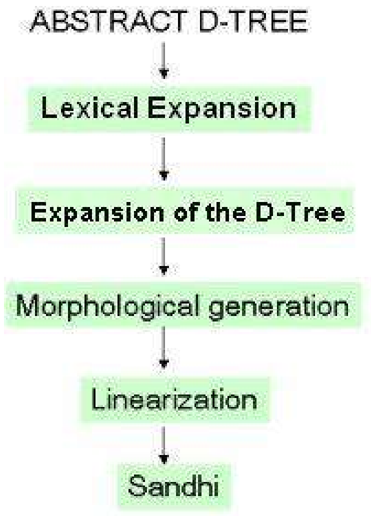
FGW 1.0 Quick Reference

By Juan C. Ruiz Anton, ruiz@trad.uji.es

This document is the user manual for FGW, a natural language software tool that offers a window-oriented user interface that allows the development and testing of syntactic realization grammars using **Dependency Grammar** as the representation formalism (see Fraser 1994, Mel'čuk 1998). For a particular language, the user must supply a generation grammar and a lexicon. The generation grammar include rules for the expression of morphosyntax, as well as rules for linearization and morphological synthesis. The lexicon file provides the grammatical, morphological and semantic information for the words of the described language.

1. Architecture of FGW

The general procedure for syntactic realization used in the FGW system is outlined in the following picture:



The starting point of the realization process is an abstract dependency tree (**D-tree**), in the form of a semantic formula as (1) (corresponding to the English sentence *I can have read the books*), that is usually introduced by the user via a special input window or a batch file.[1]

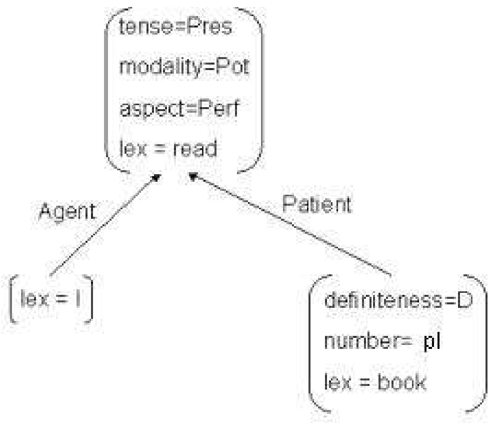
(1) `Pres Pot Perf: read Agent=(I) Patient=(d pl: book)`

The format of semantic formulas is described in detail in § 2.

[1] For mopre details on D-Trees, see §19 *a)*

The semantic formula (1) describes a predication in which the predicate `read` has two arguments, an agent (`I`) and a patient (`book`). Predicates may be associated with operators such as tense (`Pres`: present), aspect (`Perf`: perfect), modality (`Pot`: potential) or number (`pl`: plural).

The predication (1) is equivalent to the D-tree below, in which operators have been rendered as features of the form `ATTRIBUTE=VALUE`, while the predicates `read`, `I` and `book` appear as values of a feature `lex`:



A D-Tree such as this one is displayed by FGW in the following way:

```
  --<1> MAIN: read
        |--<2> Agent: I
        |--<3> Patient: book

FEATURE MATRICES
================
<1>: [lex=read,tense=Pres,modality=Pot,aspect=Perf]
<2>: [lex=I]
<3>: [lex=book,definiteness=d, num=pl]
```

The process of syntactic realization consists of the application of successive cycles of rules that modify this initial tree, by adding new features or new nodes, by changing the dependence of the existing node, or by applying a linear order to its nodes.

a) **Lexical Expansion**:

In this phase, each feature matrix in the D-tree is expanded by adding to it all the features associated in the lexicon to its `lex` (predicate). For the predicate *read*, for example, these features might be `cat=V` (lexical category = verb), `vclass=trans` (transitive) and `sem=action`. See § 5 for more information.

The result of this expansion is shown below, with the new features at the end of each feature matrix:

```
 --<1> MAIN: read
        |--<2> Agent: I
        |--<3> Patient: book

FEATURE MATRICES
================
<1>: [lex=read,tense=Pres,modality=Pot,aspect=Perf,cat=V,vclass=trans]
<2>: [lex=I,cat=PP,per=1,num=sg]
<3>: [lex=book,definiteness=d, num=pl,cat=N,per=3]
```

***b)* <u>Expansion of the D-Tree</u>**:

This step in the derivation is responsible for the transformation of the abstract D-tree (as derived from the previous phase of lexical expansion) into a fully-fledged surface D-tree, complete with auxiliary words and all the information necessary for morphological generation and word ordering.

This is done with the help of a special type of rules called **expression rules** (v. § 10), that apply top-down, starting from the top node of the D-Tree. These rules can make for a great deal of manipulation in the D-trees, including:

- Addition of features
- Insertion of new nodes, either superordinate or subordinate to the current node
- Movement of nodes (from one dependent to another in the D-tree)

Only cumulative transformation of D-trees is permitted. Expression rules never delete or change the value of the features in the nodes of the D-tree, with one unique exception: in certain restricted circumstances it is possible to change the lexical category of a node (for example, in order to transform a clause into a nominalization) (see **recategorization** in § 12.2 **C**).

In our example, expression rules are responsible for the insertion of new nodes for the auxiliares *can* and *have*, and for the determiner *the* as a dependent of *book*, thus yielding an expanded D-Tree as the following:

```
 --<5> MAIN: can
        |--<4> COMP: have
        |   |--<1> COMP: read
        |--<2> Agent/Subject: I
        |--<3> Patient: book
        |   |--<6> DET: the


FEATURE MATRICES
================
<1>:
[lex=read,vform=pastPart,aspect=Perf,cat=V,vclass=trans,domain=clause]
<2>: [lex=I,cat=PP,per=1,num=sg]
<3>: [lex=book,definiteness=d, num=pl,cat=N,per=3]
<4>: [lex=have,vform=base,cat=V,domain=clause,modality=Pot]
<5>: [lex=can,tense=Pres,cat=V,domain=clause,illoc=DECL]
<6>: [lex=the,cat=DET]
```

***c)* <u>Morphological generation</u>**:

This step produces a surface form for each lexeme, taking into account the morphological and spelling rules declared in the grammar file. In our example, `[lex=book,num=pl]` would produce *books*.

***d)* <u>Linearization</u>**:

In this phase, the nodes of the D-tree are arranged in a linear sequence of words. This task is performed by means of **linear precedence rules** (§ 13). These rules rely mainly on the functions connecting the nodes with their governors, as well as on the information present in the matrices of the nodes.

e) **Sandhi**:

After linearization of the D-tree, the great bulk of the syntactic realization process is completed. Some minor phonological adjustments between the final words in the clause may be still needed. This is the task of the sandhi rules. For example, a Sandhi rule of English would state that the combination of the auxiliary *do* and the negative particle *not* yields *don't*.

2. Predications

A semantic formula represents a **predication**, i.e. an abstract object describing the semantic structure of a sentence. As we have seen, these formulas are the starting point for the process of sentence realization in FGW.

A predication describes a state of affairs centered around a **predicate** (denoting a particular action, process or state), which is accompanied by a variable number of participant entities. For example, in the sentence *The woman found a ring in the garage*, the predicate is expressed by the verb *find*, and the participants are expressed by the nominal phrases *the woman*, *a ring*, and *in the garage*.

In terms of its content, the predication must include all that information that is necessary and sufficient for deriving the surface form of the corresponding sentence.

Semantic formulas must adhere to the following format:

[Operator$_1$ ... Operator$_n$:] Predicate [Participant$_1$ ... Participant$_n$]

For example, the English sentence *The woman found a ring in the garage* could be represented as the following predication:

(1) `DECL Past: find Ag = (D sg: woman) Pat = (d sg: ring) Loc = (D sg: garage)`

Where `Ag` = Agent, `Pat` = Patient, `Loc` = Location, `d` = definite and `sg` = singular.

Basically, this example represents the content of the English sentence *The woman found a ring in the garage*. Notice, in particular, the combination of the predicate *find* and the arguments Agent, Patient and Location.

Let us look now at the main components of a predication: the operators, the predicate, and the participants:

2.1 Operators

The **operators** represent different semantic or grammatical categories that are relevant to the content of the predication. In this example, `DECL` and `Past` are operators, meaning respectively 'declarative' and 'past tense'. All operators used in actual predications must have been explicitly declared by the user.

Note that in this definition, we consider a **grammatical category** to be a closed class of semantic distinctions that, in a particular language, are expressed grammatically (that is, by means of inflectional morphemes or grammatical words, like determiners, prepositions or auxiliary verbs). Usually, grammatical categories cover a restricted range of distinctions in a given domain, such as number, tense, mood or aspect.

When a grammatical category is present in a language, one or another of the alternative forms must be used in real cases. Thus, in the case of English, in which nouns are marked for number, one or the other of the two alternative forms (singular or plural) must always be used; there is no possibility of avoiding the choice, even when the number distinction appears to be conceptually irrelevant.

The grammatical categories of one language never exactly coincide with those of another. As Roman Jakobson once wrote, languages differ essentially in what they must express, and not in what they may express (Jakobson 1963). For example, there are languages that do not express tense, but aspect; some other languages only differentiate between past and non-past tense; and a few languages distinguish several degrees of past or future tense: near past, remote past, and so on.

2.2 Predicate

The **predicate** (*find* in our example) represents the semantic nucleus of the predication. Generally, it is a lexical item (typically a verb, an adjective or a noun).

In some cases, the predicate may be a nominal. Thus, in semantic formulas, classifying and equational predicate nominals may be dealt with as terms introduced by the label `Pred`, and. Cf. (2) and (3), respectively:

(2) `DECL Pres: Pred=(painter) Patient =(D prox sg:man)`
(i.e. *This man is a painter*)

(3) `DECL Pres: Pred=(D sg: father Poss=(I)) Patient =(John)`
(i.e. *John is my father*)

2.3 Participants

The **participants** include both arguments and adjuncts, the difference being one of lexical government: While arguments are participants required by the semantics of the predicate, adjuncts are not. The phrases *in the garage*, *the woman* and *a ring* are the surface expression of the participants in the example (1) above.

Formally, the participants fall in three types: **single nominals**, **embedded predications** and **coordinate elements**. With independence of their type, participants must be introduced by a label indicating its semantic function in the predication, and, optionally, also by its pragmatic and syntactic functions. So, for example, the participants in predication (1) are marked with the semantic function of Agent, Patient and Location (= `Loc`).

2.3.1 Nominals

Nominals are constructs that can be used to refer to an entity or entities in some world. Typically, they are expressed linguistically as noun phrases. FGW distinguishes three types of nominals: simple nominals, coordinate nominals and coreferential nominals.

A. Simple nominals

Simple nominals range from very simple items such as personal pronouns (*you*, *he*, *they*) and proper names (*Albert*, *Mary*) to complex noun phrases (*the strange man I saw yesterday in the park*), that contain operators and a set of semantic restrictors or delimiters.

In FGW, the format of simple nominals is very close to that of predications:

[Operator$_1$ ... Operator$_n$:] [Index =] Predicate [Del$_1$ ... Del$_n$]

Note that the operators, the index and the delimiters (`Del`) are optional.

The operators represent all those grammatical elements which take the form of determiners, quantifiers and semantically-based inflection (e.g. number, definiteness).

The `Index` is an exclusive identifier for the nominal. Indexes are represented by a letter `x` followed by an integer: `x1`, `x2`, etc. Indexes are normally used when the nominal in which it appears is referred to in another place of the predication (see example(8) below).

The predicate represents the semantic nucleus of the predication, typically a noun or a pronoun.

The delimiters are embedded nominals or predications, enclosed in parentheses and introduced by a label indicating its semantic function.

Some examples may make these details clearer:

(4) `d pl:child`

The operators are `d>` (definite) and `pl` (plural). The predicate is `child`. This example corresponds in English to the noun phrase *the children*.

(5) `prox 3:dog`

Here the operators are `prox` (proximal deixis) and `3` (a quantifier, meaning 'three items'). The predicate is `dog`. In English: *these three dogs*.

(6) `d sg:dog Poss=(you)`

The point of this example is the delimiter `Poss=(you)`. The label `Poss` refers to the semantic function 'Possessor'; the embedded term has a pronominal predicate (`you`). The corresponding English expression is *your dog*.

(7) `d sg:dog Restr=(black)`

This example includes a delimiter `Restr=(black)`. The label `Restr` stands for the semantic function 'Restrictor'. The embedded construct (`black`) is a reduced predication (i.e. a predication without operators). In English, this corresponds to a noun phrase with an adjective complement: *the black dog*.

(8) `d sg x1=car Restr = (Past: buy A = (d sg:teacher) Patient = (REL:x1))`

This is a rather complex example. The delimiter, again introduced by the semantic function Restrictor (`Restr`), is a full-fledged predication, complete with operators and participants. Note the occurrence of the index `x1`, which is coreferenced in the second argument of the embedded predication (an example of coreferential nominal). This example corresponds in English to a noun phrase with a relative clause: *The car that the teacher bought.*

B. Coordinate nominals

Predications may include coordinate nominals, introduced via the following format:

```
{ Term [Conj Term]* Conj Term }
```

This schema defines a set of at least two simple nominals, connected by conjunctions. These conjunctions may be `'&'` or `'v'` (and) and `'^'` (or). For example, the predication below would correspond in English to the sentence *Maria and the boy ran*:

(9) `DECL Past: run Agent={ (Maria) & (sg: boy) }.`

C. Coreferential nominals

Coreferential nominals are used to represent linguistic expressions that have an antecedent in the same predication, including reflexive and anaphoric personal pronouns, as well as relativized term positions (in many languages expressed as relative pronouns).

These nominals comply with the following format:

```
Operator:Index
```

where the operator is normally `ANA` (anaphoric) or `REL` (relative), and the index must be co-indexed with the index of another term in the same predication. For example, in (10) below, the construct `(ANA:x1)` is a well-formed term, co-indexed with the first argument of the main predicate, (`man`).

(10)
```
DECL Pres: want Agent = (d sg: x1=man) Patient = (Pres: drink Agent =
(ANA:x1) Patient = (i: vodka))
```
(i.e. *The man wants to drink vodka*)

2.3.2 Embedded predications

Complete predications may be embedded in the position of a participant role (either argument or adjunct). This is the way of representing complement clauses (ex. 10), relative clauses (ex. 8) and adverbial clauses (ex. 11):

(11)
```
DECL Past: eat Agent = (sg: x1=he) Cause=(Past: hungry Patient=
(ANA:x1))
```
(i.e. *He ate because he was hungry*)

2.4 Semantic and pragmatic functions

As we have seen, the participants are introduced in the predication by the semantic function that they play with regard to the predicate. When relevant grammatical and pragmatic functions may be included as well. For example, in the following predication, the first term is both marked as the semantic function Agent and the grammatical function Subject:

(12) `DECL Af Past: find Agent/Subject = (d sg: woman) Pat = (sg: ring)`

In FGW, the semantic and the pragmatic functions that are considered relevant in the target language are declared in the grammar file.

As for grammatical functions (e.g. subject and object) they are not regarded as a universal feature of language. When they are considered relevant for the language being described, grammatical functions must be also declared in the grammar file. They may be referred to in rules via the attribute `gf`.

2.5 Universality of predications

The form of predications is argued to be largely similar (but not necessarily equal) across languages. Put differently: linguistic expressions of different languages may differ considerably in surface form; their corresponding underlying predications can be practically identical.

In this view, differences between languages reside in the concrete language-dependent operators and predicates that 'fill in' the formal predication structures for each language, and the expression rules and linearization statements which determine the actual form in which final sentences are realized (cf. Dik 1991).

As an illustration, examine the following examples from English and Yaqui (an Amerindian language of the Uto-Aztecan family, spoken in areas of southern Arizona and northern Mexico):

(13) *These children have been singing in the church.*

(14)	*hu-me*	*usi-m*	*teopo-po*	*bwika-k*
	this-PL	child-PL	church-LOC	sing-PERF

A comparison of these examples reveals various differences in surface form. To begin with, Yaqui has Subject-Adjunct-Verb order; moreover, the kind of location which English typically expresses with the preposition in is expressed by a case suffix *-po* in Yaqui; definiteness is not grammatically coded. Finally, the verb *bwika* expresses perfective aspect by means of a *-k* suffix, as against the use of an auxiliary *have* in English.

In spite of all these differences, the underlying predications (15) and (16) are taken to be highly similar; they could be, respectively:

(15) `Pres Prog Perf: sing Agent = (prox D pl: child) Loc = (d sg: church)`

(16) `Perf: bwika Agent = (prox pl: usi) Loc = (sg: teopo)`

Apart from the obvious lexical differences (such as *sing* = *bwika*, *child* = *usi*, etc.), the structural differences between the underlying predications are minor: some operators (definiteness, tense, progressive aspect) are specified in English, but are not present in Yaqui. The actual surface differences between (15) and (16) will result from the rules (obviously different in each language) which serve to map underlying predications onto linguistic expressions.

3. Data types

FGW uses the following data types:

STRING

Typical strings start with a letter and are followed by any number of letters, numbers or undescores ("_").

It is also possible to have strings including blanks and non-letter characters, as long as they are written enclosed in a pair of quotes.

Strings are case-sensitive; that is, `num` and `Num` are different names.

Examples of valid strings:

```
     num
     agr1
     Subj_Agreement
     soufflé
     Fuß
     'in spite of'
```

In contrast, `1person` is an invalid string, because it begins with a number.

INTEGER

A number.

FEATURE

An `ATTRIBUTE=VALUE` pair.

Example:

```
     num=pl
```

VARIABLE

A string starting with a dollar sign (`$`).

Examples:

```
     $num
     $number_agreement
```

CONSTRAINTS

A group of one or more constraints on features.

See section below.

A*PATH*

A path is a sequence of path elements (separated by slashes) that denote function labels in a D-Tree, calculated from the current node:

PATH-ELEMENT$_1$ / ... / PATH-ELEMENT$_n$

The last `PATH-ELEMENT` is always a feature attribute.

Examples:

```
Obj/num
GOV/Subj/case
```

A note on paths

There are four types of path elements: alternatives, multiples, variables and simple attributes.

- An **alternative** element describes a disjunction. Its format is `(A | B | ... |C)`, where `A`, `B`, `C` are strings.
- A **multiple** element describes a successive sequence of similar functions. There are two formats allowed: `*F` for a sequence of zero or more functions `F`, and `+F` for a sequence of at least one node with function `F`.
- A **variable** element is represented as a string beginning with `$`.
- A **simple** element (representing one simple function label), represented as a string.
- Simple attributes may refer to actual relational labels (semantic, pragmatic and grammatical functions), as well as to a small set of **metarelations** such as `GOV` (that refers to the governor, the node which the current node depends on, `ANTECEDENT` (the antecedent of a pronominal/anaphoric node) and `DEP` (any dependent of the current node)

A path is always evaluated[2] starting from the current node. For an illustration, consider the D-tree portrayed graphically in fig. 2. If the current node is **A**, then the path `Object/Object/num` is evaluated to `pl`. If the current node is **B**,then the path cannot be completed, subsequently failing.

[2] On path evaluation, see § 10 *d)*

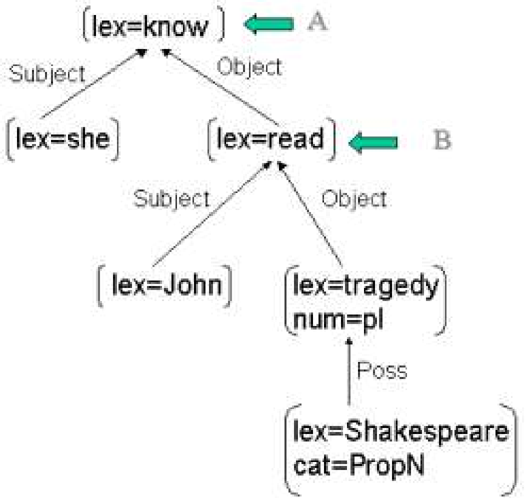


Fig. 2

In paths the label `GOV` refers to the governor of the corresponding node. For example, in fig. 2 the path `GOV/Subject/lex`, starting from the node **B**, evaluates to `she`.

Constraints

Most FGW rules allow the inclusion of conditions that have to be met by the D-tree before their application. These conditions are expressed via **constraints**. For example, rule (1) below states that the subordination of the determiner *the* is constrained by the existence in the current node of the feature `definiteness=D`:

```
(1) EXPRESSION Def_Article_insertion:
        IF definiteness=D:
            SUB Det = the.
```

<u>Types of constraints</u>

ATTRIB **=** *FVAL* Succeeds if the current node includes a feature *ATTRIB*=*VAL*, such that the *FVAL* matches[3] *VAL*. **Example**: `num=pl`
ATTRIB **= (** FVAL_1**|...|**FVAL_n**)** Succeeds if the current node includes a feature *ATTRIB*=*V*, such that *V* must match one value FVAL_i of the disjunctive set. **Example**: `tense=(Pres|Past)`
$\mathit{CONSTRAINT}_1$ **,** $\mathit{CONSTRAINT}_2$ Succeeds if Both $\mathit{CONSTRAINT}_1$ and $\mathit{CONSTRAINT}_2$ succeed. **Example**: `gen=masc, num=sing`

[3] On feature matching, see § 19 *b)*

CONSTRAINT$_1$ | *CONSTRAINT*$_2$

Succeeds if either *CONSTRAINT*$_1$ or *CONSTRAINT*$_2$ succeeds.

Example: `vform=Inf | class=Invar`

NOT *CONSTRAINT*

Succeeds if the specified *CONSTRAINT* fails.

Example: `NOT tense=Pres`

ATTRIB **= $**

Succeeds if the current node includes a feature *ATTRIB* with any value.

The character `'$'` is called the **anonymous variable**.

Example: `class=$`

ATTRIB **=** *VARIABLE*

Succeeds if:

(a) If the *VARIABLE* is bound to a value *VAL*: the current node must includes a feature *ATTRIB*=*V*, such that *VAL* matches *V*.

(b) If the *VARIABLE* is free: the current node must includes a feature *ATTRIB*=*V*, and *VARIABLE* is then instantiated to *V*.

Example: `num=$N`

ATTRIB **=** *PATH*

Succeeds if the current node includes a feature *ATTRIB*=*V*, such that the result of evaluating the specified *PATH* matches *V* (see § 19 *d)*).

Example: `gen=ANTECEDENT/gen`

PATH **=** *FVAL*

Succeeds if the result of evaluating the *PATH* matches the *FVAL*.

Example: `Subj/gen=masc`

PATH **=** *VARIABLE*

Succeeds if:

(a) If the `VARIABLE` is bound to a value `VAL`: the result of evaluating the specified `PATH` matches `VAL`.

(b) If the `VARIABLE` is free: the current node must includes a feature `ATTRIB=V`, and `VARIABLE` is then instantiated to the result of evaluating the `PATH`.

Example: `Subj/gen = $G`

PATH$_1$ **=** *PATH*$_2$

Succeeds if the result of evaluating *PATH*$_1$ matches the result of evaluating *PATH*$_2$.

Example: `Subj/binding = Poss/binding`

ATTRIB **!=** *FVAL*

Succeeds if the current node includes a feature *ATTRIB=VAL*, such that the *FVAL* does not match *VAL*.

Example: `gen != masc`

ATTRIB **!= (** *FVAL*$_1$**|...|***FVAL*$_n$**)**

Succeeds if the current node includes a feature *ATTRIB=V*, such that *V* does not match any value *FVAL*$_i$ of the disjunctive set.

Example: `vform != (Inf|Ger)`

ATTRIB **!=** *PATH*

Succeeds if the current node includes a feature *ATTRIB=V*,such that the result of evaluating *PATH* does not match *V*.

Example: `num != Subj/num`

PATH **!=** *FVAL*

Succeeds if the result of evaluating *PATH* does not match *FVAL*.

Example: `Subj/num != masc`

PATH **!= (** FVAL$_1$**|...|**FVAL$_n$**)**

Succeeds if the result of evaluating *PATH* does not match any value *FVAL*$_i$ of the disjunctive set.

Example: `Subj/sem != (action | process)`

EXIST *PATH*

Succeeds if there is a node at the end of the PATH[4].

Example: `EXIST Subj`

EXIST *PATH* **[***CONSTRAINTS***]**

Succeeds if there is a node at the end of the PATH in which the specified *CONSTRAINTS* are met.

Example: `EXIST Subj[num=pl]`

NOT *ATTRIB*

Succeeds if there is no *ATTRIB* feature in the current node.

This constraint is a notational variant of '**NOT** ATTRIB=$'.

Example: `NOT num`

[4] On path traversal, see 19 e).

INTEGER$_1$ **<** *INTEGER*$_2$

Succeeds if *INTEGER*$_1$ is lower than *INTEGER*$_2$.

INTEGER$_1$ **>** INTEGER$_2$

Succeeds if *INTEGER*$_1$ is greater than *INTEGER*$_2$.

RANDOM < *TARGET*

Succeeds if *TARGET* (an integer in the range 1-10) is greater than a random number generated by FGW.

Example: see below, note (3).

Notes:

1. Parentheses may be used to enclose groups of constraints, in order to avoid ambiguity, or simply for the sake of clarity. For example:

```
asp=Prog | (asp=Perf, tense=(Pres|Past)) | (pol=Neg, NOT
tense=Pres)
```

2. Consider the two notations for the expression of negative constraints introduced so far:

   (a) `NOT A=V`
   (b) `A!=V`

   If the current node contains a feature `A=X`, both notations produce the same result (i.e. both succeed). But if the current node does not contain any feature with the attribute `A`, the notation (a) succeeds, while the notation (b) fails, because it introduces a requirement of existence ('there is a feature A, whose value is not V') that is absent in notation (b) (that reads 'the constraint A=V does not hold').

3. Random constraints may be used to capture certain linguistic phenomena that appear to be determined by arbitrary (or poorly known) conditions. Alternatively, it can be also thought of as a way of making some rules optional. For example, in Yaqui, imperatives are normally marked in the morphology of verbs, but sometimes they are not. A way of implementing this behavior in FGW is by means of the following expression rule:

```
EXPRESSION Imperative:
    IF illoc=IMP, random > 7:
        mood := imp.
```

   Note that the target of this particular random constraint isfairly high, thus reducing the probability of failure for the constraint. A lower integer (for example, 3) would involve fewer chances of success for the rule.

4. Writing Grammar Descriptions in FGW

A description of a language is distributed in two files: a **lexicon file** (containing all the information specific of words and word classes) and a **grammar file** (containing feature declarations and realization rules of different types). Optionally, an additional transfer file may be needed, in the circumstances explained in § 7. All these are plain text files. FGW accepts extended ASCII, but not Unicode (due to a limitation of the compiler).

The **lexicon file** has a `.lex` extension, and contains two types of data:

- Lexical entries (§ 5)
- Default feature statements (§ 6)

If a transfer file is required by the application, its name must be declared also in the lexicon file, introduced by the headword `TRANSFER`, enclosed in quotes, and ended by a period. For example:

```
TRANSFER 'en-Yaqui.trf'.
```

The **transfer file** (if any) includes different types of transfer declarations (§ 7).

The **grammar file** has a `.grm` extension. It may be regarded as divided in two parts: a first part consisting of several types of feature declarations, and a second part containing the syntactic and morphological rules that are used for the realization of surface sentences.

The first part includes the following classes of declarations:

- Language declaration (§ 8)
- Declaration of features and functions (§ 9)
- Feature-redundancy rules (§ 12)

The second part comprises rules for the manipulation of D-trees:

- Expression rules (§ 10) and cycles (§ 11)
- Linearization rules (§ 13)
- Morphological rules (§ 14)
- Spelling rules (§ 15)
- Sandhi rules (§ 16)

Some general stipulations on the format of the grammar file should be kept in mind:

- All the declarations and rules must end with a period.
- The declarations section must precede the rules section.
- Declarations can be written in any order within their section.
- The relative order of cycles, linearization rules and morphological blocks is significant. Rules of this sort are applied in the order in which they occur in the file.

Comments:

All these files may include comments. They are prefixed by a percent sign ('`%`') and its scope extends until the end of the line. For example:

```
    % This is a comment
```

When not at the beginning of the line, comments should be separated by a blank from the remainder of the text. Otherwise it could result in errors in compile time.

5. Lexical Entries

Format:

```
    Headword : [Feature1, ..., Featuren].
```

Where `Headword` is a string referring to a predicate of the described language, and `[Feature1, ..., Featuren]` is a matrix of the morphosyntactic and semantic features of this word.

Notice that a lexical entry must be always ended with a period.

Note that:

- The feature list `[Feature1, ..., Featuren]` necessarily must contain two features named `cat` and `lex`. Feature `cat` refers to the lexical category of the word, and `lex` stands for the lexeme, which is normally the morphological root.

- The list of possible values for the feature `cat` may be declared by the user in the feature declaration part of the grammar file.

Examples:

(1) a. `wolf : [cat=N, lex=wolf, lexpl=wolv, sem=animate].`
    b. `cantar : [cat=V, lex=cant, vclass=1, gloss=sing].`

Example (1a) declares the feature matrix of the English noun *wolf*. The features `cat`, `lex` and `sem` (semantics) are straightforward; the feature `lexpl` declares the root for the plural form. [NOTE: This feature should be used in morphological rules as the basis for suffixation of*–es*, thus producing *wolves* instead of **wolfes*].

Example (1b) declares the feature matrix of the Spanish verb *cantar*. Notice that the feature `lex` refers to the root of the verb, as opposed to the headword, that corresponds to the conventional dictionary word, the infinitive form.

Types of features used in lexical entries:

A. `ATTRIB = FVAL`

The standard type of feature, with an atomic value (`FVAL`).

See the features in examples (1a) and (1b) above.

B. **lex(***CONSTRAINTS***) =** *IrregLex*

Irregularity feature: Declares an irregular lexeme (*IrregLex*) that replaces the standard lexeme when the specified *CONSTRAINTS* are met in the feature matrix of the word.

The irregular lexeme form may be terminated with a hyphen, what means that it is a root, able to take inflectional morphemes (see (2b) below). The absence of a hyphen indicates that the lexeme is a suppletive form, unable to admit further morphemes. *IrregLex* may also be zero (represented as Ø). This possibility may be used, for example, to deal with zero-copula in certain contexts (see example (2c)):

(2) a. akwiya : [lex=akwiya, cat=N, gloss=goat, gen=fem, lex(pl=T)=awaki].
b. amar : [lex=amar, cat=V, gloss=say, lex(tns=Pres)=omer-].
c. hay : [lex=hay, cat=V, gloss=be, lex(tns=Pres)=Ø].

Example (2a) is a lexical entry for the word *akwiya* ('goat') in Hausa, an African language of Nigeria. The last feature is an irregularity feature stating that the lexeme of this word should be *awaki* when the noun is plural (pl=T).

Similarly, the lexical entry (2b) for Modern Hebrew *amar* ('to say') includes an irregularity feature by which this feature has a root *omer* in present tense. This root may take affixes (as marked by the hyphen after the lexeme).

Example (2c) (also for Modern Hebrew) introduces a reduced version of the lexical entry for the copula. The relevant detail here is the feature lex(tns=Pres)=Ø, according to which the lexeme is zero in the present-tense form, as in the sentence *ha-mekonit hadasha*, 'the car is new', literally 'the-car (Ø) new'.

C. **val = <** $SF_1=M_1$, ..., $SF_n=M_n$**>**

Declares a **valency feature**, as a set of valency specifications, each one being a pair *SF=M*, in which *SF* is a identifier of semantic function (such as Agent or Location), and *M* is a standard feature matrix.

Valency features are applied immediately after lexical expansion (see § 1*a*). The result is that for every feature specification of the form *SF=M* in the valency feature of a predicate *P*, the matrix of the dependents of *P* matching the semantic function *SF* is overwritten with the features of *M*. For example, the lexical entry for the German preposition *während* ('during') should state that its complement (COMP) must be in genitive case (Gen). This may be expressed as val=<COMP=[case=Gen]> in:

(3) während : [lex=während, cat=P, gloss=during, val=<COMP=[case=Gen]>].

In order to understand how this feature works, suppose the following D-tree for the German sentence *Das habe ich während des Urlaubs gemacht* ('I made it during my vacation').

```
 --<1> MAIN: machen
        |--<2> Agent: Ich
        |--<3> Patient: das
        |--<4> Duration: Urlaub

FEATURE MATRICES
================
<1>: [lex=machen,cat=V,tns=past,asp=Perf]
<2>: [lex=Ich,cat=PP,per=1,num=sg]
<3>: [lex=das,cat=N,num=sg]
<4>: [lex=Urlaub,cat=N,num=sg]
```

This D-tree portrays a relationship between three participants, *Ich* ('I'), *das* ('it') and *Urlaub* ('vacation'); this latter is labelled with the semantic function `Duration` (i.e. The span of time in which the event is supposed to have had place).

The preposition *während* (with the information specified in (3)) is inserted into the D-tree as a result of the application of an expression rule, triggered by the presence of the function `Duration`. The resulting D-tree is shown in fig. 3 (some details have been simplified).

```
 --<1> MAIN: machen
        |--<2> Agent: Ich
        |--<3> Patient: das
        |--<5> Duration: während
        |   |--<4> COMP: Urlaub

FEATURE MATRICES
================
<1>: [lex=machen,cat=V,tns=past,asp=Perf]
<2>: [lex=Ich,cat=PP,per=1,num=sg]
<3>: [lex=das,cat=N,num=sg]
<4>: [lex=Urlaub,cat=N,num=sg]
<5>: [lex=während,cat=P, val=<COMP=[case=Gen]>]
```

Once the word *während* has been inserted into the D-tree, the valency feature applies, with the result of the feature `case=Gen` being added to the matrix of the `COMP` *Urlaub*. This ensures the actual genitive marking in the final word form.

6. Default feature statements

Format:

DEFAULT *CONSTRAINTS*: $ATR_1=VAL_1, \ldots, ATR_n=VAL_n$

If the *CONSTRAINTS* are met in the feature matrix of a lexical entry, the specified features $ATR_1=VAL_1, \ldots, ATR_n=VAL_n$ are default-unified to the same matrix.

These rules specify default features for the lexical entries. Feature defaults typically represent the value which is most frequent or least marked in the described language. For example, the feature `vclass` (verb class) may often be assigned the default value `transitive`, since most verbs in many languages are transitive; therefore, the feature `vclass=transitive` may be omitted from the specification of particular lexical entries. In doing so, only a minority of non-transitive verbs need to be specifically marked in the lexicon.

Operation:

Default feature statements are applied after lexical insertion: either in the replacement of the predicates of the input formula, or in rule-governed insertion, as in the statements **SUB**, **SUPER**, **MERGE** and **HEAD**.

Examples:

(4) a. `DEFAULT (cat=N): per=3, sem=inanim.`
b. `DEFAULT (cat=V): vtype=trans,sem=action.`

Example (4a) states that a word of category noun (`cat=N`) gets the features third-person (3) and inanimate semantic-type (`inanim`) by default. This means that these features are only added if no other feature of the same attribute is present in the feature matrix. The same is true for verbs (example (4b)), in which default features are transitive type (`trans`) and action semantic-type.

7. The Transfer File

Ideally, the predicates of the initial D-tree ought to correspond to lexical items of the described language; In occasions, however, they may be more conveniently represented by words of a metalanguage, chosen by the user at their own choice (English, Esperanto, Latin, or whichever other).

The equivalences between the lexical units of this metalanguage and those of the target language must be made explicit in a special text file called a **transfer file**. If a transfer file is used, this should be made explicit in the lexicon file with a line of the form

```
TRANSFER FileName.
```

A transfer file is made up of the following types of entries:

A. Word$_1$ = Word$_2$ [+ M]

A metalanguage word Word$_1$ correspond to the word Word$_2$ in the described language. `M` is an optional matrix of features that are overwritten over the whole matrix of the corresponding node after lexical expansion.

Examples:

(a) `nothing = ima.`
(b) `what = ima + [int=Q].`

These examples are from Quechua, an Amerindian language spoken in Peru, Bolivia and Ecuador. The first example (5a) states the equivalence between English *nothing* and Quechua *ima* (English is used here as a convenient metalanguage). The second example puts forward a equivalence between *what* and also *ima*, but in this occasion *ima* adds an interrogative feature (`int=Q`) to its feature specification (as found in the lexicon file).

B. Word$_1$ **=** Word$_2$ Term$_1$ ... Term$_n$

A metalanguage word Word$_1$ correspond to the word Word$_2$ in the described language, together with the dependents defined by the sequence of terms Term$_1$ ... Term$_n$. Each Term$_i$ is composed of a semantic function and a term formula (see).

This entry is for cases in which the translation of the predicate introduces additional lexical elements in the matrix. For example, in Estonian, the sense of 'to snow' is expressed by a verb *sadama* ('to fall') that is obligatorily accompanied by a subject noun *lume* ('snow').

In FGW these complex cases of lexical expression may be dealt with my means of composite correspondences, such as:

```
snowV = sadama U=(indef:lume).
```

This entry simply states that the metalanguage predicate `snowV` ('snow' as a verb) is translated by the sequence of verb (*sadama*) plus a dependent with the semantic function `U` ('Undergoer') whose predicate is *lume*. This dependent is also marked with an indefinite (`indef`) operator.

In practice, this means that a semantic formula as (a) below (corresponding to the sentence 'it snows in winter') is processed as if it were (b) (with the other English-Estonian equivalences already done), which expectedly is the right input predication in Estonian:

(a) `DECL Pres: snowV Temp = (winter)`
(b) `DECL Pres: sadama U=(indef:lume) Temp = (talv)`

C. $\mathtt{W} = \{ \mathtt{Mword}_1 : \mathtt{Rest}_1, \ \ldots.\ \mathtt{Mword}_n : \mathtt{Rest}_n \}$

Where `W`, `Mword` and `Rest` are all strings.

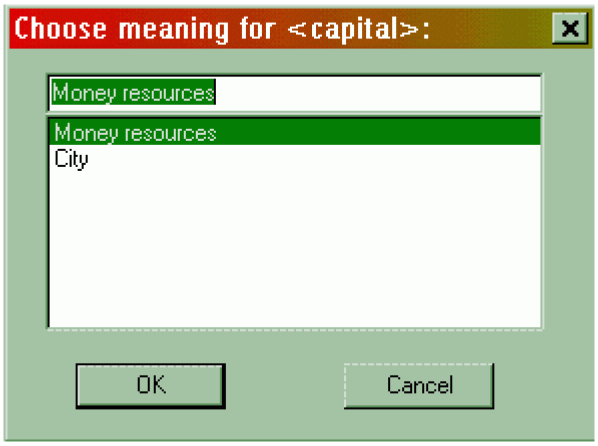
Polysemy entries declare correspondences among a polysemic word of the metalanguage `W` and the lexical items in the target language that cover the semantic domain of `W`. These lexical items are represented as pairs $\mathtt{Mword}_i$: $\mathtt{Rest}_i$, where $\mathtt{Mword}_i$ is a metalanguage word, and $\mathtt{Rest}_i$ is a string (ordinarily enclosed in quotes) that describes in plain language the selection of $\mathtt{Mword}_i$.

For example, consider the following instance of a polysemic entries from Japanese:

(a) `capital = { capital1:'Money resources', capital2:'City'}.`
(b) `capital1 = shihon.`
(c) `capital2 = shufu.`

The polysemy entry (a) states that the equivalents of the metalanguage (English) word `capital` corresponds in Japanese to (at least) two words: one for the meaning of 'money resources', labelled `capital1`, and another for a city, labelled `capital2`. These are, respectively, *shihon* and *shufu* in Japanese, as shown in (b)-(c).

The underlying predications may contain specific metalanguage words (such as `capital2`) or more general ones (such as `capital`). When an input predication contains a polysemic word, FGW asks the user to choose which of its senses should fit in that particular context of the predication. This is made through a pop-up window (see figure below) in where the selection is assisted by the descriptive statements for every distinct sense.



Choose meaning for <capital>:

Money resources

Money resources
City

Once the selection is made, the processing goes along, replacing the polysemic word in the inpur predication with the mores specific word chosen by the user.

8. Language Declaration

Format:

```
        LANGUAGE: STRING.
```

Declares the name of the described language.

Example:

```
        LANGUAGE: English.
```

9. Declaration of features and functions

Format:

ATTRIB: VAL$_1$ ... VAL$_n$.

Description:

Enumerates a list of the values that a particular category (ATTRIB) can take in the described language.

VAL may be a string, an integer or a hierarchy of the form VAL=(VAL$_1$ ... VAL$_n$). A hierarchy is a formal representation of a network of values in which some values are subtypes of other values. Hierarchical values can be recursive to an unlimited depth.

Hierarchical feature values are useful to simplify the formulation of constraints. A subtype always matches its supertype (but not vice versa); therefore, given a hierarcal value A=(B C) a constraint such as attrib=A succeeds on a matrix containing a feature attrib=B or attrib=C.

Examples:

`tense = Pres Past Fut.`	The category tense has the values Pres (present), Past and Fut (future).
`voice = Act Pass.`	The category voice has two values: Act (active) and Pass (passive).
`case = nom gen dat comit instr loc.`	The category case has the values nom (nominative), gen (genitive), dat (dative), comit (comitative), instr (instrumental) and loc (locative).

`tense = Pres Past=(ImmPast RemPast)`
`        Fut.`
	An example of a hierarchical value. The category tense includes the values Pres (present), Past and Fut (future). In addition, Past may be further subdivided in ImmPast (immediate past) and RemPast (remote past).

`cat = N V ADV ADJ=(A DET=(ART DEM Q))`
`      CJ PT.`
	An example of a recursive hierarchical value. The category ADJ (adjectival) is further subdivided in A (adjective) and DET (determiner). This, in its turn, is subclassified in ART (article), DEM (demonstrative) and Q (quantifier).

Declaration of functions

The declaration of functions is formally very similar to the declaration of features, with one difference: the attribute (`ATTRIB`) must be necessarily one of the following identifiers: `sf` (semantic functions), `pf` (pragmatic functions) or `gf` (grammatical functions). Hierarchical values are also possible. For example:

```
a. sf = Ag Pat Exp Restr Poss Loc=(In Inter Super Sub) Ben So Instr.
b. gf = Subj Obj.
c. pf = Topic Focus.
```

The declaration (a) assigns the following values for `sf` (the semantic function): `Ag` (Agent), `Pat` (Patient), `Exp` (Experiencer), `Restr` (Restrictor), `Poss` (Possessor), `Loc` (Location) (with the subtypes `In`, `Inter`, `Super` and `Sub`), `Ben` (Beneficiary), `So` (Source) and `Instr` (Instrument). In the same vein, `Topic` and `Focus` are declared as values of `pf` (pragmatic function) in (b), and `Subj` (Subject) and `Obj` (Object) as values of `gf` (grammatical function) in (c).

10. Expression rules

Format:

`EXPRESSION` `NAME :` $STATEMENT_1$`; ... ;` $STATEMENT_n$ `.`

Result:

The sequence of statements are applied in strict order. An expression rule fails as soon as one of the statements fails.

Description:

The task of expression rules is to extend the D-tree by introducing all the formal apparatus by which semantic categories and functions are expressed in the surface form of sentences in a particular language. These rules normally involve addition of features to the D-tree, as well as insertion of nodes for auxiliary words, or relocation of nodes in cases of topicalization, interrogatives, etc.

Types of statements in expression rules:

FGW allows many types of statements to be used in expression rules. These types may be grouped in the following classes:

- Conditional statements (§ 10.1)
- Feature-manipulating statements (§ 10.2)
- Node-inserting statements (§ 10.3)
- Node-moving statements (§ 10.4)

10.1 Conditional statements

A. **IF** *CONSTRAINTS*: *STATEMENTS*$_1$ [**ELSE**: *STATEMENTS*$_2$] [**END**]

If the *CONSTRAINTS* are met in the current node, then the *STATEMENTS*$_1$ are applied; if the *CONSTRAINTS* fail, the *STATEMENTS*$_2$ are tried.

If any of the *STATEMENTS* sequence fails, or if no **ELSE**-section exists, the **IF**-statement succeeds anyway.

Note that the **ELSE**-section of the rule is optional.

Example:

```
IF gf = Obj, sem = animate: case := oblique
ELSE: case := nominative
```

This example says that if the current node N has the grammatical function Object (gf=Obj) and is semantically animate (sem = semantics), then the feature case=oblique is unified to the feature matrix of the N; alternatively, the feature case=nominative is to be unified.

B. **ON** SELECTOR **= {**
TRIGGER$_1$:STATEMENTS$_1$
...
TRIGGER$_n$:STATEMENTS$_n$ **}**

This is a kind of concise multiway **IF**-statement. The SELECTOR may be a string referring to a feature attribute in the current D-tree, or a path expression, pointing to a feature attribute located elsewhere in the D-tree. The TRIGGER may be a feature value, or a disjunction of feature values (in the form (VALUE$_1$ | ... | VALUE$_n$). The anonymous variable '$' may be used as a value as a kind of default case at the end of the TRIGGER:STATEMENTS sequence.

Example:

To illustrate this kind of statements, consider the following expression rule from Cantonese, that subordinates a classifier in accordance with the semantic class of the noun:

```
EXPRESSION Classifiers:
    IF (dem=$ | quant=INTEGER):
        ON class=
        { tool:          SUB cls = bá
          machine:       SUB cls = ga
          building:      SUB cls = gâan
          cloth:         SUB cls = gihn
          (animal|pair): SUB cls = jek
          flat:          SUB cls = jêung
          cyl:           SUB cls = jî
          long:          SUB cls = tìuh
          round:         SUB cls = lâp
          square:        SUB cls = fûk
          $:             SUB cls = go
        }.
```

This rule applies on nominal nodes including a demonstrative (note the anonymous variable attached to the feature `dem`) or any numeral (`quant=INTEGER`). Then, according to the value of a `class` feature, a particular classifier is selected.

This rule shows instances of the trigger as an attribute string (`class`). This is the right case when the selector feature is located in the matrix of the current node. When the selector is supposedly found in another node, a path description to that node may be used.

Quechua offers a good example of paths in the selector of **ON**-statements. In this language, a clitic may be attached to the focus constituent of the clause, indicating the evidence that the speaker has about the reality of the described event (grammaticality, this phenomenon pertains to the field of modality, and it is called evidentiality). Suppose one category in Quechua called `evidentiality`, with three possible values: `Dir` (if the evidence is prima facie, directly witnessed by the speaker), `Ind` (if the evidence is indirect or reportative, for example for hearsay), and `Conjc`(if the evidence has a conjectural basis). This situation may be described by an expression rule as the following:

```
EXPRESSION Evidentiality:
     IF pf = Focus:
         ON GOV/evidentiality =
         { Ind:   SUB Pr = si
           Dir:   SUB Pr = mi
           Conjc: SUB Pr = cha
         }.
```

Consider the D-tree below, which underlies the Quechua sentence *payqa t'antatan mikhushan* ('He is eating bread') (Faller 2002: 11):

```
--<1> MAIN: mikhu
        |--<2> Subject/Topic: pay
        |--<3> Object/Focus: t'anta

FEATURE MATRICES
================
<1>:
[lex=mikhu,cat=V,tns=Pres,asp=Progr,gloss=eat,evidentiality=Dir]
<2>: [lex=pay,per=3,num=sg]
<3>: [lex=t'anta,cat=N,gloss=bread]
```

Note that this rule applies in the domain of the Focus node, which is the locus for the subordination of the clitic, whereas evidentiality is a category of the proposition. Thus, the selection of the evidentiality feature must be made in the governor (`GOV`) node.

10.2 Feature-manipulating statements

A. `PATH := VAL`

Strict Unification: It unifies the feature specified by the equation `PATH=VAL` with the feature matrix of the current node.

Note:

In FGW, unification is a procedure that adds a feature into a feature matrix, provided that this matrix does not contain conflicting information.

Given a feature `ATTR=V` and a matrix `M`, several situations are possible:

a. If `M` already contains a feature `ATTR=V`, then `M` remains unchanged.

b. If `M` contains a feature `ATTR=X`, such that `X` is different to `V`, then unifiaction fails.

c. Otherwise, `ATTR=V` is added to `M`.

Unification of `A=V` to `B` works differently depending on the nature of `A=V`:

a. `A` is an atomic attribute: It fails whenever the unificable feature `A=V` is incompatible with a feature already present in `B`.

b. `A` is a path. The path is first evaluated, and the result of this evaluation is unified to `B`.

c. `V` is a variable. If the variable may be instantiated (because it has been declared within the same statement) is value is unified to `B`; otherwise, `B` remains unchanged.

Examples:

```
EXPRESSION Object_Sentence:
    IF gf = Obj:
        nominalization := complement;
        case := acc.
```

In certain languages (Quechua, Tamil, among others), object sentences are expressed in the form of a nominalization in accusative case. This may be described via an expression rule such as `Object_Sentence`. It states that if the grammatical function of the current node is object (`fg=Obj`), then the features `nominalization=complement` and `case=acc` are unified to the matrix of the node.

```
EXPRESSION Null_Subj:
    IF EXIST Subj[cat = PP, NOT pf]:
        Subj/Null := TRUE.
```

This rule may handle null subjects in a number of languages. It says that if the current node governs a `Subj` node(subject) with lexical category `PP` (personal pronoun) and with no pragmatic function (`pf`) specified, then the feature `Null=TRUE` is unified to the matrix of this dependent `Subj` node. The feature `Null=TRUE` should be the responsible of the zero-expression of the involved pronouns in the morphological component (see).

B. `PATH` **`&=`** `VAL`

Default unification: it adds the feature specified by the equation `PATH=VAL` to the feature matrix of the current node.

Unlike strict unification, default unification succeeds if the specified feature cannot be added into the matrix of the current node. Compare the following examples (where `'+'` stands for 'unifies'):

a. **Strict unification**:

`gen := fem + [cat=N,num=pl]` => [cat=N,num=pl,gen=fem]

`gen := fem + [cat=N,gen=masc,num=pl]` => FAILS

b. **Default unification**:

```
gen &= fem + [cat=N,num=pl] => [cat=N,num=pl,gen=fem]

gen &= fem + [cat=N,gen=masc,num=pl] => [cat=N,gen=masc,num=pl]
```

C. **RECAT** *Function* **AS** *NewCategory* (**Recategorization**)

This statement changes the value of the `cat` (category) feature of the node identified by the specified *Function* to *NewCategory*. *Function* may be the function label of a dependent node, as well as `HEAD` (if the recategorization applies on the current node). The old category is kept in the feature matrix as a feature of attribute `bascat` (= base category).

This statement always succeeds, except if the feature matrix does not contain a `cat` feature.

Consider the example:

```
        EXPRESSION Recateg_V:
            IF cat=V, tense=(Pres|Past), NOT illoc=IMP:
                RECAT HEAD AS PCP.
```

This rule recategorizes a verb node as a participle (`PCP`) when the tense is present or past, and the illocution (`illoc`) is not imperative (`IMP`). Note that in this case recategorization can be necessary because the verbal participle may be inflected like an adjective (with gender and number agreement, for example), instead of like a verb.

D. **ASSIGN** GF SF_1 [... **|** SF_n]

States which semantic feature or features (`SF`) (and in which order) are elegible to fill the specified grammatical function `GF`.

Consider the following examples:

```
        EXPRESSION Subj_Assignment:
                IF Pat/pf=GivTop:
                        ASSIGN subj Pat;
                        voice := Passive
                ELSE: ASSIGN Subj Ag | Exp | Pat.
```

This rule states that the `Subj` (subject) function is assigned to the `Pat` (patient) if it has the `GivTop` (Given Topic) pragmatic function; in addition, a feature `voice=Passive` is unified to the current matrix. Otherwise, `Subj` is assigned to the dependent having the first possible semantic function in the set `Ag` (Agent), `Exp` (Experiencer) or `Pat` (Patient).

```
    EXPRESSION Obj_Assignment:
        ASSIGN Obj Pat.
```

This rule, in its turn, states that the `Obj` function (Object) is assigned to Patients (if any exists).

10.3 Node-inserting statements

A. **SUB** *F* = *W* [**+** *FeatureList*]

Inserts a new node *W* in the D-tree as a dependent of the current node with function *F*.

If the D-tree already contains a dependent node with function `F`, the procedure does not inserts the node, but still succeeds.

The statement fails if the word `W` is not found in the lexicon.

In any other case the statement succeeds, by inserting the subordinate node `W`. The semantic function of the inserted node will be `F`, and its feature matrix is is the the lexical entry of the word `W`, eventually unified with the optional *`FeatureList`*, and finally expanded by the application of default feature statements (v. § 6).

Example:

```
EXPRESSION Article:
    IF def=d:
        SUB Determiner = the.
```

This rule inserts a node *the* as a dependent with `Determiner` function of the current (definite) node.

Consider the lexical entry for *the*, and the D-tree below (for the sentence *I saw the dogs*):

```
the : [lex=the, cat=Art].
```

```
--<1> MAIN: see
        |--<2> Agent: I
        |--<3> Patient: dog

FEATURE MATRICES
================
<1>: [lex=see,cat=V,tns=Past]
<2>: [lex=I,cat=Pro]
<3>: [lex=dog,cat=N,def=d,num=pl]
```

The application of rule `Article` should produce the D-tree below, through the subordination of the `Determiner` node:

```
--<1> MAIN: see
        |--<2> Agent: I
        |--<3> Patient: dog
        |    | <4> Determiner: the

FEATURE MATRICES
================
<1>: [lex=see,cat=V,tns=Past]
<2>: [lex=I,cat=Pro]
<3>: [lex=dog,cat=N,def=d,num=pl]
<4>: [lex=the,cat=Art]
```

B. **`SUPER`** `W [+` *`FeatureList`*`] [,` *`DepInfo`*`]`

Inserts a new node `W` in the D-tree, as the governor of the current node `N`. The former superordinate of `N` becomes the governor of the inserted node.

The relation between the current node and its new governor may be determined in the rule by means of the optional *DepInfo* section. If no function is proposed, FGW uses the function COMP (complement) by default.

DepInfo may have the following formats:

a. **DEP AS** *Function* [**+** *NFeatureList*]

b. **DEP +** *NFeatureList*

The format (a) is used to declare a function different to the pre-defined function COMP.

Both formats also allow to declare a list of features (*NFeatureList*) to be unified with the feature matrix of the original current node N.

Operation:

If the word W is not in the lexicon, SUPER still succeeds, but the D-tree remains unchanged.

Otherwise, the lexical entry of W is unified with the optional *FeatureList* (if declared), and expanded by the application of the corresponding default feature statements. This new node is inserted as the governor, and becomes the new current node.

In addition, the valency feature of the inserted node is eventually applied on its dependent (that is, the former current node N), and the optional *NFeatureList* is then unified with the resulting matrix of N.

Notes:

A typical use of SUPER is for the insertion of prepositions, conjunctions and auxiliary verbs. For instance, in Modern Persian, purpose clauses are encoded by the preposition *tâ*, while the subordinate clause appear in subjunctive mood. This may be easily described by the following expression rule:

```
EXPRESSION Purpose_SF:
    IF sf=Purpose:
        SUPER tâ, DEP + [vform=Subjv].
```

The rule superordinates a node *tâ*, and unifies the feature vform=Subjv (subjunctive) on the matrix of the current node.

To see this rule in action, consider the D-tree in fig. 9, that represents the semantic content of the Persian sentence *bâ mâshin safar kard tâ zud berasad* ('s/he travelled by car in order to arrive sooner'). Note that the feature structure is rather simplified for ease of presentation.

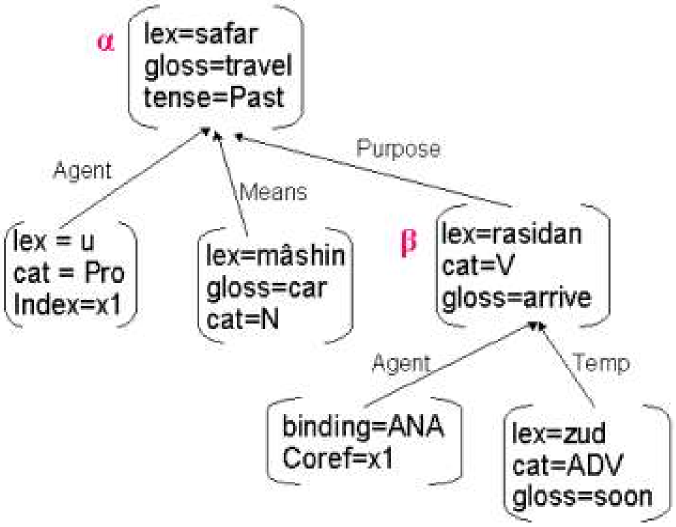


Fig. 9: D-tree before application of rule `Purpose_SF`

Let us suppose that the current node is *rasidan*, identified by β. (the word *berasad* in the surface sentence is a subjunctive third-person singular form). It is connected to its governor (α) by the semantic function `Purpose`. Other details of the D-tree are not relevant to this discussion. Since the constraint `sf=Purpose` matches the configuration of the current node, the rule `Purpose_SF` applies, and the node *tâ* is inserted, subordinate to α but superordinate to β. In addition, the feature `vform=Subjv` is unified with the feature matrix of β. This transformed D-tree is shown in fig. 10:

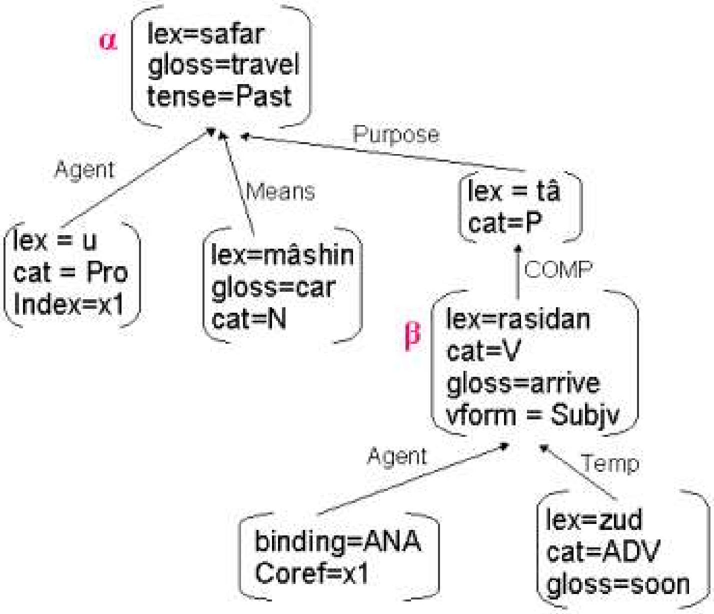


Fig. 10: D-tree after application of rule `Purpose_SF`

C. `MERGE W [+ FeatureList] [, DepInfo]`

This statement is similar to `SUPER` (see above for syntax details) in that it adds a new node in the D-tree as superordinate of the current node. The difference is that whereas in `SUPER` the current node keeps all its dependent after the insertion, in `MERGE` the dependents of the current node are all made dependents of the new node.

Operation:

Consider a current node `N`:

- If `W` is not in the lexicon, `MERGE` still succeeds, but the D-tree remains unchanged.
- Otherwise, the lexical entry of `N` is unified with the optional *`FeatureList`* (if declared), and expanded by the application of the corresponding default feature statements. This new node is inserted as the superordinate of `N`, and replaces it as the current node. The index of the merged node (`N`) is also changed, to make it dependent of `W`.

  In addition, the valency feature of the inserted node is eventually applied on `N` (now its dependent).

Note:

The name `MERGE` intends to suggest the idea that the current node and the new superordinate node are somehow 'fused' together in a complex but syntactically rigid construction, in which interleaved elements are not normally allowed. This construction has been called **clause union** or **clause reduction** by some scholars (cf. Noonan 1985:74, Aissen and Perlmutter 1983.

Consider, for example, the situation of the clause union construction formed by an auxiliary verb plus a participle in Spanish. In this language the construction is close-knit, in the sense that in-between elements (such as adverbs) are never found. The situation is very different from that of English, or even more drastically, German:

(25) a. *Pedro sin duda ha leído el libro*

b. *Peter has undoubtedly read the book.*

c. *Peter hat zweifellos das Buch gelesen.*

In a similar vein, the Spanish construction works as a group in subject-verb inversion:

(26) a. *¿Ha leído Pedro el libro?*

b. *Has Peter read the book?*

c. *Hat Peter das Buch gelesen?*

The facts seem to indicate that the Spanish construction is a kind of compound verb, that shares a same set of dependents. In FGW terminology, Spanish *ha* and *leído* are merged, whereas the English or German construction is syntactically very different in that the auxiliary is a node superordinate to the participial verb, the subject is a dependent of the auxiliary, and the object is a dependent of the participle.

Use of variables in `SUPER` and `MERGE`:

The argument of the insertion statements `SUPER` and `MERGE` may be a variable, assuming it is bound in the same statement. This is a convenient way of introducing valence-governed words in a D-tree. Consider rule `GovernedPrep`:

```
EXPRESSION GovernedPrep:
    IF prep=$P:
        SUPER $P.
```

An expression rule of this sort combines well with valency features, such as the following,in the entry of the English verb *look*:

```
look : [lex=look, cat=V, val=<Patient=[prep=at]>].
```

Note that the valency feature (`val`) of *look* includes one valency specification for the `Patient` argument, with the feature `prep=at`. This is a way of declaring that the patient of *look* is introduced by the preposition *at*.

D. **`COPY`** (*`ATTRIB`*$_1$ `....` *`ATTRIB`*$_n$)

This statement copies a set of features (identified by their attributes) from the former current node to a superordinate node. Thus, it is normally used in the same rules in which insertion statements (via `SUPER` or `MERGE`) appear. Features that are prefixed with a hyphen are deleted from their original node after been copied.

Note:

As an illustration, consider the following rule, that inserts a copula in predicate adjective clauses:

```
    EXPRESSION Copula_Insertion:
        IF cat=A:
            SUPER super;
            COPY (domain -tense -illocution -polarity).
```

The percolation statement in `Copula_Insertion` is necessary to complete the copula node with all the features that might be necessary for further expression rules (such as `illocution` or `polarity`), as well as for morphological processing (`tense`).

E. **`HEAD`** `W [+` *`FeatureList`*`]`

This statement inserts the word `W` in an lexically empty node (i.e. a node with no `lex` feature), optionally overwriting it with the specified *`FeatureList`*.

Operation:

`W` may be a bound (instantiated) variable.

`FeatureList` may also contain variables and evaluation paths (see Hebrew rule below).

The matrix of `W` is retrieved from the lexicon, and its category defaults are calculated; after that, it is overwritten by `FeatureList`, and the result matrix overwrites, in its turn, the matrix of the current node.

It fails if:

- The current node is not lexically empty (i.e. it has a `lex` feature).
- `W` is not in the lexicon.

Note:

Lexically empty nodes are allowed only in a few restricted cases:

a. In clauses containing a `Pred` function, such as:

```
DECL Pres: Pred=(i sg: town Restr=(big)) U/Top=(Cracow)
```

i.e. 'Cracow is a big city'

b. In bound terms, of the form `Operator:Variable`:

```
ANA:x1
REL:x1
```

c. In terms with an empty head, represented by a zero (also Ø):

```
d sg: 0 Restr=(big)
```

Examples:

```
EXPRESSION Reflexives:
    IF binding=REFL:
        HEAD zix.
```

This rule (from Yidish) inserts the pronoun *zix* as the expression of any reflexive term (represented by the feature `binding=REFL`).

```
EXPRESSION Rel_Pronoun:
    IF binding=REL, gf=Obj:
        HEAD hu +[gen=ANTECEDENT/gen, num=ANTECEDENT/num].
```

This rule (from Hebrew) expresses an object relative term by means of *hu* (a personal pronoun). In addition, the features `gen` (gender)and `num` (number) are evaluated on the antecedent of the relative, and their values are then added to the matrix of the word (overwriting any conflicting feature already existing in it).

10.4 Node-moving statements

A. **`RAISE`** *`PATH`*

The node `N` identified by the *`PATH`* is extracted from its position in the D-tree, and is subordinated directly under the current node. `N` keeps its own semantic, pragmatic and grammatical functions.

If `N` bears a grammatical function, an anaphoric copy of it is left in its original position in the D-Tree, with the matrix `[binding=ANA,COINDEX=α]`, where α is the index of the antecedent node.

This always succeeds, even in the case that the node determined by the path does not exist.

Examples:

```
EXPRESSION Potential:
    IF mod=Pot:
        SUPER tavânestan;
        RAISE COMP/Subj;
        COPY num per tns asp illoc -pol).
```

This rule (from Persian) superordinates a node *tavânestan* in a clause node marked with the feature of potential modality (`mod=Pot`),and then raises the `Subj` (subject) of the `COMP` node, also copying the features of number (`num`), person (`per`), tense (`tns`), aspect (`asp`) and illocution (`illoc`), and moving the feature of polarity (`pol`).[5]

[5] On feature copying and moving, see §10.3 *e)*.

```
EXPRESSION Subj_Raising:
    IF NOT Subj:
        RAISE +COMP/Subj.
```

This rule raises the subject at the end of a sequence of `COMP` nodes. This typically occurs with auxiliary verbs.

B. **MOVE TO** (*CONSTRAINTS*) **[THRU** *BARRIERS* **]**

Where *BARRIERS* is a sequence of functions of the form (*FUNC*$_1$ | ... *FUNC*$_n$).

The current node is extracted from its position in the D-tree, to be made a dependent of the first node that meets the specified *CONSTRAINTS* (its **landing node**). The displaced node keeps its semantic and pragmatic functions, but adopts the grammatical function `EXTRA` (extraposed) in its new location. An anaphoric copy of the displaced node is left in its original position in the D-Tree.

The landing node may be the governor itself, if it meets the constraints. Otherwise, the procedure applies recursively, following the chain of governors upwards.

If *BARRIERS* are declared, the landing node must be labelled with one of the specified functions.

The statement fails if the current node is lexically empty (i.e. it has no `lex` feature), or no landing node may be found in the D-Tree.

Example:

```
EXPRESSION Topic_Extraposition:
    IF pf=Top:
        MOVE TO (sf=MAIN).
```

This rule moves a dependent with the pragmatic function `Top` (Topic) upwards to become a dependent of the top node (i.e. the node that includes the feature `sf=MAIN`, assigned automatically by FGW to the top node of a D-Tree).

C. **LOWER** F *CONSTRAINTS* **TO** *PATH*

A dependent of the current node with function `F` that meet the specified *CONSTRAINTS* is displaced as a dependent of the node specified by the *PATH*.

Unlike `RAISE` and `MOVE` statements, no anaphoric copy of the displaced node is kept in its original position.

If no node can be lowered, the procedure does not fail, leaving the D-Tree unchanged.

Example:

```
EXPRESSION Copula_Insertion:
    IF tense=$, cat=A:
        MERGE be, DEP AS Attr;
        LOWER Deg TO Attr.
```

This rule merges a copula *be* in tensed adjective-headed nodes, and more significatively for the current discussion, lowers any degree dependent (introduced by a semantic function Deg) as a dependent of the adjective, not of the new copula governor.

As an example, suppose the following initial D-Tree:

```
--MAIN-[1] sick
        |--U/Top-[2] he
        |--Deg [3] very
```

Suppose that node [1] includes, at least, the feature tense=Past, together with the lexical features of *sick* (among them, cat=A). Therefore, rule Copula_Insertion applies, merging a node *be*. The intermediate result is as follows:

```
--MAIN-[4] be
        |--Attr [1] sick
        |--U/Top-[2] he
        |--Deg [3] very
```

Subsequently, lowering of the Deg node puts the node [3] as a dependent of the node [1]:

```
--MAIN-[4] be
        |--Attr [1] sick
        |         |--Deg [3] very
        |--U/Top-[2] he
```

11. Cycles (of expression rules)

Expression rules are organized in **cycles**. A cycle is a sequence of rules that apply (in the specified order) in the current node whenever certain conditions are met.

The format of cycles is as follows:

CYCLE (*CONDITIONS*): ERS_1 ... ERS_n.

Where each *ERS* is either a expression rule name, or a disjunctive set of expression rule names, expressed as (*Name*$_1$ | ... | *Name*$_n$).

In a disjunctive set, only one expression rule may be applied in each node. These are tried in left-to-right order; once one rule succeeds and produces a side-effect in the D-Tree (for example, by adding a features or new nodes), the remaining rules in the disjunctive set are discarded.

Consider the following example:

```
CYCLE (cat=N):
        Definiteness
        Possessive
        ( Subj_Marking | Obj_Marking | SemRole_Marking ).
```

According to this cycle, in every nominal node (cat=N) the Definiteness rule is applied first; then the Possessive rule, followed by a disjunction of three rules: Subj_Marking, Obj_Marking or SemRole_Marking. In disjunctions, rules are applied left to right; once a rule succeeds, the remaining rules to its right are discarded.

12. Feature-Redundancy rules

Format:

```
IF CONSTRAINTS: UNIF-STATEMENT1; ...; UNIF-STATEMENTn.
```

Feature-redundancy rules declare cases of dependence between features in a D-tree, in the sense that the occurrence of certain features (declared in *CONSTRAINTS*) normally requires adding certain features to the matrix of the current node. This is made by the use of the statements for default unification and strict unification.

Examples:

a. `IF cat = (N | ADV): domain := nominal.`
b. `IF domain=nominal, quant = (1 | indef): num &= sg.`
c. `IF domain=nominal, quant = $: num &= pl.`

These rules state that:

- Rule (a): if the current node in the D-tree contains a feature `cat` that is either `N` (noun) or `ADV` (adverb), then a feature `domain=clause` is unified to the node.

- Rule (b): if the current node includes the feature `domain=nominal`, as well as a feature `quant` (i.e. quantification) with either the value value `1` or `indef`, then an additional feature `num=sg` (singular) is default-unified to the current matrix. Note that if it already includes a `num` feature, this matrix remains unchanged.

- Rule (c): if the current node has the feature `domain=nominal`, and a feature `quant` with any value (that is the meaning of the anonymous variable `'$'`), then the default-unified feature is `num=pl`.

Notice two additional properties of feature-redundancy rules:

- Relative order is significant: one statement may add features that can appear as constraints in subsequent statements. This is the case of the feature domain in rule (a) vs. rules (b)-(c).

- The properties of unification as a computational operation may be used advantageously to simplify the formulation of these rules. For example, if rule (b) applies successfully, then rule (c) will necessarily fail, since both rules introduce opposite values of the feature `num`). Therefore, the constraints of rule (c) do not need to be excessively detailed, since in any case the wrong results will be ruled out by the unification procedure.

Feature-redundancy rules operate in the first phase of the process of realization, just after lexical expansion (v. § 1*a*) They are applied in the very same order in which appear in the grammar file. Accordingly, the application of a particular declaration may always change the context for the application of the subsequent feature-redundancy rules.

13. Linearization rules

The linearization phase task is to arrange the nodes of the D-tree in a sequence of words. It is performed by means of separate linear precedence rules. These rules rely mainly on the functions connecting the nodes with their governors, as well as on the information present in the matrices of the nodes.

These linearization rules may be classified in two types, according to the syntactic nature of the nodes they refer to. First, rules that order dependents with regard to their governors (**GOV-DEP rules**). In this sense, a dependent may appear before or after the governor. We may talk about the dependents that precede the governor as occupying the **pre-field**, in contrast with dependent that follow the governor, namely occupying the **post-field**.

A second kind of linearization rules are those that order the siblings (co-dependents) in every field (**Sibling rules**).

Linearization rules comply with the following format:

LP *NAME* : *NODE-SPEC*$_1$ **<** *NODE-SPEC*$_2$.

Where *NAME* is a string, and each *NODE-SPEC* is a node specification, composed of a function label and an optional list of constraints in brackets. The function label may be any grammatical, semantic or pragmatic function, or one of the metafunctions GOV (governor), DEP (any dependent), ARG (any argument), ADJUNCT, or SIB (a sibling). In GOV-DEP rules, one of the node specifications must include the GOV metafunction.

A such rule declares that the node that satisfies *NODE-SPEC*$_1$ is ordered before all the nodes satisfying *NODE-SPEC*$_2$.

Examples:

(a) LP AdvV: NEG < GOV.

= A negative particle (NEG) procedes its governor.

(b) LP QV: DEP[int=Q] < GOV.

= An interrogative dependent precedes its governor.

(c) LP P1V: LP ARG: ARG[NOT pos=Focus] < ADJUNCT.

= Non-focal arguments precede adjuncts.

Operation:

Linear precedence rules are ranked: every rule takes precedence over the remaining rules declared below in the grammar file. This feature allows to arrange rules according to their specificity: first the more particular ones; the more general at the end.

In English, For example, governors typically precede their dependents, but some dependents precede their governors, as is the case of determiners, subjects, and others. This may be described as in:

```
% Specific Rules
LP DetN:  Det < GOV.
LP SubjV: Suvj < GOV.

% General rule
LP DefR:  GOV < DEP.
```

14. Morphological rules

The task of morphological rules is to describe the formal modifications involved in the productive inflection of words. Each morphological operation apply on the lexeme `L` of a word, so as to form a new lexeme `L'` that reflects the modification (such as the addition of an affix) produced by the inflection.

These formal modifications operated on the lexemes include, inter alia:

- Suffixation: as in English *cat* + Plural → *cats*
- Prefixation: as in Swahili *tabu* 'book' + Singular → *vitabu*
- Infixation: as in Tagalog *sulat* 'write' + Past → *sumulat*
- Reduplication of the lexeme: as in Nahuatl *cih* 'hare' + Plural → *ciicih*
- Modification of the lexeme: as in Arabic *ktb* 'write' + Past → *katab*

Before attaching a prefix or a suffix, spelling rules (§ 15) are applied between the lexeme and the affix.

Morphological operations are organized in **blocks**, according to the proposal by Anderson (1992). Each block contain a disjunctive set of operations (called **M-operations**), that are tried in strict order. As soon as an M-operation succeeds, the block finishes its application, and the processing goes on with the next block.

The format of blocks is the following:

> **BLOCK** Name *CONSTRAINTS*:*M-OPERATION*$_1$ ... *M-OPERATION*$_n$.

Such that if the specified *CONSTRAINTS* meet in the matrix of a word `W`, the sequence of morphological operations *M-OPERATION*$_1$ ... *M-OPERATION*$_n$ is applied until one of them succeeds, what produces a morphological inflection of the lexeme of `W`.

The format of M-operations is:

A. *CONSTRAINTS*::*INFLECTION*

If the *CONSTRAINTS* are met in the matrix of the current word `W`, then the specified inflection is applied onto the lexeme of `W`.

B. OTHERWISE::*INFLECTION*

The specified inflection is applied as a kind of default operation.

NOTE: This only can be the last statement in a block.

C. *CONSTRAINTS*::**EXIT**

An **exclusion statement**: Statements of this kind implement negative contexts in which a block (or part of it) cannot be applied. If the associated constraints are met, the current block is abandonned.

FGW accepts the following types for *INFLECTION*:

Prefixation

Format: *Af*- (Variant: **PREFIX** *Af*)

The affix *Af* is prefixed to the lexeme of the current word.

Suffixation

Format: -*Af* (Variant: **SUFFIX** *Af*)

The affix *Af* is suffixed to the lexeme of the current word.

Infixation

Format: -*Af*- (Variant: **INFIX** *Af*)

The affix *Af* is infixed to the lexeme of the current word (see examples below).

Syncretism (replacement of stem)

Format: **STEM** *F*

The lexeme of the current word is replaced by the value of the feature whose attribute is *F*.

NOTE: The affixes (*Af*) in the above inflection types are normally strings, but they may be also variables, provided they are bound in the condition part of the M-operation statement (see example below).

Operation:

For every word `W` in a D-Tree, the general procedure of morphological synthesis is as follows:

- If the feature matrix of `W` contains an irregularity feature (v. §5 **B**) whose constraints are satisfiable on the matrix itself, the lexeme feature (`lex`) of the word is replaced by the declared irregular form. If this form is suppletive, the morphological synthesis terminates; otherwise, the procedure goes on with the new lexeme. This scheme implements the **principle of lexical priority** of Dik's Functional Grammar.

- The blocks of rules are applied on the matrix of `W` in strict order.

- The M-operations within the same block are mutually exclusive. They are tried in the same order in which they occur in the block, but as soon as the first operation succeeds, the other operations within the same block, regardless of whether they are applicable or not, are discarded from applying. The block is then finished, and the next block is tried.

- The morphological operations interact in a cyclic manner with the spelling rules, with phonological or graphemic adjustment following each morphological operation.

- If no block is applied, the lexeme of W remains unchanged.

Note that the relative order of the blocks in the grammar file reflects the relative order of their corresponding morphemes in relation with the base. That is, the fact that a word has the form stem+affix$_1$+affix$_2$ is reflected in the grammar file by placing the block of rules that attaches the affix$_1$ before the block that introduces the affix$_2$.

Examples:

This an example of a whole block:

```
BLOCK Agr (cat=V,vform=(prs|ptf|ptps|yap)):
    illoc=IMP | NOT finite | invar=t :: EXIT
    per=p1,num=sg :: -man
    per=p2,num=sg :: -san
    per=p3        :: -Di
    per=p1,num=pl :: -miz
    per=p2,num=pl :: -siz.
```

This block (called Agr) describes agreement inflection for a certain group of verb forms in Uzbek: as indicated by the overall group of constraints, verbs whose form is either present (prs), future (ptf), perfect (ptps) or present-progressive (yap).

Notice that the first statement is an exclusion statement: the block is not applied if the current verb is imperative, not finite, or invariable (i.e. it includes the lexical feature invar=t). The rest of the statements are operations of suffixation.

Exclusion statements can appear in any position within the block. When it is the first statement in a block, it works as a global restriction; when it occurs in the middle of the block, it works as a restriction for only the following section of it.

Now, some examples of different M-operations:

(a) Prefixation:

```
cat=V, pol=Neg :: = na-
```

This statement (from Persian) says that negative polarity (pol=Neg) is marked in verbs by the prefix *na-*.

(b) Syncretism:

```
cat=N, num=pl :: STEM lexpl
```

This statement (from Hausa) replaces the standard lexeme of a plural noun with the value of the feature lexpl (if any). For example, for the word *kogi* ('river') it must be *koguna*, as shown in the following lexical entry:

```
kogi : [lex=kogi,cat=N,gloss=river,lexpl=koguna]
```

Use of variables in inflection:

```
cat=N, num=pl, pl=$A :: SUFFIX $A
```

This statement (from Yidish) attaches a suffix that is obtained from the value of the feature pl. This is a lexical feature, found in lexical entris for nouns, such as the following:

```
    arbet     : [lex=arbet,cat=N,gloss=work,gen=fem,pl=n].
    barne     : [lex=barne,cat=N,gloss=pear,gen=fem,pl=s].
    bild      : [lex=bild,cat=N,gloss=image,gen=neu,pl=er].
```

(c) Infixation:

```
    cat=V, focus=Actor, tns!=Fut :: INFIX um
```

According to this statement (from Tagalog), the verb receives an infix -*um*- if the focus constituent is the Actor and the tense is not future.

The right treatment of infixation is a little tricky. In order to operate properly, FGW has to know where the infix is supposed to be attached. This is made by inserting the place-holder character '%' in the lexeme of the words that are possible hosts for infixation. For example, the entry for Tagalog *punta* 'go' could be:

```
    go : [cat=V,gloss=go,lex='p%u%nta',Red=pu].
```

Note the use of quotes in order to demarcate the lexeme string when it contains non-alphanumeric characters. The utility of the feature Red=su will be explained below.

Place-holders are consumed from left to right. If a rule inserts an infix, this is put in the place of the first place-holder ('%'). If other rule needs to insert a new infix, it will be placed in the next place-holder (if any), and so on. Consequently, the treatment of infixation depends heavily on the relative order of the morphological blocks.

Reduplication (and even internal modification) may be simulated in FGW by the use of infixes and place-holders. Consider reduplication. In Tagalog, many verbs reduplicate the first syllable of their lexeme in most cases of present or future tense. This may be described by the M-operation below:

```
    cat=V, tns!=Past, Red=$A :: INFIX $A
```

This rule gets the value of the feature Red (reduplication) in the form of a variable, and treats it as an infix.

15. Spelling rules

Spelling rules state the changes of graphemic or phonologic nature that are produced in the concatenation of two strings X and Y (X+Y), normally a lexeme and an affix.

These rules conform to the following format:

MP *Name*: [*Context*$_{Left}$] A/B **+** [*Context*$_{Right}$]

The substring A at the end of the first element of a concatenation is replaced with the substring B when immediately following the specified left context and immediately preceding the specified right context.

MP *Name*: [*Context*$_{\mathrm{Left}}$] **+** A/B [*Context*$_{\mathrm{Right}}$]

The substring `A` at the beginning of the second element of a concatenation is replaced with the substring `B` when immediately following the specified left context and immediately preceding the specified right context.

Notice that:

- The sign + refers to the morpheme boundary in the concatenation, and needs to occur immediately before or after the `A/B` pair (thus signalling whether the change occurs in the first or the second element of the concatenation.

- `A` and `B` may be a character or a substring.

- The trigger form `A` may be expressed also as 0 (zero). This facility is used to implement epenthesis (see examples below).

- The replacee form `B` may be expressed also as zero. This facility is used to implement deletion of the trigger (see examples below).

- The context forms include characters, substrings, disjunctive sets, and abbreviations.

- Context forms may be repeated to a undefined extension prefixing them with an asterisk.

Disjunctive sets are groups of characters or substrings that may occur in a context position. They have to be enclosed in curly brackets, as shown in the examples below.

Abbreviations are an easy and compact way to refer to disjunctive sets that are used repeatedly in the formulation of spelling rules; normally they correspond to phonological natural classes, such as vowels, labial consonants, sibilants, etc. Abbreviations must be previously declared, introduced with the keyword **AB**, as the following examples illustrate:

```
    AB Voc: {a e i o u}.
    AB Sib: {s z sh zh ch}.
```

Different examples of spelling rules may illustrate the points discussed above:

(a) `MP V_Del: Voc + V/0.`
(b) `MP D_Assim: {p t k s sh ch f x} + D/t.`
(c) `MP D_Def: + D/d.`
(d) `MP vIns: o 0/v + {o e a}.`

Rules (a)-(ac) are from Uzbek, and rule (d) is from Punjabi.

Rule (a) says that the morphophoneme V is deleted after a vowel (`Voc` is an abbreviature, defined as above). For instance in Uzbek, the possessive first-person singular suffix is *-Vm*. The concatenation *olma* ('apple') + *Vm* produces *olmam* ('my apple').

Rule (b) exemplifies a case of assimilation: the morphophoneme *D* is realized as *t* just after a unvoiced consonant (here represented by the disjunctive set `{p t k s sh ch f x}`. For example, *eshik* ('door') + *Da* (locative) produces *eshikta* ('at the door').

Rule (c) is an example of a default feature (without any context). it states that the morphophoneme *D* realized as *d*. Note that this rule only makes sense if placed after all the other possible rules that handle *D*.

Rule (d) adds an epenthetic `v` between a morpheme ended by *-o* and a morpheme beginning with a front vowel (expressed here by the disjunctive set `{o e a}`). For example, in Punjabi *ho* ('be') + *e* produces *hove*.

Operation:

Spelling rules are applied in the very same order in which they are declared in the grammar file. Usually, the first rules are more specific, whereas the last rule is always the most general or default case. Consider the rules (a-b) above, and the explanations given there.

16. Sandhi rules

Sandhi rules carry out phonological modifications between the final words in the clause. Some examples are the cases of so-called contraction (*do* + *not* > *don't*) and cliticization (Spanish: *da* + *me* + *lo* > *dÃ¡melo* 'give me that').

These rules rules have the format:

SANDHI: $\texttt{Word}_1$ + $\texttt{Word}_2$ = *CONTRACTION*.

For example:

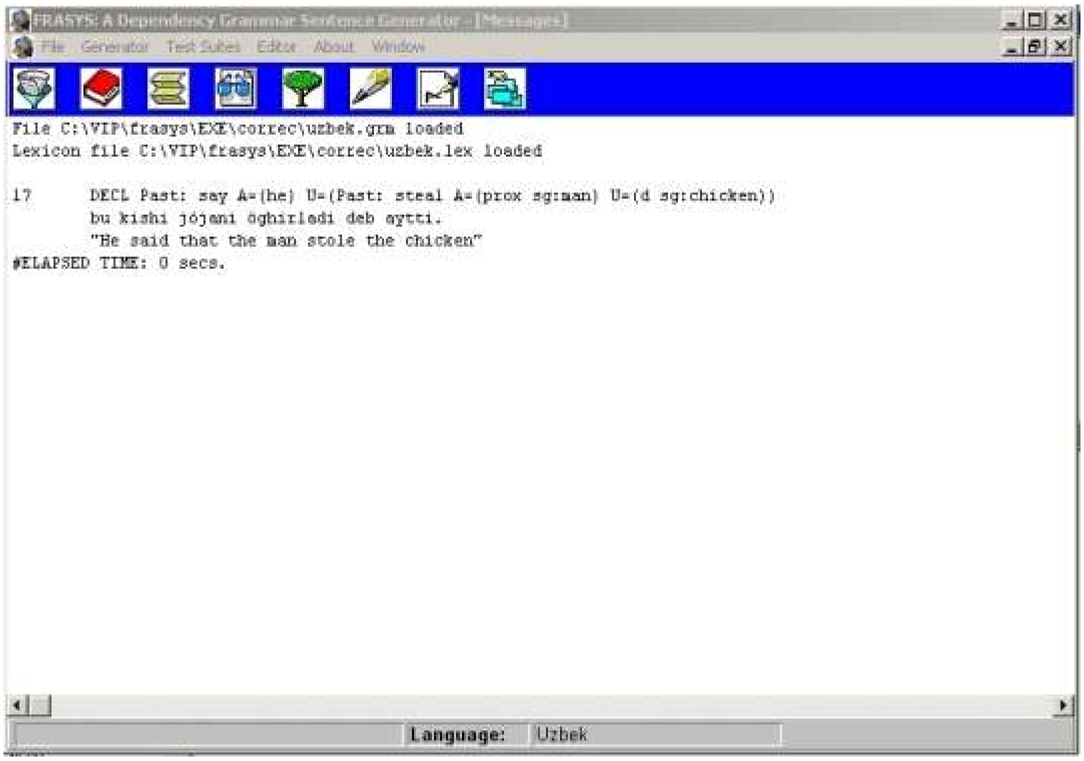
```
SANDHI: do + not = 'don\'t'.
SANDHI: de + el = del.
```

Double quotes are used in the first example because of the use of the quote (that is escaped with the backslash in order to tell the compiler that it does not indicates the end of the string.

17. Setting spypoints

FGW also offers the possibility of placing spy-points within the expression rules. They consist of the keyworsd **SPY** followed by a string (its identifier). For example, `SPY here`. They are proper statements, and must be separated with by a semicolon of the surrounding constraints. Their effect is to show the D-Tree corresponding to the point of derivation in which it is encountered. Used sensibly, spy-points are a useful tool to find out if a rule fails because of one particular constraint, or not.

18. The User Interface


FRASYS: A Dependency Grammar Sentence Generator - [Messages]
File Generator Test Suites Editor About Window
Language: Uzbek

```
File C:\VIP\frasys\EXE\correc\uzbek.grm loaded
Lexicon file C:\VIP\frasys\EXE\correc\uzbek.lex loaded

17      DECL Past: say A=(he) U=(Past: steal A=(prox sg:man) U=(d sg:chicken))
        bu kishi jójani óghirladi deb aytti.
        "He said that the man stole the chicken"
#ELAPSED TIME: 0 secs.
```

The user interface of FGW distributes the main functions over a the following menus:

File Menu:

Load Grammar File

(Also accessible by clicking the icon on the toolbar.)

This command loads the contents of a grammar file into memory.

If an error is detected, a message shows up in the output window, and the approximate location of the error is displayed. A common error is to omit the period that should be written after each declaration or rule.

Load Lexicon

(Also accessible by clicking the icon on the toolbar.)

This command loads the contents of a lexicon file into memory.

Load Test suite

(Also accessible by clicking the icon on the toolbar.)

This command loads a suite file of test predications into memory, that may be subsequently generated, one by one (by means of the option `Choose Test Predication` in the `Generator` menu), or in batch mode (by the option `Process Test Suite` in the same menu).

Test suite files should have the `'.tst'` extension. Note that as text files, they must be edited with an appropriate text-editor (such as the Windows Notepad).

A test suite file contains a set of test items, to be used mostly for checking the operation of a developing grammatical description.

Each test item is a sequence of four distinct lines:

- The first line is a separator, expressed by two hyphens (`'--'`).
- The second line is the 'translation' of the predication in a language familiar to the user.
- the third line is a predication, formulated in FGW notation.
- the last line is a string with the expected produced form. This is used in order to check out if the process of realization has succeeded.

The file may contain single-line comments. They may appear only between after the separator line, and are signalled by a prefix `'%'` in the first column of the line.

Here is an example of a test item:

```
--
% Comment
The child is playing
DECL NonPerf: playV A/Top=(sg: child)
usi yewe
```

<u>Save data in VIP files</u>

This option dumps all statements, rules and lexicon in memory into a file, using Visual prolog (VIP) internal format.

This option is only for debugging purposes; it is not useful for the novice user.

<u>Exit</u>

This command terminates the execution of FGW.

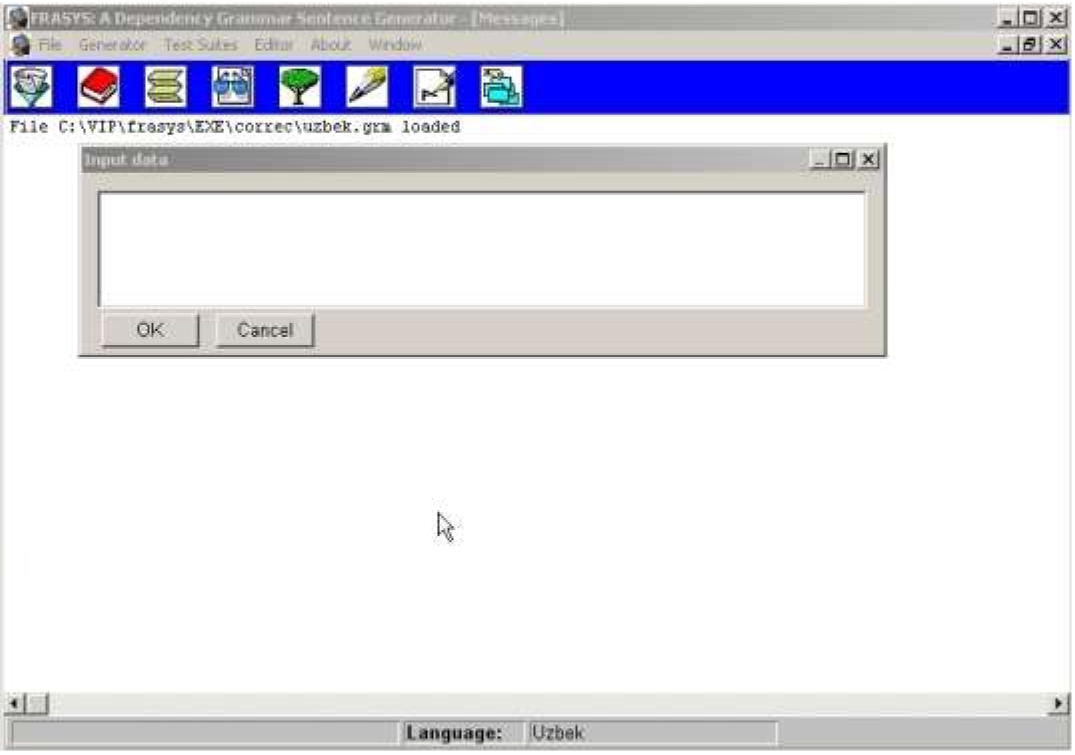
Generator Menu:

<u>Input Clause</u>

(Also accessible by clicking the 🖉 icon on the toolbar.)

This command allows the user to test the grammar of the target language, by introducing an underlying predication. FGW generates a sentence encoding this meaning in that language, provided that the corresponding grammar and lexicon files have been previously loaded.

This command activates a dialog window as shown below, where the predication can be written, or pasted by the combination of the keys Shift + Ins.



```
File C:\VIP\frasys\EXE\correc\uzbek.grm loaded
```

Retry Last Clause

(Also accessible by clicking the icon on the toolbar.)

Thos option opens a window containing the last input predication, that can be modified or transformed before trying it back again.

Choose test Predication

(Also accessible by clicking the icon on the toolbar.)

This option allows the user to select one underlying predication from a previously loaded test suite file, and produce its surface realization in the target language.

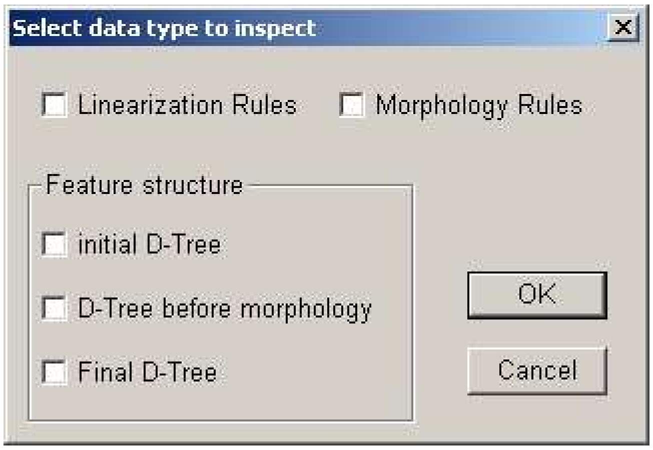
Lexicon Statistics

This option displays some information on the number of different lexical categories used in the entries of a loaded lexicon file.

Trace Mode

(Also accessible by clicking the icon on the toolbar).

This option opens a check box that allows the user to identify which rules will be applied in the derivation, and also inspect the details of different intermediate matrices produced during the derivation. Several or all of these options may be selected simultaneously.



- **Linearization Rules**: This shows which linearization rules have been applied for the lienarization of the nodes of the D-Tree. It displays information as the following:

  ```
  1 / 2 by rule #DEF
  2 / 4 by rule #Top
  ```

  These lines say (a) that the node 1 is ordered in front of the node 2 by the application of the lienarization rule `DEF`, and (b) that the node 2 precedes the node 4 by virtue of the rule `Top`.

- **Morphology Rules**: This option shows the name of the blocks that have been applied during the phase of morphological synthesis.

- **Initial D-tree**: This option displays the D-Tree after lexical expansion. (The only difference with the input predication is that the metalinguistic predicates have been translated into the target language.)

- **D-Tree before morphology**: This option displays the D-Tree after the application of the insertion rules, immediately before the linearization phase.

- **Final D-Tree**: This option displays the D-Tree after the application of the placement rules, and before the application of the rules of morphological synthesis.

D-Trees are displayed in text mode, as the following example shows:

```
 -- MAIN:  var (be_located)
        |-- U/Top:  resturon (restaurant)
        |-- Loc:  yoni (near)
        |       |-- Restr:  köl (lake)

FEATURE MATRICES:
================
: [pol=Af, domain=clause, lex=var, cat=V, gloss=be_located, vclass=intr,
illoc=DECL, tns=Pres]
: [domain=term, ntype=com, per=3, lex=resturon, cat=N, gloss=restaurant,
def=d, num=sg]
: [domain=term, per=p3, lex=yoni, cat=N, gloss=near, ntype=adv]
: [domain=term, ntype=com, per=3, lex=köl, cat=N, gloss=lake, def=d, num=sg]
```

Each node is identified by a number, in brackets; after it, the functions that connect the node to its governor, followed by the lexeme (and evenetually the gloss) of the corresponding node. Note also that dependency is marked by indentation.

The feature matrices of each node are listed just below the D-Tree, identified by their node number.

Trace Tree

(Also accessible by clicking the 🌳 icon on the toolbar).

This option toggles the trace mode in or off. When in trace mode, FGW shows information about the side-effects of expression rules.

Consider the following example:

```
1 ------------------------------
  Promotion of U (Subj_Assignment)
  Unification (Tense)
  Unification (SV_Agreement)
2 ------------------------------
  No expression rules applied!
3 ------------------------------
  Unification (PossN_Agreement)
4 ------------------------------
  No expression rules applied!
```

For each node (identified by its number) the modifications performed by expression rules on the D-Tree are listed; for example the message `Promotion of U (Subj_Assignment)` above after reports the promotion of the current node (1) to Subject status, performed by the `Subj_Assignment` rule.

The number that appear in the display of this option is the same used in D-Trees, as obtained in Trace Mode.

Test Suites Menu:

Process Test Suite

This option allows the user to generate all the predications contained in a test suite file, provided they are loaded already into memory. The results are saved in a text file called after the language name (as declared in the grammar file), prefixed with the prefix `Test-`. For example, if the name of the language is 'English', the production of a test suite will be saved in a file called `Test-English.txt`).

Error Report

Produce Hand-out

Editor Menu:

Set editor

This option allows the user to set the path to its preferred text editor.

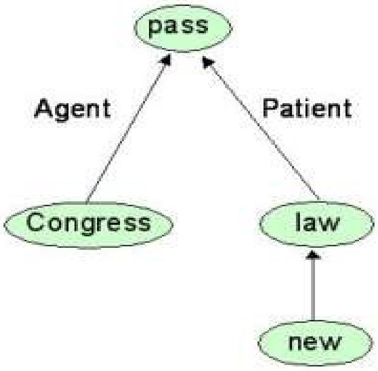
Launch editor

This option launchs a text editor, provided it has been previously set.

19. Some Terminology

a) Dependency Tree (D-Tree)

A Dependency Tree is a representation of the structure of a sentence as a network of nodes, in which there are nodes for individual words but no nodes for phrases. The nodes are all connected by relations of semantic and syntactic or pragmatic nature. For example, in the sentence *The Congress passed the new law* , the nouns *Congress* and *law* depend on the verb *pass* (as Agent and Patient, respectively), and the adjective *new* depends on *law* (as a Modifier), but there is no explicit node for the noun phrase constituted by the noun *law* and the adjective *new*. These relationships may be displayed graphically by means of dependency trees (D-trees) as the following:



b) Feature Matching

A feature `F=V`$_1$ in a D-Tree and a constraint `F=V`$_2$ match if:

- V_1 and V_2 are identical strings;
- V_2 is a disjunction of the form `(A|B|C)`,such that V_1 is one of its possible values;
- V_2 is an anonymous variable (ex. `num=$`)
- V_1 is a subtype of V_2,as stated in the feature declarations of the grammar file.

c) Feature Matrix

A set of features, normally displayed in list format:

```
[domain=term, sem=inanim, ntype=com, per=3, lex=moshina, cat=N,
gloss=car, def=d, num=sg]
```

d) Path Evaluation

For a path `E`$_1$`/.../E`$_{n-1}$`/E`$_n$, the sequence of features `E`$_1$`/.../E`$_{n-1}$ is followed until its end. Then, the feature `E`$_n$`=VALUE` is looked for; if it is found, the evaluation results to `VALUE`; otherwise, a **delayed** feature is created. Delayed features are evaluated at the end of the application of all cycles of expression rules.

The evaluation of a path fails if the final node cannot be located.

e) Path Traversal

This procedure returns the index of a node in the D-Tree that is linked to the current node by a given relation.

A path element may be of the following types:

- A disjunction, expressed as `(F`$_1$` | ... | F`$_n$`)`.
- A multiple function (expressed `*F`): zero or more instances of the required function.
- A plus function (expressed `+F`): one or more instances of the required function; at least one.
- `DEP`: any dependent of the dependent node.
- A variable (a name prefixed by `'$'`): matches any dependent of the current node.
- `GOV`: the governor of the current node.
- `PRED`: the semantic predicate of the current node (if any). Otherwise, the `PRED` is the node itself . The semantic predicate is a semantic function that must be declared in advance.
- `MAIN`: The highest node of the D-Tree.
- `ANTECEDENT`: if the current node is anaphoric (i.e. it has a `binding` feature with a `Coref=ID` feature), the antecedent is other node in the D-Tree that has the feature `Index=ID`.
- an atomic function (F): it must match any of the relations (semantic, pragmatic or grammatical) of the current node.

References

ANDERSON, Stephen R. (1992). *A-Morphous Morphology*. Cambridge University Press

AISSEN, J. and David PERLMUTTER (1983) Clause reduction in Spanish. In D. Perlmutter (ed.) *Studies in Relational Grammar 1*, University of Chicago Press.

DIK, Simon C. (1991) "Functional Grammar". Chapter 7 in F. G. Droste & J. E. Joseph (eds.) *Linguistic Theory and Grammatical Description.* Benjamins. pp. 247-274.

FRASER, Norman (1994). Dependency Grammar. In R. Asher (ed.) *Encyclopædia of Language and Linguistics*. 860-4: Pergamon.

JAKOBSON, R. (1963). On linguistic aspects of translation. In Reuben A. Brower (ed.) *On translation*, Harvard University, Press; Also in R. Jakobson 1971 *Selected Writings* vol. II.

MEL'ČUK, I. A. (1988). *Dependency syntax : theory and practice.* Albany: State University Press of New York.

NOONAN, Michael (1985) Complementation. In T. Shopen (ed.) *Language typology and syntactic description*. Vol. II, Cambridge University Press, 42-140.