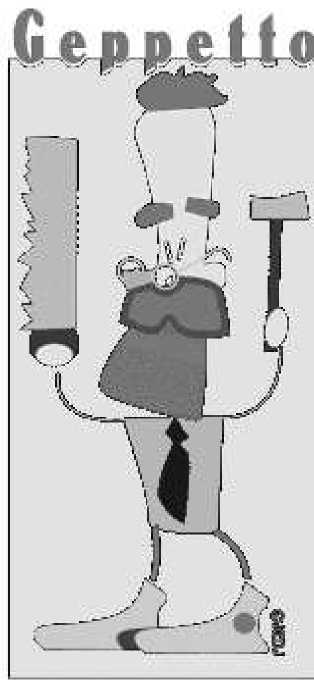

The Geppetto Development Environment

Version 2.1 User Manual

Fabio Ciravegna, Alberto Lavelli, Daniela Petrelli, and Fabio Pianesi
e-mail: {cirave|lavelli|petrelli|pianesi}@irst.itc.it

4th November 1997

Istituto per la Ricerca Scientifica e Tecnologica
Loc. Pantè di Povo I-38050 Trento, Italy

Contents

1 Introduction **4**

2 The TFS Formalism **5**
2.1 The syntax of basic TFS descriptions 5
2.2 Declarations 6
2.3 External constraints 7
2.4 Appropriateness and typing 8
2.5 Static typing 9
2.6 Unifiers 9

3 Linguistic Systems **11**
3.1 Compilation 12
3.2 Macros 12

4 External Interfaces **12**
4.1 External Interface to KB 13
4.1.1 TFS::COMPUTE-CONSISTENCY Input 13
4.1.2 TFS::COMPUTE-CONSISTENCY Output 15
4.1.3 TFS::COMPUTE-CONSISTENCY Body 15

5 Graphical User Interface **16**
5.1 Terminology, Notation and References 16
5.2 The GEPPETTO Monitor Window 17
5.2.1 The Menu Bar 17
5.3 The Resource Composer Window 19
5.4 Editing Linguistic Resources 20
5.4.1 The Macros Browser 21
5.4.2 The Lexicon Browser and the Grammar Browser windows 22
5.4.3 The Hierarchy Browser window 23
5.5 The Linguistic System Editors 24
5.5.1 The Macro Editor 25
5.5.2 The Lexical Item Editor and the Grammar Rule Editor 26
5.5.3 The Hierarchy Item Editor 29
5.6 Testing and Debugging 29
5.7 Parser and Generator Chart Browser 31
5.7.1 Parser Chart Browser 31
5.7.2 The Generator Chart Browser 33

6 Installation **33**
6.1 System Requirements 33

7 Acknowledgments **34**

A Known Bugs **35**
A.1 Monitor 35
A.2 Browsers 35
A.3 Hierarchy 35
A.4 Miscellanea 36

B Application Programmer Interface for Parsing **37**
B.1 Parsing with GEPPETTO 37
B.2 Inserting a Preprocessor 37
B.2.1 Recognizing Words in a Sentence 38
B.2.2 Creating Word Items 38
B.2.3 Creating a Word Item Instance 39

B.2.4 Prototypical Lexical Entry . . . 40
B.2.5 Using a Preprocessor: an Example . . . 41
B.3 Creating a New Variable . . . 42
B.4 Accessing the Final Logical Form . . . 42
B.5 Inserting Edges in the Chart at Parse Time . . . 42
B.6 Data Structures . . . 42
B.6.1 The CHART . . . 43
B.6.2 Edges . . . 43

C Linguistic Resources and Processors **44**
C.1 A Linguistic System for Italian . . . 44
C.1.1 Type Hierarchy . . . 44
C.1.2 Grammar . . . 47
C.1.3 Lexicon . . . 48
C.1.4 Macros . . . 49
C.2 Parsers . . . 51

List of Figures

1 GEPPETTO Monitor at start up time in `Edit Resources` layout (the default layout). . . . 17
2 The Architecture Composer window. . . . 19
3 The Grammar Browser window. . . . 20
4 The Macros Browser window. . . . 21
5 The Lexicon Browser window. . . . 22
6 The Hierarchy Browser window. . . . 23
7 The Macro editor. . . . 26
8 The Lexical Item editor. . . . 27
9 The Grammar Rule editor. . . . 27
10 The Hierarchy Item editor. . . . 28
11 GEPPETTO Monitor window in `Test System` layout. . . . 30
12 GEPPETTO Monitor window in `Edit & Test` layout. . . . 30
13 The Parser Chart Browser in sequential layout . . . 31
14 The Parser Chart Browser in in&out layout . . . 32
15 The Generator Chart Browser . . . 33

1 Introduction

GEPPETTO is an environment aiming at facilitating the development of linguistic modules and resources for NLP. GEPPETTO provides facilities for:

- editing and debugging grammars and lexica;
- linking linguistic data to a parser and/or a generator;
- integrating domain knowledge stored in KBs;
- using already available specialized processors (e.g. morphological analyzers).

GEPPETTO is based on a Typed Feature Logic [Carpenter, 1992] oriented formalism for specifying linguistic data. TFL specifications are compiled into a graph format, where each node represents a Typed Feature Structure (TFS). The TFL standard formalism has been modified to accommodate:

- *Declaration statements* specifying, for instance, that a certain object is not an ordinary TFS. In case its properties must be assessed by other, possibly external, modules, such a fact can be specified by means of ...
- ... *External constraints*. They provide explicit links to external modules, e.g. morphological processors, independent KBs, etc.
- *Directives* for the unifier. For instance, it is possible to force the unifier to consider certain paths in the first place. Such paths may be those that have been observed to cause more frequent failures [Uszkoreit, 1991].
- *Macros*.

Declarations statements and external constraints have been employed to interface the grammars and lexica produced by means of GEPPETTO to existing KBs containing domain knowledge.

GEPPETTO provides a number of unification algorithms among which the user can select the one that best suits his/her purposes. The algorithms have been designed to:

- carefully control and minimize the amount of copying needed with non-deterministic parsing schema [Wroblewski, 1987] [Kogure, 1990]; as is well known, this is crucial to reduce both time and space requirements at runtime;
- provide a better match between the characteristics of the unifiers and those of the linguistic processors using them. It can be observed, in fact, that different linguistic processors may profit of different unification algorithms. GEPPETTO allows the user to choose the unification algorithm best suited to the needs of the particular linguistic processor at hand.

Other relevant characteristics are the following:

- Types are implemented as bit-vectors [Ait-Kaci *et al.*, 1989]. This move permits to efficiently handle very large type hierarchies as well as providing a straightforward way to account for type disjunction.
- A memory management schema has been designed to control the amount of garbage collection done at run time. In particular, TFS-graph nodes creation can be reduced by accessing a stock of available nodes.

The editing environment offers graphical facilities for editing linguistic data:

- A grapher for editing the type inheritance hierarchy. The grapher displays mouse sensible nodes, allowing the user to input the TFL specifications for each type of the hierarchy.

- Browsers for grammar rules, lexical items and macros.
- Specialized windows for TFL constraint editing.
- TFL-syntax error checking.
- Debugging facilities.

GEPPETTO makes available a library of linguistic processors: chart parsers (island driven, CYK) and generators (non-deterministic Head-Driven Bottom-Up [Pianesi, 1993]) together with the possibility of integrating user's designed ones. As said above, the user can specify the unification algorithms more appropriate to the chosen linguistic processor.

GEPPETTO has already been used for the development of Italian grammars and lexica within applicative projects: for instance, for the Italian tactical component of a Natural Language generation system (LRE-GIST project) and for the development of a prototype for Information Extraction from texts.

As it should be clear from the above description, one of the key feature of GEPPETTO is the customizability of the environment to users' needs. As for the types of users, three main categories are supported:

- system manager
- grammar developer
- application developer

According to such different users' needs, 2 different environments are under development:

- development
- delivery

The system described in this document is the development environment used by the grammar writer.

Fabio Pianesi has designed and implemented the TFS formalism, the unifiers and the graphical interface of Version 1.0 of GEPPETTO. Alberto Lavelli has implemented the two parsers (CYK and head-driven) integrated within GEPPETTO and has written both the grammars and the lexica developed for Italian; he has also supported everybody else in using Common Lisp. Fabio Ciravegna has designed and implemented the external interface to knowledge bases; he also tested GEPPETTO in the development of a prototype for Information Extraction from texts. Daniela Petrelli has designed and implemented the graphical interface of Version 2.x of GEPPETTO.

2 The TFS Formalism

2.1 The syntax of basic TFS descriptions

The syntax of the TFS description is reproduced below:[1]
(1)

a. Desc → Type-Desc | Conj-Desc | Disj-Desc | Path-equation | Path-Disequation | Path-Assignment

b. Type-Desc → '(' Type-Label ')' | '(' Type-Desc1 '+' Type-Desc1 ')' | '(' Type-Desc1 '/' Type-Desc1 ')' | '(' '~' Type-Desc1 ')' | '(' '{' (Type-Desc1) $^+$ '}' ')'

[1] Please, note that currently full-blown disjunctive descriptions are not supported. The only type of disjunction is that of atomic types.

c. Type-Desc1 → Type-Desc | Type-Label

d. Conj-Desc → '(' Desc '&' Desc ')' | '(' Desc '&' Conj-Desc ')'

e. Disj-Desc → '(' Desc 'or' Desc ')' | '(' Desc 'or' Disj-Desc ')'

f. Path-Assignment → '(' '<' (Feat)+ '>' '←' '(' Type-Desc ')'[2]

g. Path-equation → '(' '<' (Feat)+ '>' '=' '<' (Feat)+ '>' ')'

h. Path-disequation → '(' 'NOT' Path-equation ')'

According to (1)a, a TFS description can be: a type description, a conjunctive or a disjunctive description, a path equation or a path disequation, or, finally, a type assignment to a path. The simplest form of type description, cf. (1)b, consists of a type label enclosed in parenthesis. More complex forms permit to specify operations on types: type joins (operation symbol "+"), type difference (operation symbol " / "), type negation (operation symbol "∼"). All these infix operation symbols can apply to type descriptions, recursively, as shown in (1)b. Finally, a type description can be any non-empty sequence of type descriptions.

Conjunctive and disjunctive descriptions are sequences of TFS descriptions linked by the symbols "&" and "or" respectively.

Path assignments are statements specifying the type that is to be attached at the end of a given path (where a path is a sequence of features, as usual), cf. (1)f.

Path equations, cf. (1)g, require two paths to share their terminal points and are used to explicitly introduce shared values.

Path disequations, cf. (1)h, are the negation of a path equation. They require that the two paths figuring in the embedded path equation statement never share their value.

Besides this much of basic syntax for TFS specifications, a number of additional features are available.

2.2 Declarations

(2)

Declarations → '(' 'DECL' Decl-Spec ')'
Decl-Spec → '(' Decl-Head Decl-Body ')'
Decl-Head → 'SPECIAL-NO-ENV'
Decl-Body → 'PATH+' Path $^+$

A declaration is a way to specify that certain typed feature structures (or nodes thereof) behave in a certain manner. A TFS declaration breaks down into the keyword "DECL" and a declaration specification, the latter consisting of a declaration head and a declaration body. Typically, declaration bodies are introduced by the keyword "PATH+" followed by a non empty sequence of feature paths. Decl-Head states that all the TFSs attached at the end of each path behaves in the way it specifies. At the moment, the only "Decl-Head" available is "SPECIAL-NO-ENV", which requires the TFSs to be simple special nodes (cf XX for a discussion of special nodes). Notice, finally, that when a declaration appears within a TFS specification it has lexical scope in the sense that it does not affect the previous portions of specification. Care must be taken, then, to ensure that no inconsistency arise. Consider for instance the following TFS-specification:

(3)
(......
< A B C > = < D E F >
......
(DECL (SPECIAL-NO-ENV (PATH+ < A B C >)))
.......)

[2]Note that the *left-arrow* character (←) is used in the syntax of path assignment because the first version of the TFS system was developed in the Medley environment, where such character is shown instead of the *underscore* character (_); in the current version of the system you have to use the *underscore* character instead of the *left-arrow* character.

When translating such a TFS-specification, the compiler first encounters the path equations and assigns the nodes at the end of both paths the default status of normal TFS nodes. The following declaration, however, requires one of such nodes to be special, a conflicting requirement that causes the compiler to issue an error message. The correct statement is as follows:

```
(4)
(......
(DECL (SPECIAL-NO-ENV (PATH+ < A B C >)))
......
< A B C > = < D E F >
......)
```

2.3 External constraints

External constraints are a mechanism which allows a module built from GEPPETTO to exchange information with external modules. In general, we will maintain that every non-TFS based module is external. To such an extent, therefore, a knowledge base is an external module, morphological processors are external modules and the Lisp itself is an external module. Combine with the possibility of declaring certain nodes as special, external constraints permits TFS to carry along information that would otherwise be non computable (or less efficiently computable) within the TFS system, reuse available resources and so on. In a sense, external constraint are interfaces between the TFS system and external processors.

As for the implementation, external constraints are sorts of procedural attachments. For a given TFS F, an external constraint is a function which computes from the values associated with certain addresses within F (that is, with certain sub-TFSs of F) and returns a value that must be stored at another specified address within F. The syntax of the specification for an external constraint is the following one:

```
(5)
External-constr-spec → '[' constraint-name '(' parameters ')' place ']'
parameters → path *
place → path
```

Here, *constraint-name* is the name of the constraint, *parameters* is a sequence of paths specifying the addresses where the values are to be found and *place* is the path at the end of which the returned value must be stored.

Important notice. As said above, external constraints typically let values be computed by external modules and store the result as non-TFS objects. Such non-TFS objects can be lisp symbols, numbers, lisp lists, etc.. It is, therefore, necessary to instruct the unifier that the addresses where such non-TFS values are to be stored must not undergo the normal process of unification. Therefore, it is necessary that every path mentioned within an external constraint specification be previously declared as SPECIAL-NO-ENV.

At the moment, the following external constraints names are available.

- **kb-consistency**. This calls a function performing consistency checks on a knowledge base. It takes 5 arguments, as explained in XX below.
- **kb-consistency-lex**. Takes 5 arguments and performs consistency checks on semantic arguments introduced lexically; see XX.
- **kb-type-compatibility**. It computes an ISA test. Takes 4 arguments.
- **kb-type-intersection**. It computes the intersection of two semantic types.
- **run-type-lisp-symbol**. It returns a lisp symbol. It takes no arguments
- **create-a-constant**. It returns a semantic constant. It takes one argument, see XX.
- **set-logical-form**. It computes a logical form (see XX) from a structural representation. It takes one argument.

- **type-equal**. It performs an equality test on semantic types.

An external constraint can be required to be executed only during the time a certain process is active, e.g. only during parsing or at the time a lexical item is drawn from the lexicon and used by the parser, etc. To control that the execution of an external constraint is performed at the right time a *process-flag* is used. The process flag can assume one of the following values:

- `:AT-PARSE-TIME`. The constraint is to be executed only when the parser attempts at constructing a new edge.
- `:AT-USE-DATA-DAG-TIME`. This is reserved to TFSs for lexical items and grammatical rules. It specifies that, when the TFS is extracted from the grammar or from the lexicon, certain computations must be performed before the TFS can be used for parsing purposes.
- `:AT-COMPILE-TIME`. This instructs the compiler that it must execute the corresponding computations.

The evaluation of an external constraint can be delayed until certain conditions are not met. For instance, an external constraint which requires the KB to compute a logical form can be delayed until all the relevant information are available. The delay cannot exceed, however, the extent of the process. Hence, if the external constraint computing logical forms has the process flag `:AT-PARSE-TIME` its execution can be delayed until the system is in parsing mode. It is an error if an external constraint survives the process which must evaluate it. For instance, if an external constraint that must be executed `:AT-PARSE-TIME` is still active on the final edge, that edge is not well-formed.

2.4 Appropriateness and typing

Upon compilation of one TFS description the compiler produces a feature structure which, for our purposes, can be represented as a graph whose nodes are labeled by types. Such feature structures are not typed, since no general typing scheme has been introduced yet and any feature can be applied to any type. To restrict such a possibility, a typing scheme is needed that regiments the association between features and types and provides information about the kind of values the feature can have in that context. When considering this question, we talk of the **appropriateness** of a feature for a given type. Two typing discipline can be considered: **weak** and **strong** typing. The former only requires that whenever we attach a feature to a given type, that feature be appropriate for that type and takes on an appropriate value. Strong typing does something more: whenever we have a given type, all the features that are appropriate must be present and given an appropriate value. Both regimes support type inference, e.g. from a given (non-disjunctive) description (which may even be impoverished in terms of typing information) we can deduce a unique minimum (strongly or weakly) well typed satisfier. However, since strong typing is more expensive, in terms of space requirement, GEPPETTO supports only weak typing.

The notion of appropriateness can be rendered by means of a partial function mapping types and features into types, this way specifying if the feature is appropriate for the given type and, furthermore, providing restrictions on its values.

Definition. Appropriateness Specification - An appropriateness specification over *(TYPE, $\leq$)* and *FEAT* is a partial function *Approp: (FEAT$\times$TYPE) $\rightarrow$ TYPE* such that:

- (**minimum introduction**) For every feature $f \in FEAT$, there is a minimum type *MinIntro(f)* such that *Approp(f, MinIntro(f))* is defined;
- (**upward closure**) If *Approp(f, σ)* is defined and $\sigma \leq \tau$, then *Approp(f, τ)* is also defined and *Approp(f, σ)$\leq$Approp(f, τ)*.

The *minimum introduction condition* requires that for every feature there is a minimal type for which it is appropriate. GEPPETTO does not require the user to define such a minimal type, but computes it at compile time. The *upward closure condition* secures that appropriate

features are inherited in the type hierarchy. That is if f is appropriate for a given type it is also appropriate for all its descendants. Furthermore, the appropriateness values reflect the ordering in the type hierarchy; in other words, the appropriateness function is monotonic.

The editing facilities of GEPPETTO simply require the user to input, for a given node, the appropriateness specification for the features that are not inherited. The latter are not accessible to the user and feature inheritance is taken care of by the system itself.

The typing discipline serves to secure that, for a given node in a typed feature structure F and a given feature, the value of such a feature in F is appropriate, according to the appropriateness specification. If this is the case for every feature in F then the feature structure is said to be *(weakly) well typed.*

2.5 Static typing

A system is statically typed if all the type inference can be performed at compile time. In the normal case, after having unified two (weakly or strongly typed) FSs, type inference must be performed on the result in order to make it well-typed. This is not necessary with static typing, since the unification of two well-typed FSs always produces a failure or a well-typed FS. A common way to achieve this result is to impose conditions on the appropriateness specification. [Carpenter, 1992] analyses two such conditions: constant appropriateness and join preservation.

- **Constant appropriateness** - This amounts to requiring that the appropriateness function be, for every feature, a constant, i.e. for every $f \in FEAT$ and $\sigma, \tau \in TYPE$, $Approp(f, \sigma)=Approp(f, \tau)$, when both are defined. With this we have that: - in the case of weak typing, if F and F' are weakly well typed then $F \cup F'$ is weakly well typed - in the case of strong typing, we need to modify the algorithm for unification so that, whenever a node is created for which features are appropriate that are not present in the two unifiands, such features are introduced and given suitable values. With this modification, and if the appropriateness condition is loopless, the unification of two strongly well-typed FSs produces a strongly well-typed FS.
- **Join Preservation** - Considering features as functions from types into types, join preservation amounts to requiring that features behave like an homomorphism, i.e. $f(\sigma \cup \tau)=f(\sigma) \cup f(\tau)$. Recast in the terminology of the appropriateness function, join preservation amounts to the following: $Approp(f, \sigma \cup \tau)=Approp(f, \sigma) \cup Approp(f, \tau)$, if both are defined, else, if $Approp(f, \sigma)$ is defined then $Approp(f, \sigma \cup \tau)=Approp(f, \sigma)$, else it is undefined. Again, the unification of two (weakly or strongly) well-typed FSs will produced a well-typed FS.

GEPPETTO supports the *join preservation* strategy. It does so automatically, warning (at compile time) the user when the relevant conditions have been violated.

(Both static typing discipline can be shown not to cover all the cases. Sometimes, it is actually necessary to perform (at least part of) type inference at run time, that is when two TFSs are unified.)

2.6 Unifiers

At present, three unifiers are available in GEPPETTO: a destructive one, a dynamic copy version and a third unifier that is called *asymmetric*. For an explanation of the way grammar rules encode constituency, see [Shieber, 1986; 1992].

Each unifier consists of two modules: the unification algorithm and the copy schema.

The destructive unifier destructively modifies one of the two unifiands to accommodate the result TFS. It can only be executed on unifiands that will not be reused. In case the

original information must be preserved, the two unifiands must be copied and the unification algorithm must be applied to the copies. At present, the destructive unifier is used only by the compiler.

Notice that, in order to accommodate external constraints, the object all the unifiers operate upon are not TFSs but dotted pairs consisting of a TFS and a list of active external constraint; let us call such an object a *TFS-constraint-pair*.

The relevant functions for the destructive unifier are the following:[3]

- `D-TFS-UNIFY`. It takes three arguments: *TFS-constraint-pair1*, *TFS-constraint-pair2* and a *process-flag*. The *process-flag* is used by the module resolving external constraints, see XX. This function returns a TFS-constraint pair such that the TFS component is the result of the unification and the constraint list is what remains after the result of the unification.

- `DESTRUCTIVE-TFS-EMBED-1`. It takes four arguments: *TFS-constraint-pair1*, *TFS-constraint-pair2*, a *path* and a *process-flag*. This function embeds the TFS corresponding to the first TFS-constraint-pair into the second TFS-constraint-pair at the address specified by *path*. Notice that embedding functions are the ones used by the parser.

- `TFS-COMPLETE-COPY-TOP`. It takes a TFS-constraint-pair and returns a copy of it.

Dynamic copy unifier. As before, it consists of two modules, the unifier and the copy algorithm. It differs from the destructive version in that it does not require previous copy of to-be-preserved TFSs. Rather, it performs unification in the usual manner and saves the results in temporary stores. In case the unification succeeds the copy algorithm produces copy of all and only the nodes that have been changed during the unification phase. This unifier is used by the head driven parser. The relevant functions are:

- `TFS-UNIFY`. Its arguments are the same as those of `D-TFS-UNIFY`. It unifies the two TFSs according to the dynamic copy schema.

- `TFS-EMBED-1`. This is the embedding function for the dynamic copy schema. Its arguments are the same as those of `DESTRUCTIVE-TFS-EMBED-1`.

- `TFS-COPY-TOP`. This function applies after the unification has succeeded. It takes a TFS and return a copy of it.

Asymmetric dynamic copy unifier. It is a mixture of the previous two algorithms. It takes two TFS-constraint pairs such that the TFS of the first is to be treated as in the destructive mode and the second as in the dynamic copy mode. However, to avoid useless copying, it temporary stores the results, whenever possible, in the nodes of the first TFS. If the unification succeeds, the algorithm completely copies the first TFS. The second TFS is copied according to the dynamic schema. This algorithm is exploited by the CYK parser where the first TFS invariably corresponds to the grammar rule. An advantage of this unification schema, is that it can apply to more than two TFSs at time. Thus, given TFS_1,...., TFS_n, it uses TFS_1 in a (delayed) destructive mode and all the other TFSs in a dynamic-copy mode. The relevant functions are:

- `EXTENDED-ASIMM-EMBED-TOP`. This function takes a number of keys: - two TFS-constraint pairs, corresponding respectively to the first and the second constituent in a CYK-style combine task; a TFS-constraint pair for the rule; - two embedding paths, corresponding to the addresses for the first and second TFSs, respectively, in the grammar rule TFS. Finally, it has a key for the process flag (see above) and a key for the *extraction path*, that is for the address singling out the sub-TFS to be returned.

[3] All the functions listed in the following are in the package `TFS` and the symbols are not exported from such package.

- `TFS-ASIMM-COMPLETE-COPY`. This is the copy function, taking the TFS to be copied.

It should be noticed that, in the case of the dynamic copying schema and of the asymmetric one, the calls to the copying algorithm are managed by both `TFS-UNIFY` and `TFS-EMBED-1`.

3 Linguistic Systems

The basic objects that GEPPETTO deals with are *linguistic systems*. They consist of sets of data descriptions accessible to the user for the purpose of editing. Therefore, these objects are not the ones unification algorithms operate upon. The latter are obtained by compiling the linguistic system (see Section 3.1).

The components of a linguistic system are:

- **A Type Hierarchy.** This is a set of types together with their mutual relations, the feature appropropriateness specifications and (possibly) a TFS description attached to each type. Notice that type inheritance is not limited to feature appropriateness but extends to the TFS descriptions associated to each type. This fact might not be apparent at the editing phase, since TFS description inheritance is computed only during the compilation phase.

- **A Grammar.** Essentially, this is a set of TFS descriptions, each specifying a grammar rule. For the TFS system, a grammar rule is simply a TFS which is instance of a given type. At the moment, the following types are defined for grammar rules:

  - `psr`, the general type for phrase structure rules;
  - `prs-1`, the type for unary rules;
  - `psr-2`, the general type for binary phrase structure rules;
  - `psr-2-left-head` and `psr-2-right-head`, the types for left- and right-headed binary rules, respectively.

  The general format of a phrase structure rule is similar to the one adopted in [Shieber, 1992]. Each constituent is represented by a feature attached to the top level node of the corresponding TFS. In the present system `C0` corresponds to the lefthand side constituent whereas the righthand side constituents are identified by the features `C1`, `C2` and so on. At the editing level, the user must specify the TFS description for the rule. When a rule has been compiled and upon usage, during parsing or generation, the TFS structure for a given constituent of the righthand side of a rule is directly unified at the relevant address. For example, the TFS for the first son is "embedded" at the address `C1`.

  NOTE that in the current version of GEPPETTO there is a strict assumption regarding the name of the more general type for phrase structure rules: such type is `psr` and its name is contained in the Lisp variable `TFS::*PHRASAL-RULE-ROOT-NODE*`. Currently there is no way of changing this value other than setting a new value for the variable in the Lisp listener (and then recompiling the hierarchy); anyway, we suggest not to do that. In future releases of GEPPETTO we plan to introduce a suitable functionality in the API.

- **A Lexicon.** As with rules, a lexical item is a TFS which is an instance of a given type. When editing a lexical item, the user must essentially specify the corresponding TFS description.

- **A set of Macros.** For an introduction to macros, see XX.

3.1 Compilation

As said above, the objects contained in a linguistic system are not the ones the unification machinery operates upon. The latter are the minimal satisfiers for the TFS descriptions contained in a linguistic system and are obtained by "compiling" the linguistic system itself. Compilation produces data structure (TFSs) which are appropriate for the basic level machinery (unification algorithms). Compilation can be broken down into the following steps:

- Type hierarchy compilation. During this phase, the original type hierarchy is mapped into a more efficient representation by means of bit vectors (see [Ait-Kaci *et al.*, 1989]). Furthermore, the feature appropriateness functions are also given a more efficient representation for the use at run time and a number of missing information are computed, e.g. type intros. Finally, the TFS descriptions associated with each type are compiled into TFSs.
- Grammar compilation. A minimal satisfier is assigned to each TFS description in the grammar.
- Lexicon compilation. A minimal satisfier is assigned to each TFS description in the lexicon.

Among other things, the computation of a minimal satisfier for a TFS description involves:

- macro expansion: each macro appearing in the TFS description is expanded;
- inheritance of TFS description satisfiers: given a TFS minimal satisfier F for a TFS description, the TFS description for each type appearing in F is expanded in F.

3.2 Macros

A TFS macro is a convenient way to express generalisations and statements that can be reused, with slight changes, in different places. They are very similar to the old PATR-II style macros. A TFS macro has a name, which always start with the "at" sign '@'. Then it has a body and a number of parameters. The body of a macro is similar to normal TFS descriptions, the only difference being that it can contain parameters. The latter are metavariables that receive a value when the macro is evaluated (e.g. during compilation). Parameters always start with the "?" sign. Therefore, `?NAME` is a valid parameter for, say, the macro `@OK`. Notice that the usual conventions must be followed. More precisely, a lowercase parameter, e.g. `?type`, refers to a metavariable ranging on types whereas an uppercase parameter, e.g. `?FEATURE`, ranges on features (or paths). From the point of view of its usage, a macro "is" a TFS description and can be used whenever such an object is appropriate. Among other things, this means that one or more macros can appear in the body of another macro. When a macro is used, its parameters must be assigned values:

(@NAME $?para1 value1$?para2 value2 ...)

Therefore, a legal TFS description could look as follows:

```
(Desc1 & Desc2 & (@NAME $?PARA1 value1 $?para2 value2) ...)
```

where `?PARA1` ranges on features, so that value1 must either be a feature or a path, and `?para2` ranges on types. Notice, finally, that parameters in use, as in the last example, must be preceded by the dollar sign.

4 External Interfaces

Just from the beginning the formalism has been designed in such a way that it could be interfaced with external sources of information and knowledge, if necessary. This can be the case, for example, of the use of a morphological processor, a preprocessor for proper names and/or numbers, dates, etc. or of a knowledge base.

The interaction between the TFS formalism and the external source of knowledge is accomplished through the use of a combination of the external constraints described in section 2.3. Each of such external constraints *per se* do not do anything particularly meaningful, it is only their interaction that allows to import in TFS descriptions the external information in a usable way.

Something general about external interfaces

- get-old-type-labels
- get-...-type
- get-LF
- prototypes for numbers

4.1 External Interface to KB

As mentioned it is possible to ask the unifier to interact with some external knowledge bases (KB) and their representation and reasoning system, via the external constraints. In section 2.3 the following external constraints were mentioned among the others:

- **kb-consistency**. This calls a function that performs consistency checks on a knowledge base;
- **kb-consistency-lex**. it performs consistency checks on semantic arguments introduced lexically;
- **kb-type-compatibility**. It computes an ISA test.

The execution of all these constraints performs a call to a function in the TFS package whose name is COMPUTE-CONSISTENCY. Here follows that function documentation.

```
Function:   TFS::COMPUTE-CONSISTENCY
Parameters: Test-Type
            LogForm-1 LogForm-2
            VAR-1 VAR-2
            &optional ARG-NAME
```

This is a function that receives two logical forms and returns a third one IFF the unification of two given variables is semantically acceptable. It is generally used for semantic discrimination.

4.1.1 TFS::COMPUTE-CONSISTENCY Input

The function takes as input 6 parameters (one optional).

- *Test-type* is one of the following:
  - :KB-CONSISTENCY-LEX: the test is requested by the unifier during the building of some lexical items; it is used to build the logical form of the lexical item. It is something not to be checked by the KB for consistency, but just a request for building a logical form composed by the append of the two received as input. One logical form is always NULL and the other should be returned and the variables unified.
  - :KB-CONSISTENCY: it is a request to test for a specific predicate. The input QLF must be appended (they are lists) to form a unique QLF and the two variables in var-1 and var-2 must be unified (i.e. one of the two must be substituted with the other in the consed QLF). The semantic test to be performed is that of consistency of the resulting QLF.

- :KB-TYPE-COMPATIBILITY: this is a request for checking for the unification of two variable types. This is necessary in constructions like appositive clauses, where the semantic head of a variable must be unified with another. In many cases there will not be any differences between the way this test is implemented and the way :KB-TYPE-COMPATIBILITY will (for example when a knowledge representation tool is able to test for consistency of logical forms). But in other cases the implementation will differ (especially when the KRT used is not very powerful).

- *LogForm-?* second and third parameters are two structures containing two pieces of information:
  - A CONVERSION TABLE: this is a table were the unification of variables is recorded by associating the name of one of the two with the other. The flow of variable unification then generates a chain of renomination in the table. For example if var_a and var_b are unified, the entry for var_a is left as it is and that of var_b is set to var_a. The table is used to retrieve a unique referent for any unified variables: for example when in the following steps var_a is unified with var_c (and consequently the entry for var_a is set to var_c), it is possible to demonstrate that the transitive property of unification still holds by following iteratively the chain in the table:

    $var_b \rightarrow var_a \rightarrow var_c$

    The table is then used to assert that if $var_a \cup var_b$ and $var_a \cup var_c$, then $var_b \cup var_b$.
  - A QUASI LOGICAL FORM: this is a list of predicates in AND. No disjunction is supported in Version 2.1. Each predicate is either a list of two or three elements:
    * 2-elements predicates: they have a form like: "(SL::COMPANY X)" and are used to define the type of a variable.
    * 3-elements predicates: they are something like: "(SL::HAS-COLOR X Y)" and define relations holding between two variables. In this case Y is the color of X. The first variable is the object modified by the predicate, whereas the second is the "value" of the predicate.

    For example a logical form like:

    ```
    ``((TO-GO X) (ACTOR X Y) (PERSON Y))''
    ```

    means that X is an action of going performed by a person. In general relations are expressed via their domains and ranges. For example an expression like "The book about cars" is expressed via the following form:

    ```
    ``((BOOK X) (ON-ARGUMENT-rel Z) (CAR Y) (DOMAIN Z X) (RANGE ZY))''
    ```

    That means that "ON-ARGUMENT-rel" is a two places relation whose fillers are a book and an argument.

- *VAR-?*: these are the variables to be unified during the composition of two QLFs. VAR-1 is to be found in LogForm-1, VAR-2 in Logform-2.

- *Arg-name*: It is an optional parameter indicating which is the predicate that will hold between the two variable. Sometimes (NOT ALWAYS!) it is known by the syntax and passed to the semantic module. The function must be very careful in accepting the truth of that parameter because sometimes something very strange can be passed; it is always a good rule to check for the presence in one of the Logforms for the specific predicate. If not found, the function must look for the correct predicate by herself by searching the two QLF.

4.1.2 TFS::COMPUTE-CONSISTENCY Output

The function can return two type of output:

- a LogForm structure: if the test has been successfully performed;
- :DELAY: if the function does not receive enough information for answering the test; this happens for example when one of the QLFs or variables is T;
- NIL if the test was not successful.

4.1.3 TFS::COMPUTE-CONSISTENCY Body

A simple body of the function performing just the append of QLFs (i.e. no KRT is used to test the consistency) is provided in the file "semantic-check.lisp". Here follows the same code commented.

```
(defun COMPUTE-CONSISTENCY (Test-Type LogForm-1 LogForm-2
                            VAR-1 VAR-2 &optional ARG-NAME)

    ;; incomplete information. return :delay
  (COND
    ((or (null VAR-1) (equal var-1 (list t))
          (null VAR-2) (equal var-2 (list t))))
      :DELAY)

    ;; enough information received
    (T
     (let* ((ris-fin NIL)
           ris-test
           var-a var-b bottom
           table-1 table-2 table-fin)

       ;; separation of QLF from variables renomination tables
       (setq lf-1    (GET-LOGICAL-FORM-OBJECT-LOGICAL-FORM LogForm-1)
             table-1 (GET-LOGICAL-FORM-OBJECT-RENAMING-TABLE LogForm-1)
             lf-2    (GET-LOGICAL-FORM-OBJECT-LOGICAL-FORM LogForm-2)
             table-2 (GET-LOGICAL-FORM-OBJECT-RENAMING-TABLE LogForm-2))

      ;; the previous variable unifications are taken into account
      ;; variables are assigned to bottom of the chain in the
      ;; normalization table
      (setq var-a (GET-VAR-RENAMING VAR-1 table-1)
            var-b (GET-VAR-RENAMING VAR-2 table-2))

      ;; normalization of the two QLF using the variables
      (setq LF-2 (normalize-LF LF-2 var-b (list VAR-2 ))
            LF-1 (normalize-LF LF-1 var-a (list VAR-1 )))

      ;; one of the two var is decided to be the result of
      ;; their unification
      (cond ((eq var-a var-b) (setf bottom var-b))
            ((constant? var-b) (setf bottom var-b))
            (t (setf bottom var-a)))
      ;; normalization of QLF using bottom
      (setq LF-2 (normalize-LF LF-2 bottom (list var-b))
            LF-1 (normalize-LF LF-1 bottom (list var-a)))
```

```
;; the two QLF are appended
;; HERE INSERT THE SEMANTIC CHECK OPERATING ON
;; THE RESULT OF  APPEND
(setq ris-fin (APPEND LF-1 LF-2))

;; if the test result is not NIL
(when ris-fin
  ;; create a new conversion table for variables
  ;; by merging the two input tables
  (setq table-fin (MERGE-RENAMING-TABLE table-1 table-2))

  ;; renaming of the variables in the resulting table
    (RENAME-VARIABLE var-b  bottom table-fin)
    (RENAME-VARIABLE var-a  bottom table-fin)

  ;; create the resulting structure
  (MAKE-LOGICAL-FORM-OBJECT  ris-fin table-fin)))))
```

5 Graphical User Interface

5.1 Terminology, Notation and References

Throughout this manual some keywords are used to identify portion of the user interface. These are the fundamental definitions:

- Pane identifies a single elementary graphical component of the user interface;
- Frame is an organized set of Panes grouped for functionalities and graphically separated from others;
- Window is similar to Frame but it is physically separated.

A *slanted style* is used to indicate pane, frame or window names.

Other self-explaining terms, such as buttons, scrollable lists, dialog boxes etc., are used.
When not explicitly said differently, *to click* means to press the left mouse button.
In general, a MOTIF look and feel has been adopted getting advantage of some of its sophisticated widgets (i.e. file browser or dialog boxes) wherever possible. Please refer to MOTIF Style Guide for doubts on look and feel.

A `typewriter style` indicates a command that can be accessed via menu or button. `Command 1`→`Command 2` indicates the command path to activate `Command 2`.

Geppetto user interface has been studied using a participatory design approach. User interface design principles and guidelines have been taken into account in the implementation phase [Ciravegna *et al.*, July 1997].
To let the user better understand and use the Geppetto environment, explicit references to the design rational are reported along this chapter.
The system has been tested with users, but it can surely be improved, so suggestions and criticisms are welcome (`geppetto@irst.itc.it`).
To visually identify which pane is editable and which is not, a different background color has been used throughout all the Geppetto interface windows. So the panes where is not possible to type in will have the same color as the window background, while the editable ones will have a lighter color (in the default Geppetto X resources).

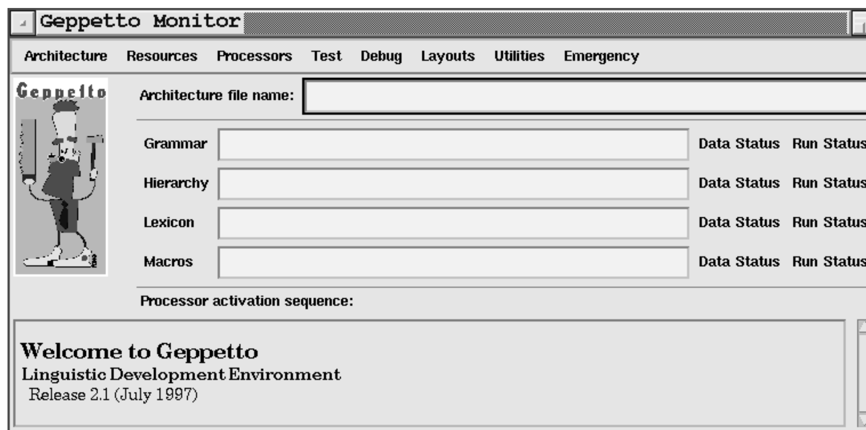

Figure 1: GEPPETTO Monitor at start up time in `Edit Resources` layout (the default layout).

5.2 The GEPPETTO Monitor Window

GEPPETTO is an interactive environment for developing Linguistic Engineering Architectures (LEAs or *architectures* for short) based on Typed Feature Logic. A set of linguistic resources and a (set of) processor(s) have been identified as core elements of an architecture. The set of linguistic resources is composed by a type hierarchy, a grammar, a lexicon and a set of macros; at least one processor (a parser or a generator) has to be added to complete an architecture.

The development of an architecture has been logically broken into different phases: editing the linguistic resources and/or testing the architecture. GEPPETTO has three different layouts for the *Monitor* window to better support these phases. The layout opened at startup time (figure 1) supports editing (see discussion below in this section) while the other two are meant to support testing (see section 5.6) or the two activities intermixed.

Apart from the *Menu Bar* described in detail below, the editing layout mode contains:

- the *Geppetto Logo* pane; clicking on the logo an information box is displayed;
- the *Architecture Feedback* pane that displays information on the current architecture. Apart from the architecture filename, it reports the linguistic resource filenames together with their status with respect to the need of saving and compiling them (`Saved` or `Modified` for the data file and `Compile` or `Run` for the internal structure). The list of the currently active processor sequence completes the architecture status feedback (not available in Version 2.1).
- the *Message* pane, where actions performed by the system and by the user are recorded.

5.2.1 The Menu Bar

The *Menu Bar* provides many actions grouped in eight sets: `Architecture`, `Resources`, `Processors`, `Test`, `Debug`, `Layouts`, `Utilities`, and `Emergency`. Each group is discussed below.

Architecture It contains the actions to be performed on the architecture as a whole.

- `Load...` loads an already existing architecture file script that contains the reference to the resource files and the processor setup. Users select it through a file browser box.
- `New` resets GEPPETTO to let the user create a new architecture from scratch.
- `Load and Compile Resources...` is the same as executing the two commands `Load...` and `Resources`→`Compile` in sequence (see above and below).
- `Save` saves the current architecture definition as a script file.

- `Save As...` saves the architecture in new file script. The user is prompted with the window for selecting the new filename.
- `Quit` exits the GEPPETTO environment.

Resources It contains commands that act both on the whole resource set (`Compose...`, `Save As...`, `Compile`) and on the single data sets.

- `Compose...` opens the *Resource Composer* window that allows to select the resource files (see 5.3). The selected grammar, hierarchy, lexicon and macro files will be the current architecture resources;
- `Save As...` opens the *Resource Composer* window to specify a new filename for each resource file;
- `Compile` compiles all the linguistic resources of the current architecture. It is mandatory to do that before entering the testing phase, i.e. before analyzing a sentence the linguistic system must be compiled. Note that the compilation command does not imply any action of saving the compiled data on file; the effect of the compilation is simply that of translating the data from the format in which they are written on file into an internal format, suitable of being executed by the unification engine. Currently there is no way of saving on file the results of the compilation.

After linguistic resources has been loaded (using `Composer` or `Architecture→Load`) or created (with `Architecture→New`), the specific menus for the four linguistic resources (grammar, type hierarchy, lexicon, macros) are available. On each of the four components it is possible to perform the following actions:

- `Edit...` opens the browser of the component. This browsers gives an overview on the resource as a whole and allows to access to each single resource element (see below for a complete description);
- `Compile` compiles this component (Not available for the Macros);
- `Save` updates the file of the component;
- `Save As...` (not yet implemented).

Processors This menu contains the commands related to the processors use.

- `Set Flow` allows to set the current sequence of processors activation in the current architecture. `Parser` and `Generator` commands open windows where the available processors are listed in an acceptable activation sequence: the user simply clicking on them can change the current setting;
- `Add Processor` allows to add a user defined processor to the ones already available. (not implemented in Version 2.1)

Test Most of these commands are available only in testing phase (i.e., after an architecture has been loaded and compiled).

- Run Test Suite runs the processor on a Test Suite. It is not available in Version 2.1.
- Set Log File... allows the user to set the file where the processing results will be saved during following executions;
- `Load Test Input...` copies the content of a user selected file into the Test Input pane.
- `Load Test Suite...` (Not implemented in the Version 2.1)
- Analyze Corpus runs the processor sequence on a corpus. It is not available in Version 2.1.

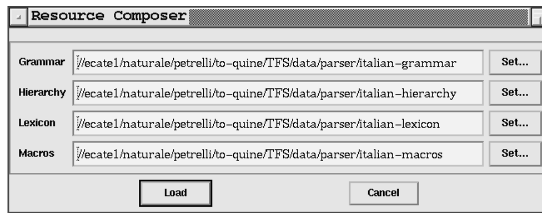

Figure 2: The Architecture Composer window.

Debug These commands starts the available debugging tools.

- Chart Browser starts the *Chart Browser* tool;
- Set Tracer sets the processors to work verbosely. It is not available in Version 2.1.

Layouts These commands change the Monitor window layout:

- `Edit Resources` sets the default layout, the one presented in figure 1, is focussed on giving the user the maximum feedback on the linguistic resources status;
- `Test System` sets the testing layout (see section 5.6 for the Test Frame Layout) supports the user in testing providing more space to input and result;
- `Edit & Test` sets a complete layout that has both editing and testing features.

Utilities It contains mainly commands for operating on GEPPETTO running mode.

- `Delivery System` saves the architecture in a run time format. (Not implemented in the Version 2.1)
- General
  - `Clear Messages` clears the *Message* pane.
  - `Clear Results` clears the *Test Results* pane: it is active only in testing layout.
  - `Clear Input` clears the *Test Input* pane: it is active only in testing layout.
  - `Garbage Collection` starts a global garbage collection.
- Preferences
  - `Work Silently` tells GEPPETTO not to display the less relevant messages, such as when opening browsers or editing windows (the default value is verbose).
  - `Memory Management` sets the memory usage (not available in Version 2.1).

Emergency It contains commands to reset a few environment conditions so that it is possible to recover from an emergency situation without exiting from GEPPETTO.

NOTE that if, for any reason, GEPPETTO collapses, it is sometimes possible to recover most of the done work invoking the following Lisp function: `(tfs-ui::recover-tfs-monitor)`

5.3 The Resource Composer Window

As previously discussed, a core concept in GEPPETTO is that of linguistic resources. These are: a grammar, a type hierarchy, a lexicon, and a set of macros. Each data set is saved in a different file.

The *Resource Composer* window, shown in figure 2, supports the user in defining the linguistic resources configuration, setting the four file names. The `Set` buttons allow to select a file through a *File Browser* window; the file name can be even directly typed in.

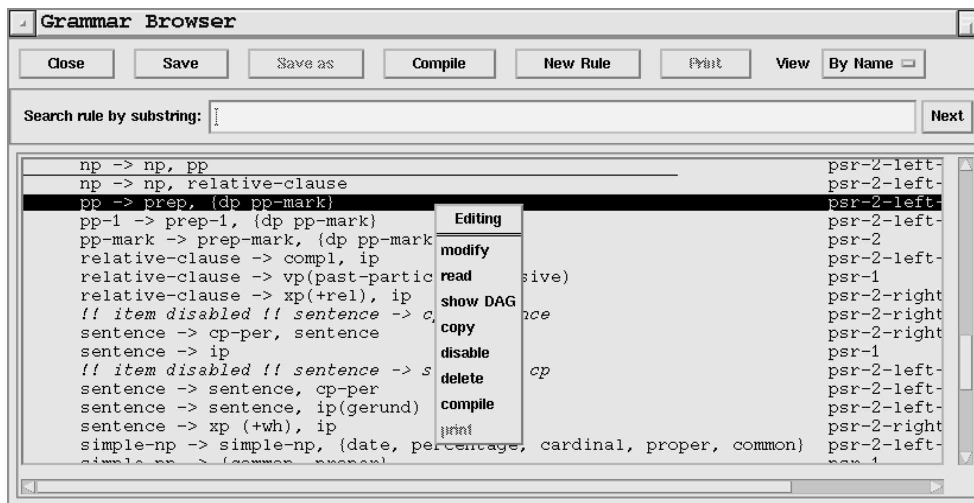

Figure 3: The Grammar Browser window.

Linguistic resources files can be shared among different linguistic system (the default mode). This means that modifies on one component effect others architectures. To avoid this it is necessary to create new resource files, to say to load an architecture or to compose resources, and save the component in a different files using `Resources`→`Save As` command. Giving a new file name to a component, a new empty resource set is created. This is useful when a component has to be created from scratch.

NOTE that in Version 2.1 no checking format is implemented, so be careful in selecting the filename: you risk to REWRITE AN EXISTING FILE WITHOUT ANY FURTHER NOTIFICATION!!!

5.4 Editing Linguistic Resources

Geppetto provides two levels of editors for each of the four linguistic data components. The one accessed by `Edit` works at an overview level (following called browser level) on the whole resource data set. Single resource items are then accessible through pop-up menu commands.

Except for the hierarchy one, the other three browsers share the same template. In this section the general description is provided, while specific features are described in dedicated sections below. The template browser window is divided in three parts (see figure 3).

The upper part contains the general-purpose actions (`Close`, `Save`, `Save as`, `New` *element*, `Print`) together with specific ones.

The `New` *element* button opens the specialized editor on an empty element.

`Save as` and `Print` commands are not provided in Version 2.1.

A few words for clarifying some terminology or behavior that can appear misleading. The data associated with each of the linguistic system components (i.e., type hierarchy, grammar, lexicon, and macros) are usually loaded from file; all the changes to such data do not affect the data stored on file until an explicit `Save` action is performed. The `Close` action simply closes the corresponding window but it is in no way equivalent to an abort of the current session of editing; the next time the user will open such window, the data shown will be the same as immediately before closing the window (even if a `Save` action has not been performed).

In the middle frame, commands to look up an item from the list below are provided. It is not necessary to fully type the element name: as soon as a key is pressed, the list is automatically scrolled to display the first matching element in the center of the pane. Pressing the RETURN key, this first matching element is automatically selected. A beep advises if no element matches the current selection string. Is is also possible to search for an element containing a substring, typing '*' as first character. The `Next` button can be used to find the next occurrence of the string in the list below. The search is case sensitive, so the name must

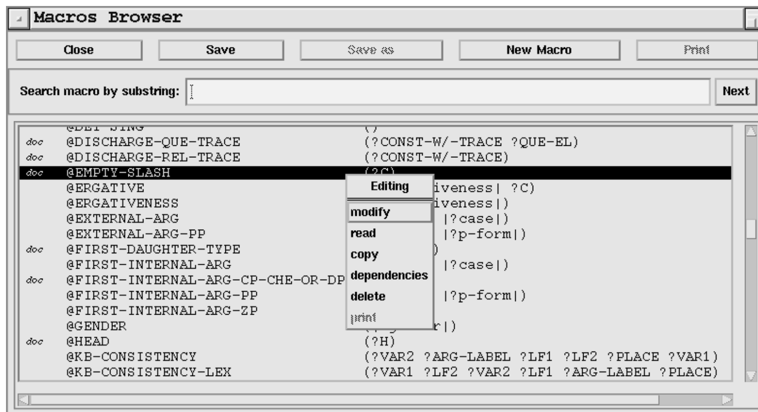

Figure 4: The Macros Browser window.

be rightly typed, including special characters, such as @. This is because uppercase letters as well as non alphanumeric characters may be used as essential part of item names.

The lower pane displays a scrollable list of the elements currently belonging to the component. Only main attributes are displayed here: a complete inspection of element attributes is possible using the editor.

The first column shows if documentation on the element is available: a click on *doc* displays the information box (see 5.5 for editing documentation facility).

A click on a list item selects it and pops up the menu. A click on a selected element deselects it. If an element is selected, click the right mouse button pops up the menu without deselecting it.

A multiple selection is still possible: keep the META key (the one with the diamond) pressed meanwhile clicking on elements. To pop up the menu when multiple selection is active, click (left or right mouse buttons) everywhere on the pane.

Mouse and search selection can be combined for multiple selection.

The menu presented for a single selection contains a set of commands common to all the three browsers, plus a set of special commands defined only for that linguistic system component. The common set includes: `Modify`, `Read`, `Copy`, `Delete` and `Print`.

The design of the command position is part of the rational behind the user interface.

The commands displayed on the top pane act on the whole set of elements, whereas the ones in the pop-up menu act on the selected element(s) (single or multiple). So the `Print` command on top, prints all the elements (the list) whereas the `Print` in the pop-up prints the single element (the full description as presented in the *Editor* window).

5.4.1 The Macros Browser

It provides only the general-purpose commands `Close`, `Save`, `Save as`, `New Macro` and `Print`. `New Macro` calls the *Macro Editor* on an empty macro record.

The macros list has three columns:

- Doc: if present, it shows that documentation for this macro is available: click on *doc* to see it;
- Name: the name of the macro;
- Parameters: the parameters of the macro.

The pop-up menu available for a single selection contains:

- `Modify` opens the editor on the selected macro; only one editor in modify mode can be open at any time;

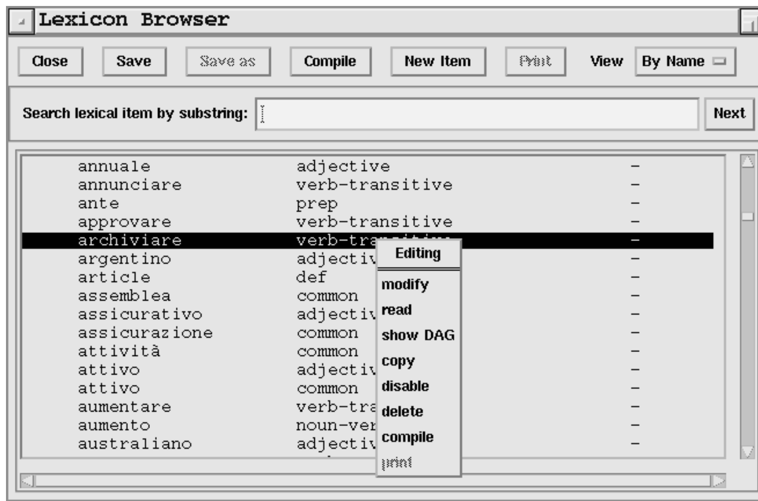

Figure 5: The Lexicon Browser window.

- `Read` opens the editor on the selected macro in read-only mode;
- `Copy` allows to edit a new macro copying the attributes of the selected one, i.e. the macro editor is open duplicating the values of the attributes; the macro name must be changed to have it added to the macros list;
- `Dependencies`, when available, displays in a separate window the oriented graph of the macro dependencies, i.e. all the macros that use such macro and all the macros used by such macro;
- `Delete` removes the selected macro from the macro set;
- `Print` prints the full description of the selected macro (not available in Version 2.1).

When multiple selection occurs, only `Delete` and `Print` are available in the pop-up menu.

5.4.2 The Lexicon Browser and the Grammar Browser windows

The *Lexicon Browser*, shown in figure 5, and the *Grammar Browser*, shown in figure 3, have the same functionalities, so a common description is given.

Besides the general commands described above, these browsers have a `Compile` command shown as button in the top part. This command compiles the lexicon/grammar.

Another element in the global command part is the `View` option button that allows to change the elements order from alphabetic on name to alphabetic on type and vice versa.

The lexicon list has four columns:

- Doc: if present, it shows that documentation for the corresponding item is available: click on *doc* to see it;
- Name: the name of the item;
- Type: the type of the item;
- Score: a score (or a probability computed using a corpus) associated with the item (in Version 2.1 only the score for grammar rule is implemented).

The pop-up menu available for a single selection contains:

- `Modify` opens the editor on the selected item (note that only one item at a time can be open in such modality);

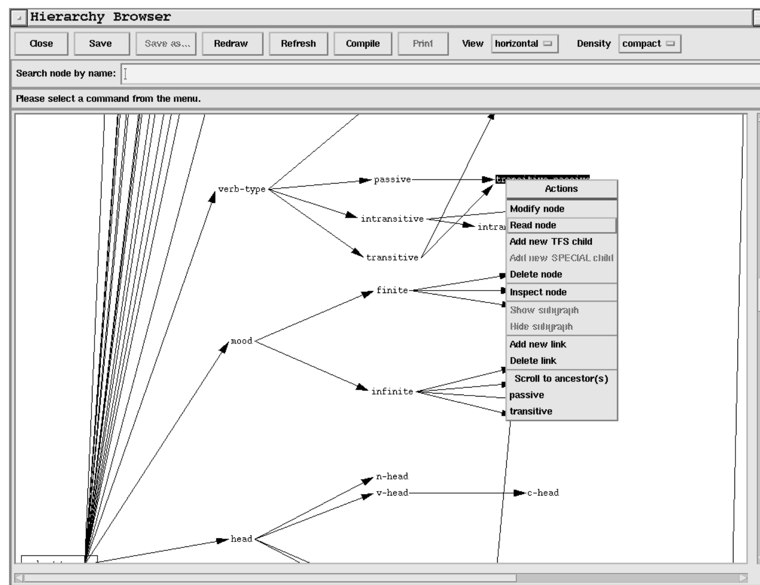

Figure 6: The Hierarchy Browser window.

- `Read` opens the editor on the selected item in read-only mode (you can open as many items as you like in read-only mode; compare with `Modify` above);
- `Show DAG` opens a windows that shows the DAG (Direct Acyclic Graph) associated with the selected item;
- `Copy` opens the editor on a copy of the selected item;
- `Disable/Enable` disable/enables the use of the selected item; i.e., the item is excluded from (included in) the runtime resources. This command can be used to run the system temporarily excluding some items, without deleting them. Disabled items are displayed in *italics*;
- `Delete` deletes the selected item;
- `Compile` compiles the selected item;
- `Print` prints the selected item (not implemented in Version 2.1).

When multiple selection occurs, only the commands `Show DAG`, `Delete` and `Print` are available.

5.4.3 The Hierarchy Browser window

The *Hierarchy Browser* (see figure 6) is essentially a grapher that allows the user both to browse and to modify the relationships between the types in the hierarchy.

The *Hierarchy Browser* window has four parts.

The upper part, as for the other browsers, contains the global commands. Besides the general-purpose commands, `Redraw`, `Refresh` and `Compile` are provided.

`Redraw` draws the whole hierarchy and is useful when the hierarchy is displayed only partially (see `Show subtree` and `Hide subtree` below) to restore the previous situation. `Refresh` draws the nodes and edges locally and can be used when the presentation is not perfect.

`Compile` compiles the hierarchy requiring the grammar and the lexicon to be recompiled too.

Option menus to change the hierarchy `View` (vertical or horizontal orientation) and the node `Density` (low, medium, high) allows to arrange the big graph in a more readable way.

The second section, below the commands, allows to scroll the hierarchy graph through a search by name.

The third is the message pane, where useful information on how manipulate nodes are displayed.

The lower scrollable pane displays the hierarchy graph.

It is possible to move a node in a new position: click the middle mouse button on the node and drag the mouse to the new position. *Anyway, the new position is not recorded from a working session to another one; so do not spend too much time to reorganize the hierarchy layout.*

A menu pops up when clicking the left mouse button on a node:

- `Modify node` opens the *Hierarchy Item Editor* on the selected node;
- `Read node` opens the *Hierarchy Item Editor* on the selected node in read-only mode;
- `Add new TFS child` creates a new TFS node as child of the selected node and opens the *Hierarchy Item Editor* on an empty node;
- `Add new SPECIAL child` creates a new SPECIAL node as child of the selected node and opens the *Hierarchy Item Editor* on an empty node (not available in Version 2.1);
- `Delete node` deletes the selected node;
- `Inspect node` opens the inspector on the selected node;
- `Show subtree` shows the subgraph having the selected node as root (fragile);
- `Hide subtree` shows the graph after hiding all the descendants of the selected node (fragile);
- `Add new link` allows to create a new link between the selected node (the ancestor) and another existing node (the descendent);
- `Delete link` deletes the link between the selected node and another node.

The *Hierarchy Browser* has been implemented using Grasper; the graphs produced by Grasper are not always completely satisfactory:

1. longs arcs connecting nodes that are very far from each other are sometimes interrupted in the middle: if this happens try to use the command Refresh;
2. if two nodes which are descendants of the same degree of the `bottom` are connected by an arc, such arc is drawn horizontally, to say overlaying other nodes;
3. the redrawing command, that is supposed to be useful to solve the problem reported in 1, automatically scrolls the window to the top left corner.

Moreover some Grasper commands seem to be fragile and not deterministic. We are working to solve some of the most annoying problems. Anyway if you are in trouble you can close the browser and open it again: this action recreates the hierarchy as it was displayed the first time, to say possible new positions of nodes are lost.

5.5 The Linguistic System Editors

As for the browsers, GEPPETTO provides a specialized editor for each linguistic system components.

Anyway, a common template has been designed: specific commands and panes are explained in dedicated paragraphs below.

The template editor window is composed by five parts:

- Documentation: here general information can be edited; this information is shown when clicking on *doc* column in the browser list (see 5.4). The difference between Documentation and Comment (see below) is that the former is meant as documentation on the item as a whole (for example for 'asbestos' lexical item this is the right place for information like: "the term 'asbestos' is used in medical text as a cause of cancer, whereas it is considered a raw material in commercial texts"), whereas in the latter information on the implementation of the lexical entry can be inserted (for example a comment like "3/12/96: substituted the macro '@Adjective' with '@Adjective- 1' because..." should be put here). This distinction has been inserted also to better support team development of LEA: a user has the possibility to leave documentation accessible at the browser level in order to let other colleagues to get an idea of the rational behind;
- Attributes frame: in the second part some special attributes of the element can be edited; the layout design changes depending on the edited component;
- Body: here the body of the element can be edited; what should be typed in here changes from component to component;
- Comment: in this pane comments related to the implementation (i.e. TFS description content) might be inserted; this comments are intended to be useful only at this level, to say they are not visible from outside the editor.
- Command (frame): three common commands are provided (some of the editors have further specific commands):
  - `OK`, after performing the check, the current values are saved and the editor window is closed;
  - `Check` runs a test to verify the correctness of the current values of the attributes (this check is usually a purely syntactic one);
  - `Cancel` closes the editor without saving the changes.

The specialized editor can be called both on an empty element to create a new instance or on an existing one to modify it. `Modify` can be accessed through the pop-up menu in a browser; `New` is activated by the `New` *element* in the specialized browser.

Please, note that currently in GEPPETTO there are only limited capabilities of undoing the actions performed. When you are editing an item you can decide to abort what you are doing before exiting the editor, but once you have exited the editor using `OK`, the changes are not reversible (at least, not automatically). Such capability will be added in future versions of the system.

5.5.1 The Macro Editor

The *Macro Editor*, see figure 7, is slightly different from the template described above.

It is used to modify an already existing macro, to create a new one from a copy or from scratch.

The *Attributes frame* contains only the macro name (that must start with @) and the parameters.

The only specialized command is `Dependencies` that shows the dependency graph as described in 5.4.1.

The *Body pane* contains the TFS description associated with the macro; it must be a list of Lisp S-expressions connected via "&". Please, refer to section 2.1 for details on the syntax of TFS descriptions.

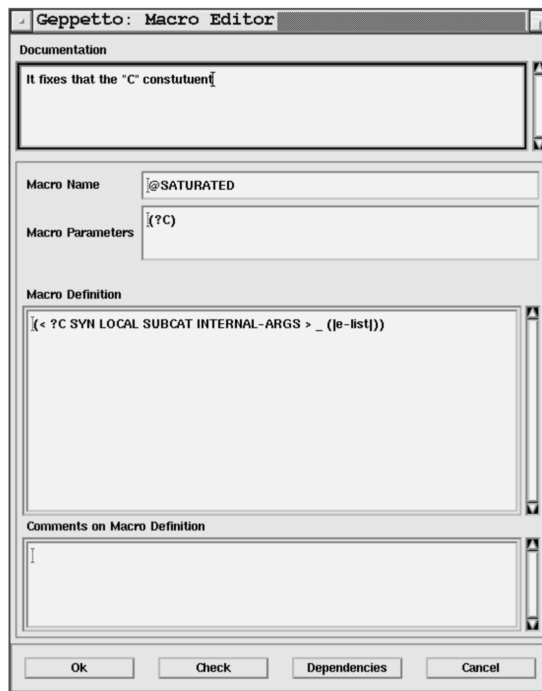

Figure 7: The Macro editor.

5.5.2 The Lexical Item Editor and the Grammar Rule Editor

As for the browsers, the *Lexical Item Editor* (figure 8) and the *Grammar Rule Editor* (figure 9) have the same functionalities, so a common description is given below.

The editor is used to modify the values of an element of the lexicon/grammar or to create a new one (from scratch or copying an already existing one).

Specific attributes contain: the name, the type (i.e. the lexical category, for instance *noun*) of the lexical item, the score (or a probability computed using a corpus) associated with the lexical item (not implemented in Version 2.1).

As for macros, a TFS description associated with the lexical item must be a list of Lisp S-expressions connected via "&". For example the definition for "company" could be:

```
((@COMMON-SING)
  &  (@COMMON-SEM |$?pred| SEMLEX::ORGAN-CONCEPT)
  & (@LOGICAL-FORM))
```

Refer to section 2.1 for details on the syntax of TFS descriptions.

The specific commands for lexical items and grammar rules are

- `Compile` compiles the lexical item (grammar rule);
- `OK & Compile` compiles the lexical item (grammar rule) and closes the window; more precisely, some checks are performed, the current values are saved, then the lexical item (grammar rule) is compiled and the editor window is closed; the rationale behind the use of this command is that the linguistic resource is ready to be used without the need of any further global compilation;
- `Show DAG` opens a window that shows the DAG associated with the lexical item (grammar rule).

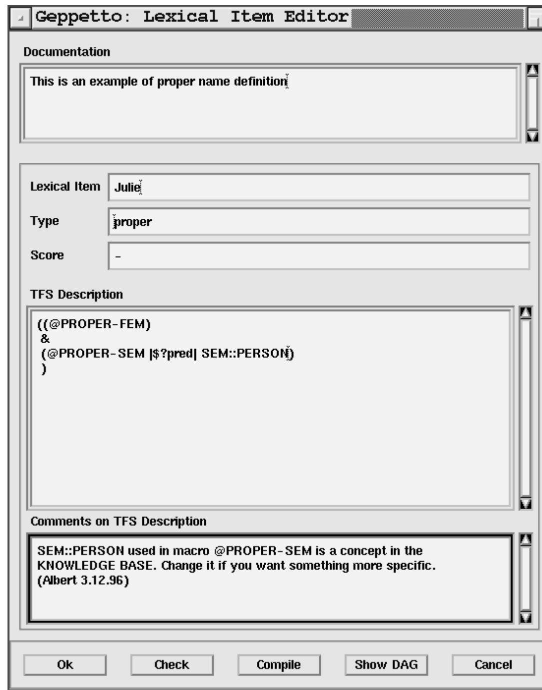

Figure 8: The Lexical Item editor.

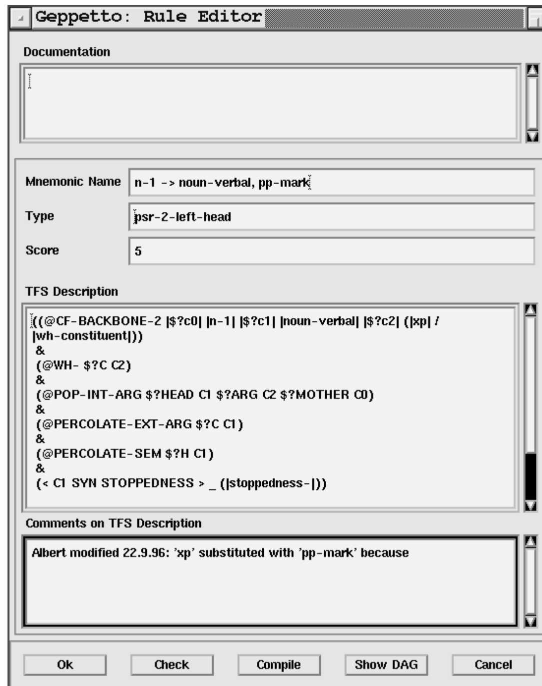

Figure 9: The Grammar Rule editor.

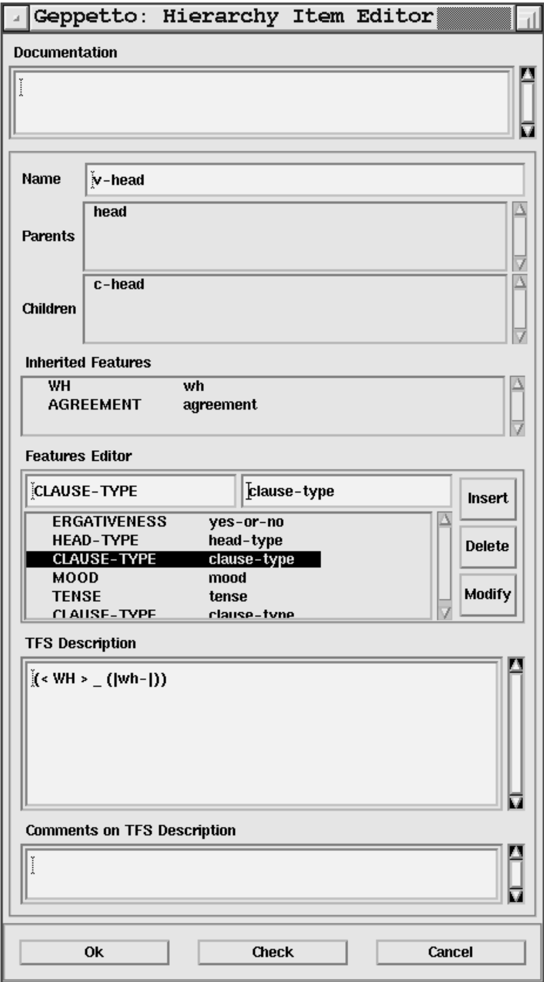

Figure 10: The Hierarchy Item editor.

5.5.3 The Hierarchy Item Editor

The *Hierarchy Item Editor* (figure 10) allows the user to modify the characteristics associated with a particular type.

The attribute frame is quite different from the previous one; it contains:

- Name is the name of the type;
- Parents shows the types that are parents of the current type;
- Children shows the types that are children of the current type;
- Inherited Features shows the feature inherited by the ancestors;
- Features Editor frame allows to modify the features (and their possible values) locally associated with the type.

  Note that when you use a certain type as the restriction of a local feature, such type must have already been defined in the hierarchy.

  In the upper part of such an editor, two editable panes contain the item that is currently edited, split in a left and a right part (containing the feature and its appropriate value, respectively).

  The lower part contains a selectionable list of the items already associated with the type. The item to edit can be selected clicking on it: the two components will be automatically inserted in the upper panes for further modifications.

  In the right part of the editor, there are three commands: `Insert`, `Delete`, and `Modify`. `Insert` is used to add the currently edited feature to the list; `Delete` removes the selected feature from the list; `Modify` substitutes the selected feature with the modified one.

  In future releases of GEPPETTO we plan to add some functionalities in order to give the user some help, e.g. providing a list of already defined types and/or already used features.

Parents, *Children* and *Inherited Features* panes are not editable because these lists are automatically computed when the Hierarchy Item Editor opens.

5.6 Testing and Debugging

GEPPETTO allows to test the current LEA using predefined file(s) or extemporary typed input. In the first case the user has to select the file through `Test`→`Load Test Input` command or directly typing into the *Current Suite* pane the pathname (press RETURN to load the file).

Release 2.1 allows to load a single file; i.e., testing against test suite (meant as a set of files) is not supported.

User can also directly type into the *Test Input* pane the input for the processor. This means that it is possible to load a predefined input file and than extemporarily modify the content before starting the run.

Figure 11 shows the GEPPETTO Monitor `Test System` layout. It is activable via `Layouts`→ `Test System`.

Figure 12 shows the GEPPETTO Monitor `Edit & Test` layout; this layout combines the features of the other two. It is activable via `Layouts`→`Edit & Test`.

To run the processor on the input click the `Run` button. The `Stop` button can be used at any time to interrupt an ongoing analysis/generation. Feedback from the analysis/generation process is displayed in the *Message* pane, while analysis/generation results are recorded in the *Test Results* pane.

When the analysis/generation is finished, the following buttons are available for inspecting the output produced by the linguistic processor:

- `Clear Input` clears the *Test Input* pane.
- `Clear Results` clears the *Test Results* pane.

Figure 11: GEPPETTO Monitor window in `Test System` layout.

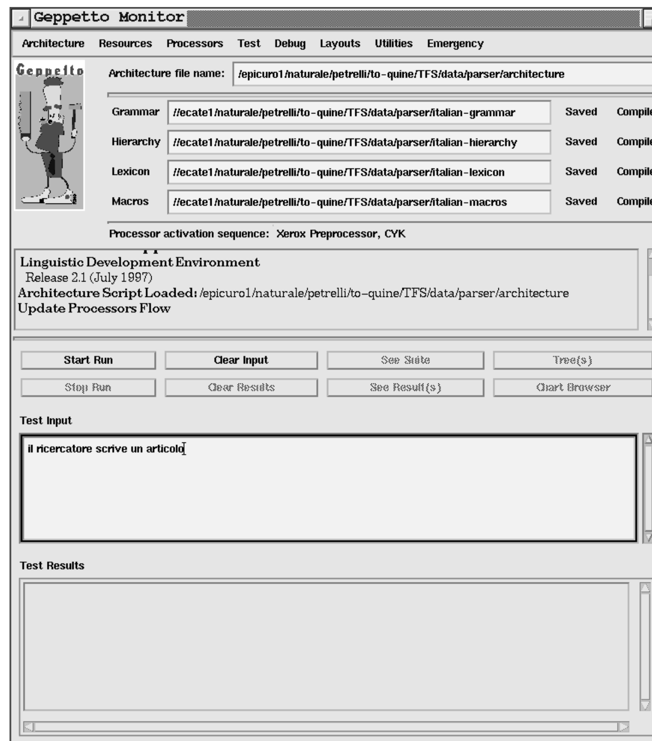

Figure 12: GEPPETTO Monitor window in `Edit & Test` layout.

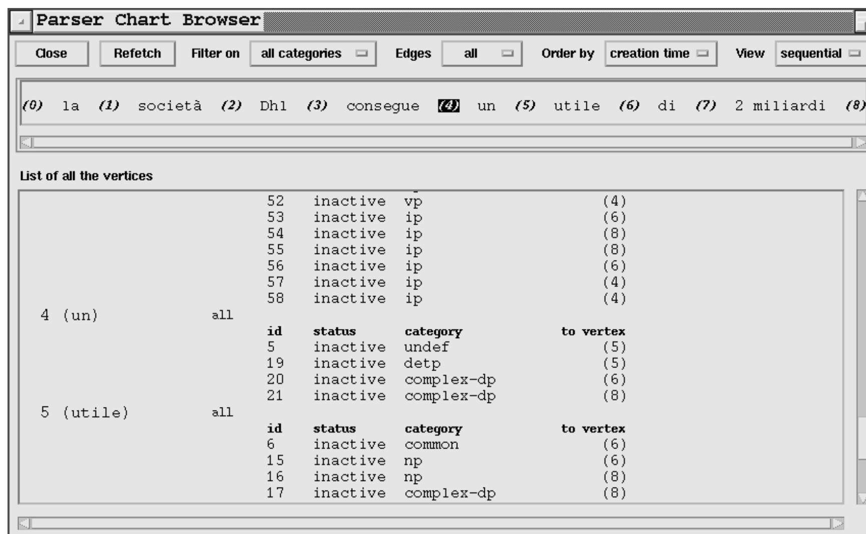

Figure 13: The Parser Chart Browser in sequential layout

- `See Input` open a window to allow a better browsing of input. Not available in Version 2.1.
- `See Result(s)` for interpretation, writes the logical forms in the *Test Results* pane; a supplementary window is opened to allow an easier inspection of the results (not implemented in Version 2.1).
- `Chart Browser` opens the chart browser window (see section 5.7).
- `Tree(s)` (available only for analysis in Version 2.1) opens up to ten parse tree windows (if any) produced by the last run of the processor.

5.7 Parser and Generator Chart Browser

The Chart Browser is a tool that helps users in debugging chart based structures. It aims to be the starting point of a set of debugging tools, such as inspector, semantic DAGs display, etc.

Depending on the current processor (parser or generator) a different chart browser is activated.

Anyway, both chart browsers are organized in the same way: the upper part contains general commands; the middle part displays the input (enriched with details) used to browse chart contents and the lower part allows the user to investigate edges.

5.7.1 Parser Chart Browser

The Geppetto Parser Chart Browser (Figures 13 and 14) are based on standard chart-parsing concepts: vertices and edges. For their meaning, vertices have been used to enrich the analysed sentence and as sensible points to interactively browse the parser output.

The "enriched input sentence" (i.e., the sentence with sensible vertices inside), is displayed in the middle part of the Parser Chart Browser. A click on a vertex makes it the current selection and changes the edge list(s) presented in the pane(s) below.

For each vertex the edges can be grouped in incoming and outgoing. This suggested two different layouts: *sequential* (Figure 13) is focussed on the whole interpretation and shows the list of vertices together with their edges; *in & out* (Figure 14) is focussed on browsing through vertices and shows the incoming and the outgoing edges in two adjacent panes. The user can select one of the two layouts from the `View` option menu displayed on the top.

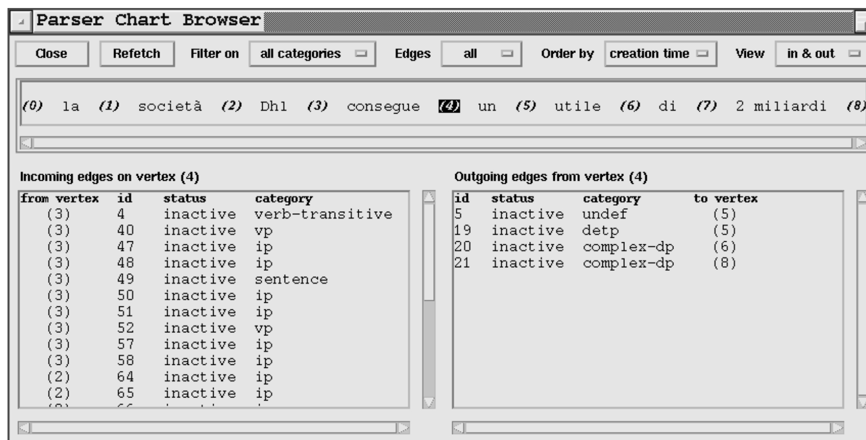

Figure 14: The Parser Chart Browser in in&out layout

In sequential layout, the list contains both vertex and edges lines. The displayed attributes of a vertex are the identifier (number and connected word) and view type (active, inactive, all). A popup menu is displayed when you click on a vertex line. It allows to

- change the edge show policy for the current vertex (there are three policies: showing all the edges, showing the inactive or the active ones only; if you are using the CYK parser, note that there are no active edges in the chart);
- inspect the data structure associated to the current chart vertex.

For each vertex only the outgoing edges are displayed in the sequential layout. Information displayed for each edge includes: id, status, category and destination vertex (to-vertex); the pop-up-menu presented when clicking on edge line allows to

- show the edge DAG;
- show the edge parse tree;
- inspect the data structure associated to the current edge.

The to-vertex attribute of an edge line, available in both layouts, is an active element: the associated pop-up-menu allows to

- scroll the chart browser sentence to see that vertex;
- go to the vertex scrolling the chart browser sentence and the vertex list;
- inspect the data structure associated to the chart vertex where the current edges enters.

In the in&out layout what changes is the way edges are displayed. For each vertex, incoming and outgoing edges are displayed in two different panes: the IN-vertex on the left, the OUT-vertex on the right. Clicking on a vertex (i.e. a number) in the sentence pane, the edges entering the vertex are shown in the IN-vertex pane, and the edges exiting the vertex are shown in the OUT-vertex pane. The possible gestures and their semantics inside IN and OUT panes are the same as those for the sequential layout described above.

Finally, there are four commands in the Parser Chart Browser window:

- **Close** hides the chart browser;
- **Refetch** re-reads the data containing the chart; NOTE: this function is automatically triggered when a new sentence is run!

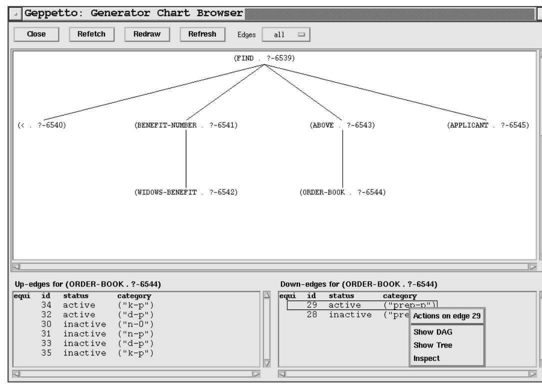

Figure 15: The Generator Chart Browser

- **Filter on** activates a filter on the categories of the edges shown (all categories, specific lexical categories);
- **Edges** sets the display policy for edges (all, active only, inactive only);
- **Order by** sets the order in which edges are displayed (creation time, length);
- **View** changes the Parser Chart Browser layout from sequential to in&out and vice versa.

5.7.2 The Generator Chart Browser

The Geppetto Generator Chart Browser (Figure ??) is composed by three parts: a command line on the top (buttons), a .... in the middle (active tree/DAG???), a pair of edges list below (up and down edges).

The commands allow to change the edge view (all, only active, only inactive), to redraw/refresh the tree presentation, refetch the new structure created after a new run.

The graph is used as starting point for the browsing through the edge structures created by the generator during the processing.

Clicking on nodes it is possible to select ..... : the corresponding edges are displayed in the two panes below.

Edges are, again, sensible: clicking on them a pop-up with possible commands are displayed. It is possible to: inspect the edge structure, display the tree ...., and display the .... DAG.

6 Installation

6.1 System Requirements

The basic algorithms have been tested and run on Allegro Common Lisp 4.2 (or later), Lucid Common Lisp 4.0.1 and Medley 2.0. The version of the Geppetto graphical environment described in this report runs only under Allegro Common Lisp 4.2 (or later) with CLIM 2.x and Grasper. There is an older version of the graphical environment running exclusively under the Medley environment.

As for hardware requirements, the Geppetto graphical environment runs on a Sparc5 with 64 Mega RAM.

In order to have a satisfactory appearance of the graphical interface of Geppetto, it is necessary to set the Xdefaults for the application. You need to ask your system administrator to put the file **Geppetto** in the `app-defaults` directory (at our site such directory is `/lib/X11/app-defaults`, but its exact place in the file system can depend on how X11 was installed). Such file is provided together with the Geppetto distribution and its name **MUST** be Geppetto; the file must be installed on each machine on which you plan to run Geppetto. You can override those defaults putting your own preferences in `~/.Xdefaults` (as for any other X application).

7 Acknowledgments

We would like to thank Dominique Estival for suggesting us the name Geppetto. The development environment was originally named GTP (Grammar Toolbox Prototype); Dominique Estival always said GPT instead of GTP, and GPT was easy to transform into GePpeTto; Geppetto is no longer an acronym (but it is fancier, isn't it?) but we liked it and so decided to adopt such name for the system.

We would also like to thank Livio Milanesio from 'the PACKARDT imaging' (`http://www.media-net.it/pckdt/`) for designing both the Geppetto and Pinocchio logos and for allowing their free use in our project.

References

[Ait-Kaci *et al.*, 1989] Hassan Ait-Kaci, Robert Boyer, Patrick Lincoln, and Roger Nasr. Efficient implementation of lattice operations. *ACM Transactions on Programming Languages and Systems*, 11(1):115–146, 1989.

[Carpenter, 1992] B. Carpenter. *The Logic of Typed Feature Structures*. Cambridge University Press, Cambridge, Massachusetts, 1992.

[Ciravegna *et al.*, July 1997] Fabio Ciravegna, Alberto Lavelli, Daniela Petrelli, and Fabio Pianesi. Participatory Design for linguistic engineering: the case of the Geppetto Development Environment. In *Proceedings of the ACL/EACL'97 Workshop on Computational Environments for Grammar Development and Linguistic Engineering*, Madrid, July 1997.

[Kogure, 1990] Kiyoshi Kogure. Strategic lazy incremental copy graph unification. In *Proceedings of the International Conference on Computational Linguistics*, pages 223–228, Helsinki, Finland, 1990.

[Pianesi, 1993] Fabio Pianesi. Head-driven bottom-up generation and Government and Binding: a unified perspective. In Helmut Horacek and Michael Zock, editors, *New Concepts in Natural Language Generation: Planning, Realization and Systems*, pages 187 – 214. Pinter Publishers, London, 1993.

[Shieber, 1986] S. M. Shieber. *An Introduction to Unification-Based Approaches to Grammars*. Center for the Study of Language and Information, Stanford, California, 1986.

[Shieber, 1992] S. M. Shieber. *Constraint-Based Grammar Formalisms*. MIT Press, Cambridge, Massachusetts, 1992.

[Uszkoreit, 1991] Hans Uszkoreit. Strategies for adding control information to declarative grammars. In *Proceedings of the Annual Meeting of the Association for Computational Linguistics*, pages 237–245, Berkeley, California, USA, 1991.

[Wroblewski, 1987] David A. Wroblewski. Nondestructive graph unification. In *Proceedings of AAAI-87*, pages 582–587, Seattle, WA, 1987.

A Known Bugs

Work on GEPPETTO is in progress and the Release 2.0 is still in beta testing: so some bugs are still present. We are working to fix all the problems.

If you find something that seems to be a bug, please browse the following lists before contacting us at `geppetto@irst.itc.it`.

The following lists contain all the bugs known at this moment: each list refers to a linguistic system component. The imperfections in GEPPETTO which we are aware of are usually mentioned in the description of the interested component in section 5 (they are emphasized using bold face character).

They are listed in order of importance (the most important first) and sometimes a strategy for coping with them is suggested.

A.1 Monitor

- At startup the GEPPETTO logo is not displayed and an error is reported: this means that there is a conflict with some other program (e.g., Netscape) for the colour resources. There are two possible solutions:

  - selecting a document with fewer colours in it (this solution sometimes works if the other program is Netscape);
  - exiting the other program (if the first solution does not work).

  Such kind of conflict for colour resources may also happen in the middle of an interaction with GEPPETTO when the user opens a window such as the ones used by the Inspector and the Debugger of Allegro Composer. The solutions are the same as those mentioned above.

- After creating a *New...* linguistic system, if you try to run a sentence the system asks you to load it even if it should be already loaded.

- While the system is compiling, the *Monitor* window should not be closed or changed in layout, otherwise an error occurs.

A.2 Browsers

- It might happen that if a browser is closed with an element still selected, when it is open again it does not show anything until scrolling, but the elements aren't accessible and the browser window close does not work any more: to avoid this bug pay attention in exiting the browsers with no element selected (when it occurs a quit and restart of GEPPETTO is needed); unfortunately we are not able to reproduce it in a deterministic way, so it is difficult to fix it.

- Typing a space while the focus is on a browser causes the window being automatically closed, because the Close button is the default action.

- Typing any character (different from the space) while the focus is on a browser, its list pane is scrolled to the very end; even if the characters are not recorded and a beep bells, they are temporarily shown in the lower part of the window.

A.3 Hierarchy

This is the most fragile component of GEPPETTO. Some strange behaviours are due to GRASPER, the tool used to draw the graphs.

A.4 Miscellanea

- For reasons (still) unclear, it may happen that some actions on DAGs, Parse Trees, Dependencies and Hierarchy (in general, actions that affect Grasper) do not work properly anymore: in this case exit GEPPETTO and start it again.

B Application Programmer Interface for Parsing

In this section the GEPPETTO's API for parsing is described. The current documentation is partial and requires the load of the patch PATCH.01.lisp.

The API is thought for being used together with the delivery system, but can also be used for modifying the standard GEPPETTO's behavior.

Currently the documented API concerns:

- running the parser on a sentence from the lisp interpreter (or from a program).
- modifying the GEPPETTO behaviour during parsing:
  - inserting a preprocessor for lexical analysis;
- requesting data from the environment:
  - asking for final logical forms;
  - inspecting data structures (i.e. chart and edges).

B.1 Parsing with GEPPETTO

When using the standard graphical interface if the "Start Run" button is pushed in the GEPPETTO monitor, the following function is run:

```
CHART::PARSE-STRING-SENTENCE
  input: a string containing a sentence
```

this is a function that calls the parser. When using the delivery system it is possible to run the parser using the same function from the lisp interpreter.

NOTE!! if the `CHART::PARSE-STRING-SENTENCE` is run from the lisp interpreter and the graphical interface is active some unexpected behavior can be obtained. For example the usual actions of activation/deactivation of push buttons are not performed. So the GEPPETTO monitor is not affected in any way by the function. But if the chart is inspected the internal data are changed.

There is a parameter that controls the maximum number of edges that the parser is allowed to produce; this parameter is `CHART::+edge-count-limit+` and is initially set to 700.

B.2 Inserting a Preprocessor

A preprocessing phase is defined as all the processing needed before parsing; it takes as input a string containing the sentence and returns a list of items that represent lexical edges to be inserted in the chart.

When an input is typed in the input pane of the monitor, a string is passed to the function `CHART::PARSE-STRING-SENTENCE` (see subsection B.1). This function calls a preprocessor on the input string and then loads the chart and runs the parser.

The default preprocessing for GEPPETTO is just the recognition of the words contained in the input string. More complex preprocessors can be provided by the user.

In GEPPETTO the method for preprocessing is

```
CHART::PREPROCESS
   input: PRE::+type-of-output+
          sentence-string
   output: a list of items representing ''words''
```

The `PRE::+type-of-output+` parameter controls the kind of output produced; the parameter can have two values : `:STRING` or `:WORD-ITEM`. As it will be seen other values can be added by the user.

According to the value of the parameter `PRE::+type-of-output+`, there are two possible types of output:

- strings: just a string of a word to be searched in the dictionary: this is what is generally required by applications without any requirements of input preprocessing. Words are searched in the dictionary as they are in the input string and the chart is loaded;
- instances of the class `CHART::WORD-ITEM` (see subsection B.2.2): this is what is required by applications that use sophisticated preprocessing phases where for example numbers, compound proper names, etc. are recognized.

The default preprocessing functionality provided by the default GEPPETTO preprocessor is just the recognition of words in the input string. Here follows the code:

```
(defparameter PRE::+type-of-output+ :STRING)

;; case for just returning strings
(defmethod PREPROCESS ((PRE::+type-of-output+ (eql :STRING))
                       sentence-string)
  (pre::split-string-into-words sentence-string))


;; case for returning word-items
(defmethod PREPROCESS ((PRE::+type-of-output+ (eql :WORD-ITEM))
                       sentence-string)
  (let ((count 0))
    (loop for word in (pre::split-string-into-words sentence-string)
        collect (make-instance 'word-item
                  :item-word (string-downcase word)
                  :item-start count
                  :item-end (incf count))
        )))
```

When more sophisticated preprocessing is needed it is possible to ADD a new method with a `PRE::+type-of-output+` different from `:STRING` or `WORD-ITEM`.

Once that the user has selected (or created) the right method for preprocessing, it is necessary to insert the results in the chart; this is done automatically by GEPPETTO when receiving the list of strings or word items through the method `ADD-ITEM` (see section B.5).

Some facilities are provided for inputting the result of a user specific preprocessor.

In the following we will see two main functionalities: words recognition in the sentence string and creation of word-items.

B.2.1 Recognizing Words in a Sentence

When the changes to the preprocessor are just a redefinition of the algorithm for word recognition, it is possible to REWRITE the following function:

```
PRE::SPLIT-STRING-INTO-WORDS
   input: sentence-string: the string of a sentence
   output: a list of strings
```

B.2.2 Creating Word Items

When a powerful preprocessor is used, the user may want to insert more complex items than strings. For example s/he may want to introduce an item that is a complex proper name of type PERSON. To do that it is necessary to build WORD-ITEM instances for providing GEPPETTO with that information.

There are two kinds of word items: those for standard words to be looked for in the dictionary and those for special tokens (i.e. tokens that are not likely to be found in the lexicon, such as proper names, etc.).

A word item is an instance of a class with the following definition:

```
(defclass word-item ()
    ((word :accessor item-word :initarg :item-word
           :documentation "")
     (start :accessor item-start :initarg :item-start
            :documentation "")
     (end :accessor item-end :initarg :item-end
          :documentation "")
     (lexcat :accessor item-lexcat :initarg :item-lexcat
          :initform NIL
          :documentation "only for standard words")
     (DAG :accessor item-DAG :initarg :item-DAG :initform NIL
            :documentation "only for special tokens")
     )
  (:documentation ""))
```

word is the word string.

start and *end* are the indices of the current word in the sentence (and in the chart).

lexcat is the lexical category of the word. It can be provided only for standard words; it is used for constraining the search in the vocabulary to a single lexical category. For example when a Part of Speech tagger is available, the user can want to look in the dictionary just for the word "run" as a "noun" (and not as a verb). The lexical category here must be a label (i.e. a string) taken from the type hierarchy. Subsumption relations are taken into account, so for example it is possible to use "constituent" for asking for any constituent lexical types. If the parameter is NIL, it is meant as "bottom", i.e. any lexical categories is accepted.

DAG: it is to be provided only for those items that are not to be found in the dictionary; for example for composed proper names recognized by a preprocessor. The way for building a DAG is explained below.

The class `WORD-ITEM` is general enough for many purposes; if the a more sophisticated treatment is required, the user can define a user class *as a subclass of WORD-ITEM* where additional information is provided. In that case it is also necessary to define a new `ADD-ITEM` function for allowing the additional information to be inserted in the chart (see subsection B.5). Anyway the redefinition of the `ADD-ITEM` method requires extensive knowledge in the way GEPPETTO implements data structures; we strongly recommend to avoid the redefinition of `ADD-ITEM` when possible.

B.2.3 Creating a Word Item Instance

For words that are to be searched as usual in the lexicon it is just necessary defining the following information in their `WORD-ITEM` instance.

```
CHART::CREATE-A-NORMAL-WORD-ITEM
  input: word
         &key start
              end
  output: a list of word items
```

This function returns a list of `WORD-ITEM` instances for the current entry.

word a string as the edge will be found in the final parse tree.

start and *end* are indices in the chart.

item-lexcat cat is necessary only when the word has been already disambiguated, so only the entries with a specified lexical category are to be retrieved.

In case the item is not to be searched in the dictionary (e.g. because it is a special token returned by the preprocessor such as a compound proper name), for creating the corresponding word item(s), it is necessary to call the function:

```
CHART::CREATE-A-SPECIAL-WORD-ITEM
  input: word
         prototypical-word
```

```
          &key start
               end
               logform-list
               mainvar
    output: a list of word items
```

This function returns a list of `WORD-ITEM` instances for the current entry.

The input parameters are the following:

word a string as the edge will be found in the final parse tree.

prototypical-word a Prototypical Lexical Entry. It is discussed below.

start and *end* are indices in the chart.

logform-list is a list of logical form to be inserted in the word-item instances; the list is provided for allowing multiple items to be generated from a single prototypical word. For example the user could want to say that a proper name indicates either a person or the name itself[4]. The function outputs a word-item instance for each logical form provided as input.

NOTE!

- the logical form must include a unary predicate including the mainvar! For example the kind of predicate that must be present in the logical form is `''(SL::NUMBER-CONCEPT X)''` if X is the mainvar.
- The only variable GEPPETTO is able to see is the mainvar: in the final DAG the mainvar will not be present, as it will be substituted by GEPPETTO with a unique variable in order to avoid the use of the same variable for more than one item. BUT this is not the same with other variables in the logical form. If there are other variables in the logical form the check for uniqueness of variables is left to the users. For creating a unique name see subsection B.3.

mainvar it indicates which is the mainvar in the logical form.

VERSION 2.0B LIMITATION!! Currently it is possible to create special words items only for objects without pending constraints, i.e. for lexical categories such as nouns, numbers, proper names, but not for verbs or prepositions.

B.2.4 Prototypical Lexical Entry

The key issue for building a special word item is a prototypical lexical entry; this is an *dummy entry in the lexicon* that is used only for building DAGs at run-time [5].

For example an entry for numbers can be the following:

```
(TFS-LEX-ITEM :ROOT "!number!"
      :TYPE "cardinal"
      :CONSTRAINTS ((@COMMON-SEM |$?pred| (SL::NUMBER-CONCEPT))
                    & (@LOGICAL-FORM))
      :COMMENT "Dummy entry for numbers")
```

The prototypical entry establishes the main features of the final term: for example it states that all the numbers have an associated lexical category that is called "cardinal". The only feature that is undetermined is the LOGICAL FORM. This is because numbers differ just for their value. It is then necessary to insert the value in the logical form. For example for creating the logical form `''((SL::NUMBER-CONCEPT 123))''`. The logical form of the prototype as defined in the previous example is instead `''(SL::NUMBER-CONCEPT VAR-1)''`.

As mentioned in the previous section for creating a complete DAG from a prototypical lexical entry, it is necessary to call the function: `CHART::CREATE-A-SPECIAL-WORD-ITEM` that receives as input the logical form(s) to be inserted to replace the underspecified.

[4]For example in the sentence "Mark is a child" the term "Mark" refers to an individual. In the sentence "Mark is the name of that child" the term "Mark" refers to a name.

[5]The only distinction between a dummy entry and a normal entry is the associated word. The user is warned to use always names that are strings impossible to be found in a sentence; a good idea is to use exclamation marks or semicolons around their names; for example the dummy entry for number could be "!number!" or ":number".

B.2.5 Using a Preprocessor: an Example

As conclusion of the section on preprocessing we show an example of the use of a sophisticated preprocessor. Let's suppose to have the sentence: "XYX Ltd. showed a profit of 123 billion Lire in 1993"

It is necessary to re-define the method `CHART::PREPROCESS`; for example as follows:

```
(defparameter PRE::+type-of-output+ :FROM-MY-OWN-PREPROC)

(defmethod PREPROCESS
      ((PRE::+type-of-output+ (eql :FROM-MY-OWN-PREPROC))
      sentence-string)
  (my-preprocessor
      (MY-split-string-into-words sentence-string)))
```

If we suppose that the preprocessor's internal results are as follows:

```
``XYX''           ``proper''   NOTKNOWN         0 1
``XYX Ltd.''      ``proper''   COMPANY          0 2
``Ltd.''          ``unknown''                   1 2
``showed''        ``mainv''                     2 3
``profit''        ``common''                    3 4
``of''            ``prep''                      4 5
``123''           ``cardinal''   123            5 6
``billion''       ``common''                    6 7
``123 billion''   ``cardinal'' 123,000,000,000  7 8
```

For creating the correct output the function `CHART::PREPROCESS` should return:

```
(APPEND
    (create-a-special-word-item "XYX" "!proper!"
              :start 0 :end 1
              :logform-list '(((NOTKNOWN X)))
              :mainvar 'X)
    (create-a-normal-word-item "Ltd." :start 1 :end 2)
    (create-a-special-word-item "XYX Ltd." "!proper!"
              :start 0 :end 2
              :logform-list '(((COMPANY X)))
              :mainvar 'X)
    (create-a-normal-word-item "showed"  :start 2 :end 3)
    (create-a-normal-word-item "a"  :start 3 :end 4)
    (create-a-normal-word-item "gross"  :start 4 :end 5)
    (create-a-normal-word-item "profit"  :start 5 :end 6)
    (create-a-normal-word-item "of"  :start 5 :end 6)
    (let ((num-value-sym 123)
          (var-rel (TFS::run-time-lisp-symbol)))
      (create-a-special-word-item "123" "!number!"
              :start 6 :end 7
              :logform-list `(((USER::QUANTITA X)
                                (USER::QUANTITA-VALUE ,var-rel )
                                (USER::+DOMAIN+  ,var-rel X)
                                (USER::+RANGE+   ,var-rel ,num-value-sym)
                                (USER::VALORE ,num-value-sym)))
              :mainvar 'X)
  )
   (create-a-normal-word-item "billion"  :start 7 :end 8)
```

```
  (let ((num-value-sym 123000000000)
        (var-rel (TFS::run-time-lisp-symbol)))
    (create-a-special-word-item "123000000000" "!number!"
            :start 6 :end 8
            :logform-list `((((USER::QUANTITA X)
                              (USER::QUANTITA-VALUE ,var-rel )
                              (USER::+DOMAIN+  ,var-rel X)
                              (USER::+RANGE+   ,var-rel ,num-value-sym)
                              (USER::VALORE ,num-value-sym)))
            :mainvar 'X)
))
```

B.3 Creating a New Variable

For creating a new variable it is possible to call the function:

```
TFS::RUN-TIME-LISP-SYMBOL
```

It returns a new variable id that is guaranteed to be unique.

B.4 Accessing the Final Logical Form

Once Geppetto is run on a sentence it is possible to retrieve the final logical form of the edges covering the whole sentence by calling:

```
CHART::GET-LOGICAL-FORMS
```

It returns a list of logical forms.

B.5 Inserting Edges in the Chart at Parse Time

During parsing a new edge can be added to the chart in any moment by calling the function:

```
CHART::ADD-ITEM
   input: a word item
          left-vertex
          right-vertex
   side effect: adds a lexical edge in the chart.
```

word item is either a word or an instance of the WORD-ITEM class (see subsectionword-item).

left-vertex and *right-vertex* are the starting and ending vertices of the lexical edge to be inserted in the chart. If just the indices of the vertex are available the vertices are found through the instruction

```
(AREF *CHART* index)
```

For adding the results of a preprocessor it is better not using `ADD-ITEM`, but the function `PREPROCESSING` as mentioned in subsection B.2.

B.6 Data Structures

Geppetto uses some internal structures for representing data such as the chart. The present subsection is provided **only for allowing the user to undestand the internal mechanisms**. Note that the data structures **can be changed across versions, so the user MUST NOT implement functions accessing the internal data structures**. The Application Programming Interface is provided for that purpose.

B.6.1 The CHART

Currently the CHART is a vector (`CHART::*CHART*`) whose elements are vertices; a vertex is an element with the following structure:

- index: a number from 0 to n (where n is the length of the sentence)
- word: the word form associated to the vertex;
- max-right-ext: an integer indicating the furthest vertex reachable using an edge exiting the current vertex (see max-edges);
- max-edges: longest edges exiting the current vertex;
- left-e-sels: a structure implementing the subsumption relation among edges entering the vertex;
- right-e-sels: a structure implementing the subsumption relation among edges entering the vertex;
- active-e-sels: a list of the edge-selections for which a task is currently present in the agenda

Moreover in the controlled version of the parser the following additional information are provided:

- lex-cats: lexical categories for the form in the field "word";
- sem-cats: semantic categories for the form in the field "word";

The chart has two special vertex types: the leftmost vertices and the rightmost: they are the vertices starting and closing each sentence; they have an additional field, namely `COMPLETE-PARSES`; there all the edges spanning all the sentence are listed.

B.6.2 Edges

Currently each edge is an instance of the class `CHART::EDGE` with the following slots:

- `ID`: an integer identifying the edge;
- `CAT`: a symbol indicating the grammatical category of the edge;
- `FROM-VERTEX`: the index of the vertex in the chart the edge is exiting;
- `TO-VERTEX`: the index of the vertex in the chart the edge is entering;
- `DOTTED-RULE`: the dotted rule involved by the edge: an instance of the class `dotted rule`;
- `CHILDREN`: a list of edges whose composition generated the edge;
- `DAG`: the DAG of the edge;
- `EQUI-EDGES`: edges equivalent to the current edge; used only in context free parsing where it is possible to consider equivalent edges spanning the same input and with the same category;
- `CURRENT-UNIFICATION`: indicates if the unification producing the DAG of the edge was asymmetric (in which case the value of the slot is :ASIMM) or not (in which case the value of the slot is NIL)
- `PREFERENCE`: an instance of the `CHART::PREFERENCE` class: only for the controlled parser.

Edges are of two types: *active edges* or *inactive edges*. Active edges represent partially instantiated rules, whereas inactive ones fully recognised rules; `CHART::ACTIVE-EDGE` and `CHART::INACTIVE-EDGE` are subclasses of the class `CHART::EDGE`.

C Linguistic Resources and Processors

C.1 A Linguistic System for Italian

A sample linguistic system (type hierarchy, grammar, lexicon, and macros) for Italian is provided together with the GEPPETTO environment.[6] This part of the Appendix describes the data contained in such linguistic system trying to give some hints on how to modify and make use of it in order to build your own linguistic system. We are sorry but almost all the comments contained in the data files are written in Italian.

The *type hierarchy* is the basis of the whole system, where all the basic grammatical concepts are defined and related through multiple inheritance. Both the *grammar* and the *lexicon* are seen as set of instances of two particular types of the hierarchy, respectively `psr` (phrase structure rule) and `constituent`, and their subtypes. The *macros* are kind of shortcuts, a way for hiding low-level details and making the linguistic system more modular and easy to modify and maintain.

Some general remarks on the linguistic system: it is written adopting a HPSG-like style, but without being particularly strict or faithful in such attitude. Currently the type hierarchy supports only grammar rules that are at most binary, but that can be easily modified adding further types to the hierarchy.

C.1.1 Type Hierarchy

First of all, some notational conventions. The names of types must be written in lowercase, while the names of the features must be in uppercase. Each type has a set of features appropriate for it (directly defined or inherited by its ancestors), together with a restriction on the value of every feature. Types can have TFS descriptions, too.

Here follows a brief description of the main types defined in the hierarchy (this list will give you only a partial view of the hierarchy; in order to have a complete understanding of the structure of the hierarchy, run the grapher of the hierarchy and browse the types); each item is organized in the following way:

- the name of the type;
- a documentation string;
- the features appropriate for such type;
- its subtypes;
- some comments.

First of all, a couple of types of general utility:

- **list**: it is used for subcategorization information
- **d-list**: *difference lists* are used in dealing with long-distance dependencies

Here are the types specifically related to grammar rules:

- **psr**: *phrase structure rule*

  Appropriate features:

  `[C0: constituent]`

  - **psr1**: *unary phrase structure rule*

    Appropriate features:

    `[C0: constituent]` (inherited)

    `[C1: constituent]`

[6] Just before delivering the GEPPETTO environment, we have added a few English words to the lexicon so that you can run some English sentences (but with the Italian grammar).

– **psr2**: *binary phrase structure rule*

  Appropriate features:

  `[C0: constituent]` (inherited)

  `[C1: constituent]`

  `[C2: constituent]`

  * **psr2-left-head**: binary rule whose (syntactic and semantic) head is its leftmost right-hand side element
  * **psr2-right-head**: binary rule whose (syntactic and semantic) head is its rightmost right-hand side element

Now, the types related to grammatical constituents; please, note that the part concerning long-distance dependencies (i.e. the features `BINDING` e `PSEUDO-SLASH`) is a little bit messy.

- **constituent**: grammatical constituent

  Appropriate features:

  `[SYN: syn]`

  `[SEM: sem]`

  – **adjectival**
    * **adjective**
    * **ap**

  – **adverbial**
    * **adverb**
    * **advp**

  – **nominal**

  – **prepositional**

  – **prepositional-mark**

  – **sentential**

  – **verbal**

  – **xp**

  – **xp-predicative**

  – **det**

  – **punctuation-symbol**

  The types listed above are only some of the subtypes of **constituent**; in order to have a complete understanding of the structure of this part of the hierarchy, browse the grapher of the hierarchy.

- **syn**: syntactic information

  Appropriate features:

  `[LOCAL: local]`

  `[BINDING: d-list]`

  `[STOPPEDNESS: stoppedness]`

  `[UNBALANCED-PUNCT-SYMBOL: yes-or-no]`

  The feature `LOCAL` contains all the local syntactic information, whereas `BINDING` is used in order to deal with long-distance dependencies. The two features `STOPPEDNESS` and `UNBALANCED-PUNCT-SYMBOL` are related to an attempt of taking into account punctuation during parsing.

- **local**: local syntactic information

  Appropriate features:

  `[HEAD: head]`

  `[SUBCAT: subcat]`

  `[PSEUDO-SLASH: constituent]`

  The feature `HEAD` contains head information of the constituent. The feature `SUBCAT` contains information on subcategorization frames.

- **head**: head information;

  Appropriate features of `head` or of some of its subtypes:

  `[AGREEMENT: agreement]`

  `[CASE: case]`

  `[DEFINITENESS: definiteness]`

  `[MOOD: mood]`

  `[TENSE: tense]`

- **agreement**

  Appropriate features:

  `[GENDER: gender]`

  `[NUMBER: number]`

  `[PERSON: person]`

- **case**
  - **nom**
  - **acc**
  - **dat**

- **definiteness**
  - **definite+**
  - **definite-**
  - **generic**

- **gender**
  - **fem**
  - **mas**

- **number**
  - **sing**
  - **plu**

- **person**
  - **first**
  - **second**
  - **third**

- **mood**

- **finite**
    * **conditional**
    * **indicative**
    * **subjunctive**
- **non-finite**
    * **gerund**
    * **imperative**
    * **infinitive**
    * **participle**

- **tense**
  - **past**
  - **present**
  - **future**

- **sem**: semantic information; the current structure of information under the feature `SEM` is a little bit awkward, because there are some redundancies due to the coexistence of two different approaches to the representation of semantic knowledge: one based on a structured semantic representation where semantic information (basically a predicate-argument structure) is spread all over the SEM (sub)DAG and another where Quasi Logical Forms (QLFs) are stored as lists under a single TFS node. Currently the structural semantic form is used only in lexical items for automatically building the QLF associated to such item. In the further processing only the QLF is actively in order to interact with the external source of knowledge containing semantic information.

  A general advice regarding the semantic features (I mean, all the features that are below the `SEM` feature of the type `constituent`) is that up to now you should not try to modify them. We are currently working in order to make this part modifiable and customizable by the resource developer, but now it is hardwired; so, please, be careful and do not try to modify this part of the hierarchy!

C.1.2 Grammar

As already said, the grammar is written adopting a HPSG-like style, but without being particularly strict or faithful in such attitude, and currently the grammar rules are at most binary.

The coverage of the grammar is not very wide.

The convention in choosing the name of the grammar rules has been that of using names suggesting the context-free backbone of the rule (plus some other relevant information).

Each grammar rule is characterized by six elements: name, documentation, type (a subtype of `psr`), TFS description, score, and comments. The rationale of having both documentation and comments is that the former gives information on the general behaviour of the entry (relevant for a quick understanding), whereas the latter makes remarks on some lower-level details. In spite of this plentifulness of tools for documenting and commenting the grammar rules, there are few comments in the sample grammar.

- name: it has only a mnemonic meaning and is not used for the indexing of the rules;
- documentation: it describes the general behaviour of the rule;
- type: it must be a subtype of `psr`;
- constraints: a TFS description which states some constraints on the feature structures associated to the rule;
- score: currently it is a hand-assigned indication of the probability of the rule; in the future it will be extracted automatically analyzing some corpora;

- comments: remarks on some hacks present in the TFS description of the rule.

Here follows an example of grammar rule; i.e., the rule that, after having built the VP (or the AUX1P or the AUX2P, in case auxiliaries are present), recognizes the subject:

```
(TFS-RULE :NAME "ip -> xp, {aux1p, aux2p. vp}"
          :DOCUMENTATION ""
          :TYPE "psr-2-right-head"
          :CONSTRAINTS
           ((@CF-BACKBONE-2 $?\c0 |ip|
                            $?\c1 (|xp| / |wh-constituent|)
                            $?\c2 ({ |aux1p| |aux2p| |vp| }))
            &
            (@WH- $?C C1)
            &
            (@MOOD |$?mood| |finite| $?C C2)
            &
            (@CLAUSE-TYPE |$?clause-type|
                ({ |declarative| |polar-interrogative| |content-interrogative| })
                $?C C2)
            &
            (@POP-EXT-ARG $?HEAD C2 $?ARG C1)
            &
            (@SATURATED $?C C2)
            &
            (@PERCOLATE-SLASH $?H C2)
            &
            (@AGREE $?C1 C1 $?C2 C2)
            &
            (@PERCOLATE-SEM $?H C2)
            &
            (< C0 SYN UNBALANCED-PUNCT-SYMBOL > = < C2 SYN UNBALANCED-PUNCT-SYMBOL >))
          :COMMENT ""
          :SCORE 5)
```

C.1.3 Lexicon

Currently the lexicon is composed by word forms because a morphological analyzer has not yet been integrated in the environment.

Lexical entries are characterized by five elements: name, documentation, type (a subtype of `constituent`), TFS description,, and comments. The rationale of having both documentation and comments is that the former gives information on the general behaviour of the entry (relevant for a quick understanding), whereas the latter makes remarks on some lower-level details. In spite of this plentifulness of tools for documenting and commenting the lexical entries, there are few comments in the sample lexicon.

- name: it is the key used by the parser in order to retrieve entries from the lexicon;
- documentation: it describes the general behaviour of the lexical entry;
- type: it must be a subtype of `constituent`;
- constraints: a TFS description which states some constraints on the feature structures associated to the lexical entry;
- comments: remarks on some hacks present in the TFS description of the lexical entry.

Here follows an example of very simple lexical item, for the Italian word "articolo" (i.e. article, paper):

```
(TFS-LEX-ITEM :NAME "articolo"
              :DOCUMENTATION ""
              :TYPE "common"
              :CONSTRAINTS ((@COMMON-MAS-SING)
                            &
                            (@COMMON-SEM |$?pred| SEMLEX::ARTICOLO)
                            &
                            (@LOGICAL-FORM))
              :COMMENT "")
```

And now a more complex one, that for the verb "proporra'" (i.e., "it will propose"); most of the complexity of such an entry is hidden in the definition of the macro `@VERB3-CP-DI`, i.e. the macro that states the syntactic and semantic behaviour of the verbs with 3 arguments: the subject, the indirect object and a CP of non-finite mood whose COMPL is the preposition "di".

```
(TFS-LEX-ITEM :NAME "proporra'"
              :DOCUMENTATION ""
              :TYPE "verb-transitive"
              :CONSTRAINTS ((@NUMBER |$?number| |sing|)
                            &
                            (@PERSON |$?person| |third|)
                            &
                            (@TENSE $?C < > |$?tense| |future|)
                            &
                            (@MOOD $?C < > |$?mood| |indicative|)
                            &
                            (@VERB3-CP-DI |$?pred| SEMLEX::PROPORRE
                             |$?arg1| SEMLEX::AGENT |$?arg2| SEMLEX::PATIENT
                             |$?arg3| SEMLEX::ACTION)
                            &
                            (@LOGICAL-FORM))
       :COMMENT "")
```

C.1.4 Macros

As already said, macros are meant to be kind of shortcuts in order to make the writing of the linguistic system easier and more modular.

There are a number of more or less arbitrary conventions in the syntax of macros that are justified (?) by the need of parsing them.

Each macro is characterized by five elements: name, parameters, documentation, body, comments. The rationale of having both documentation and comments is that the former gives information on the general behaviour of the macro (relevant for a quick understanding), whereas the latter makes remarks on some lower-level details. In spite of this plentifulness of tools for documenting and commenting the macros, there are few comments in the sample macro set.

- name: it must start with the character `@` (for instance, `@AGREE`);
- parameters: the list of the parameters of the macro; each parameter name must start with the character `?` and is in lowercase if it represents a type and in uppercase if it represents a feature path; when calling a macro, the name of the parameters must be further prefixed with the character `$` (more or less as the keyword parameters of Common Lisp lambda list). For example, the macro `@AGREE` has got two parameters, `?C1` and `?C2`; when using the macro in some TFS description associated with a rule, I would write `(@AGREE $?C1 C1 $?C2 C2)`;
- documentation: it describes the general behaviour of the macro;

- body: the body of the macro; it is similar the any other TFS description associated with rules or lexical entries, except for the fact that all the parameters of the macro must be used in the body (the parallelism with the macros of Common Lisp is no longer valid; in Common Lisp you are able not to use some parameters, possibly being warned by the compiler, while here you cannot ignore the parameters);
- comments: remarks on some hacks present in the TFS description of the macro.

And now an example of macro, `@AGREE`:

```
(TFS-MACRO :NAME "@AGREE"
           :PARAMETERS ((?C2 . FEATURE) (?C1 . FEATURE))
           :DOCUMENTATION "checks that the two constituents ?C1 e ?C2
                           share the feature AGREEMENT"
           :BODY (< ?C1 SYN LOCAL HEAD AGREEMENT > = < ?C2 SYN LOCAL HEAD AGREEMENT >)
           :COMMENT "")
```

And now the definition of the macro `@VERB3-CP-DI`, used above in the definition of a lexical entry:

```
(TFS-MACRO :NAME "@VERB3-CP-DI"
           :PARAMETERS ((|?pred| . TYPE)
                        (|?arg1| . TYPE)
                        (|?arg2| . TYPE)
                        (|?arg3| . TYPE))
           :DOCUMENTATION "it is useful for control verbs such as 'proporre'
                           (proporre a qualcuno di fare qualcosa) where the person
                           who is proposed something is also the subject of the
                           subordinate clause"
           :BODY
            ((@STOPPEDNESS-LEX)
            &
            (@EXTERNAL-ARG |$?type| |dp| |$?case| |nom|)
            &
            (@FIRST-INTERNAL-ARG-PP |$?type| |pp-mark| |$?p-form| \a)
            &
            (@SECOND-INTERNAL-ARG |$?type| |cp| |$?form| |di|)
            &
            (< SYN LOCAL SUBCAT INTERNAL-ARGS REST REST > _ (|e-list|))
            &
            (@KB-CONSISTENCY $?LF1 < SEM HEAD LF >
                             $?LF2 < SEM ARG3 FILLER
                                         ARG1 FILLER HEAD LF >
                             $?VAR1 < SEM ARG2 FILLER
                                          HEAD MAIN-VAR >
                             $?VAR2 < SEM ARG3 FILLER
                                          ARG1 FILLER HEAD MAIN-VAR >
                             $?ARG-LABEL < SEM ARG2 ROLE >
                             $?PLACE < SEM HEAD LF >)
            &
            (@VERB3-SYN-SEM-MAP)
            &
            (@VERB3-SEM |$?pred| |?pred| |$?arg1| |?arg1|
                        |$?arg2| |?arg2| |$?arg3| |?arg3|))
           :COMMENT "")
```

And now a few comments on some of the macros, grouped according to the part of the linguistic system involved.

As far as the grammar is concerned:

- `@CF-BACKBONE-1` and `@CF-BACKBONE-2` are used for specifying the context-free backbone of the rules (respectively for unary and binary ones);
- `@PERCOLATE-EXT-ARG`, `@PERCOLATE-INT-ARGS`, `@PERCOLATE-SEM` and a number of other macros all starting with the prefix `@PERCOLATE-` are meant to make some information percolate from one of the right-hand side elements to the left-hand side element of the rule;
- `@AGREE` imposes the constraint that two constituents have a common agreement.

As for the lexicon (please, remember that currently the lexicon is composed by word forms):

- `@CASE`, `@GENDER`, `@NUMBER`, `@PERSON`, `@MOOD`, `@TENSE` are used for specifying the value of the corresponding feature in the lexical entries;
- there are macros specifying the general syntactic and semantic behaviour of the most frequently used words; for instance, transitive verbs (`@VERB2`), and verbs with more complex subcategorization frames.

C.2 Parsers

As already mentioned, two parsers are currently provided within the GEPPETTO environment, i.e. a Head-driven one and a CYK-like one. The main difference from the point of view of the grammar developer is that, if you want to use the CYK parser, obviously your grammar must contain rules that are at most binary.

The simplest way of checking the behaviour of the linguistic system is that of running the parser, clicking the Run button in the GEPPETTO monitor window, and then inspecting the results (parse trees, DAGs) always through the functionalities provided by the graphical interface.

If you like to process in some way the results produced by the parser, you can access them using the following functions:[7]

- `GET-COMPLETE-PARSES`: takes no arguments and returns the list of all the edges representing complete analyses of the sentence (i.e. all the edges of category `sentence` spanning all the input sentence);
- `GET-EDGE-TYPE`: takes an edge and returns the type of such edge; if such type is not disjunctive it returns simply a single element (a string), otherwise returns a list of strings (i.e. a disjunction);
- `DAG`: takes an edge and returns the DAG associated to such edge (as a matter of fact, the value returned is a dotted pair consisting of the DAG and the possible external constraints associated with such DAG);
- `CHILDREN`: takes an edge and returns the list of children of such an edge; when the parameter is a preterminal edge, it returns the word associated with such edge (i.e. a string);

In order to make the processing more efficient, during the compilation of the grammar the context-free backbone of every rule is computed. Always during compilation, the syntactic head associated with each rule is computed in order to index the rules for the parsing.

[7] Unless explicitly stated otherwise, all the functions are in the `CHART` package, and not exported by such package.