FUF: the Universal Unifier
User Manual
Version 5.0

Michael Elhadad

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

23 October 1991

Abstract

This report is the user manual for FUF version 5.0, a natural language generator program that uses the technique of unification grammars. The program is composed of two main modules: a unifier and a linearizer. The unifier takes as input a semantic description of the text to be generated and a unification grammar, and produces as output a rich syntactic description of the text. The linearizer interprets this syntactic description and produces an English sentence. This manual includes a detailed presentation of the technique of unification grammars and a reference manual for the current implementation (FUF 5.0). Version 5.0 now 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.

1. Introduction

1.1. How to read this manual

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

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

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

This manual contains material for people interested in any of these. It starts with an introduction to functional unification, its syntax, semantics and terminology. The next chapters deal with the ‘‘grammar development’’ tools: tracing and indexing, a presentation of the morphology component and the dictionary. The next chapter is devoted to typing in FUF: type definition, user-defined unification methods and expression of complete information. One chapter is devoted to the new features of versions 4 and 5, and to control specification. Finally the last chapter is a reference manual to the package. One appendix is devoted to the possible non-linguistic applications of the formalism, and compares the formalism with programming languages.

Note that this manual does **not** describe or document the example grammars provided as examples with the unifier. These grammars contain a brief documentation on-line and are accompanied by example inputs. It is hoped that this constitute enough data for people to use the grammars.

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). 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 [17]. It comes down to enriching the input-fd with directives coming from the grammar and indicating word order, syntactic constructions, number agreement and other features.

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

2. Getting Started

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

2.1. Main User Functions

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

```
(UNI 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.l
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.)

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

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

The functions `complexity` and `avg-complexity` measure how complex is a grammar, that is how much time unification with this grammar requires. They are documented in section 9 on indexing.

3. FDs, Unification and Linearization

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

3.1. What is an FD?

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

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

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

3.2. A Simple Example of Unification

We present here a minimal grammar that contains just enough to generate the simplest complete sentences. It is included in file "gr0.l" in the directory containing the examples. A little more complex grammar, handling the active/passive distinction, is available in "gr1.l", and a more interesting one in "gr2.l".[1]

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

[1]Note that the simplest grammars presented in the manual use the standard phrase structure approach S -> 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.

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

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

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

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

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

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

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


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:

```
CLISP> (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 hierarchy of constituents is traversed breadth first.

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

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

CLISP> (uni ir02)

>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 5.5. The linearizer works as follows:

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

Note that when the fd does not contain a morphological feature, the morphology module provides reasonable defaults. More details on morphology are provided in section 10.

Note also that 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 BENEF are missing):
John likes.
```

Finally, note that if one of the constituent sent to the morphology is not a known morphological category, the morphology module can not 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 you have done something wrong. You should check the list of legal morphological categories (see section 10) or you should check why a high level constituent is sent to the morphology (your fd is too flat). You can use the function `morphology-help` to have on-line help on what the morphology module can do.

4. Writing and Modifying Grammars

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

1. Determine the input to use. In general, input is given by an underlying application. If not, the criterion to decide what is a good input is that it should be as much ‘‘semantic’’ as possible, and contain the fewest syntactic features as possible.

2. Identify the types of sentences to produce.

3. For each type of sentence, identify the constituents and sub-constituents, and their function in the sentence. A constituent is a group of words that are ‘‘tied together’’ in a clause. A constituent in general plays a certain function with respect to the higher level constituent containing it. For example, in ‘‘John gives a book to Mary,’’ the group ‘‘a book’’ forms a constituent, of category ‘‘noun-group,’’ and it plays the role of the ‘‘object upon which action is performed’’ in the clause. Such role is often called ‘‘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.

5. Precise characterization of FDs

5.1. Generalities: features, syntax, paths and equations

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

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

A ‘‘leaf’’ is a primitive fd. It can be either a symbol, a number, a string, a character or an array.

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

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

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

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

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

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

The notation {^4 x} is equivalent to {^ ^ ^ ^ x}. It is convenient when dealing with deeply embedded constituents.

The value of a pair can be a path. In that case, it means that the values of the pair pointed to by the path and the value of the current pair must always be the same. In this case, the two features are said to be unified. In the previous example, the features at the paths `{verb number}` and `{prot number}` are unified. 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.

Since version 3, 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.[2]

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

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

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

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

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:

[2]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.

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

CLISP> (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

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

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

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

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

The following attributes have a special unification behavior: `alt, opt, ralt, pattern, cset, fset, test, control` and `cat` (or the currently specified cat attribute). The following values have a special unification behavior: `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.

5.3. Disjunctions: The ALT and RALT keywords

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

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

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

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

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

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))))
```

5.4. Optional features: the OPT keyword

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

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

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

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

`opt` is used exactly in the same way as `alt`.

5.5. Control of the ordering: the PATTERN keyword

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Appendix II describes some advanced uses of pattern unification.

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

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

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

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

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

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

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

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

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

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

5.6. 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 2 following pairs are correctly unified:

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

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

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

1. If a feature `(cset (c1 ... cn))` is found in the FD, the constituent set is just `(c1 ... cn)`.
2. If no feature `cset` is found, the constituent set is the union of the following sub-fds:

a. If a pair contains a feature `(cat xx)`, it is considered a constituent.

b. If a sub-fd is mentioned in the pattern, it is considered a constituent.

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

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

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

5.7. The special value NONE

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

5.8. The special value ANY - The Determination stage

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

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

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

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

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

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

Note that in practice, ANY is used VERY rarely.

5.9. The special value GIVEN

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

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

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

NOTE: The `under` construct is related to the `given` value. It is presented in Section 6.2.

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

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

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

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

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

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. Types in Unification

This version of the unifier implements three notions of types:

- Typed features
- Procedural types with user-defined unification methods
- Constituent types with the FSET special feature

The idea of using types in unification is relatively recent. So this manual first motivates the use of types in FUGs. The following subsections describe each one of the three methods of typing available.

6.1. Why Types

6.1.1. Typed features

6.1.1.1. A limitation of FUGs: no structure over the set of values

To formally define a grammar, we define *L* as a set of labels or attribute names and *C* as a set of constants, or simple atomic values. A string of labels (that is an element of L^*) is a path, and is noted $\{l_1...l_n\}$. A grammar defines a domain of admissible paths, $\Delta \subset L^*$. Δ defines the skeleton of well-formed FDs.

In functional unification, the set of constants *C* has no structure. It is a flat collection of symbols with no relations between each other. All constraints among symbols must be expressed in the grammar. In linguistics, however, grammars assume a rich structure between properties: some groups of features are mutually exclusive; some features are only defined in the context of other features.

```
                  | Question
                  | Personal
      | Pronoun --|
      |           | Demonstrative
      |           | Quantified
Noun  |
      | Proper
      |           | Count
      | Common ---|
                  | Mass
```

Let’s consider a fragment of grammar describing noun-phrases (NPs) using the systemic notation given in [30]. Systemic networks, such as this one, encode the choices that need to be made to produce a complex linguistic entity. They indicate how features can be combined or whether features are inconsistent with other combinations. The configuration illustrated by this fragment is typical, and occurs very often in grammars. The schema indicates that a noun can be either a pronoun, a proper noun or a common noun. Note that these three features are mutually exclusive. Note also that the choice between the features `{question, personal, demonstrative, quantified}` is relevant only when the feature pronoun is selected. This system therefore forbids combinations of the type `{pronoun, proper}` and `{common, personal}`.

The traditional technique for expressing these constraints in a FUG is to define a label for each non terminal symbol in the system. The resulting grammar is as follows:

```
((cat noun)
 (alt (((noun pronoun)
        (pronoun
         ((alt (question personal demonstrative quantified)))))
       ((noun proper))
       ((noun common)
        (common ((alt (count mass))))))))
```

This grammar is, however, incorrect, as it allows combinations of the type `((noun proper) (pronoun question))` or even worse `((noun proper) (pronoun zouzou))`. Because unification is similar to union of features sets, a feature `(pronoun question)` in the input would simply get added to the output. In order to enforce the correct constraints, it is therefore necessary to use the meta-FD NONE (which prevents the addition of unwanted features) as shown below:

```
((alt (((noun pronoun)
        (common NONE)
        (pronoun
         ((alt (question personal demonstrative quantified)))))
       ((noun proper) (pronoun NONE) (common NONE))
       ((noun common)
        (pronoun NONE)
        (common ((alt (count mass))))))))
```

The input FD describing a personal pronoun is then:

```
((cat noun)
 (noun pronoun)
 (pronoun personal))
```

There are two problems with this corrected FUG implementation. First, both the input FD describing a pronoun and the grammar are redundant and longer than needed. Second, the branches of the alternations in the grammar are interdependent: you need to know in the branch for pronouns that common nouns can be sub-categorized and what the other classes of nouns are. This interdependence prevents any modularity: if a branch is added to an alternation, all other branches need to be modified. It is also an inefficient mechanism as the number of pairs processed during unification is $O(n^d)$ for a taxonomy of depth d with an average of n branches at each level.

6.1.1.2. Introducing Typed Features

The problem thus is that FUGs do not gracefully implement mutual exclusion and hierarchical relations. The system of nouns is a typical taxonomic relation. The deeper the taxonomy, the more problems we have expressing it using traditional FUGs.

We propose extracting hierarchical information from the FUG and expressing it as a constraint over the symbols used. The solution is to define a subsumption relation over the set of constants C. One way to define this order is to define types of symbols, as illustrated below:[3]

[3]This notion of typing is similar to the Ψ-terms defined in [1].

```
(define-feature-type noun (pronoun proper common))
(define-feature-type pronoun (personal-pronoun question-pronoun
                              demonstrative-pronoun quantified-pronoun))
(define-feature-type common (count-noun mass-noun))

The grammar becomes:
((cat noun)
 (alt (((cat pronoun)
        (cat ((alt (question-pronoun personal-pronoun
                    demonstrative-pronoun quantified-pronoun)))))
       ((cat proper))
       ((cat common)
        (cat ((alt (count-noun mass-noun)))))))

And the input: ((cat personal-pronoun))
```

The syntax of the new function `define-feature-type` will be presented in section 6.2. Once types and a subsumption relation are defined, the unification algorithm must be modified. The atoms *X* and *Y* can be unified if they are equal OR if one subsumes the other. The result is the most specific of *X* and *Y*.

With this new definition of unification, taking advantage of the structure over constants, the grammar and the input become much smaller and more readable. There is no need to introduce artificial labels. The input FD describing a pronoun is a simple `((cat personal-pronoun))` instead of the redundant chain down the hierarchy `((cat noun) (noun pronoun) (pronoun personal))`. Because values can now share the same label CAT, mutual exclusion is enforced without adding any pair `[l:NONE]`.[4] Note that it is now possible to have several pairs $[a{:}v_i]$ in an FD F, but that the phrase ''the `a` of F'' is still non-ambiguous: it refers to the most specific of the v_i. Finally, the fact that there is a taxonomy is explicitly stated in the type definition section whereas it used to be buried in the code of the FUG. This taxonomy is used to document the grammar and to check the validity of input FDs.

6.1.2. Typed Constituents: The FSET Construct

A natural extension of the notion of typed features is to type constituents: typing a feature restricts its possible values; typing a constituent restricts the possible features it can have.

Type declarations (in the grammar):

```
(determiner ((fset (definite distance demonstrative possessive))))
```

Input FD describing a determiner:

```
(determiner ((definite yes)
             (distance far)
             (demonstrative no)
             (possessive no)))
```

The `fset` feature specifies that only the four features listed can appear under the constituent `determiner`. This statement declares what the grammar knows about determiners. `Fset` expresses a completeness constraint as defined in LFGs [8]; it says what the grammar needs in order to consider a constituent complete. Without this construct, FDs can only express partial information. The exact syntax of `fset` is given in Section 6.3.

---

[4]In this example, the grammar could be a simple flat alternation ((cat ((alt (noun pronoun personal-pronoun ... common mass-noun count-noun))))), but this expression would hide the structure of the grammar.

Note that expressing such a constraint (a limit on the arity of a constituent) is impossible in the traditional FU formalism. It would be the equivalent of putting a NONE in the attribute field of a pair as in NONE:NONE.

Without FSET:
```
((cat clause)
 (alt (((process-type action)
        (inherent-roles ((carrier NONE)
                         (attribute NONE)
                         (processor NONE)
                         (phenomenon NONE))))
       ((process-type attributive)
        (inherent-roles ((agent NONE)
                         (medium NONE)
                         (benef NONE)
                         (processor NONE)
                         (phenomenon NONE))))
       ((process-type mental)
        (inherent-roles ((agent NONE)
                         (medium NONE)
                         (benef NONE)
                         (carrier NONE)
                         (attribute NONE)))))))
```

With FSET:
```
((cat clause)
 (alt (((process-type action)
        (inherent-roles ((FSET (agent medium benef)))))
       ((process-type attributive)
        (inherent-roles ((FSET (carrier attribute)))))
       ((process-type mental)
        (inherent-roles ((FSET (processor phenomenon))))))))
```

In general, the set of features that are allowed under a certain constituent depends on the value of another feature. Figure 6.3 illustrates the problem. The fragment of grammar shown defines what inherent roles are defined for different types of processes (it follows the classification provided in [6]). We also want to enforce the constraint that the set of inherent roles is ‘‘closed’’: for an action, the inherent roles are agent, medium and benef *and nothing else*. This constraint cannot be expressed by the standard FUG formalism. `fset` makes it possible.

Note also that the set of possible features under the constituent `inherent-roles` depends on the value of the feature `process-type`. The first part of the Figure above shows how the constraint can be implemented without `fset`: we need to exclude all the roles that are not defined for the process-type.[5] Note that the problems are very similar to those encountered on the pronoun system: explosion of `none` branches, interdependent branches, long and inefficient grammar.

The `fset` (feature set) attribute solves this problem: `fset` specifies the complete set of legal features at a given level of an FD. `fset` adds constraints on the definition of the domain of admissible paths Δ of a grammar. The syntax is the same as `cset`. Note that all the features specified in `fset` do not need to appear in an FD: only a subset of those can appear. For example, to define the class of middle verbs (*e.g.*, ‘‘to shine’’ which accepts only a medium as inherent role and no agent), the following statement can be unified with the fragment of grammar shown in the previous figure:

[5] Note that this is not even correct, since any other attribute (besides the names of roles) could still be accepted by the grammar.

```
((verb ((lex "shine")))
 (process-type action)
 (voice-class middle)
 (inherent-roles ((FSET (medium)))))
```

The feature `(FSET (medium))` can be unified with `(FSET (agent medium benef))` and the result is `(FSET (medium))`.

Typing constituents is necessary to implement the theoretical claim of LFG that the number of syntactic functions is limited. It also has practical advantages. An important advantage is good documentation of the grammar. Typing also allows checking the validity of inputs as defined by the type declarations.

6.1.3. Procedural Types

FUF also implements a third notion of type in unification: procedural types correspond to user-defined data-structures that are unified by special-purpose unification methods. The unification method describes how elements of the type fit in a partial order structure. Typed features are explicitly described (extensionally) partial orders. With procedural types, the partial order is intensionally described by a Lisp procedure.

Procedural types therefore allow the grammar to integrate complex objects that could hardly be described by standard FDs alone. Examples of procedural types are `pattern` (with the pattern unification method enforcing ordering constraints), `cset` (with the cset unification method checking for set equality) and `tpattern` defined in `gr7.l` in the example directory which implements the semantics of tense selection.

There are limitations to the use of procedural types:

1. Procedurally typed objects are always considered as leaves in an FD: that is, no matter how complex is the object, the unifier does not know how to traverse it from the outside. It is viewed as a black box. There is no notion of ‘‘path’’ within the object.
2. Typed objects can only be unified with objects declared of the same type.
3. It is the responsibility of the user to make sure that the unification method actually implements a real partial-order.

The following is a trivial (read: useless) example of how procedural types can be used. The syntax of `define-procedural-type` is described in Section 6.4.

```
;; Unification of 2 numbers is the max: order is the total order of
;; arithmetic  (which is also a partial order!)

(defun unify-numbers (n1 n2 &optional path)
  (max n1 n2))

(define-procedural-type 'num 'unify-numbers :syntax 'numberp)

> (u '((num 1)) '((num 2)))
((num 2))

> (u '((num 1)) '((num 0)))
((num 1))

;; Unification of 2 lists is the list with the more elements.
;; That defines a (probably not very useful) total order on lists.
(defun unify-lists (l1 l2 &optional path)
  (if (> (length l1) (length l2))
      l1
    l2))

(define-procedural-type 'list 'unify-lists :syntax 'sequencep)

> (u '((list (1 2 3))) '((list (1 2 3 4))))
((list (1 2 3 4)))

> (u '((list (1 2))) '((list (a b c d))))
((list (a b c d)))
```

6.2. Typed Features: define-feature-type

6.2.1. Type definition

Typed features are hierarchies of symbols which are interpreted by the unifier. They are defined by using the function `define-feature-type`.

```
(DEFINE-FEATURE-TYPE <name> <children>)
-> Asserts that <children> are the immediate specializations of <name> in a
   type hierarchy.

Example:

(define-feature-type mood (finite non-finite))
;; finite and non-finite are specializations of the mood symbol.

(define-feature-type epistemic-modality (fact inference possible))
;; The symbols fact, inference and possible are specializations of the
;; epistemic-modality symbol
```

The function `subsume` tests if a symbol (or an object in general) is a specialization of another symbol.

```
(SUBSUME <symbol> <specialization>)
-> T is <specialization> is a specialization of <symbol>
   NIL otherwise.

Example:  (subsume 'mood 'finite)
          -> T

          (subsume 'epistemic-modality "must")
          -> T

          (subsume 'finite 'mood)
          -> NIL
```

The function `reset-typed-features` resets the working space and deletes all feature type definitions from memory. It is recommended to call it before loading a new grammar to avoid any side effect from previously defined types.

```
> (reset-typed-features)
```

6.2.2. The under Construct

When a type hierarchy is defined, it is possible to check if an input value is more or less ‘‘instantiated’’ within the hierarchy. Using the `under` construct, one can check if a value is more specific than a symbol within a hierarchy. The syntax is the following:

```
((a #(under <v>)))  will unify with ((a w)) only if w is a specialization
of v or v itself.

Example:

> (define-feature-type a (aa ab ac))
> (define-feature-type aa (aaa aab))

> (u '((x #(under aa))) '((x a)))     ;; a is not a specialization of aa
:fail

> (u '((x #(under aa))) '((x nil)))   ;; nil is the least specific of all
:fail                                 ;; it is not a specialization of aa

> (u '((x #(under aa))) '((x aab)))   ;; Ok, aab is a specialization of aa
((x aab))

> (u '((x #'(under z))) '((x z)))     ;; Even if z is not in a hierarchy
((x z))                               ;; can check for its presence
```

NOTE: when `under` is used with a symbol `z` which is not part of a type hierarchy, `#(under z)` unifies with `z` only. In particular, it will NOT unify with `NIL`. So the expression `((a #(under z)))` is equivalent to the expression `((a given) (a z))`. In fact, `under` is the typed extension of the notion of `given`. Note that `((a #(under z)))` is the equivalent of LFG’s notation $a =_c z$.

6.3. Typed Constituents: the FSET Construct

The `fset` special attribute expresses a completeness constraint in an FD.

```
;; Value of FSET is a list of symbols.
((a  ((fset (x y z))
      (x 1))))
```

`FSET` specifies that all symbols which are NOT listed is its value have a value of `NONE`, that is, they are not defined at this level of the fd.

```
;; b is not in the fset of a
> (u '((a ((fset (x y z))))) '((a ((b 1)))))
:fail
```

Two `FSET` descriptions can be unified. The result is an `FSET` whose value is the intersection of the two values.

```
> (u '((fset (x y z))) '((fset (v w x z))))
((fset (x z)))

;; ((fset nil)) is equivalent to NONE (no feature accepted)
> (u '((a ((fset (x y))))) '((a ((fset (a b))))))
((a none))
```

6.4. Procedural Types

Procedural types are defined by a name and a special unification method, optionally a syntax checker function can be declared. The unification method actually defines a partial order over the elements of the type.

```
(DEFINE-PROCEDURAL-TYPE <name> <function> :syntax <checker> :copier <copier>)

Declares <name> to be a special attribute, whose value can only be
interpreted by <function>.

The special types are considered "atomic" types (unifier cannot access to
components from outside).

The unification procedure must be deterministic (no backtracking
allowed) and must be a real "unification" procedure: that is, the type must
be a lattice (or partial order).

<FUNCTION> must be a function of 3 args: the vals to unify and the path
where the result is to be located in the total fd.
It must return :fail if unification fails, otherwise, it must return a
valid object of type <type>.

NOTE: <FUNCTION> must be such that NIL is always acceptable as an
argument and is always neutral, ie, (<FUNCTION> x nil) = x.
NOTE: <FUNCTION> must be such that (<FUNCTION> x x) = x

<CHECKER> must be a function of 1 arg:
It must return either True if the object is a syntactically correct
element of <TYPE>, otherwise, it must return 2 values:
NIL and a string describing the correct syntax of <TYPE>.

<COPIER> must be a function of 1 arg:
it must copy an object of <TYPE> that has no cons in common with its
argument.  By default, COPY-TREE is used.

NOTE: (<COPIER> x) = (<FUNCTION> x nil)
```

The following example shows one use of procedural types:

```
;; Unification of 2 numbers is the max: order is the total order of
;; arithmetic  (which is also a partial order!)

(defun unify-numbers (n1 n2 &optional path)
  (max n1 n2))

(define-procedural-type 'num 'unify-numbers :syntax 'numberp)

> (u '((num 1)) '((num 2)))
((num 2))

> (u '((num 1)) '((num 0)))
((num 1))

;; Only values of the attribute num can be unified together...
;; a and num are not compatible!
> (u '((num {a})) nil)
:fail
```

7. EXTERNAL and Unification Macros

The `External` specification allows the grammar writer to produce the constraints of a grammar in a ‘‘lazy’’ way, specifying pieces of the grammar only when needed. `External` also provides a way of developing ‘‘macros’’ in a grammar.

The mechanism is the following: `(u x <external>)` stops the unification, call the external function specified in the external construct, and uses the value returned to continue as in `(u x <value>)`. External functions expect one argument: the path where the value they return will be used.

The syntax is the following:

```
((a EXTERNAL))

or

((a #(EXTERNAL <function>)))
```

In the short form (`external` only), the external function used is the value of the variable `*default-external-value*`. Otherwise, the name of the function is explicitly specified.

There are two reasons to use an `external` construct:

1. The same portion of the grammar is repeated over and over in different places. Extract this repeating portion, give it a name as a portion, and use the function as a ‘‘macro’’ in the grammar. An example of this sort can be found in file gr6.l in the example directory. The macro is called `role-exists`.

2. There are constraints that are better expressed at run-time, when some other parameters, external to the unification process, have been calculated. The `external` construct actually allows a coroutine-like interaction between two processes. This can be used for example to implement a cooperation similar to the one described in the TELEGRAM system [2] between a planner and the unifier. A similar mechanism can be used in the following setting: a unification-based lexical chooser must interact with a knowledge base to decide what lexical items to use. The input given to the lexical chooser only contains pointers to concepts in the knowledge base. When the lexical chooser must make a decision, it needs more information from the knowledge base. The `external` construct allows the lexical chooser to pull information from the knowledge base *only when it needs it*. Therefore, the input does not need to contain all the information that might be needed but only the entry-points to the knowledge base necessary to identify the additional information that may turn out to be relevant under certain conditions.

8. Tracing

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

8.1. External vs. Internal Traces: switches

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

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

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

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

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

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

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

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

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

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

(trace-disable-all)   disable all flags.

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

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

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

(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
```

8.2. Tracing of alternatives and options

The most useful trace of the unification is generated by giving a name to an alternative of the grammar. It is done by adding an atomic name after the keywords `alt`, `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 with PASSIVE) ...)`.) The effect will be that all unification done subsequently will be traced, producing the following output:

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

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

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

The possible messages printed when the grammar is traced are:

```
Move in the alternatives:
        ENTERING ALT f: BRANCH #i
        FAIL IN ALT f
        When the alt is indexed (cf section 9):
        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
```

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

8.3. Local tracing with boundaries

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

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

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

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

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

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

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

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

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

8.5. 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:

```
--> Entrering 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.

9. Indexing and Complexity of grammars

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

9.1. Indexing

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

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

The program allows indexing of `alt` and `ralt` constructs.[6] The indexing tells the unifier how to choose one branch out of the alternation based on the value of the index only, and without considering the other branches ever. The following example illustrates the technique.

```
Example taken from gr4

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

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

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

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

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

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

A grammar is always indexed at the top-level by the `cat` feature (or the currently set cat attribute). It makes more sense to index on the features that will be bound in the input or at the moment the `alt` or `ralt` will get tried, but it never hurts to index an `alt`, so it is recommended to index whatever is indexable.

Note the syntax of an `alt` or `ralt` constructs:

```
(alt { traceflag | (trace on traceflag) | (:trace flag) }
     { (index on path) | (:index path)}
     { (demo str) | (:demo str)}
     { (bk-class class) | (:bk-class class) }
     { (:order {:random | :sequential}) }
     { (:wait <list>) }
     { (:ignore-unless <list>) }
     { (:ignore-when   <list>) }
     ( fd1 ... fdn ))

The annotations (trace, index, bk-class, order, wait and ignore)
can be in any order.
```

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

```
Example taken from gr4:

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

If the index identifies more than one branches in the disjunction, then the index will serve to prune the disjunction from all the banches that do not match the index, and the search will continue with the remaining branches as usual. For example:

```
((alt (index on cat)
   (((cat clause)
     (mood declarative)
     ...)
    ((cat clause)
     (mood non-finite)
     ...)
    ((cat np)
     ...)))
  ...)
```

When this disjunction is unified with the input `(cat clause)`, two branches are retained by the index matcher (branches 1 and 2). Branch 3 does not match the index and is therefore eliminated. The unifier will then proceed with the alt as if it only contained branches 1 and 2.

Note that if a branch does not contain a bound value for the index, it is as if it contains the value NIL, and it will therefore always retained in the alt after the index matching.

9.2. Complexity

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

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

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

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

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

10. 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.

10.1. Lexical categories are not unified

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

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

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

10.2. 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.

10.3. 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.

10.4. **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.
```

11. The Dictionary

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

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

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

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

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

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

The default value starts as follows:

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

12. Control in FUF

Pure functional unification can be too slow for practical tasks. FUF 5.0 offers several control tools that allow the grammar writer to make a grammar more efficient. This section summarizes very briefly how control is specified in FUF 5.0.

The approach has been to add annotations to the grammars that can be used by the unifier to improve performance. In a sense, this annotation approach is similar to the "optional type declarations" in Common Lisp. An important constraint that has been maintained is that the annotations do not change the semantics of the grammar, but uniquely the order in which the unifier processes it.

All the control annotations are applied to disjunctions. From the measurements made on FUF working on large grammars, it was found that optimization of conjunctions was not necessary, as the average length of conjunctions over the course of a unification is quite small (on the order of 10) and time spent processing them is quite small. In contrast, the overhead associated with dealing with disjunctions is quite high.

The control techniques for disjunctions implemented in FUF are the following:

- indexing
- dependency-directed backtracking
- lazy-evaluation / freeze
- conditional evaluation

In addition to these control mechanisms, a series of "impure" mechanisms ease the integration of unification-based processing in a larger practical system:

- type hierarchies and procedural typing
- external functions
- "given" checking

This chapter explains and gives examples on how the control features of FUF operate.

12.1. Index

Indexing is also discussed in Section 9. We discuss it in the general context of specifiying control in FUF.

There are many occurrences of disjunctions where the choice between the different branches of the disjunction can be determined based on the value of a single "key" attribute. For example:

```
(alt (
       ((person first)
        ...)
       ((person second)
        ...)
       ((person third)
        ...)
     ))
```

In this case, the `person` attribute can be used as a key to ‘‘jump’’ to the right branch when it is available. To indicate such cases, FUF allows the use of the "index" annotation:

```
        (alt (:index person) ...)
```

The way indexing works is as follows: if when the disjunction (alt) is unified the current input has an instantiated value for the attribute person, then based on this value the unifier will only consider the correct branch and will not use any backtracking point evaluating this disjunction. If the value of person is not instantiated yet, then the disjunction is unified normally.

This is an easy and trivial thing to say about grammars but it does boost the performance dramatically, as it avoids all the overhead associated with putting and removing backtracking points.

Note that the index does not have to be at the toplevel of the branches (you can index on an embedded attribute, any path will do), and does not have to uniquely discriminate among the branches - if you have a disjunction with projections {a,a,b,c,d} on the indexed attribute and the current value of the attribute is a then the disjunction will remain but will be filtered as just {a, a}.

INDEX is used in all examples and is easy to add to existing grammars. No grammar should be left without INDEX.

12.2. Dependency-directed Backtracking and BK-CLASS

The top-down regime implemented in FUF generally handles the semantic constraints found in the input efficiently. However, the input to a text generator must also include pragmatic constraints that are inherently non-structural. What makes these constraints non-structural is that they may be expressed at different levels of the syntactic tree. They can therefore trigger non-local backtracking, beyond the scope of a single constituent. Such non-local backtracking can be very inefficient because, when the failure occurs, there is a large number of decision points in the stack between the choice that caused the failure and the current frame.

The situation can be illustrated by the following schema, where each frame corresponds to a decision made by the unifier regarding a non-deterministic construct found in the grammar (each frame is a backtracking point):

```
| Frame TOP  |  ---------> failure because of feature P
| Frame T100 |  )
| Frame T099 |  )
...          )----> Lots of decisions that do not
...          )      affect the value of P.
...          )
| Frame T016 |  )
| Frame T015 |  ---------> feature P is set to wrong value
| Frame T014 |
...
| Frame T001 |
```

To avoid the cost of a blind backtracking, we introduce the `bk-class` construct. `bk-class` implements a version of dependency-directed backtracking [3] specialized to the case of FUF. `bk-class` relies on the fact that in FUF, a failure always occurs because there is a conflict between two values for a certain attribute, at a certain location in the total FD. We want to be able to express the fact that in the Figure above, frame TOP depends on the decision made in frame T015 and not on any of the intermediary decisions.

Remind that a failure can only occur in a leaf of the graph representing the total FD, when two atomic values conflict for the same attribute. The idea is that the location of certain failures can be used to identify the only decision points in the backtracking stack that could have caused the failure. This identification requires additional knowledge that must be declared in the FUG. More precisely, we first allow the FUG writer to declare certain paths to be of a certain `bk-class`. We then require the explicit declaration in the FUG of the choice points that correspond to this `bk-class`.

For example, the following statements are used in grammar gr7:

```
(define-bk-class 'manner-conveyed 'manner)
(define-bk-class 'manner 'manner)
(define-bk-class 'lexical-verb '(ao manner))
(define-bk-class 'ao-conveyed 'ao)
(define-bk-class 'ao 'ao)
```

The first one specifies that any path ending with the symbol `manner-conveyed` is of class manner. In addition, we tag in the FUG all `alts` that have an influence on the handling of the manner constraint with a declaration `(bk-class manner)` as in:

```
;; In GR7 -- CLAUSE branch -- ADJUNCTS region
(alt manner (:demo "Is there a manner role?")
  (:bk-class manner)
  (((manner none))
   ;; can it be realized by other means - delay with ANY
   ((manner ((manner-conveyed any))))
   ;; if cannot be realized any other way, resort to an adverb
   ((manner-comp ((cat adv)
                  (concept {^ ^ manner concept})
                  (lex {^ ^ manner lex})))
    (manner ((manner-conveyed yes)))
    (pattern (dots manner-comp verb dots)))
   ;; or to a pp
   ((manner-comp ((cat pp)
                  (lex {^ ^ manner lex})
                  (concept {^ ^ manner concept})
                  (opt ((prep ((lex "with")))))))
    (manner ((manner-conveyed yes)))
    (pattern (dots verb dots manner-comp dots)))))
```

This fragment extracted from grammar GR7 illustrates a possible use of the `bk-class` construct. The example is detailed in a paper accompanying the manual (available in file .../doc/bk-class). The fragment above implements the following decisions: if there is no `manner` role specified in the input, then nothing needs to be done. If there is a manner role, then the grammar must decide how to realize it. One option is to realize it as either an adverbial adjunct or as a PP adjunct. Another option, shown in the next figure, is to realize the `manner` role by selecting a verb which would carry the manner meaning. The three possibilities are illustrated by the three sentences, where the constituent realizing the manner role is in italics:

1. The Denver Nuggets *narrowly* beat the Boston Celtics 101-99.

2. The Denver Nuggets beat the Boston Celtics *by a slight margin* 101-99.

3. The Denver Nuggets *edged* the Boston Celtics 101-99.

The fragment of the grammar implementing the choice of the verb is shown in the next figure:

```
;; Lexicon for verbs: mapping concept - lex + connotations
((cat lex-verb)
 (alt verbal-lexicon (:demo "What lexical entry can be used for the verb?")
   (:index concept)
   (:bk-class (voice-class transitivity))
   (
    ;; Need to express the result of a game
    ((concept game-result)
     (transitive-class transitive)
     (voice-class non-middle)
     (process-class action)

     (alt (:bk-class (ao manner))
       (
        ;; The verbs edge or nip express the manner
        (({manner} ((concept narrow)
                    (manner-conveyed yes)))
         (lex ((ralt ("edge" "nip")))))

        ;; The verbs stun or surprise express an evaluation of agent
        (({AO} ((concept rating)
                (carrier {agent})
                (orientation -)
                (ao-conveyed yes)))
         (lex ((ralt ("stun" "surprise")))))

         ;; Neutral verbs
         ((lex ((ralt ("beat" "defeat" "down"))))))))

    ;; More verbs for other concepts
    ....
    ((concept move)
     (lex ((ralt ("walk" "run"))))))))
```

Examples of input that exercise these parts of the grammar are provided in file ir7. A simple example is shown in the next figure:

```
(defun t1 ()
  (format t "t1 --> The Denver Nuggets edged the Celtics.~%")
  (setf t1 '((cat clause)
             (process-type action)
             (process ((concept game-result)))
             (agent ((cat proper)
                     (lex "The Denver Nugget")
                     (number plural)))
             (medium ((cat proper)
                      (lex "the Celtic")
                      (number plural)))
             (tense past)
             (manner ((concept narrow))))))
```

The overall flow of control through this grammar concerning the realization of the manner role is as follows:

The unifier first processes the input at the clause level. When it reaches the region dealing with adjuncts, it checks the `manner` role. There is a `manner` role in input `t1`, so the grammar has a choice between the three modes of realization listed above - verb, adverb or PP. The grammar first tries to ‘‘leave a chance to the verb’’ - more precisely, it leaves a chance to some other unspecified constituent in the grammar to account for the `manner` role. If the verb happens to convey the manner, then nothing else needs to be done. This case is illustrated by the following trace of unification of example `t1` in file `ir7`.

```
Trace of example t1

;; Set up the traces
> (load "$fug5/examples/gr7")
> (load "$fug5/examples/ir7")
> (trace-disable-all)
> (trace-enable-match "manner")
> (trace-enable-match "game-result-lex")
> (trace-bk-class t)
;; Run example
> (uni (t1))

t1 --> The Denver Nuggets edged the Celtics.
>STARTING CAT CLAUSE AT LEVEL {}

-->Entering alt MANNER -- Branch #1
-->Fail in trying ((CONCEPT NARROW) (LEX "narrowly")) with NONE
   at level {MANNER}
>Special path {MANNER} caught by class (MANNER) after 1 frame

-->Entering alt MANNER -- Branch #2
-->Updating (MANNER-CONVEYED NIL) with ANY at level {MANNER MANNER-CONVEYED}
-->Success with branch 2 in alt MANNER

>Special path {VERB TRANSITIVE-CLASS} caught by class (TRANSITIVITY)
 after 1 frame

>STARTING CAT PROPER AT LEVEL {SUBJECT}
>STARTING CAT VERB-GROUP AT LEVEL {VERB}
>STARTING CAT PROPER AT LEVEL {OBJECT}
>STARTING CAT LEX-VERB AT LEVEL {VERB LEXICAL-VERB}

;; choose a verb that expresses the manner
-->Entering alt GAME-RESULT-LEX -- Branch #1
-->Fail in trying NONE with RATING at level {AO CONCEPT}
-->Entering alt GAME-RESULT-LEX -- Branch #2
-->Updating (MANNER-CONVEYED ANY) with YES at level {MANNER MANNER-CONVEYED}
-->Updating (LEX NIL) with "nip" at level {VERB LEX}
-->Success with branch 2 in alt GAME-RESULT-LEX

[Used 59 backtracking points - 12 wrong branches - 0 undos]
The Denver Nuggets nipped the Celtics.
```

But if for any reason (like the interaction between manner and argumentation detailed in the bk-class paper and illustrated in examples found in file ir7 or because the input already specifies the verb) the grammar fails to account for the `manner` role in other constituents, the `ANY` constraint put under the `manner-conveyed` feature will fail at determination time. At this time, we know we failed because of the `manner-conveyed` feature, so if we backtrack, we want to backtrack to the latest choice that involved the `manner` role realization. With the `bk-class` annotations, the unifier will backtrack *directly* to the last choice point of class `manner`, ignoring all intermediate decisions. This case is illustrated by the trace of the unification of example `t2` from file ir7. In `t2` the verb is specified and set to be ‘‘beat’’ which does not convey the `manner` role. In this case, the unifier needs to backtrack as illustrated:

```
> (uni (t2))
t2 --> The Denver Nuggets narrowly beat the Celtics.
>STARTING CAT CLAUSE AT LEVEL {}

;; First leave a chance to the verb
-->Entering alt MANNER -- Branch #1
-->Fail in trying ((CONCEPT NARROW) (LEX "narrowly")) with NONE
   at level {MANNER}
-->Entering alt MANNER -- Branch #2
-->Updating (MANNER-CONVEYED NIL) with ANY at level {MANNER MANNER-CONVEYED}
-->Success with branch 2 in alt MANNER

>STARTING CAT PROPER AT LEVEL {SUBJECT}
>STARTING CAT VERB-GROUP AT LEVEL {VERB}
>STARTING CAT PROPER AT LEVEL {OBJECT}
>STARTING CAT LEX-VERB AT LEVEL {VERB LEXICAL-VERB}

;; Now choose the verb: it must be beat
-->Entering alt GAME-RESULT-LEX -- Branch #1
-->Fail in trying NONE with RATING at level {AO CONCEPT}
-->Entering alt GAME-RESULT-LEX -- Branch #2
-->Updating (MANNER-CONVEYED ANY) with YES at level {MANNER MANNER-CONVEYED}
-->Fail in trying "beat" with "nip" at level {VERB LEX}
-->Fail in trying "beat" with "edge" at level {VERB LEX}
-->Entering alt GAME-RESULT-LEX -- Branch #3
-->Updating (LEX "beat") with "beat" at level {VERB LEX}
-->Success with branch 3 in alt GAME-RESULT-LEX

;; Problem now: manner is not conveyed! Backtrack...
[Used 64 backtracking points - 17 wrong branches - 1 undo]
>Fail in Determine: found an ANY at level {MANNER MANNER-CONVEYED}
>CURRENT SENTENCE: The Denver Nuggets beat the Celtics.

;; Backtrack because of manner-conveyed
>Special path {MANNER MANNER-CONVEYED} caught by class (AO MANNER)
 after 2 frames

;; Which makes lexical-verb fail:
-->Fail in alt GAME-RESULT-LEX at level {VERB LEXICAL-VERB}

;; Now We skip 23 frames on the stack! We go up DIRECTLY to the choice of manner: express it as an adverb
>Special path {VERB LEXICAL-VERB} caught by class (MANNER) after 23 frames
-->Entering alt MANNER -- Branch #3
-->Updating (CAT NIL) with ADV at level {MANNER-COMP CAT}
-->Enriching input with (CONCEPT {MANNER CONCEPT}) at level {MANNER-COMP}
-->Enriching input with (LEX {MANNER LEX}) at level {MANNER-COMP}
-->Updating (MANNER-CONVEYED NIL) with YES at level {MANNER MANNER-CONVEYED}
-->Unifying (DOTS START DOTS) with (DOTS MANNER-COMP VERB DOTS)
-->Trying pattern : (DOTS START DOTS MANNER-COMP VERB DOTS)
-->Adding constraints : NIL
-->Success with branch 3 in alt MANNER

;; Redo intermediary decisions
>STARTING CAT PROPER AT LEVEL {SUBJECT}
>STARTING CAT VERB-GROUP AT LEVEL {VERB}
>STARTING CAT PROPER AT LEVEL {OBJECT}
>STARTING CAT LEX-VERB AT LEVEL {VERB LEXICAL-VERB}

;; Choose verb again
-->Entering alt GAME-RESULT-LEX -- Branch #1
-->Entering alt GAME-RESULT-LEX -- Branch #2
-->Entering alt GAME-RESULT-LEX -- Branch #3
-->Updating (LEX "beat") with "beat" at level {VERB LEX}
-->Success with branch 3 in alt GAME-RESULT-LEX

;; This time it works
[Used 106 backtracking points - 58 wrong branches - 137 undos]
The Denver Nuggets narrowly beat the Celtics.
```

Note that in this second example, if the bk-class feature is disabled (by typing (clear-bk-class)), the unifier does not find an acceptable solution after 20,000 backtracking points. This indicates that bk-class is not merely an optimization but is often required to make unification practical.

In the case of non-structural constraints like argumentation or manner as illustrated in grammar gr7 and input ir7, the dependency-directed mechanism implemented in FUF with `bk-class` nicely complements a general top-down control regime. This mechanism improves FUGs efficiency while preserving their desirable properties - declarativity and bidirectional constraint satisfaction.

12.3. Lazy Evaluation and Freeze with wait

The `bk-class` mechanism is useful in general to correct directly a decision made a long time ago, as soon as it appears that the decision was wrong. But this implies that all the work done between the wrong decision and the realization that it is wrong must be re-done once the original wrong has been corrected. For example, in the following case:

```
Frame TOP   ---------> failure because of feature P
Frame T100  )
Frame T099  )
...        )----> Lots of decisions that do not
...        )      affect the value of P.
...        )
Frame T016  )
Frame T015  ---------> feature P is set to wrong value
Frame T014
...
Frame T001
```

The decisions corresponding to frames T016 to T100 are going to be done twice because T015 made the wrong choice originally. While making a decision twice is not too bad compared to the cost of exhaustively searching all the choices corresponding to frames T016 to T100 backwards, it is still sub-optimal.

One could reach a more efficient solution if the decision corresponding to frame T015 is delayed and the unifier commits to a value for P only after frame T100. The `wait` control specification is used to achieve this effect under certain conditions.

Wait specifies that the decision corresponding to a disjunction depends on the value of certain features. The syntax is as follows:

```
(alt TP (:wait P) ...)
```

This declaration indicates to the unifier that the alt TP depends on the value of the feature P. If when it is first met feature P is instantiated, then the alt is evaluated normally. If however P is not yet instantiated, the whole disjunction is delayed: it is put on hold, on an agenda. The rest of the grammar is evaluated, and periodically, the unifier checks the agenda to determine if one of the frozen alts can be awakened.

The following example illustrates a possible use of `wait`:

```
;; In conjunction: do verb ellipsis?
;; If the same verb is used in both conjuncts
;; you can ellide it in the second conjunct:
;; --> John ate the burger and Mary the fish.
;; The ellipsis is done by adding a GAP feature to the second verb.
;; Wait until verbs for both conjuncts have been selected.
;; This alt does NOT determine what the verbs should be.
(alt verb-ellipsis (:wait {constituent1 process lex}
                          {constituent2 process lex})
  (((cat clause)
    ({^ constituent1 process lex} {^ ^ ^ ^ constituent2 process lex})
    ({^ constituent2 process gap} yes)
    (verbal-ellipsis yes))
   ((verbal-ellipsis no)))))
```

This fragment is extracted from grammar gr10. It is part of the branch of the grammar dealing with conjunction. This alt implements the decision whether to use a verb ellipsis in a conjunction of two clauses. The verb in the second conjunct can be elided if it the same lexical entry as the verb in the first conjunct. This match is checked by testing that the {constituent1 process lex} path can be unified with the {constituent2 process lex} path. When it can be unified, the {constituent2 process} is enriched with the feature (gap yes), which the linearizer will interpret by skipping the verb in the final sentence.

Now remember that the standard control flow in FUF is top-down breadth-first in the tree of constituents. A conjunction of clauses is made up of the following constituents:

```
COMPLEX-CLAUSE
   |
   |---- ELEMENTS
   |        |
   |        |--- CONSTITUENT1
   |        |        |
   |        |        |--- PROCESS
   |        |        |--- PARTICIPANTS
   |        |        |--- SYNT-ROLES
   |        |
   |        |--- CONSTITUENT2
   |        |        |
   |        |        |--- PROCESS
   |        |        |--- PARTICIPANTS
   |        |        |--- SYNT-ROLES
   |        |        |
```

When the branch of the grammar specifying the conjunction is traversed, the choice of the lexical item that will realize the process is still three levels down. The decision whether to use an ellipsis concerns only the conjunction grammar, so it must be located in the conjunction branch, not in the branch of the grammar deciding on lexical choice for the process. But it depends on this later decision - we do not want to influence the choice of the verbs in the conjunct by the fact that they are conjoined. So the decision is instead delayed until the verbs in both conjuncts have been determined.

Note that if the input already specifies the verbs, then the decision can be evaluated right away. It is only delayed when needed.

There can be complications when using `wait` when two decisions wait for each other, in a deadlock configuration. I have not yet encountered such situations in practice but in such cases, a combination of `wait` and `bk-class` could ensure that the deadlock can be broken in an efficient way.

12.4. Conditional-evaluation with ignore

The `ignore` control annotation allows the grammar writer to evaluate certain decisions only under certain conditions. The idea is to just ignore certain decisions when there is not enough information, there is already enough information or there is not enough resources left. The 3 cases correspond to the annotations:

```
(alt (:ignore-when <path> ...) ...)
(alt (:ignore-unless <path> ...) ...)
(alt (:ignore-after <number>) ...)
```

`Ignore-when` is triggered when the paths listed in the annotation are already instantiated. It is used to check that the input already contains information and the grammar does not have to re-derive it.

`Ignore-unless` is triggered when a path is not instantiated. It is used when the input does not contain enough information at all, and the grammar can therefore just choose an arbitrary default. A real example of the use of `ignore-unless` is given below.

`Ignore-after` is triggered after a certain number of backtracking points have been consumed. It indicates that the decision encoded by the disjunction is a detail refinement that is not necessary to the completion of the unification, but would just add to its appropriateness or value. IGNORE-AFTER IS NOT IMPLEMENTED IN FUF5.0.

The main problem with these annotations is that their evaluation depends on the order in which evaluation proceeds, and that this order is not known to the grammar writer. But in conjunction with `wait`, this issue is not necessarily a problem as `wait` establishes constraints on when a decision is evaluated. However, this indicates that you have to be EXTREMELY careful with these annotations, as they affect the semantics of the grammar.

The following example is extracted from grammar gr10. It is actually the same case of verb ellipsis in conjunction presented in the section on `wait` above. In this decision, we also want to specify that the whole decision of whether to use ellipsis or not is only relevant in the case of a conjunction of clauses. This is expressed by addition the `ignore-unless` annotation as shown in the following figure:

```
;; In conjunction: do verb ellipsis?
;; This decision ONLY applies to CLAUSEs.
;; Ignore it in conjunctions of NPs and other cats.
(alt verb-ellipsis (:wait process)
  (:ignore-unless ((cat clause)))
  (((cat clause)
    ({^ constituent1 process lex} {^ ^ ^ ^ constituent2 process lex})
    ({^ constituent2 process gap} yes)
    (verbal-ellipsis yes))
   ((verbal-ellipsis no)))))
```

13. Reference Manual

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

13.1. Unification functions

13.1.1. *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)`

13.1.2. *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.

13.1.3. *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 specifiy 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.

13.1.4. u

Type: function

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

Arguments:

- `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.)

13.1.5. 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 13.1.4 for an explanation of the limit argument.

13.1.6. 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 13.1.3 for an explanation of the `cat-attribute` argument. Refer to paragraph 13.1.4 for an explanation of the `limit` argument.

13.1.7. 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.

13.1.8. 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 13.1.3 for an explanation of the `cat-attribute` argument. Refer to paragraph 13.1.4 for an explanation of the `limit` argument.

13.1.9. 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`.

13.1.10. 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 13.1.4 for an explanation of the `limit` argument.

13.1.11. 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 funtion 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 13.1.4 for an explanation of the `limit` argument.

13.1.12. 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 13.1.4 for an explanation of the `limit` argument.

13.2. Checking

13.2.1. 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.

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

message	condition
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.[7]
An EXTERNAL pair must be of the form (ATT external-spec): ~s	External-spec can be simply the symbol `external` or a construct `#(external <fctn>)`.
--- Warning: The argument of external must be a function.	In a `#(external <fctn>)`, <fctn> is not recognized as a defined function.

[7]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 UNDER specification must be an array of the form #(UNDER symbol): ~s	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 **#(under <x>)**, <x> must be a symbol defined in a type hierarchy.
--- Warning: The argument of under does not have specializations defined. Use `(define-feature-type ~s (spec1 ... specn))`.	In a **#(under <x>)**, <x> must be a symbol defined in a type hierarchy. If the type hierarchy does not exist, it can be defined with the `define-feature-type` 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.

13.2.2. 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`).

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

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 refering 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. Not 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.

13.2.3. 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.

13.2.4. 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.

13.2.5. 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`.

13.2.6. 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.
```

13.3. Tracing

13.3.1. *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%

13.3.2. *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%

13.3.3. *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`

13.3.4. *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: `#\%`

13.3.5. *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`

13.3.6. 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.

13.3.7. 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.

13.3.8. 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.

13.3.9. 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.

13.3.10. trace-disable-all

Type: function

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

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

13.3.11. 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.

13.3.12. 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.

13.3.13. trace-enable-all

Type: function

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

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

13.3.14. 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.

13.3.15. 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.

13.3.16. trace-on

Type: function

Calling form: (`trace-on`)

Description: `trace-on` turns on tracing.

Initially, tracing is off.

13.3.17. 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.

13.3.18. 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 backtracking point of the corresponding class. In addition, the number of frames that have been skipped thanks to the bk-class specification is printed. Examples of trace are provided in the section on bk-class.

13.3.19. trace-category

Type: function

Calling form: (`trace-category` *cat* `&optional` *on-off*)

Arguments: `cat` can be either a symbol (the name of a category), or the symbol `:all` specifying that all categories are to be traced, or a list of symbols (names of categories). If `on-off` is non-nil, the specified categories are traced, if it is `nil`, they are untraced.

Description: `trace-category` turns on and off tracing for all or certain categories.

When a category is traced, the unifier emits a message each time it starts unifying a constituent of that category. When :all is used, all categories are traced. This is particularly useful to identify in what constituent the unifier is failing and to follow the top-down breadth-first traversal of the constituents during unification. A usefule sequence of calls is:

```
(trace-on)
(trace-disable-all)
(trace-category :all)
```

This sequence will trace the progression of the unifier at the level of the constituents, as in the following example:

```
> (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.
```

13.4. Complexity

13.4.1. avg-complexity

Type: function

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

Arguments:

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

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

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

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

13.4.2. complexity

Type: function

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

Arguments:

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

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

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

13.5. Manipulation of the dictionary

13.5.1. *dictionary*

Type: variable

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

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

The entries contain the following properties:

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

13.5.2. lexfetch

Type: function

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

Arguments:

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

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

13.5.3. lexstore

Type: function

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

Arguments:

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

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

13.6. Linearization and Morphology

13.6.1. call-linearizer

Type: function

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

Arguments:

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

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

13.6.2. gap

Type: feature.

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

13.6.3. morphology-help

Type: function.

Calling form: `(morphology-help)`

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

13.7. Manipulation of FDs as data-structures

13.7.1. FD-intersection

Type: function

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

Arguments:

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

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

13.7.2. FD-member

Type: function

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

Arguments:

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

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

13.7.3. FD-to-list

Type: function

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

Arguments:

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

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

13.7.4. gdp

Type: function

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

Arguments:

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

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

`gdp` works only if the special variable *input* is accessible and bound to the total fd containing *fd*. In normal environments, the function `top-gdp` should be used instead.

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

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

13.7.5. gdpp

Type: function

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

Arguments:

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

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

`gdpp` works only if the special variable *input* is accessible and bound to the total fd containing *fd*. In normal environments, the function `top-gdpp` should be used instead.

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

13.7.6. top-gdp

Type: function

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

Arguments:

- *fd* is an fd.
- *path* is a valid path structure.

description: works like `gdp` but considering fd as the total fd. Can therefore be used even if `*input*` is not bound in the current environment. This is the user-level function to access constituents of fds. .

13.7.7. top-gdpp

Type: function

Calling form: (`top-gdpp` *fd path*)

Arguments:

- *fd* is an fd.
- *path* is a valid path structure.

description: works like `gdpp` but considering fd as the total fd. Can therefore be used even if `*input*` is not bound in the current environment. This is the user-level function to access pairs in fds. .

13.7.8. alt-gdp

Type: function

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

Arguments:

- *fd* is an fd.
- *path* is a valid path structure with disjunction specifiers.

description: works like `gdp` but works even if fd contains disjunctions (`alt`, `ralt` or `opt` constructs). Paths when disjunctions are allowed are different from paths in regular fds. They 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 refering 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. Not 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.

This function is less robust than `gdp`, it does not check for cycles, does not check for FSETs violations and does not check for user-defined special types.

13.7.9. list-to-FD

Type: function

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

Arguments:

- *list* is a regular lisp list.

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

Note: The notation {l ~3 a} refers to the third element of the list l if l is a list in FD form. That is, the path {l ~3 a} is equivalent to {l cdr cdr car a}.

13.8. Fine tuning of the unifier

13.8.1. *any-at-unification*

Type: variable

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

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

Standard Value: T

13.8.2. *use-given*

Type: variable

Description: If `*use-given*` is `T`, `given` and `under` constructs have their usual semantics. If it is `nil`, then `given` is considered as a normal symbol and a construct `#(under s)` is considered equivalent to simply `s`.

Standard Value: `T`

13.8.3. *use-any*

Type: variable

Description: If `*use-any*` is `T`, `any` constructs have their usual semantics (as determined by `*any-at-unification*`). If it is `nil`, then `any` is considered as a normal symbol.

Standard Value: `T`

13.8.4. *keep-cset*

Type: variable

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

Standard Value: `nil`

13.8.5. *keep-none*

Type: variable

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

Standard Value: `T`

13.8.6. *agenda-policy*

Type: variable

Description: Can be either `:force` or `:keep`. Determines what to do with frozen alts at the end of unification:

- If `:keep`: keep them unevaluated in result
- If `:force`: force their evaluation at determination time.

The value should only be :keep in situations where several grammars are used in a pipeline architecture, and the frozen decisions are passed along from one stage to the next.

Standard Value: `:force`

Appendix I
Installation of the Package

I.1. Finding the files

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

```
Language : Common Lisp
System   : At Columbia, available
           on Lisp-A (Symbolics), in directory >elhadad>fuf>
           on the SUN workstations (SUNOS/Lucid), in ~elhadad/Fug/freeze
           (define environment variable "fug5" to this value:
            under csh: setenv fug5 ~elhadad/Fug/freeze
            under ksh: fug5=~elhadad/Fug/freeze; export fug5)
          Examples are in the subdirectory named "examples".

Start    : on Lisp-A:  (load ">elhadad>fuf>fug")
           on the SUNs: % cl   #need to have /lisp/bin is path
                        CL> (load "$fug5/fug5")
```

The file FUG5.L will load all the required modules. Examples are in the files GR0, GR1 and up for the grammars, and in files IR0, IR1, ... and up for the inputs. The examples are of increasing complexity.

```
To try the examples, type:

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

I.2. Porting to a new machine

The program is contained in 24 files of source and 25 files of examples. All the source files should be grouped in a directory, that we will call here `$fug5`, and the example files in a subdirectory of `$fug5` called `examples`.

Once this is done, you probably need to edit the file FUG5.L. This file loads all the required modules and defines a few functions useful for compiling or loading the package. In the file FUG5.L, the function `require` is used to load all submodules. `require` takes as first argument the name of a module, and accepts a second optional argument, the name of the file containing that module.

You must change the second arguments of all the `require` statements in file FUG5.L and update there the name of the directory, from `$fug5` to the name of your directory.

You also need to edit the first line of the functions `compile-fug5` and `reload-fug5` and change there the name of the directory from `$fug5` to the new name.

When the file FUG5.L is updated, load it in your common-lisp environment and follow these 4 steps:

```
(load "$fug5/fug5")

(in-package "FUG5")

(compile-fug5)

(reload-fug5)
```

NOTE TO UNIX USERS: if you run CommonLisp under Unix, and your version of Lisp can read environment variables and expands such variables in file names (for example, `(load "~userx/file1")` is a valid statement, or `(load "$var/file2")`), then you don't need to edit the file FUG5.L. All you need to do is to define the environment variable "`fug5`" to be the complete pathname of the directory containing the source files.

Once this installation is done, all you need to do to load the package is `(load "$fug5/fug5")` (with `$fug5/` replaced by the name of your directory if you are not under Unix).

I.3. Packages

The whole package is loaded in package `'FUG5`. The easiest way to access it is to type:

```
(in-package "FUG5")   ;; note the upper-case

or

(use-package "FUG5")
```

The following symbols are exported from package 'FUG5 (they are the external symbols of the package, cf [Steele-84, chapter 11, p171-192]):

External Symbols of package FUG5	
File	Symbols
checker.l	fd-p fd-syntax fd-sem grammar-p get-error-pair
complexity.l	complexity avg-complexity
determine.l	*keep-cset* *keep-none*
external.l	*default-external-function*
graph.l	*any-at-unification* *use-given*
lexicon.l	*dictionary* lexfetch lexstore
linearize.l	call-linearizer morphology-help
path.l	gdp, gdpp top-gdp, top-gdpp alt-gdp
top.l	*u-grammar* *lexical-categories* *cat-attribute* uni uni-fd unif list-cats u u-disjunctions
trace.l	trace-on trace-off internal-trace-on internal-trace-off trace-category trace-enable trace-disable trace-enable-all trace-disable-all trace-enable-match trace-disable-match all-tracing-flags *trace-marker* *all-trace-off* *all-trace-on* *trace-determine* *top*
type.l	define-feature-type define-procedural-type reset-typed-features

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

External Symbols of package FUG5 (keywords)	
File	Symbols

top.l	`===`
	`* already exists in LISP`
	`trace already exists in USER`
	`@`
	`^`
	`alt`
	`any`
	`cat`
	`control`
	`cset`
	`dots`
	`*done*`
	`gap`
	`given`
	`index`
	`lex`
	`mergeable`
	`none`
	`opt`
	`pattern`
	`pound`
	`punctuation`
	`test`

All these symbols are documented for reference in section 13. If you use the package FUG5 in another package, only these symbols will be imported.

Appendix II
Advanced Features

II.1. Advanced Uses of Patterns

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

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

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

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

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

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

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

II.2. Advanced uses of CSET

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

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

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

II.3. Long Distance Dependencies and the GAP feature

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

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

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

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

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

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

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

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

```
Examples:
((a 1)
 (test (equal #@{a} #@{b})))

is equivalent to the nicer:

((a 1)
 (b {a}))

((a 1)
 (test (numberp #@{a})))
```

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

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

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

III.1. The member/append example

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

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

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

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

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

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

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

III.2. Representing lists as FDs

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

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

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

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

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

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

III.2.1. NIL and variables

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

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

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

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

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

III.2.2. The "~" notation

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

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

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

The ~n notation can be used to refer to elements of lists represented as FDs. The path {l ~4} refers to the fourth element of the list l. It is equivalent to the path {l cdr cdr cdr car}.

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

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

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

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

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

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

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

A program in FUG can be viewed as a collection of procedures, each procedure being represented by a category. In the member example of section III.1, an input containing the feature *(cat member)* will be sent to the *member* procedure.

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

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

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

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

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

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

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

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

III.6. Analogy with PROLOG programs

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

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

- The notion of success and failure in unification are equivalent to the "yes" and "no" of PROLOG programs.
- Simple statement can be combined using the connectives AND and OR: both FDs and PROLOG

statements make use of conjunction and disjunction.

- Both notations rely heavily on unification, and refinement of partial descriptions to perform computations.

III.7. Use of Set values in linguistic applications

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

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

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

Appendix IV
Non standard features of the implementation and restrictions

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

IV.1. Typed features

This implementation supports the definition of types of symbols as described in the section on typing.

IV.2. The FSET special attribute and typed constituents

This implementation supports the special attribute `FSET` to implement the notion of typed constituent and express completeness constraints. An fd `((fset (a b c)) (a 1))` can only have defined values (non-`none` values) for the attributes a, b and c. All other attributes are considered as having value `none`. Two `FSET` features can be unified, and the result is the intersection of the two values.

File `test1.l` in the example directory contains examples of use of `fset` feature.

IV.3. User-defined types

User defined types with special unification procedures can be defined in this implementation. Refer to the `define-procedural-type` entry in the reference manual for details. File `test2.l` in the example directory contains examples of use of procedural types.

IV.4. Limits on disjunction in input

To work best, the unifier requires the input to be a simple FD, containing no disjunction (`alt`, `ralt` or `opt`). It can contain patterns. `test`s and `control`s are not allowed in input.

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

If disjunction are found in an FD given to `fd-sem`, a warning message is printed. `fd-sem` also issues warnings if its argument contains `pattern`s or `cset`s.

It is possible to use input fds containing disjunctions in two ways: one way is to first ‘‘normalize’’ the input fd and randomly choose one fd compatible with the disjunction-full fd but containing no disjunction at all. The function `normalize-fd` performs this operation. The other way to use disjunctions in input is to use the low-level unifier function `u-disjunctions` . Note that in general, `u-disjunctions` can be extremely inefficient unless some control mechanisms are properly used.

IV.5. Mergeable constituents in patterns

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

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

IV.6. Indexing of alternation

This implementation allows indexing of `alt`s, as described in section 9. The notation used is:

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

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

IV.7. Test and Control

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

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

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

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

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

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

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

IV.8. GIVEN and UNDER

The special value `given` and the related construct `under` are defined in this implementation. A feature `(att given)` is unified with an input fd, if the input contains a real value for attribute `att` at the beginning of the unification. When typed features are defined, a feature `(att #(under symb))` is unified with an input fd, if the input contains a value for attribute `att` which is *more specific* than `symb` at the beginning of the unification.

Note that if `symb` has no specializations defined in a type hierarchy, the notation `(att #(under symb))` is equivalent to `((att given) (att symb))`.

`given` and `under` are useful to check the presence of required features in inputs.

IV.9. EXTERNAL specifications and macros

The implementation supports the `EXTERNAL` construct which provides a form of delayed dynamic expression of constraints and a mechanism for a macro facility in grammars.

A feature `(att #(external <fctn>))` in a grammar is interpreted as follows: the unifier calls the function <fctn> with one argument, the path at which the external feature is being unified. The function must return a valid fd <F>. The unifier then continue unification with this returned fd instead of the as if the feature had been `(a <F>)` in the first place.

It is therefore possible to dynamically decide, at run-time, what constraints will be used in the grammar.

IV.10. User-defined CAT parameter

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 specifiy 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.

IV.11. Resource limited processing

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.)

IV.12. BK-CLASS Control Specification

The idea behind `bk-class` is to take advantage of the knowledge of where a failure occurs in the graph being unified to filter the stack of backtracking points. Note that a failure always happens at a leaf of the graph because of a conflict between two atomic values. The path leading to the point of the failure is called the "failure address".

Note also that backtracking points are all associated with a disjunction construct in the grammar.

In FUF, you can annotate each disjunction to either catch (restart) or ignore (propagate higher on the stack) failures occurring at certain pre-declared failure addresses.

The details are:

1. Define classes of failure addresses as regular expressions on paths. These are called the bk-classes (for backtracking classes). All failures that occur at an address which is not a member of any bk-class are treated normally by all backtracking point.
2. Annotate certain disjunctions (and therefore the backtracking points they place on the stack) with the list of classes to which they are allowed to respond:

```
        (alt (:bk-class c1 ... cn) ...)
```

3. During backtracking, a backtracking point checks if the failure address is part of a bk-class, and if it is, whether it is a member of the bk-classes it is declared to process. If it is, the b-point restarts the computation (by proceeding to the next alternative in the disjunction), otherwise it just ignores the failure and propagates it to the rest of the stack.

Through this mechanism, only the decisions that are ‘‘relevant’’ to the current failure need to be undone.

In practice, this mechanism when it is applicable provides huge performance gains, but it is rarely applicable: the problem is to define the bk-classes and it is a difficult task. You need to experiment a lot with it though as in general, bk-class has lots of potential. The problem is deciding when it is applicable in the context of a big grammar.

Note that this mechanism relates to the ‘‘cut’’ of Prolog but it also has a big advantage: if the `bk-class` annotations are done consistently (which cannot be checked automatically), then the semantics of the grammar is not modified by the annotation in the sense that all the results produced by the ‘‘pure grammar’’ would also be produced by the grammar with the annotation.

Note also that this mechanism is implemented in such a way that when it is not used in the grammar, there is very little overhead in processing time.

An example of use of bk-class is provided in grammar GR7 .

IV.13. WAIT Control Specification

The idea is to freeze a decision until enough information is gathered by the rest of the unification process to make the decision efficiently. The annotation is:

```
        (alt (:wait (<path1> <value1>) ...) ...)
```

When the unifier meets such a disjunction, it first checks if all the `pathi` have a value which is instantiated (`valuei` is used with typed symbols). If they are all instantiated, it proceeds as usual. If they are not, then the disjunction is put on hold (frozen) and the unification proceeds with the other constraints in the grammar. Whenever a toplevel disjunction is entered, the unifier checks if one of the frozen disjunctions can be thawed. If all the decisions to be made are frozen, any one is forced.[8] `Wait` is probably the most powerful of all control constructs. It is just implemented and there is not much experience with it but it appears to be working. In any case, it does not break anything when it is not used. The main benefit to be expected from `wait` is that it allows to order decisions dynamically based on what the input contains.

IV.14. IGNORE Control Specifications

The idea is to just ignore certain decisions when there is not enough information, there is already enough information or there is not enough resources. The 3 cases correspond to the annotations:

```
(alt (:ignore-when <path> ...) ...)
(alt (:ignore-unless <path> ...) ...)
(alt (:ignore-after <number>) ...)
```

`Ignore-when` is triggered when the paths are already instantiated. This is used when the input already contains information and the grammar does not have to re-derive it.

`Ignore-unless` is triggered when a path is not instantiated. This is used when the input does not contain enough information at all, and the grammar can just choose an arbitrary default.

`Ignore-after` is triggered after a certain number of backtracking points have been consumed. This indicates that the decision encoded by the disjunction is a detail refinement that is not necessary to the completion of the unification, but would just add to its appropriateness or value. **IGNORE-AFTER IS CURRENTLY NOT IMPLEMENTED**.

The main problem with these annotations is that their evaluation depends on the order in which evaluation proceeds, and that this order is not known to the grammar writer. But in conjunction with wait, this issue is not necessarily a problem as "wait" establishes constraints on when a decision is evaluated.

---

[8]NOTE: These 2 issues require further optimization: when do you awake a frozen disjunction and which one do you force first can be decided by some sort of dependency analysis, but one that can be done statically by building incrementally a complex data-structure, otw the cost of the analysis would not be justified. Practically, the first issue is most important: it determines how much overhead is added on ALL disjunctions when freeze is used. I am experimenting currently with a "granularity" control system, which controls how often the agenda of frozen disj. is checked. This is currently not implemented.

Appendix V
Changes from Version to Version

V.1. New Features and Modifications in Version 3

- Paths are now represented with curly braces instead of simple lists.
  Old notation: (a (^ ^ b))
  New notation: (a {^ ^ b})
  The emacs routine found in file src/convert.el converts from the old format to the new format.
- Typed features are now allowed.
- Paths can now be in the left position of a pair: ({a} x) is legal.
- External is introduced.
- Procedural types are introduced.
- Fset is introduced.

V.2. New Features and Modification in Version 4

- bk-class is introduced.
- cset and pattern can contain paths.
- ordinal and cardinal are added to the morphology.

V.3. New Features and Modification in Version 5

- syntax of alt is transformed with keywords :demo, :trace, :wait, :bk-class, :ignore-when, :ignore-unless and :order acceptable in any order. Old syntax is still recognized.
- wait is introduced.
- ignore-unless and ignore-when are introduced.
- @ notation is changed to #@ notation (to avoid conflict with CLASSIC).
- ^n and ~n notations are introduced

References

[1] Ait-Kaci, H.
A Lattice-theoretic Approach to Computation Based on a Calculus of Partially Ordered Type Structures.
PhD thesis, University of Pennsylvania, 1984.
UMI #8505030.

[2] Appelt, D. E.
TELEGRAM: A Grammar Formalism for Language Planning.
In *Proceedings of the Eigth National Conference on Artificial Intelligence,* pages 595 - 9. Karlsruhe, West Germany, August, 1983.

[3] de Kleer, J., Doyle, J., Steele, G.L.Jr, Sussman, G.J.
Explicit Control of Reasoning.
Artificial Intelligence: an MIT Perspective.
MIT Press, 1979, pages 93-116.

[4] Elhadad, M.
Types in Functional Unification Grammars.
In *Proceedings of ACL'90,* pages . Pittsburgh, 1990.

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

[6] Halliday, M.A.K.
An Introduction to Functional Grammar.
Edward Arnold, London, 1985.

[7] Johnson, Mark.
CSLI Lecture Notes. Volume 14: *Attribute-value Logic and the Theory of Grammar.*
University of Chicago Press, Chicago, Il, 1988.

[8] Kaplan, R.M. and J. Bresnan.
Lexical-functional grammar: A formal system for grammatical representation.
The Mental Representation of Grammatical Relations.
MIT Press, Cambridge, MA, 1982.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

[30] Winograd, T.
Language as a Cognitive Process.
Addison-Wesley, Reading, Ma., 1983.

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

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

Index

Table of Contents