-structure, it is useful to contrast functional unification (FU) with the more well known structural unification (SU) as used in PROLOG for example, and to distinguish FDs from the first-order terms used in SU.

The most important difference is that functional unification is not based on order and length. Therefore,  $\{a:1, b:2\}$  and  $\{b:2, a:1\}$  are equivalent in FU but not in SU, and  $\{a:1\}$  and  $\{b:2, a:1\}$  are compatible in FU but not in SU (FDs have no fixed arity). The following quote from Knight summarizes the distinction between feature structures and the first order terms used in SU:

- Substructures are labeled symbolically, not inferred by argument position.
- Fixed arity is not required.
- The distinction between function and argument is removed.
- Variables and coreference are treated separately. [Knight, p.105]

A comparison between the FD notation and the first-order term notation illustrates these differences. The following FD and first-order term can be used to represent the fact that *Steve builds a crane that is 2 lbs and 4 feet high*:

```

((process build)
 (agent Steve)
 (object ((concept crane)
          (weight 2)
          (height 4))))

build(Steve, Crane(C1, 2, 4))4

```

Contrasting these two notations for the same example illustrates the differences:

- *Symbolic labels for substructures*: the arguments, that is the agent and the medium, are clearly labeled in the feature structure notation. In particular, a term like `Crane(C1, 2, 4)` is particularly difficult for a human to interpret.
- *Fixed arity*: features can be added at will to an FD. FDs are used to represent *partial information*. This is not the case for first-order terms. If the knowledge representation changes to include width to crane descriptions, in addition to weight and height, all the terms need to be updated, since `Crane(n, w, h)` is not compatible with `Crane(n, w, h, l)`. The FD notation is always partial and leaves the possibility of adding new features as needed.
- *Function and argument*: first-order terms have a head (the function) which plays a central role in the unification process. This is not the case in FDs. All information plays the same role.<sup>5</sup>
- *Variables and coreference*: in standard unification, a variable is used to mean two distinct things: that the value of the role is unknown (there are no constraints on it), and that the value of the role is the same as all other objects referred to with the same variable. So for example, in a term such as `like(X, X)`, the use of the variable `X` means that we don't know who likes whom, and that it is known that the agent

---

<sup>4</sup>Other representations are of course possible using first-order notation. Some of them have some of the advantages of features structures. For example: `build(B1)`, `agent(B1, Steve)`, `object(B1, C1)`, `crane(C1)`, `weight(C1, 2)`, `height(C1, 4)`. In fact, any FD can always be translated in a one-to-one mapping to a class of restricted first-order terms.

<sup>5</sup>In most cases, however, one feature plays a special role: for example, the `cat` attribute can specify the category or type of a description, but this is not built into the syntax, and several such "special" attributes can coexist in the same FD, allowing a reader to adopt several perspectives on the same FD.

and the object of `like` must be the same objects. The distinction between these two functions of variables is best explained in [Ait-kaci]. In FDs, coreference and unspecification are represented by two different syntactic devices: variables are features which are unspecified (they simply do not appear in the FD, or appear with an empty value), coreference is handled with *paths*. For example, to express the constraint that the medium of `like` must refer to the same object as its agent, the following FD can be used:

```
((process like)
 (agent {object}))
```

## 5.4. Disjunctions: The ALT and RALT Keywords

`alt` stands for “alternation”. The syntax for using `alt` is:

```
((att1 val1)
 (att2 val2)
 ...
 (ALT {annotations*} (fd1 fd2 ... fdn))
 ...
 (attn valn))
```

The meaning of a pair with an `alt` attribute is the following: the unifier tries 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)
 ...)))
```

When the order is truly not relevant and there is no reason to choose a default branch, then you can use the `ralt` keyword instead of `alt`. `ralt` has exactly the same syntax as `alt` and also expresses a disjunction, but the unifier will choose one of the branches at random instead of always trying the first untried branch. (`ralt` stands for “random alt”)

Alternatively, the `:order` annotation can be used to specify whether the branches should be order in random or sequential order. The syntax is as follows:

```
(alt (:order :sequential)      is equivalent to (alt (fd1...fdn))
 (fd1 ... fdn))

(alt (:order :random)          is equivalent to (ralt (fd1...fdn))
 (fd1 ... fdn))
```

An `alt` can be embedded within another `alt` or it can be the value of a feature as in:

```
((a (alt (1 2 3 4))))
```

## 5.5. 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))
```

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

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

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.

NOTE: because of the presence of `dots` and `pound`, the unification of patterns is a non-deterministic operation. It can produce several results for a given input, and there is no way to predict 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).

The following example illustrates the fact that pattern unification is non-deterministic in general:

```
Pattern Unification:
p1: (pattern (dots a dots b dots))
p2: (pattern (dots c dots d dots))

Compatible Results:
(pattern (dots a dots b dots c dots d dots))
(pattern (dots a dots c dots b dots d dots))
(pattern (dots a dots c dots d dots b dots))
(pattern (dots c dots a dots b dots d dots))
(pattern (dots c dots a dots d dots b dots))
(pattern (dots c dots d dots a dots b dots))

Pattern Unification:
p3: (pattern (dots a dots b))
p4: (pattern (dots b c))
Pattern Unification fails.
```

Patterns are eventually interpreted by the linearization component to produce a string out of an FD.

Appendix ADVANCED describes some advanced uses of pattern unification.

## 5.7. Explicit Specification of Sub-constituents: the CSET Keyword

The unifier works top-down 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 unifier 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 correspond 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 (unordered). Therefore the following 2 pairs are correctly unified:

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

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

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

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

FUF does not rely exclusively on `csets` to find the constituents to be recursively unified. FUF generally tries to infer the value of `cset` from the value of `pattern` and an observation of the features of the current FD (with the assumption that features containing a `cat` attribute are constituents). The exact procedure followed to identify the implicit constituent set of an fd is:

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, explicit `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. For larger grammars, however, you should put the emphasis on a clean constituent structure, and therefore you should carefully use the explicit CSET facilities instead of blindly relying on FUF's inferencing. In this case, the advanced CSET facilities described below will prove helpful.

### 5.7.1. Implicit and Incremental CSET Specification

CSET specification is the means by which a programmer describes the constituent structure of sentences. Given the importance of this task and its complexity, facilities have been added to FUF to make CSET specification more flexible. It is thus possible to specify *implicit* and *incremental* CSETs. This section describes these features of the CSET specification. It presents advanced facilities and can be skipped in first lecture.

The general form of a CSET specification is:

```
(CSET (= c1 ... cn)
      (== b1 ... bm)
      (+ a1 ... ap)
      (- d1 ... dq))
```

The simple syntax (CSET (c1 ... cn)) is equivalent to (CSET (= c1 ... cn)).

As usual in FDs, all of the features of the CSET are optional. Each feature in the CSET feature is named as follows:

- The = sublist is called the absolute CSET.
- The == sublist is called the basis CSET.
- The + and - sublists are the increment CSETs.

The idea behind the use of incremental CSETs is to gradually refine the CSET description, by adding partial information - thus folding the constituent structure description into the general “partial information, gradual refinement” methodology of FD specification. The incremental CSET specifications are added to the basis CSET, which can either be specified explicitly, using the == notation, or be the result of the implicit CSET inference described above.

The actual CSET of an FD is computed by applying the following procedure:

1. If the absolute CSET (=) is present, it becomes the value of the actual CSET. The absolute CSET feature is used when you want to disable all incremental computations.
2. Else: If the basis-CSET is present,
  - a. let BSET = basis-CSET, else let BSET = the implicit CSET, computed as described above (from pattern and cat inspection).
  - b. If the +increment-CSET is present, let BSET = BSET union +increment.
  - c. If the -increment-CSET is present, let BSET = BSET - -increment (set difference).
  - d. Return BSET as the actual CSET.

The result of this procedure is a list of absolute paths, pointing to the sub-constituents of the current constituent. In all cases, this actual cset is “cleaned up” by applying the following procedure:

- All duplicate paths are removed. Two paths are duplicate if they point to the same FD. That is, even if the paths are distincts but they point to conflated FDs, they are considered duplicates in the CSET.
- All leaf-constituents are removed. A leaf-constituent is a constituent whose value is not a list of pairs - for example, a symbol or a string. The idea is that there is no point in recursing on a leaf-constituent.
- All constituents which are conflated with a -increment constituent are also removed from the actual cset.

The following grammar fragments illustrate briefly the use of incremental CSET specification.<sup>6</sup>

```
((cat clause)
 (cset (== prot verb))
 (alt ((prot given)
      (subject {^ prot})
      (voice active))
      ((prot none)
      (goal given)
      (cset (- prot) (+ goal))
      (subject {^ goal}))))))
```

The first cset feature specifies that the basis cset for this clause constituent should *not* be the implicit cset (derived from pattern and cat), but instead the explicit list (prot verb). In the second cset feature, this basis cset is incrementally refined by specifying that when prot is none, prot is not part of the cset and instead goal is added to the cset. If prot is a non-leaf constituent, then the actual cset for this example will be (prot verb), if it is none, then it will be (verb goal).

```
((cat clause)
 (subject ((cat np)))
 (prot {^ subject})
 (cset (- prot)))
```

In this second fragment, no absolute cset is specified (no = feature) and no basis cset is present (no == feature). Therefore the implicit cset is computed and is found to be (subject) (because of the presence of the (cat np) feature under subject). (subject) is thus the initial basis-cset. The incremental cset (- prot) is then added to the cset. The basis-cset remains (subject) at this point. Finally, the cset is “cleaned” - and it is found that prot and subject are duplicates of each other because of the conflation (prot {^ subject}). Since prot is member of the -increment list, its “synonyms” are deleted from the basis cset, and the actual cset is found to be empty ().

The computation of implicit and incremental CSETs can interact with the more advanced control features of goal freezing (with the `wait` construct). These interactions are described in Sect. WAIT-CSET.

### 5.7.2. Unification of Incremental CSET Specifications

When two complete CSET specifications are unified, the following rules are applied: assume that (cset (= a1) (== b1) (+ c1) (- d1)) is unified with (cset (= a2) (== b2) (+ c2) (- d2)),

1. If both a1 and a2 are instantiated:
  - a. If (set-equal a1 a2) then:
    - i. If c1 union c2 is included in a1, return (cset (= a1)) Else return :fail.
    - ii. If d1 union d2 intersect with a1, return :fail Else return (cset (= a1))
  - b. else :fail.
2. If only a1 is instantiated and a2 is null:
  - a. If a1 intersects d1 union d2, then :fail.
  - b. Else (cset (= a1)) is returned.

---

<sup>6</sup>Keep in mind that this facility is above all designed for use in large grammars, and the short fragments shown here could easily be handled without incremental CSETs.

3. If neither `a1` and `a2` is instantiated, compute the unions: `b=b1 u b2`, `c=c1 u c2`, `d = d1 u d2`, and return `(cset (== b) (+ c) (- d))`.

## 5.8. 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`.

## 5.9. 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.
- tests that no frozen constraint is left.

It is important to realize that none of this can be done before the unification is finished. Section `WAIT-CSET` gives a more complete picture of the determination process and explains why there may be a need for several cycles of determination when `wait` and goal freezing are used.

Note that in practice, `ANY` is used rarely. The next special value `GIVEN` is used more often, and is easier to manipulate, except in cases where goal freezing is used.

## 5.10. The Special Value GIVEN

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` and, conversely, when you use `wait` (for goal freezing) and something does not work, it is because you should use `any` instead of `given`.

`Given` addresses the most obvious limitation of the top-down regime used by FUF when traversing the

constituent tree (and computing the cset expansion) in cases when the grammar contains the equivalent of left-recursive rules. `given` is in a sense the dual of the any meta-variable of the original FUG formalism: while any requires a feature to be instantiated at the *end* of the unification process, `given` requires it to be instantiated *before* unification starts. Thus, `given` is used to check that a constraint has been specified in the input. A `given` feature can only appear in a grammar, thus, `given` gives a different status to the two arguments of the unification function.<sup>7</sup>

```
((cat NP)
  (semr ((possessor GIVEN)))
  (cset (det head))
  (det ((cat NP)
        (possessive yes)
        (semr {^ ^ possessor})))
  ...)
```

To illustrate how `given` solves the problem of the left-recursive rules, consider how the possessive NP rule is implemented in FUF in this example. This FD implements the FUF equivalent of a rule:

```
NP -> NP(possessive) head
```

This is a left-recursive rule which, if used in a top-down control, could lead to infinite recursion of the form NP -> NP (NP (NP (NP ..... (NP head...))) Note how the cset expansion in FUF is the equivalent of the rule application in rule-based grammatical formalisms.

To avoid using this rule when there is no possessor in the input (which is the step which leads to non-termination), the grammar contains a feature (`possessor GIVEN`), checking that the semantic representation of this constituent does indeed have a possessor specified in the input. If none is specified, then the rule will fail, and the sub-constituent possessive NP will not get created. Without the use of `GIVEN`, this FD would always be successfully unified, and the cset expansion would lead to non-termination. The use of the `given` meta-variable therefore ensures that the top-down regime of FUF is goal-directed.

NOTE: The `under` construct is related to the `given` value. It is presented in Section FEATURE-TYPE.

## 5.11. The Special Attribute CAT: General Outline of a Grammar

Each constituent of an FD is generally characterized by its ‘category’. In FD terms, this 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 ef a ru r8 Vr8 forNOUN.: The Outline of a grammarmluso b:d

```
((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 SECT-NON-STANDARD for a list of the non-standard features of this implementation.

NOTE: The symbol identifying categories ('cat) can be changed in the program. It is by default 'cat, but this default can be changed by setting a new value to the `*cat-attribute*` variable or by providing an optional argument to the unification functions, as explained in the reference manual.



## 6. 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.

### 6.1. 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 must call the function `register-category-not-unified`. To find out the list of non-unified categories, call the function `categories-not-unified`.

```
CATEGORIES-NOT-UNIFIED (&optional (cat-attribute *cat-attribute*))
REGISTER-CATEGORY-NOT-UNIFIED (cat &optional (cat-attribute *cat-attribute*))
```

Note that this information depends on the `cat-attribute`, which by default is `cat` (cf. page 27).

### 6.2. 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 LEXICON.L
PRONOUN:
  agreement done on pronoun-type, case, gender, number,
  distance, person.
  irregular pronouns are defined in file LEXICON.L
VERB:
  agreement is done on number, person, tense and ending.
  irregular verbs must be put in the list *IRREG-VERBS*
  in file LEXICON.L
DET :
  agreement is done on number, definite and first letter of
  following word for "a"/"an" or feature a-an of following word.
ORDINAL, CARDINAL:
  string is determined using value and digit to determine whether
  to use digits or letters.
```

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

### 6.3. Accepted Features for VERB, NOUN, PRONOUN, DET, ORDINAL, CARDINAL 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 : {";", " ", ":", "(", ")", "...}
ORDINAL, CARDINAL:	VALUE: a number (integer or float, positive or negative)
	DIGIT: {YES, NO}

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.

### 6.4. Possible Values for Features NUMBER, PERSON, TENSE, ENDING, BEFORE, AFTER, CASE, GENDER, PERSON, DISTANCE, PRONOUN-TYPE, A-AN, DIGIT and VALUE

```

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.

DIGIT:  {YES, NO}
        Default is YES.

VALUE:  a number.

```

## 6.5. 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 (initialize-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-tuples of the form: (root present-third-person-singular past present-participle past-participle)

The default value starts as follows:

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

## 6.6. Linearization and Punctuation

The linearizer interprets the *pattern* ordering constraints and assembles the words of the sentence into a linear string. In addition, the linearizer deals with punctuation and capitalization. The general algorithm followed by the linearizer is:

1. If a feature gap is found, the linearization of the FD is the empty string.
2. Else: Identify the *pattern* feature in the current FD. If a pattern is found:
  - a. For each constituent of the pattern, recursively linearize the constituent.
  - b. The linearization of the fd is the concatenation of the linearizations of the constituents in the order prescribed by the pattern feature. (Note that during linearization, dots and pounds are ignored in the pattern.)
3. If no feature pattern and a feature lex 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 .
  - c. Identify the feature punctuation. Punctuation can contain three sub-features: *before*, *after* and *capitalize*. According to the value of these features, append or prepend punctuation to the string computed by the morphology module, and capitalize the string if requested. The linearization of the fd is the resulting string.
  - d. If the current cat is not known by the morphology system, the linearization of the constituent is a string of the form <unknown cat X: S>.
4. If no feature pattern and no feature lex is found, the linearization of the current fd is the empty string.

The linearizer also deals with inserting sequences of punctuation signs, insertion of spaces between words and the ‘liaison’ article ‘an’ as in ‘an interesting case’ or ‘an RPS’. The following rules are implemented:

1. A space is inserted between each pair of words except when:
  - a. Do not put space after an opening bracket “([“
  - b. Do not put space before a closing bracket “)]]“
  - c. Do not put space before punctuations.

2. When the indefinite singular article (“a”) is followed either by a word that starts with a vowel or by a word whose FD contains the feature (a-an yes), the form “an” is used instead of “a”. For example, if a word is produced from the FD ((lex "RPS") (a-an yes)), the string “an RPS” will be produced.
3. The first word of the string produced by the linearizer is capitalized.
4. If an FD contains the feature (punctuation ((capitalize yes))), the string produced by the linearizer is capitalized. If the value of capitalize is no, the string is not capitalized, even if it starts the sentence.
5. If an FD contains the feature (punctuation ((before ",") (after ","))), the string produced by the linearizer starts and ends with a comma. Any string can be specified in this feature.
6. Leading punctuations are removed from the final string. (There are 6 punctuation signs ,:;!?)
7. A final period (.) is added to sentence if it does not already end with a punctuation. If the mood of the sentence is a specialization of interrogative, then a final question mark (?) is added.
8. Sequences of punctuations are filtered according to the following rules:

<p>a. , , -&gt; ,</p> <p>  , .        .</p> <p>  , ;        ;</p> <p>  , :        :</p> <p>  , !        !</p> <p>  , ?        ?</p>	<p>d. : ,        :</p> <p>  : .        .</p> <p>  : ;        ;</p> <p>  : :        :</p> <p>  : !        !</p> <p>  : ?        ?</p>
<p>b. . ,        . ,</p> <p>  . .        .</p> <p>  . ;        . ;</p> <p>  . :        . :</p> <p>  . !        . !</p> <p>  . ?        . ?</p>	<p>e. ! ,        ! ,</p> <p>  ! .        ! ,</p> <p>  ! ;        ! ;</p> <p>  ! :        ! :</p> <p>  ! !        ! !</p> <p>  ! ?        ! ?</p>
<p>c. ; ,        ;</p> <p>  ; .        .</p> <p>  ; ;        ;</p> <p>  ; :        :</p> <p>  ; !        !</p> <p>  ; ?        ?</p>	<p>f. ? ,        ? ,</p> <p>  ? .        ?</p> <p>  ? ;        ?</p> <p>  ? :        ?</p> <p>  ? !        ?</p> <p>  ? ?        ?</p>



## 7. Tracing and Debugging

### 7.1. What it Means to Debug a FUF Program

When using FUF, a grammar developer is programming in the FUF programming language. The grammar is a program. The input FD is the input to the program. FUF is the interpreter and sentences are the output of the execution of a grammar on inputs. In this framework, when something “goes wrong,” the grammar developer must debug his grammar. The main sources of bugs found when developing FUF programs are:

- The input is not well formed (it is not a valid FD).
- The grammar is not well formed (it is not a valid FD).
- The input does not have the structure expected by the grammar (too flat or too deeply nested).
- The constituent structure is badly specified in the grammar (causing either too many constituents to be generated, or not enough, or infinite recursion in the constituent traversal).
- The ordering patterns are not correctly specified or two pattern specifications do not unify.
- The appropriate morphological features are not passed to the leaf-constituents, preventing the morphology module from performing the required inflections.
- Relative paths are not pointing to the place they were intended to. An ambiguous relative path is not correctly resolved.
- Given/any fail when they should not: given can interact poorly with wait, and should be replaced by any, any can cause heavy backtracking and should be replaced by given.
- Types are not defined as expected, or there is an unwanted interaction between types.
- FSET declarations are too rigorous.
- Wait and/or bk-class are not used correctly.
- An indexed feature is not found in an indexed alt.

The main problem when debugging a FUF program is to identify what caused a failure to occur. We introduce the following terminology to discuss the various debugging tools available in FUF: a failure occurs whenever the unifier attempts to unify a simple leaf symbol with a different, incompatible leaf. Failures trigger backtracking. An unexpected failure is a failure that the programmer did not expect. The initial failure is the first unexpected failure to occur during an unification. The distinction between regular failure and unexpected failure is of course subjective, but it is useful, because there are usually many failures that occur even if there are no bugs in the grammar. This happens whenever a non-indexed alt is traversed. Branches are tried in order until the appropriate branch is found. When failure occurs during the test of the initial branches, this is not the trace that “something is going wrong”. So the main problem of the FUF debugger is to, as quickly as possible, identify the initial failure - and to filter out all the irrelevant expected failures.

This task is made difficult because FUF backtracks a lot, and if the initial failure is missed, a lot of subsequent “unexpected failures” will occur when wrong branches are tried upon backtracking (in a sense, everything that happens after the initial failure should be disregarded, as the unifier is engaging on a wrong course).

This chapter presents the tracing tools available in FUF and provides advice on how to use them to quickly identify the initial failure.

## 7.2. Checking the Validity of FDs and Grammars

Before tracing the unification, it is important to check that both the input and the grammar are valid FDs. The following functions perform this checking:

```
(FD-SYNTAX fd): check that a Lisp expression FD is a well-formed FD.

(FD-P fd): check that a well-formed FD does not contain inconsistencies
(that is, (u fd nil) is not :fail, or in other terms, the FD
does not contain contradictory features, such as ((a 1) (a 2))).

(GRAMMAR-P grammar): check that a Lisp expression grammar is a well-formed
grammar.
```

When you suspect that your current input/grammar do not work, check first that they are syntactically valid using these functions.

## 7.3. Fine Tuning Tracing: Overview of FUF Tracing Functions

Several dimensions characterize the activity of the FUF unifier:

- Where in the *input* is the unification proceeding.
- Where in the *grammar* is the unification proceeding.
- How *important* is the current action of the unifier.
- What *stage* of the unification is proceeding.

It is possible to tailor the tracing behavior of FUF according to each of these dimensions. In general, FUF can output a tracing message whenever it takes an action, such as adding a feature to the total FD, selecting a branch in the grammar, backtracking because of a failure, expanding a cset and moving in the constituent tree traversal, freezing and unfreezing goals etc. Outputting all the possible trace messages is always overwhelming, and provides too much text to be useful. So the challenge of the FUF debugger is to fine-tune the tracing system to produce just enough information to locate the bugs in the grammar and/or in the input FD.

The grammar developer indicates what portions of the grammar must be traced: the grammar is traced, not the unifier. 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 following functions:

```

I GENERAL TRACING CONTROL

(trace-on) enable all trace messages to be output.
(trace-off) disable all trace messages to be output
(trace-bp &optional (frequency 10)):
    Output a dot for every frequency backtracking points.
    Useful for long computations to get a feeling of what's happening.
    Works even if (trace-off).
%break% allows the insertion of break points in the grammar.
(trace-level level)
    Determines detail level of trace to be printed. The following
    levels are defined:
        00: feature level action
        05: unimportant alt-level action
        10: alt-level action - branch number trying
        12: demo messages
        15: freeze, ignore, bk-class
        20: constituent level action
        30: important failure - end of alt

II TRACING FLAGS MANAGEMENT: ENABLE & DISABLE

(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-enable-alt alt-name :expansion t :grammar g)
(trace-disable-alt alt-name ...)
    Enable all tracing flags defined under a def-alt or
    def-conj. If expansion is nil, does not expand the
    sub-def-alt.
(trace-disable-match string)
    disable all flags whose names contain string.
(trace-enable-match string)
    re-enable all flags whose names contain string.

III CONTROL OF SPECIFIC ACTIVITY TRACING

(trace-determine :on t|nil) enable tracing of determine stage or not.
(trace-category :all|cat|(cat1...catn) t|nil) enable tracing of
    categories or not.
(trace-bk-class t|nil) list special messages concerning bk-class
(trace-wait :on t|nil) list special messages concerning goal freezing
(trace-cset :on t|nil) trace cset expansion
(trace-alts :on t|nil) detailed tracing of all alts, even unnamed.
(hyper-trace-category cat :status t|nil)
    trace category with printing of full constituent
    before unification of constituents of the traced cats

```

## 7.4. Identifying Possible Bugs: Trace-bp

To identify whether FUF enters into unbounded backtracking (which is the sign of a bug in the grammar or in the input), it is useful to first monitor how many backtracking points are being used. This can be achieved by using the `trace-bp` (trace backtracking points) function. This is the only tracing function which can be activated even if the general tracing system is turned off (with the function `trace-off`). The effect of the function is to output a dot on the screen every `n` backtracking points, where `n` is 10 by default and can be changed by giving an argument to `trace-bp`. The form of the function is:

```
(TRACE-BP &optional n)
(TRACE-BP nil): turns off printing of .
```

If the string of dots becomes longer than expected, then it is worth stopping the unifier under the suspicion that something is going wrong, and to start the tracing system with `trace-on`.

## 7.5. Levels of Tracing

In the attempt to reduce the amount of trace messages and finding the original source of failure, the most useful tool is the `trace-level` function, which filters tracing messages according to their importance. The following levels of importance are predefined:

- 30: important failure - end of alt
- 20: constituent level action
- 15: freeze, ignore, `bk-class`
- 12: demo messages
- 10: alt-level action - branch number trying
- 05: unimportant alt-level action
- 00: feature level action

The function `trace-level` sets the minimum level of tracing messages that can be printed. Thus, calling `(trace-level 20)` insures that only important messages of level 20 and 30 will be printed.

The levels are defined as follows:

- 30: An “important” failure in unification occurs - that is, all the branches of an alt have been tried and failed. This in general means that the input is not compatible with the grammar. It does not force immediate failure of the overall unification process, because some backtracking options may still be open, but it is a strong indication that something wrong is going on. In general, there is a high probability that the *first* level 30 failure message is the initial failure (cf. p.35).
- 20: Constituent Level Action - traces the constituent tree traversal of FUF. Whenever the grammar is re-accessed to unify a new constituent, a tracing message is printed. This is useful to follow the progression of the unifier through the total FD. In general, the traversal is a top-down breadth-first expansion of the constituent tree.
- 15: Control messages: The dominant control strategy of FUF is the top-down constituent traversal. Fine-tuning of this strategy is, however, possible, using the `bk-class`, `wait` and `ignore` directives. These constructs are described in more detail in Section CONTROL.
- 12: Demo messages: The alt construct allows the grammar writer to output “demo messages” when an alt is entered. These messages are considered of level 12.
- 10: Alt-level action - branch number trying: when an alt is tried, branches are tried successively, in order, randomly (for a `ralt`) or directly (for an indexed alt). A tracing message is output each time a branch is tried, indicating the name of the alt (therefore the position of the unifier within the grammar) and the number of the branch within the alt.
- 05: unimportant alt-level action: when an alt is indexed, certain tracing messages are printed to document the index search.
- 00: feature level action: each time a feature is added to the total FD by the unifier, a tracing message is

printed. This is the absolute lowest level of tracing and results in unending seas of messages.

In general, when debugging, start by setting (trace-level 30), this provides most of the time directly the location of the initial failure.

## 7.6. Tracing of Alternatives and Options

To follow the unifier as it proceeds through the grammar, the most useful trace is generated by giving a name to an alternative of the grammar. It is done by adding an atomic name after the keywords `alt`, `ralt` 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 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:
  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 l
  FAIL IN TRYING s with s AT LEVEL l
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 l
  s BECOMES A POINTER TO s AT LEVEL l
  UPDATING BOTH PATHS TO A BOUND

```

## 7.7. 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

```

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

### 7.7.1. Special Flags `%trace-on%` and `%trace-off%`

The special tracing flags `%trace-on%` and `%trace-off%` are predefined and have the effect of temporarily turning all tracing on and off. They are used as all other local tracing flags, as shown in the following example:

```

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

    ;; branch 2 of alt passive
    ((verb ((voice passive))

      %trace-off%          ;; Turn off tracing of details in this branch

      (prot any)
      (prot ((cat np)))
      (by-obj ((cat pp) (prep ((lex "by")) (np (^ prot))))
      (pattern (dots verb by-obj dots))

      %trace-on%          ;; Turn tracing back on

    )))

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

```

`%Trace-off%` is generally used to remove tracing information from a region of the grammar that has already been debugged but still leaving the alt traversal tracing messages on. The same effect is most of the time achieved by using the `trace-level` function, but `%trace-off%` allows finer tuning of the tracing behavior of FUF.

### 7.7.2. The Special Tracing Flag `%break%`

The special tracing flag `%break%` is used to trigger a break in the execution of FUF. When found by FUF, a Lisp continuable break is triggered. It is then possible to examine the current state of the unification using the Lisp debugger. Using the Lisp debugger, it is then possible to either resume unification or abort it. Within the debugger, the total fd can be inspected (and modified) by using the function `path-value` and `set-path-value`. A typical session is shown below:

```

(setq *u-grammar* '((alt ((cat clause) (pattern (s v o))
                        (s ((cat np)))
                        (v ((cat v)))
                        (o ((cat np))))
                      ((cat np) %break%)
                      ((cat v) %break%))))

;; The np and v branch are not yet written - a break is inserted
;; The developer can then insert new values manually during unification.

LISP> (uni '((cat clause)
           (s ((lex "John"))
              (v ((lex "like"))
                  (o ((lex "Mary")))))

>

>=====
>STARTING CAT CLAUSE AT LEVEL {}
>=====

>Expanding constituent {} into cset ({0} {V} {S}).
>

>=====
>STARTING CAT NP AT LEVEL {0}
>=====

Break: Break in grammar

Restart actions (select using :continue):
 0: return from break.
[1c] FUG5 23> (path-value {o})

((LEX "Mary") (CAT NP))
[1c] FUG5 24> (path-value {n})

((LEX "John") (CAT NP))

[1c] FUG5 25> :cont

>Constituent {0} is a leaf.
>

>=====
>STARTING CAT V AT LEVEL {V}
>=====

Break: Break in grammar

Restart actions (select using :continue):
 0: return from break.
[1c] FUG5 26> :reset

FUG5 27>

```

The behavior of the debugger depends on which version of Common Lisp you are using. The example shown

here is run under Franz Inc's Allegro Common Lisp. Consult your Lisp manual to find out the details of how to resume processing (the `:cont` command in ACL) and abort processing (the `:res` command in ACL). Another source of variation is whether the `{}` notation is recognized within the debugger or not. This notation is implemented using macro characters in Lisp. Macro-characters are recognized in ACL's debugger but not in Lucid Common Lisp's implementation. For that reason, the function `path-value` accepts as parameter either a path or simply a list of attributes.

The function `path-value` returns the value of a path in the current total FD. It is useful to inspect the current value of the total FD. The function `set-path-value` is also defined to change a value within the total FD. Note that its use is highly dangerous.

```
(path-value path-or-list) Return the value of path in the current total FD.

(set-path-value path-or-list FD) Set the value of a path in the current
                                total FD to FD.

Examples:

(path-value {process v})
(path-value '(process v)) ;; equivalent form when the {} notation is not
                          ;; recognized

(set-path-value {process v} '((lex "take")))

(set-path-value {process v} (u (path-value {process v}) '((tense past))))
                          ;; u performs a simple unification between fds.
```

For more general access, the total FD is accessible in the special variable `*input*` and it can be modified in any possible ways. But if you do follow this way, there is a high probability that the unification will not be able to proceed normally. Note that there is no way to easily remove a conflation from the total FD using only `path-value` and `set-path-value` because `path-value` always follows the paths until a non-path value can be returned. The following example illustrates this limitation:

```
(setq *input* '((a {b})
               (b ((b1 1)))))

(path-value {a}) --> ((b1 1))

(set-path-value {a} '((b1 2)))

(path-value {}) --> ((a {b}) (b ((b1 2))))

;; Cannot just with path-value and set-path-value remove the conflation
;; between a and b (a {b}).
```

One last word of caution: when using `set-path-value`, relative paths are made absolute before being inserted relative to the path of insertion given as parameter. Since the relative path notation can be ambiguous (cf p.15), this can, under circumstances where there is an ambiguity, create unexpected results.

## 7.8. 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.

When you use the `def-alt` and `def-conj` notation (cf. Section DEF-ALT), the functions `trace-enable-alt` and `trace-disable-alt` can be used to enable and disable all the tracing flags appearing under a given `def-alt` or `def-conj` construct. The `expansion` keyword determines if the enabling/disabling only concerns the tracing flags appearing directly in the `def-alt` construct, or also all the flags appearing in the expansion of the construct.

## 7.9. The :demo directive

Reading traces from the unifier is a particularly tedious task. The main problem is that the messages generated by the program are very similar to each other. The `:demo` directive allows the grammar writer to bring some variety to these messages. A demo-message can be used to output a specific message during the trace of the program when entering an `alt` (or a `ralt`) construct. The syntax is indicated by the following figure:

```
(alt voice (:demo "Is the voice active ~
                or passive?")
  (((voice active)
    ...)
   ((voice passive)
    ...)))
```

The message will be printed in the trace of the program (only if the tracing flag `voice` is enabled) as shown:

```
--> Entering alt VOICE
     Is the voice active or passive?
--> Trying branch #1
     ...
```

Note that the message is indented in the stream of trace messages. Such messages allow the grammar writer to put some semantic information into the trace messages, so that the whole stream of messages can be more easily interpreted.

In addition to its position at the top of an `alt` construct, a demo-message can be embedded anywhere in a grammar by using the `control-demo` function. `Control-demo` must be used within a `control` pair and produces an indented demo message in the trace stream. The following example illustrates its use:

control-demo

```
(alt from-loc (:demo "Is there a from-loc role?")
  ((from-loc none)
   %TRACE-OFF%
   (control (control-demo "No from-loc")))
   %TRACE-ON%)
  ((from-loc given)
   %TRACE-OFF%
   (control (control-demo "From-loc is here")))
   %TRACE-ON%))
```

Tracing output:

```
--> Entering alt FROM-LOC
    Is there a from-loc role?
--> Fail with branch #1
--> Entering branch #2
    From-loc is here
--> Success with branch #2
```

NOTE: The control pair containing a control-demo call should be put within a %TRACE-OFF% - %TRACE-ON% pair to avoid the printing of system trace messages regarding the control pair.

NOTE: The demo message string is passed to the format common-lisp function, and can therefore contain formatting characters accepted by that function (e.g., ~ or ~%). Refer to your CommonLisp manual for details.

## 7.10. Tracing of Specific Stages of the Unification

The following group of function enable or disable tracing of specific stages of the unification process:

```
(trace-determine :on t|nil)
(trace-category :all|cat|(cat1...catn) t|nil)
(trace-bk-class t|nil)
(trace-wait :on t|nil)
(trace-cset :on t|nil)
(trace-alts :on t|nil)
(hyper-trace-category cat :status t|nil)
```

### 7.10.1. Trace-determine

The determination stage checks that no ANY constraint is left unsatisfied at the end of the unification stage, and that all TEST constraints are satisfied. In addition, it checks that no further CSET traversal is required to restore after frozen constraints have been thawed or forced.

When trace-determine is on, these checks generate tracing messages. The specific messages are:

```
FAIL: Found an ANY at level <path>
Current Sentence: ....

TEST succeeds: <expr> at level <path>

Fail in testing <expr> at level <path>
```

The current sentence is only printed when FUF is called from the toplevel function `uni` whose function is to print a sentence. In other uses of FUF, the current function is not generated. This message is of level 30 (highest level).

In addition, each time the determination stage is reached, a line of statistics on backtracking is printed, of the form:

```
[Used 119 backtracking points - 26 wrong branches - 15 undos]
```

Three pieces of information are provided:

1. The total number of backtracking points used so far. This can be interpreted as the number of “questions” FUF asked to the grammar, or how many times a branch had to be chosen in an alt construct.
2. The number of “wrong branches” - that is, how many times FUF picked up a branch and started unification to finally undo this branch upon backtracking. This can be interpreted as the number of “wrong guesses” made by FUF.
3. The number of “undos” - that is, how many features were added to the total FD while traversing “wrong guesses” and were later removed from the total FD upon backtracking. This gives an indication of how “deep” FUF went into wrong branches before realizing that they led to failure.

When, because of the interaction between `wait` and the constituent traversal (cf Section WAIT-CSET) a new cycle of unification must be started after determination, a new line of statistics will be printed. These lines of statistics are NOT printed if the keyword parameter `non-interactive` is set to true when calling the top-level functions of FUF (for example, as in `(uni fd1 :non-interactive t)`).

These statistic lines are printed indicate whenever a unification cycle ends and a determination cycle starts. They therefore provide important information on how unification is proceeding.

### 7.10.2. Trace-Category and Hyper-trace-category

Trace-category is useful to follow the traversal of the constituent structure as it is performed by FUF. When a category `C` is traced, the following message is printed whenever a constituent of category `C` is unified with the grammar:

```
>=====
>STARTING CAT ADJ AT LEVEL {SYNT-ROLES SUBJ-COMP HEAD}
>=====
```

The function `hyper-trace-category` provides more detail: if category `C` is “hyper-traced”, the same message is printed, and in addition the whole constituent is printed:

```

>=====
>STARTING CAT ADJ AT LEVEL {SYNT-ROLES SUBJ-COMP HEAD}
>=====

>CONSTITUENT {SYNT-ROLES SUBJ-COMP HEAD} =
((CAT ADJ) (CONCEPT {SYNT-ROLES SUBJ-COMP CONCEPT}))
(POLARITY {SYNT-ROLES SUBJ-COMP POLARITY})
(LEX {SYNT-ROLES SUBJ-COMP LEX}))

```

The functions are used as follows:

```

(TRACE-CATEGORY c | :all | (c1 ... cn) &optional t | nil)
(HYPER-TRACE-CATEGORY c | :all | (c1 ... cn) &optional t | nil)
    Trace or hyper-trace a given category, or all categories
    (if parameter is :all) or a list of categories.
    If a second parameter nil is added, the category or
    categories are untraced.

```

### 7.10.3. Trace-Cset

The `trace-cset` function is used to monitor constituent traversal. Each time FUF finishes unifying a constituent, it computes the cset of the result of the unification, applying the rules described in Section 5.7. The constituents are then traversed breadth-first. The function `trace-cset` triggers messages of the following form at the end of the unification of each constituent:

```

>Expanding constituent {} into cset ({SYNT-ROLES SUBJ-COMP}
                                     {PROCESS}
                                     {SYNT-ROLES SUBJECT}).
>

>=====
>STARTING CAT AP AT LEVEL {SYNT-ROLES SUBJ-COMP}
>=====

```

If the constituent has an empty cset, it means it is a leaf in the constituent structure. In that case, the following message is printed:

```

>Constituent {SYNT-ROLES SUBJ-COMP HEAD} is a leaf.
>Constituent {PROCESS EVENT} is a leaf.
>

```

### 7.10.4. Trace-BK-Class

The interpretation of the BK-class annotation has been introduced in Section BK-CLASS. BK-class is used to allow intelligent dependency-directed backtracking. Each choice point can be marked to belong to a certain backtracking class (bk-class), and certain classes of paths can be declared to belong to corresponding bk-classes. Upon failure, the address of failure is checked. If it does not belong to a declared bk-class, regular chronological backtracking is started. If it does belong to a bk-class, then backtracking continues up to the latest choice points that belong to the same bk-class.

When tracing this behavior, the following conditions are monitored: (1) when a special failure address is met and how high up the intelligent backtracking progresses and (2) what is the latest address of failure. The following messages are printed in each case:

```
>BK: Special path <path> caught by alt <alt> of class <c> after <n> frames.
>BK: Switch from <path1> to <path2>
```

The first message is emitted whenever intelligent backtracking occurs. The second one is emitted whenever the current address of failure is modified following the rules discussed on page FAILURE-ADDRESS.

### 7.10.5. Trace-Wait

The wait annotation is used to allow goal delaying during the evaluation of a grammar. The idea is to wait until enough information is available before trying to choose between the branches of an alt. The grammar writer indicates which information is requested before the evaluation of an alt can start using the wait annotation and a sequence of path expressions which point to the features which must be instantiated with the requested information.

When enough information is already present, the alt is evaluated as usual. When there is missing information (some of the features are not yet instantiated), the alt is frozen (delayed). Frozen alts are stored on an agenda, which keeps track of the decisions which remain to be made in the future. At regular intervals (in FUF, whenever a new choice point is met), the agenda is checked, and if enough information has been gathered since the time a decision has been frozen, the decision is thawed, and evaluated immediately. After the evaluation of the thawed alt, control proceeds where it was interrupted.

In addition, at the end of the unification stage, the determination stage checks if any decision is still on the agenda. Such a situation is reached if not enough information could be gathered anyway to allow the evaluation of the frozen alts. In this case, the frozen alts of the agenda are ‘forced’ and evaluated, even though some of the requested information is missing.

Finally, there is one more configuration of control decisions which concerns the goal delaying behavior of FUF: as explained on page WAIT-IGNORE, wait annotations have priority over ignore annotations to insure that ignore annotations are only considered when enough information is available for them to make sense. When thawing a frozen alt from the agenda, after enough information has been gathered, the ignore annotations are immediately checked. If they do match, then the whole frozen alt is immediately ignored.

Corresponding to each of these configurations, the trace system emits the following messages. In every case, a unique agenda identifier (integer number) is assigned to each frozen alt:

```
>Freezing alt <alt>: waiting for <path> [agenda <n>]
>Thawing [agenda <n>]: Restarting at level <path>
>Forcing [agenda <n>]: Restarting at level <path>
>Ignoring [agenda <n>]
```

### 7.10.6. Trace-Alts

The function `trace-alt`s systematically monitors traversing of alts in the grammar, even if the alts are not traced, and outputs a message whenever a new branch is tried. It is best used in conjunction with a `trace-disable-all` setting, to follow uniquely alt traversal. The form of the output is as follows:

```
LISP> (trace-disable-all)
LISP> (trace-alt)
LISP> (uni t1)

>=====
>STARTING CAT CLAUSE AT LEVEL {}
>=====

>Fail in alt VOICE
>Fail in alt VOICE-NORMAL
>Fail in alt VOICE-NORMAL
>Fail in alt VOICE-NORMAL
>Fail in alt DATIVE-MOVE-DEFAULT
>Fail in alt DATIVE-MOVE-DEFAULT
>Fail in alt SUBJECT-SUBCAT
>Fail in alt SUBJ-COMP-CAT
>Fail in alt :ANONYMOUS
>Fail in alt :ANONYMOUS
>Expanding constituent {} into cset ({{SYNT-ROLES SUBJ-COMP}
                                     {PROCESS}
                                     {SYNT-ROLES SUBJECT}}).
>
```

Each time a branch in an alt fails, the message “fail in alt X” is printed. If the alt is not traced, the name `:anonymous` is printed instead. This is useful to find possible errors even in places which are not traced in the grammar. In general, `trace-alt` should only be used in last recourse.

## 7.11. Some Advice on FUF Debugging

This section provides some pragmatic advice on how to use the tracing system of FUF based on the experience gathered while developing large grammars. It lists common sources of confusion, warns against the misfeatures of the tracing system, common bugs and some successful bug tracking tricks.

### 7.11.1. Syntax Errors

The first source of bugs, often incomprehensible ones, is syntax errors, either in the input FD or in the grammar. So the first precaution is to check both inputs and grammars regularly. Use functions `FD-P` and `GRAMMAR-P` for that purpose. In grammars, check especially the number of parentheses occurring after ALTs.

### 7.11.2. Semantic Errors

Semantic errors in the grammar can be trivial typos or the sign of a bad design of the FD structure. In any case check the following points:

- Misspellings: wrong feature-name, wrong leaf-value-name
- Paths: in both forms, *i.e.*, `((f1 ((f2 ((f3 and {f1 f2 f3`. Make sure that paths actually point to the location in the total FD that you expect.

- The most common source of error is to include the wrong number of up-arrows ^ in a path expression. Always double-check your relative paths, especially those appearing embedded in ALTs, CONTROL, EXTERNAL when the up-arrow notation can be quite counter-intuitive.
- Observe the structure of the unified graph before unification. To this end, use the function `uni-fd` and pretty-print its output as in:
 

```
(pprint (clean-fd (uni-fd input :grammar gr)))
```

 This is often instructive. Use the function `top-gdp` to explore this output FD in detail.
- Draw a graph of the total FD with all features instantiated to help you check all your paths and levels of embedding.
- If you do not understand the current structure of the FD, insert a `%break%` in your grammar at some critical point, and use the function `path-value` to inspect the total FD during the time of the unification.
- It is risky to use absolute paths in general (SURGE does not include a single absolute path for example). Using an absolute path is a sign of desperation.

### 7.11.3. Expression of Negative Constraints

Negative constraints are used to limit the scope of acceptable FDs by the grammar and to force failure when certain FD configurations are met. The main tools in FUF for the expression of negative constraints are FSET and NONE. The main sources of confusion here are:

- FSET is too restrictive: you didn't plan on adding a feature, a new feature is not compatible with FSET.
- FSET must include explicitly all the features, including CAT and CSET.
- NONE is used in the wrong place.

### 7.11.4. Control

The overall flow of control of FUF is quite complex, and the trace messages do not make it very easy to follow. Before going on and suffering through the zillions of lines produced by the trace system, try to understand analytically what could go wrong - recheck the syntax of your fd and grammar, and recheck the structure of your FD and the structure your grammar builds. Only when this fails should you try to read the trace messages to understand what went wrong on a particular input. You should proceed in the following order:

1. Identify that there is a bug: do `(trace-off)` and `(trace-bp)`. This will emit a dot every 10 backtracking points (10 is the default) and indicate how much effort FUF has invested on your input. For example, when dealing with SURGE, a clause that takes more than 300 backtracking points (30 dots on the screen) is either very complicated or contains a bug. When you think there is a bug, interrupt the unifier (generally use `control-C`).
2. Monitor the speed of appearance of the dots. In general, FUF operates in two modes: "forward" and "backward". Forward mode is when the grammar is traversed as expected, and every new feature falls into place. Backward mode is after the occurrence of the first unexpected failure. The unifier starts backtracking and tries every other branch in an unexpected manner. In general, these tried branches fail very quickly. So in backward mode, backtracking dots tend to be emitted much more quickly than in forward mode. When the dots start piling up, it's a good sign that FUF has entered backward mode, and that a bug has been found.
3. Once you suspect there is a bug, identify the first unexpected failure. The first attempt to do that is to set `(trace-on)` and `(trace-level 30)`. This will print the most obvious candidates. If this does not work, basically, you're in trouble, and finding the bug will take time (sorry! I'm in the middle of the implementation of a graphical debugger for FUF which should help you beyond this stage).

4. The next step is to gradually lower the trace-level until you get a good understanding of where in the grammar is the source of the bug - moving to values 20, 15, 12, 10, 5 and 0 (trying trace-level 0 on a full grammar without disabling most of the tracing flags is a sign of desperation).
5. Once the approximate location of the problem is identified in the grammar, enable only the relevant tracing flags (those defined in the grammar around the problematic spot). This is achieved by using the functions (trace-disable-all) (trace-enable-alt <name-of-suspected-alt>).
6. From then on, try to understand what's happening. I find it convenient to run FUF within an Emacs buffer and to use the editor to search through the trace messages printed by FUF.
7. If you enable all tracing flags and set trace-level to 0 and you still cannot find a tracing message identifying a failure (of the form "FAIL in ...") this can be due to 2 problems:
  - a. Either the failure is occurring in a region of the grammar which does not contain tracing flags. In this case, set (trace-alt) and check the messages "fail in alt :anonymous".
  - b. Or the failure is due to a missing cat in the grammar. Often if you unify ((cat xxx)) with a grammar that has no branch for xxx, there is no failure message produced. Check your cats often.

The main points you should be looking for when debugging are:

- Endless backtracking: this can be found by setting trace-level 00 and looking for repetition in the flow of messages.
- Failure on ANY (especially in conjunction with recursion).
- Failure on given (especially in conjunction with wait): remember that an alt that is annotated with (wait path1) can be forced and therefore evaluated even if path1 is not yet instantiated. Therefore, using (path1 given) in such a situation is dangerous. In this case, switch to any.
- INDEX (especially double indexes and partial indexes). Do NOT factor together branches in an indexed alt. For example, the following index will not work (unification with ((a 1)) will fail):
 

```
(alt (:index a)
      ((a ((alt (1 2))))))
      ((a 3)))
```
- WAIT (especially in conjunction with ignore and bk-class)
- BK-CLASS: especially, since the bk-class declarations are persistent, do not forget to evaluate (clear-bk-class) when loading a new grammar. In case of doubt, evaluate (clear-bk-class) and re-evaluate all your (define-bk-class). In general, it is good to add the following prelude in all the files containing a grammar definition:
 

```
(eval-when (load eval)
      (clear-bk-class)
      (reset-typed-features))
```
- IGNORE: check that you do not ignore alts too liberally.
- Constituent Structure Definition:
  - Wrong cset (especially interaction between cat and explicit cset). In case of doubts, use an explicit cset in your grammar. Avoid over-restrictive csets (with the = notation), they often cause failure of unification with further cset features in the grammar.
  - Wrong pattern and interaction between pattern and cset when using implicit cset expansion.
  - Csets can trigger infinite recursion.
- Linearizer: The linearizer produces strings of the form <unknown cat: XXX> when a category unknown to the morphology module is found as a leaf in the constituent tree. This is often due to the following effect: all top-level unification functions have a :limit keyword option used to limit the number of

backtracking point used on a single input. The default value for limit is 10,000. When unification stops after reaching this limit, the FD is sent to the linearizer, even though it has not been completely unified, and all constituents have not been completely expanded. In that case, you can get strings such as <unknown cat: NP> because an NP has not been expanded. To correct this problem, increase the limit.

- Type Hierarchy Problems:

- Since the type declarations are persistent, do not forget to evaluate (`reset-typed-features`) when loading a new grammar. In case of doubt, use `draw-types` to verify the current state of type definitions. In general, it is good to add the following prelude in all the files containing a grammar definition:

```
(eval-when (load eval)
  (clear-bk-class)
  (reset-typed-features))
```

- Wrong `under` specification.
  - Wrong `define-feature-type`
  - Lack of `under` (when one wants only to test for the presence of a feature and not to enrich the total FD).
- Procedural types, test and control:
    - Wrong `CONTROL/TEST` function
    - `TEST` for `CONTROL` and vice-versa.
    - Wrong use of the path construct with a `TEST/CONTROL` (with the `#` notation).
    - Wrong `EXTERNAL` function.
    - Wrong procedural-type definitions.

## 8. Reference Manual

This chapter includes a list of the functions, variables and switches that a user of FUF can manipulate. They are grouped under 3 categories. In each category, the list is sorted alphabetically:

1. Unification functions
2. Checking
3. Tracing
4. Linearization and Morphology

### 8.1. Unification functions

#### 8.1.1. `*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)

#### 8.1.2. `*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.

#### 8.1.3. `*cat-attribute*`

**Type:** variable

**Standard value:** cat

**Description:** The `*cat-attribute*` variable contains a symbol. It is the default value for the `cat-attribute` argument to all the unification functions.

The CAT parameter is used to identify constituents in an fd when the `cset` attribute is not present. Through this mechanism, the unifier implements a breadth-first top-down traversal of the structures being generated.

By default, the CAT parameter is equal to the symbol `cat`. It is however possible to specify another value for this parameter. As a consequence, it is possible to traverse the same fd structure and to assign the role of constituents to different sub-structures by adjusting the value of this parameter. This feature is particularly useful when an fd is being processed through a pipe-line of grammars.

#### 8.1.4. `u`

**Type:** function

**Calling form:** (`u fd1 fd2` &optional *limit success*)

**Arguments:** p

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

- `limit` is a number. The default value is 1000.
- `success` is a function of three arguments. It must be defined as: `(defun x (fd fail frame) ...)` where `fd` is an fd, `fail` is a continuation (a function) and `frame` is an object of type `frame`. The default value is the function `default-continuation`.

**Description:** `u` unifies `fd1` with `fd2` and passes 3 values to the `success` continuation: a resulting fd, a continuation to call if more results are needed and a “stack-frame” containing information needed to run the continuation. By default, `default-continuation` just returns the unified fd. `u` is a low-level function.

It is possible to limit the time the unifier will devote to a particular call. The `:limit` keyword available in all unification functions specifies the maximum number of backtracking points that can be allocated to a particular call. Using this feature it is possible to perform “fuzzy” unification. (Note that the appropriateness of a fuzzy or incomplete unification relies on the particular control strategy used of breadth-first top-down expansion.)

### 8.1.5. u-disjunctions

**Type:** function

**Calling form:** `(u-disjunctions fd1 fd2 &key limit failure success)`

**Arguments:**

- `fd1` and `fd2` are arbitrary FDs. Both `fd1` and `fd2` can contain non-deterministic constructs, like `alt`, `ralt` and `opt`.
- `limit` is a number. The default value is 1000.
- `failure` is a function of one argument. It must be defined as: `(defun x (msg) ...)` where `msg` can be safely ignored.
- `success` is a function of three arguments. It must be defined as: `(defun x (fd fail frame) ...)` where `fd` is an fd, `fail` is a continuation (a function) and `frame` is an object of type `frame`. The default value is the function `default-continuation`.

**Description:** `u-disjunctions` unifies `fd1` with `fd2` and passes 3 values to the `success` continuation: a resulting fd, a continuation to call if more results are needed and a “stack-frame” containing information needed to run the continuation. By default, `default-continuation` just returns the unified fd. `u-disjunctions` is a low-level function. It is the only unification function accepting disjunctions in the input. For all other functions, if the input contains disjunctions, it should first be normalized by calling the function `normalize-fd`. The search for a unification works as follows: first one fd compatible with `fd1` is unified (as in `normalize`, in a blind search manner. Then, this fd is unified with `fd2`. If all tries with `fd2` fail, then the unifier backtracks and tries to find another fd compatible with `fd1`. Therefore, the choices in `fd1` are in general buried very deep in the search tree.

Refer to paragraph 8.1.4 for an explanation of the `limit` argument.

### 8.1.6. uni

**Type:** function

**Calling form:** `(uni input-fd &key grammar non-interactive limit cat-attribute)`

**Arguments:**

- `input-fd` is an input fd. It must be recognized by `fd-p`. It must not contain disjunctions.
- `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.
- `limit` is a number. It is 10000 by default.
- `cat-attribute` is a symbol. It has the value of `*cat-attribute*` 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` or of the `cat-attribute` argument. If `input-fd` contains no feature `cat` the unification fails. (cf. `unif` if this is the case.) If the input contains disjunctions, it should be normalized before being used (cf `normalize`).

Refer to paragraph 5.11 for an explanation of the `cat-attribute` argument. Refer to paragraph 8.1.4 for an explanation of the `limit` argument.

### 8.1.7. uni-string

**Type:** function

**Calling form:** (`uni-string input-fd &key grammar non-interactive limit cat-attribute`)

**Arguments:**

- `input-fd` is an input fd. It must be recognized by `fd-p`. It must not contain disjunctions.
- `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.
- `limit` is a number. It is 10000 by default.
- `cat-attribute` is a symbol. It has the value of `*cat-attribute*` by default.

**Description:** `uni-string` works exactly like `uni` except that it returns the generated string as a lisp string and does not print it.

### 8.1.8. uni-fd

**Type:** function

**Calling form:** (`uni-fd input-fd &key grammar non-interactive limit cat-attribute`)

**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-attribute`. If `input-fd` contains no feature `cat` the unification fails. (cf. `unif` if this is the case.)

Refer to paragraph 5.11 for an explanation of the `cat-attribute` argument. Refer to paragraph 8.1.4 for an explanation of the `limit` argument.

### 8.1.9. unif

**Type:** function

**Calling form:** (unif *input-fd* &key *grammar*)

**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\**.

**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*.

### 8.1.10. u-exhaust

**Type:** function

**Calling form:** (u-exhaust *fd1 fd2* &key *test limit*)

**Arguments:**

- *fd1* and *fd2* are fds. They must be recognized by *fd-p*. *fd1* cannot contain disjunctions.
- *test* is a lisp expression. It is T by default.
- *limit* is a number. It is 10000 by default.

**Description:** Unifier exhaust. Takes 2 functional descriptions and returns the list of all possible unifications until *test* is satisfied. *Test* is a lisp expression which is evaluated after each possible unification is found. In *test*, refer to the list being built as *fug3::result*. This function does NOT recurse on sub-constituents. It is at the same level as *u* or *u-disjunctions* (low-level function). If *test* is *nil*, this function will generate all possible unifications of *fd1* and *fd2*.

Refer to paragraph 8.1.4 for an explanation of the *limit* argument.

### 8.1.11. u-exhaust-top

**Type:** function

**Calling form:** (u-exhaust-top *input* &key *grammar non-interactive test limit*)

**Arguments:**

- *input* is an fd. It must be recognized by *fd-p*. It cannot contain disjunctions.
- *grammar* is a FUG. It must be recognized by *grammar-p*. By default it is *\*u-grammar\**.
- *non-interactive* is a flag. If it is *nil*, statistics are printed after each unification. Otherwise the function works silently. The default value is *nil*.
- *test* is a lisp expression. It is T by default.
- *limit* is a number. It is 10000 by default.

**Description:** Unifier exhaust with recursion. This function works like *u-exhaust* except that it also recurses on the sub-constituents of the input. It keeps producing fds until *test* evaluates to a non-*nil*. *test* is evaluated in an environment where *fug3::result* is bound to the list of all fds found so far. If *test* is *nil*, this function will

generate all possible unifications of `input` and `grammar`.

Refer to paragraph 8.1.4 for an explanation of the `limit` argument.

### 8.1.12. uni-num

**Type:** function

**Calling form:** (`uni-num input n &key grammar limit`)

**Arguments:**

- `input` is an `fd`. It must be recognized by `fd-p`. It cannot contain disjunctions.
- `n` is an integer.
- `grammar` is a FUG. It must be recognized by `grammar-p`. By default it is `*u-grammar*`.
- `limit` is a number. It is 10000 by default.

**Description:** Unifies `input` with `grammar` and backtracks `n` times. Each time, the result is processed as per `uni`. Refer to paragraph 8.1.4 for an explanation of the `limit` argument.

## 8.2. Checking

### 8.2.1. fd-syntax

**Type:** function

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

**Arguments:**

- `fd` is a list of pairs. It is `*u-grammar*` by default.
- `print-warnings` is a flag. It is `nil` by default.
- `print-messages` 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. If `print-messages` is `nil`, no diagnostic messages are printed, and the function just returns `T` or `nil`. If it is non-`nil`, diagnostic messages are printed.

\*\*\*\*\*NOTE: The following table has not been updated for Version 5.2\*\*\*\*\*

Diagnostics detected by <code>fd-syntax</code>	
message	condition
Unknown type: <code>~s</code> .	An <code>fd</code> must be either a legal leaf (symbol, number, string, character or array), a path or a valid list of pairs.
Unknow type: <code>~s</code> . Should be a pair or a tracing flag.	Within a list of pairs <code>fd</code> , all elements must be either pairs or tracing flags.
<code>~A</code> is a tracing flag. It cannot be used as an attribute in <code>~A</code> .	Tracing flags are not legal attributes in a pair.

message	condition
The attribute of a pair must be a symbol or a path: ~s.	Other types are forbidden.
An alt/ralt pair must have at most 4 args: (alt {trace} {index} {demo} (fd1 ... fdn)).	The disjunction construct being checked has more than 4 arguments.
An alt/ralt pair must have at most 4 args: (alt {trace} {index} {demo} (fd1 ... fdn)). There is no trace/index/demo flag in this pair.	The disjunction construct being checked has only 4 arguments, but one of the arguments is not properly formed.
An alt/ralt pair must have at most 4 args: (alt {trace} {index} {demo} (fd1 ... fdn)). There are no /index and demo/trace and demo/trace and index flags in this pair.	The disjunction construct being checked has only 3 arguments, but one of the arguments is not properly formed.
This alt/ralt pair must have one value: (alt (fd1 ... fdn)). There is no valid trace, index or demo flag in this pair.	No valid modifier is found in the current construct, and yet it has more than one value.
Value of alt/ralt must be a list of at least one branch: ~s.	A (alt ()) pair is invalid.
Value of special attribute ALT/RALT must be a list of valid FDs.	One of the fds in the branches of the pair is not valid.
An OPT pair must have at most 2 args: (OPT trace fd).	The OPT pair being checked has more than 2 arguments, as in (opt t a b).
This OPT pair must have at most 1 value: (OPT fd). (There is no valid tracing flag in this pair.)	A valid tracing flag must be either a symbol or a form (trace on <symbol>). The construct being checked has the form (opt f1 f2 ...) where f1 is not a valid tracing flag.
Value of OPT must be a valid FD: ~s	The fd part of the OPT is not a valid fd.
An FSET pair must be of the form (ATT VALUE): ~s.	The FSET pair has more than one value.
Value of special attribute FSET must be a list of atoms: ~s	One of the elements of the list is not a symbol.
A CSET pair must be of the form (ATT VALUE): ~s	The CSET pair has more than one value.
Value of special attribute CSET must be a list of symbols or valid paths: ~s	One of the elements of the list is not a symbol or a path.
A PATTERN pair must be of the form (ATT VALUE): ~s	The PATTERN pair can have only two values.
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 SPECIAL pair must be of the form (ATT VALUE): ~s	Special attributes can have only one value. <sup>8</sup>

<sup>8</sup>Special attributes (user defined types) can also issue user-defined syntax messages through the use of syntax-checker functions. Cf section on user-defined types for details.

message	condition
An EXTERNAL pair must be of the form (ATT external-spec): ~s	External-spec can be simply the symbol <code>external</code> or a construct <code> #(external &lt;fctn&gt; )</code> .
--- Warning: The argument of external must be a function.	In a <code> #(external &lt;fctn&gt; )</code> , <code>&lt;fctn&gt;</code> is not recognized as a defined function.
An UNDER specification must be an array of the form <code> #(UNDER symbol): ~s</code>	An UNDER value has been found that does not have the proper form.
The argument of an UNDER specification must be a symbol: ~s	In a <code> #(under &lt;x&gt; )</code> , <code>&lt;x&gt;</code> must be a symbol defined in a type hierarchy.
--- Warning: The argument of under does not have specializations defined. Use <code> (define-feature-type ~s (spec1 ... specn))</code> .	In a <code> #(under &lt;x&gt; )</code> , <code>&lt;x&gt;</code> must be a symbol defined in a type hierarchy. If the type hierarchy does not exist, it can be defined with the <code> define-feature-type</code> function.
A pair must be of the form (ATT VALUE)	The form being checked has more than two elements.
A value should be a valid fd.	In a pair (att value), value is not a valid fd.

### 8.2.2. fd-sem

**Type:** function

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

**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.
- `print-messages` is a flag. It is T by default.
- `print-warnings` 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).

\*\*\*\*\*NOTE: The following table has not been updated for Version 5.2\*\*\*\*\*

Diagnostics detected by fd-sem	
message	condition
--- Warning: Disjunctions in input FD: ~s	<code>grammar-p</code> is nil and a disjunction has been found in <code>fd</code> .
--- Warning: PATTERN or CSET should not be placed in input.	<code>grammar-p</code> is nil and a <code>pattern</code> or <code>cset</code> has been found in <code>fd</code> .

--- Warning: ANY or GIVEN should not be placed in input.	<i>grammar-p</i> is nil and a any or given has been found in <i>fd</i> .
--- Warning: EXTERNAL or UNDER should not be placed in input.	<i>grammar-p</i> is nil and a external or under has been found in <i>fd</i> .
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))).
This branch is contradictory: ~s	A branch in a disjunctive construct has been found with a contradictory value.

When `fd-sem` finds a problem in a grammar, it returns the path to the problem. Paths within grammars are different from paths in regular `fds` because of the presence of disjunctions. Paths have the same syntax, except that to go through an `alt` or `ralt` construct, additional information must be provided. The following syntax is used to denote the traversal of an `fd` with disjunctions:

```

PATH      := { att-spec* }
ATT-SPEC  := symbol | alt-spec | ralt-spec | opt-spec
ALT-SPEC  := (alt <index> {<branch>})
RALT-SPEC := (ralt <index> {<branch>})
OPT-SPEC  := (opt <index>)

```

Both `<index>` and `<branch>` must be numbers. The `<index>` information refers to the index of the `alt`, `ralt` or `opt` pair within its parent node. That is, given that a list of pairs can contain several `alts` at the same level, it is necessary to distinguish between them. The `<index>` information gives the position of the pair in the list, with index 0 referring to the first pair. If it is necessary to go further down an `alt` or `ralt` pair, then it is necessary to identify what branch must be followed. This information is given by the `<branch>` index. Note that a specifier `(alt <index>)` without a branch index **MUST** necessary be the last one in a path and refers to the whole `alt` pair. The function `alt-gdp` is used to traverse an `fd` with disjunctions using these extended paths as input. The function `get-error-pair` returns the first pair where `fd-syntax` would find an error.

### 8.2.3. `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.

### 8.2.4. 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.

### 8.2.5. get-error-pair

**Type:** function.

**Calling form:** (`get-error-pair` *fd*)

**Arguments:**

- `fd` is an fd.

**Description:** `get-error-pair` checks the syntax of an fd. If the syntax is not correct, it returns the pair containing the first offending constituent of `fd`.

### 8.2.6. normalize-fd

**Type:** function.

**Calling form:** (`normalize-fd` *fd*)

**Arguments:**

- `fd` is an fd.

**Description:** `normalize-fd` prepares an fd that can contain disjunctions to be used as input to the standard unification procedures (that do not accept disjunctions, *i.e.*, all except `u-disjunctions`). If `fd` contains disjunctions, `normalize` will return one disjunction-free fd compatible with `fd`. It is best understood as basically an equivalent to the operation (`u nil fd`). If `fd` is not semantically correct (it contains contradictions), `normalize` will return `:fail`.

Normalize is also useful to put an fd in normal form with respect to the following constraint:

```
The fd ((a ((x 1))) (a ((y 2)))) is
      ((a ((x 1)
           (y 2))))
in normal form.
```

## 8.3. Tracing

\*\*\*\*NOTE: This Section has not yet been updated for Version 5.2\*\*\*\*

\*\*\*\*Certain functions are missing\*\*\*\*

### 8.3.1. *\*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%

### 8.3.2. *\*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%

### 8.3.3. *\*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. When it is on and the determination stage fails in the context of a uni call, then the partially found sentence is linearized and printed. Cf *trace-determine* for the user-level interface to this variable.

**Standard Value:** T

### 8.3.4. *\*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:** #\%

### 8.3.5. *\*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

### 8.3.6. 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.

### 8.3.7. 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.

### 8.3.8. 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.

### 8.3.9. 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.

### 8.3.10. trace-disable-all

**Type:** function

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

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

### 8.3.11. 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.

**8.3.12. 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.**8.3.13. trace-enable-all****Type:** function**Calling form:** (trace-enable-all)**Description:** `trace-enable-all` enables all tracing flags. Initially, all tracing flags are enabled.**8.3.14. 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.**8.3.15. 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.**8.3.16. trace-on****Type:** function**Calling form:** (trace-on)**Description:** `trace-on` turns on tracing.

Initially, tracing is off.

**8.3.17. trace-determine****Type:** function**Calling form:** (trace-determine &key *on*)**Description:** `trace-determine` turns on and off tracing for the determination stage.

When tracing is on for the determination stage, a message is printed indicating the location of the any found or the failed test. In addition, if the top-level function called is `uni`, the partially unified `fd` is linearized and printed to show the progression of the unifier.

**8.3.18. trace-bk-class**

**Type:** function

**Calling form:** (trace-bk-class &optional *on*)

**Description:** trace-bk-class turns on and off tracing of the special bk-class backtracking behavior.

When tracing is on for bk-class, a message is printed whenever a path of a bk-class category is caught by a

```

> (uni a3)
>STARTING CAT DISCOURSE-SEGMENT AT LEVEL {}
>STARTING CAT UTTERANCE AT LEVEL {DIRECTIVE}
>STARTING CAT CLAUSE AT LEVEL {DIRECTIVE PC}
>STARTING CAT NP AT LEVEL {DIRECTIVE PC OBJECT}
>STARTING CAT NP AT LEVEL {DIRECTIVE PC IOBJECT}
>STARTING CAT NP AT LEVEL {DIRECTIVE PC SUBJECT}
>STARTING CAT LEXICAL-ENTRY AT LEVEL {DIRECTIVE PC VERB-CONCEPT}
>STARTING CAT NP AT LEVEL {DIRECTIVE PC OBJECT}
>STARTING CAT NP AT LEVEL {DIRECTIVE PC SUBJECT}
>STARTING CAT NP AT LEVEL {DIRECTIVE PC BENEF}
>STARTING CAT LEXICAL-ENTRY AT LEVEL {DIRECTIVE PC VERB-CONCEPT}
>STARTING CAT VERB-GROUP AT LEVEL {DIRECTIVE PC VERB}
>STARTING CAT PP AT LEVEL {DIRECTIVE PC DATIVE}
>STARTING CAT DET AT LEVEL {DIRECTIVE PC OBJECT DETERMINER}

[Used 108 backtracking points - 57 wrong branches - 103 undos]
John takes a book from Mary.

```

## 8.4. Linearization and Morphology

### 8.4.1. 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.

### 8.4.2. 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.

### 8.4.3. morphology-help

**type:** function.

**calling form:** (morphology-help)

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

## References

# Index

# notation 21  
 # notation 52  
  
 %break% 37, 50  
  
 \* notation 58  
 \*all-trace-off\* (variable) 62  
 \*all-trace-on\* (variable) 62  
 \*cat-attribute\* (variable) 27, 53  
 \*irreg-plurals\* (variable) 29, 31  
 \*irreg-verbs\* (variable) 29, 32  
 \*lexical-categories\* (variable) 53  
 \*top\* (variable) 62  
 \*trace-determine\* (variable) 62  
 \*trace-marker\* (variable) 40, 62  
 \*u-grammar\* (variable) 53  
  
 ... notation 21  
  
 :anonymous 51  
 :demo (annotation) 44  
 :order (annotation) 19  
  
 === notation 7  
  
 A-an (morphological feature) 29, 30  
 Absolute path 13, 50  
 Adj 29  
 Adv 29  
 Agreement (subject/verb) 6, 8  
 All-tracing-flags (function) 37, 44, 63  
 Alt (keyword) 6, 19, 39  
 Alt (special attribute) 49  
 Alt-gdp (function) 60  
 Ambiguity of the ^ notation 15  
 Any (special value) 25, 51  
  
 Bk-class (annotation) 47, 51, 65  
 Branch (of an alt) 6  
  
 Call-linearizer (function) 66  
 Cardinal 29  
 Case 30  
 Cat (special attribute) 26, 29, 51, 56  
 Categories-not-unified (function) 29  
 Category 26  
 Clean-fd (function) 50  
 Clear-bk-class (function) 51, 52  
 Common noun 6  
 Con 29  
 Conflation 14  
 Conjunction 13  
 Constituent 6, 22  
 Constituent traversal 7, 22, 25, 46, 47, 51  
 Constraint (feature as) 13  
 Control (keyword) 52  
 Control in FUF 47, 48  
 Control-demo (function) 44  
 Cset (keyword) 7, 22, 47  
 Cset (special attribute) 51  
 Cset (unification) 22, 24  
  
 Default (in alt) 6, 19  
 Define-feature-type (function) 52

Define-procedural-type (function) 52  
 Demonstrative 30  
 Denotation (of FDs) 13  
 Desperation (when debugging FUF) 50, 51  
 Det 29, 30  
 Determination 25, 45, 55, 56  
 Dictionary 31  
 Digit (morphological feature) 29  
 Disjunction 19  
 Distance 30  
 Dots (in pattern) 21  
 Draw-types (function) 52  
  
 EMACS (text editor) 51  
 Ending 30  
 Equations 13, 14  
 Examples 5  
 External (keyword) 52  
  
 Failure (of unification) 8, 19, 25, 39  
 Far 30  
 FD 1, 5  
 Fd-p (function) 3, 5, 49, 60  
 Fd-sem (function) 59  
 Fd-syntax (function) 57  
 Features 13  
 First (person) 30  
 Fset (special attribute) 50  
  
 Gap 66  
 Gender 30  
 Get-error-pair (function) 61  
 Given (special value) 25, 51  
 Gr0.l (file) 5  
 Gr11.l (file) 11  
 Gr4.l (file) 11  
 Grammar organization 26  
 Grammar-p (function) 3, 49, 61  
 Graph (FD as) 15, 50  
  
 Hyper-trace-category (function) 37, 46  
  
 Ignore (annotation) 51  
 Index (annotation) 51  
 Infinitive 30  
 Initial failure 35  
 Initialize-lexicon (function) 31  
 Instantiated features 25  
 Internal-trace-off (function) 63  
 Internal-trace-on (function) 63  
  
 Leaf 13  
 Lex (special attribute) 7  
 Lexical categories 29  
 Lexicon.l (file) 31  
 Limit (keyword) 52  
 Limit 54  
 Linearization 1, 9, 20, 29, 66  
 Linearize.l (file) 29  
 Linearizer 52  
  
 Mergeable constituents (in pattern) 58  
 Modal 29  
 Morphology 9, 10, 29, 32, 66  
 Morphology-help (function) 10, 29, 67  
  
 Near 30

Nil (special value) 13  
 Non-deterministic constructs 21  
 Non-interactive (keyword) 46  
 Non-standard features of implementation 27  
 None (special value) 25, 50  
 Normalize-fd (function) 15, 54, 61  
 Noun 29, 30  
 Number 30

Objective 30  
 Opt (keyword) 20, 39  
 Optional features 20  
 Order (annotation) 19  
 Order independence 19  
 Ordering constraints 6, 20  
 Ordinal 29

Pair (attribute/value) 13  
 Past 30  
 Past-participle 30  
 Path (flat description of FDs) 13  
 Path (unification) 14  
 Path 13  
 Path-value (function) 43, 50  
 Pattern (keyword) 6, 9, 20, 58  
 Pattern (unification) 21  
 Person 30  
 Personal 30  
 Phrase 29  
 Plural 30  
 Possessive 30  
 Pound (in pattern) 21  
 Prep 29  
 Present 30  
 Pronoun 30  
 Pronoun-type 30  
 Proper noun 6  
 Punctuation 29, 30, 32

Quantified 30  
 Question 30

Ralt (keyword) 19  
 Recursion 7, 22  
 Reflexive 30  
 Register-category-not-unified (function) 29  
 Relative path 13, 50  
 Relpro 29  
 Reset-typed-features (function) 51, 52  
 Restrictions 27  
 Root 30

Second (person) 30  
 Set-path-value (function) 43  
 Singular 30  
 Structure sharing 14  
 Sub-constituents 7  
 Subjective 30  
 Syntax 13

Tense 30  
 Test (keyword) 25, 52  
 Third (person) 30  
 Top-gdp (function) 50  
 Total fd 25, 55, 56  
 Trace-alts (function) 37, 49, 51

Trace-bk-class (function) 37, 47, 65  
 Trace-bp (function) 37, 50  
 Trace-category (function) 37, 46, 65  
 Trace-cset (function) 37, 47  
 Trace-determine (function) 37, 45, 64  
 Trace-disable (function) 37, 44, 63  
 Trace-disable-all (function) 37, 44, 51, 63  
 Trace-disable-alt (function) 37, 44  
 Trace-disable-match (function) 37, 44, 63  
 Trace-enable (function) 37, 44, 64  
 Trace-enable-all (function) 37, 44, 64  
 Trace-enable-alt (function) 37, 44, 51  
 Trace-enable-match (function) 37, 44, 64  
 Trace-level (function) 37, 38, 50  
 Trace-off (function) 37, 50, 64  
 Trace-on (function) 37, 50, 64  
 Trace-wait (function) 37, 48  
 Tracing (local) 40  
 Tracing (of alt) 39  
 Tracing (of opt) 39  
 Tracing 35  
 Tracing flag 40, 57, 62  
 Tracing messages 39

U (function) 53  
 U-disjunctions (function) 54  
 U-exhaust (function) 56  
 U-exhaust-top (function) 56  
 Under (special value) 26, 52  
 Uni (function) 3, 54  
 Uni-fd (function) 3, 50, 55  
 Uni-num (function) 57  
 Uni-string (function) 55  
 Unif (function) 3, 56  
 Unification (overall mechanism) 7  
 Unification 1  
 Unification functions 53  
 Unknown category 10, 29, 52

Value (morphological feature) 29  
 Verb 29, 30

Wait (annotation) 48, 51  
 Wait (control annotation) 25

^ notation (ambiguity) 15  
 ^n notation 13, 50

## Table of Contents

<b>1. Introduction</b>	<b>1</b>
1.1. How to Read this Manual	1
1.2. Function and Content of the Package	1
<b>2. Getting Started</b>	<b>3</b>
2.1. Main User Functions	3
<b>3. FDs, Unification and Linearization</b>	<b>5</b>
3.1. What is an FD?	5
3.2. A Simple Example of Unification	5
3.3. Linearization	9
<b>4. Writing and Modifying Grammars</b>	<b>11</b>
<b>5. Precise Characterization of FDs</b>	<b>13</b>
5.1. Generalities: Features, Syntax, Paths and Equations	13
5.2. FDs as Graphs	15
5.3. Functional Descriptions vs. First-order Terms	18
5.4. Disjunctions: The ALT and RALT Keywords	19
5.5. Optional Features: the OPT Keyword	20
5.6. Control of the Ordering: the PATTERN Keyword	20
5.7. Explicit Specification of Sub-constituents: the CSET Keyword	22
5.7.1. Implicit and Incremental CSET Specification	23
5.7.2. Unification of Incremental CSET Specifications	24
5.8. The Special Value NONE	25
5.9. The Special Value ANY - The Determination Stage	25
5.10. The Special Value GIVEN	25
5.11. The Special Attribute CAT: General Outline of a Grammar	26
<b>6. Morphology and Linearization</b>	<b>29</b>
6.1. Lexical Categories are not Unified	29
6.2. CATEGORIES Accepted by the Morphology Module	29
6.3. Accepted Features for VERB, NOUN, PRONOUN, DET, ORDINAL, CARDINAL and PUNCTUATION	30
6.4. Possible Values for Features NUMBER, PERSON, TENSE, ENDING, BEFORE, AFTER, CASE, GENDER, PERSON, DISTANCE, PRONOUN-TYPE, A-AN, DIGIT and VALUE	30
6.5. The Dictionary	31
6.6. Linearization and Punctuation	32
<b>7. Tracing and Debugging</b>	<b>35</b>
7.1. What it Means to Debug a FUF Program	35
7.2. Checking the Validity of FDs and Grammars	36
7.3. Fine Tuning Tracing: Overview of FUF Tracing Functions	36
7.4. Identifying Possible Bugs: Trace-bp	37
7.5. Levels of Tracing	38
7.6. Tracing of Alternatives and Options	39
7.7. Local tracing with boundaries	40
7.7.1. Special Flags %trace-on% and %trace-off%	40
7.7.2. The Special Tracing Flag %break%	41
7.8. The trace-enable and trace-disable Family of Functions	44
7.9. The :demo directive	44
7.10. Tracing of Specific Stages of the Unification	45
7.10.1. Trace-determine	45

7.10.2. Trace-Category and Hyper-trace-category	46
7.10.3. Trace-Cset	47
7.10.4. Trace-BK-Class	47
7.10.5. Trace-Wait	48
7.10.6. Trace-Alts	49
7.11. Some Advice on FUF Debugging	49
7.11.1. Syntax Errors	49
7.11.2. Semantic Errors	49
7.11.3. Expression of Negative Constraints	50
7.11.4. Control	50
<b>8. Reference Manual</b>	<b>53</b>
<b>8.1. Unification functions</b>	<b>53</b>
8.1.1. *lexical-categories*	53
8.1.2. *u-grammar*	53
8.1.3. *cat-attribute*	53
8.1.4. u	53
8.1.5. u-disjunctions	54
8.1.6. uni	54
8.1.7. uni-string	55
8.1.8. uni-fd	55
8.1.9. unif	56
8.1.10. u-exhaust	56
8.1.11. u-exhaust-top	56
8.1.12. uni-num	57
<b>8.2. Checking</b>	<b>57</b>
8.2.1. fd-syntax	57
8.2.2. fd-sem	59
8.2.3. fd-p	60
8.2.4. grammar-p	61
8.2.5. get-error-pair	61
8.2.6. normalize-fd	61
<b>8.3. Tracing</b>	<b>62</b>
8.3.1. *all-trace-off*	62
8.3.2. *all-trace-on*	62
8.3.3. *trace-determine*	62
8.3.4. *trace-marker*	62
8.3.5. *top*	62
8.3.6. all-tracing-flags	63
8.3.7. internal-trace-off	63
8.3.8. internal-trace-on	63
8.3.9. trace-disable	63
8.3.10. trace-disable-all	63
8.3.11. trace-disable-match	63
8.3.12. trace-enable	64
8.3.13. trace-enable-all	64
8.3.14. trace-enable-match	64
8.3.15. trace-off	64
8.3.16. trace-on	64
8.3.17. trace-determine	64
8.3.18. trace-bk-class	65
8.3.19. trace-category	65
<b>8.4. Linearization and Morphology</b>	<b>66</b>
8.4.1. call-linearizer	66

8.4.2. gap	66
8.4.3. morphology-help	67
<b>Index</b>	<b>69</b>