# An X-Windows Toolkit for Knowledge Acquisition and Representation Based on Conceptual Structures*

Michel Wermelinger** and José Gabriel Lopes

Centro de Inteligência Artificial/UNINOVA
Quinta da Torre, 2825 Monte da Caparica, PORTUGAL
{mw|gpl}@fct.unl.pt

**Abstract.** This paper describes GET (Graph Editor and Tools), a tool based on Sowa's conceptual structures, which can be used for generic knowledge acquisition and representation. The system enabled the acquisition of semantic information (restrictions) for a lexicon used by a semantic interpreter for Portuguese sentences featuring some deduction capabilities. GET also enables the graphical representation of conceptual relations by incorporating an X-Windows based editor.
**Keywords**: conceptual structures, knowledge representation, graphical interfaces, natural language processing.

## 1 Introduction

Conceptual structures [9] are an ambitious attempt to represent knowledge in a natural and expressive way. An implementation of the necessary machinery would enable us to test their practical suitability for semantic representation of natural language sentences, for conceptual modeling of relational databases, etc. So we decided to program a prototype in X-Prolog [1], a result of the ESPRIT project "Advanced Logic Programming Environments". One of the main reasons for this choice was the possibility to access the X-Windows functionalities in order to display conceptual structures making use of their easy to read graphical notation.

The resulting implementation, called Graph Editor and Tools (GET), is currently a *generic* tool for knowledge acquisition and representation based on conceptual structures and consists of two distinct parts: the Conceptual Graph Tools (CGT), a portable collection of Prolog predicates implementing the most important operations on conceptual graphs, and the Conceptual Graph Editor (CGE) working under X-Windows and using the primitives provided by CGT.

* This work was partially supported by JNICT, under contracts PMC-T/P/TIT/167/90 and PMCT/MIC/87439, by INIC, under project CALIPSO, by FCT/UNL, and by Gabinete de Filosofia do Conhecimento.

** Owns a scholarship (PMCT/BIC/114/90) from Junta Nacional de Investigação Científica e Tecnológica.

This paper describes CGT, CGE, and a semantic interpreter for Portuguese sentences, focussing on CGE. Finally, possible enhancements as well as some insights gained with this work regarding the utilization of conceptual graphs for Natural Language Processing are given. For a more detailed account see [12] and [13].

## 2 The Conceptual Graph Tools

The Conceptual Graph Tools (CGT) are a portable collection of Prolog predicates implementing the most important operations on conceptual graphs, a simple mark-&-sweep memory management system, and a linear notation parser and generator using Definite Clause Grammars [6]. CGT also provides facilities to manipulate graph databases made up of:

- a type hierarchy
- a set of graphs, where each may have some descriptive text associated to it
- for each concept type, a (possibly empty) set of schemata
- for each type, the associated canonical graph and/or definition

A sample database comes with the toolkit; it contains all relations defined in the Conceptual Catalog [9, Appendix B], and several basic concept types. CGT enables the user to easily create new types with their associated definitions, schemata, and canonical graphs in order to build several new databases on top of the given one.

The linear notation of conceptual graphs as parsed and generated by CGT is a bit different from the one used by Sowa. The formal definition in [12] extends the one given in [9, Appendix A.6], especially in what concerns the type and referent fields, including nested contexts. The minor differences are due to efficiency concerns and implementation restrictions (like using '\' for '$\lambda$' and '`V`' for '$\forall$' to use just ASCII). Major changes or restrictions were motivated by unclear aspects of the formalism, specially regarding coreference links. For example, should one permit any two concepts to corefer? How can inconsistencies be detected? Therefore, it was decided that coreferenced concepts must have compatible types, i.e. one is a subtype of the other. Furthermore, contexts may not corefer if their referents contain graphs because it would be difficult to check them for incompatibility.

Currently CGT has the following implementational restriction: once a concept or relation type is defined it isn't possible to change neither its definition nor its associated canonical graph. The reason is simple: if the canonical graph associated to a type $X$ changes, all graphs with a concept or relation of type $X$ must probably change, too. And if a type $Y$ is defined in terms of $X$, the graphs involving concepts or relations of type $Y$ would have to change too, and so on. This weakness can be repaired by adding a specific maintenance tool.

CGT features a SAFE[3] linear notation parser: it doesn't perform any error recovery, stopping with the first error found. It copes quite well with semantic

[3] Stop At First Error

errors (e.g. undefined referent variables, unknown type labels) but it still needs to be made more robust regarding syntax errors (e.g. a missing '`]`'). Furthermore, it forces the graphs entered by the user to be meaningful by checking them against the canonical graphs of the ocurring relation types.

As one should expect, some parts of Sowa's formalism have not yet been implemented in this first version of the Graph Tools. The most notable omissions are the first-order rules of inference and the $\phi$ operator, which translates graphs into first-order logic formulas. The latter could be modified to assert graphs as Prolog clauses in order to use Prolog's inference engine for deductions. Other things still need to be improved, specially referents and coreference links. Both will require theoretical work; the former, particularly, will need some reworking while the latter must be carefully analysed with respect to their side-effects on operations such as the canonical formation rules.

## 3 The Conceptual Graph Editor

CGE enables the user to create and manipulate conceptual structures in a graphical way, using the primitives provided by the Tools. It can be considered to be a kind of "syntax-oriented" editor, as most commands correspond to operations provided by the formalism, thereby enforcing the resulting graph to be canonical. The alternative would be to have a "visual" editor allowing to operate on single nodes and arcs, but its implementation would be more difficult because incomplete and ill-formed graphs would have to be taken into account.

The Conceptual Graph Editor was coded in X-Prolog, a superset of Prolog including the Widget[4] Description Language [1] which enables the programmer to access the X Windows functionalities in a declarative way. Therefore, the editor takes advantage of the underlying graphical interface, providing an easy way to edit graphs using windows, dialog boxes, icons, buttons, selections, and the combination of mouse and keyboard. Furthermore, the choice of the X-Windows standard increases portability and decreases the learning time for users already familiar with other graphical interfaces.

### 3.1 The Editor Window

A Conceptual Graph Editor is a window comprising five areas (see Fig. 1):

**header** This area consists of a single line of text displaying a description of the shown graph. The text may be a user defined string (as in the figure), the usual description (e.g. '`canonical graph for BUS(x) is`', '`relation AGNT(x, y) is`', etc.) or simply the word '`graph`'.

**graphical display** It is under the header and shows the edited graph(s).

**linear display** It shows the same graph as the graphical display but in linear notation. It is a normal text widget, enabling the user to edit its contents using normal Emacs commands [11] in order to create graphs which cannot be obtained with the menu commands.

[4] Window gadget—a graphical object in X-Windows terminology.

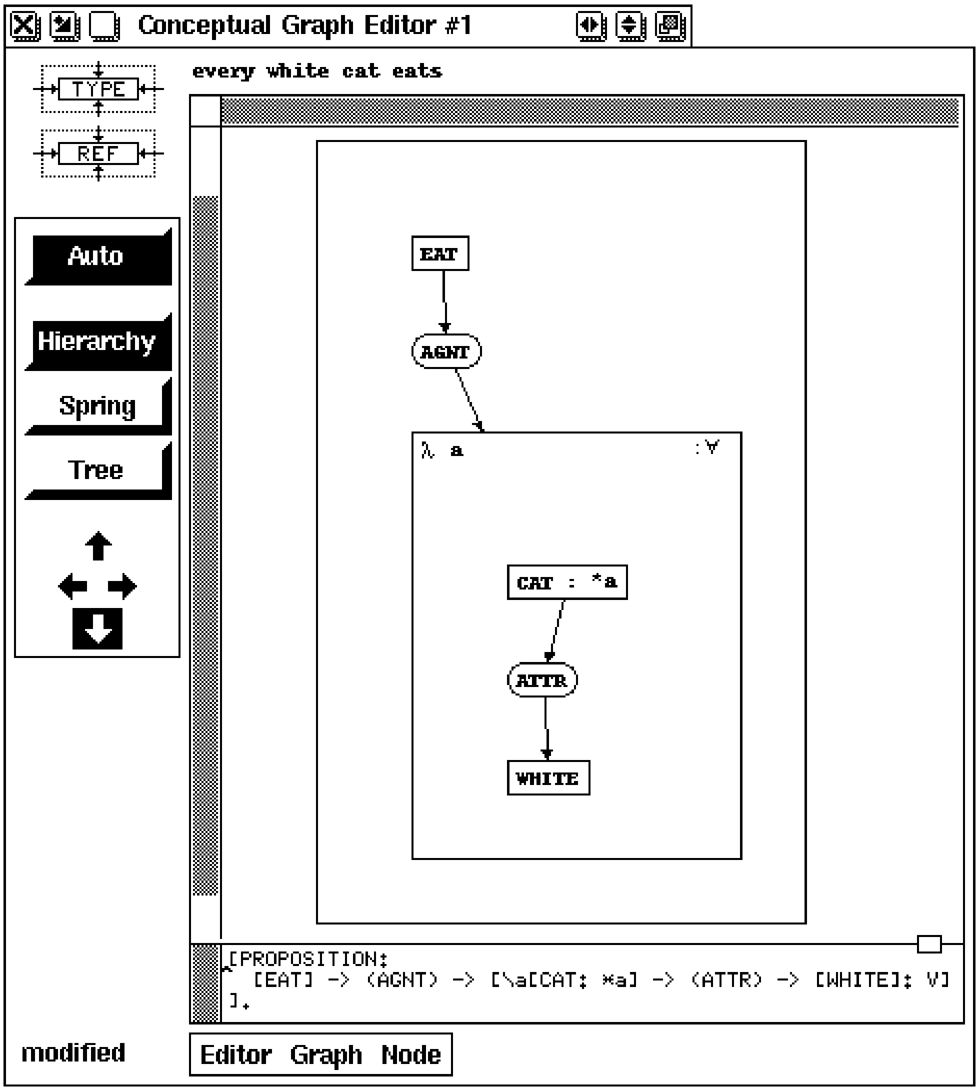


**Fig. 1.** A Conceptual Graph Editor

**menus** Under the two display areas, all possible commands to (visually) edit graphs and their nodes are provided. Most operations are directly supported by CGT.

**buttons** Located to the left of the graphical display, the two top buttons provide access to two commands without keyboard shortcuts (‘`Restrict Type`’ and ‘`Restrict Referent`’) while the other buttons provide an easy control over the way graphs are drawn in the graphical display.

The relative sizes of the graphical and linear displays may be changed by dragging the small rectangle between them with the mouse. There is also a '`modified`' label in the bottom left corner, appearing only when a graph has been added to the database but the latter hasn't been saved on disk.

Several editor windows may be open at the same time. To make better usage of the display area of the monitor, the editor windows may be iconified. In order to distinguish the various editors in an easy way, both the icons and the windows are numbered.

### 3.2 The Graphical Display

The main area within an editor window is occupied by the graphical display of graphs. In CGE, the visual appearance of graphs may be controlled by the user, either semi-automatically or manually for full control.

To make the display of conceptual graphs easier, a generic widget to handle the visualization of arbitrary graphs was used [8]. *All* graphs in the *same* context, and only them, are displayed with the *same* Graph Widget. This gives a lot of flexibility, as graphs in different contexts may be displayed in different ways. The relevant attributes of the Graph Widget for CGE users are:

**layout mode** It indicates if the display of the graph is to be done automatically (by the widget) or manually (by the user, dragging the nodes with the mouse to the desired position).

**layout function** This is the algorithm used in automatic mode to calculate the positions of the graph's nodes.

**layout style** It may be one of the four available styles (left-right, right-left, top-down, bottom-up) for automatic layout.

There are three pre-defined layout functions:

**Hierarchy** This function is mainly used for hierarchical graphs and it is the one that provides the best results for conceptual graphs.

**Tree** This function can only be used for a single graph that is in fact a tree (see Fig. 2. If used to display disconnected graphs, a mess will appear on the screen.

**Spring** This is the only iterative function, i.e. the visual appearance of the graph will depend on the original position of its nodes, whereas the other functions always display the same graph in the same way.

To the left of the graphical display there's a layout control box consisting of eight buttons. The top one controls the layout mode, the next three control the layout function and the bottom four arrows control the layout style. The buttons have a twofold purpose. By clicking on them with the mouse, the user may set the corresponding attributes in the selected context(s). On the other hand, the state of the buttons reflects the attributes of the selected context(s).

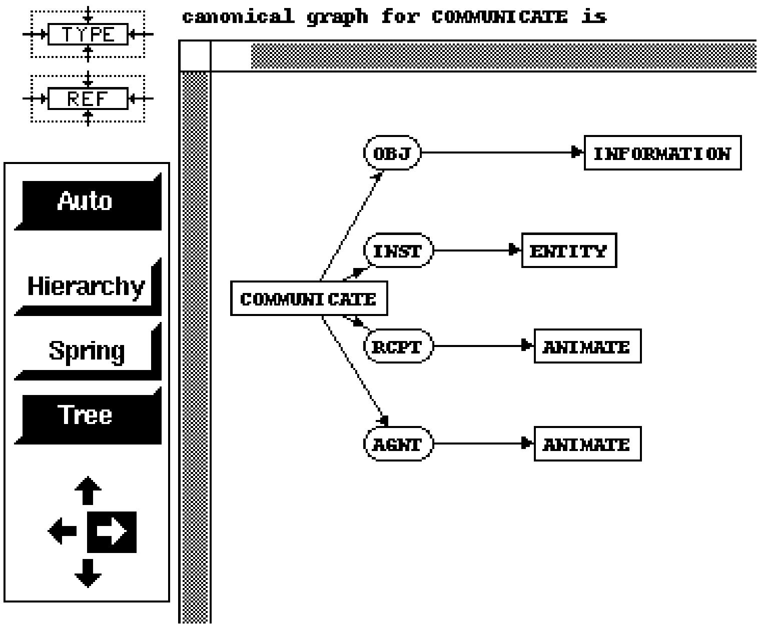


**Fig. 2.** Tree display

### 3.3 The Editor Commands

The commands available in CGE may be issued from the keyboard, selecting an entry of a menu, or clicking with the mouse on an icon. Often, there are at least two ways to invoke the same command. Most of the commands use dialog boxes (to interact with the user) and selections (to show the graphs or nodes on which to operate). There are several types of the former (acknowledge dialogs to display error messages, choice dialogs to present a set of options, etc.), taking into account the various needs for user input. Also, two kinds of selections are provided to enable some commands to distinguish their operands (e.g. the insertion operation needs to know the graph to insert and the context in which to insert it). All available commands, except those involving the way graphs are displayed, appear in the following three menus:

**Editor Menu** The Editor Menu contains commands that don't belong to the conceptual graph formalism, like loading and saving a graph, change the current graph database, deleting an arbitrary graph, and quitting the editor.

**Graph Menu** The commands in this menu operate on whole graphs. They include the canonical formation rules and the propositional rules of inference. Furthermore, there are commands to compute the depth of a graph and to check whether one graph is a generalization, a specialization, or a copy of another graph.

**Node Menu** This menu groups all commands that operate only on relations and concepts. They are divided into three groups: the restrict operation from the canonical formation rules, type expansions, and referent expansions and contractions.

There isn't an "undo" command, but most of the implemented ones have a counterpart, like drawing vs. erasing a double negation, iteration vs. deiteration, etc. To cancel the effect of *any* operation, the '`Clear Graph`' command is provided, but it should be used only in case of a mistake (e.g. the wrong graphs were joined) as it is not a canonical or propositional rule.

### 3.4 Future Work

The Graph Widget needs some recoding before being of practical use for the display of conceptual graphs: the algorithms must take the size of the nodes into account, make better usage of space, and maybe a new one must be developped for nested graphs. Because of these problems and other implementational details the visualization of coreference links wasn't implemented. Some other possible enhancements are:

- Make a type lattice editor/viewer which would provide an easy way to create new types or to select existing ones.
- Show coreference links and enable the user to edit them in a simple way (e.g. by pointing and dragging).
- Enable the user to choose for each context whether it should be displayed in normal or reduced size, thus improving the effective usage of the available display area.
- Enable the user to do some things (e.g. lambda abstraction in the type field) in a more graphical way, instead of having to write the corresponding linear notation for it.

## 4 A Semantic Interpreter

A small semantic interpreter for Portuguese sentences was built using conceptual graphs. The approach taken is similar to the one described in [10]: The lexicon associates a canonical graph to each possible meaning of a word and the interpreter proceeds in a bottom-up way when processing the syntactic tree. For each subtree it obtains a graph and its so-called "head". At the next level, the interpreter will try to join the graphs by matching directly the corresponding heads. As one can see, the only operations the interpreter needs from CGT are the canonical formation rules.

The sentences are parsed with a wide coverage Portuguese syntax description [4] using the XG formalism [7]. In its actual state, the semantic interpreter only covers a tiny subset of that description. On the other hand, it performs some deduction on the database constructed from the input sentences. The interpreter accepts three types of sentences (see the appendix for examples):

*Declarative sentences.* They denote assertions to be added to the database, stating a simple negative or positive *fact* (e.g. "The cat doesn't eat.") or a *rule* of the form "*A* if *B*", where *A* and *B* are simple facts (e.g. "The cat eats the mouse if it is hungry."). The former are represented by negative or positive contexts, respectively, and the latter uses the '`IMP`' relation [9, section 4.2].

It must be stressed that there is no anaphora resolution. As a consequence, sentences like "The cat eats the mouse if it doesn't run away." must be rewritten into "The cat eats the mouse if the mouse doesn't run away." and even in this case no coreference link between the two '`MOUSE`' concepts will be drawn. Therefore, whenever the individual is not specified the interpreter makes the simplifying assumption that the user is always referring to the same one. In the last section of this paper we provide some ideas to work around this problem.

Deductions upon graphs are made according to the following four rules, where $G/X$ is a fact represented by a proposition of polarity $X$ containing graph $G$, $A \Rightarrow B$ represents the rule "*B* if *A*", and $\leq$ denotes specialization:

$$
\begin{array}{llllll}
G1/pos & \text{and} & G2/pos \Rightarrow G3/X & \text{implies} & G3/X & \text{if } G1 \leq G2. \\
G1/neg & \text{and} & G2/neg \Rightarrow G3/X & \text{implies} & G3/X & \text{if } G2 \leq G1. \\
G1/pos & \text{and} & G2/X \Rightarrow G3/neg & \text{implies} & G2/\neg X & \text{if } G1 \leq G3. \\
G1/neg & \text{and} & G2/X \Rightarrow G3/pos & \text{implies} & G2/\neg X & \text{if } G3 \leq G1.
\end{array}
$$

The first two rules are *modus ponens*, the other two implement *modus tollens*. For example, the third rule states: if the database contains a positive fact *A* and a rule stating "not *C* if *B*", then the fact "not *B*" will also be in the database if *C* is a generalization of *A*.

Whenever a new fact or rule is entered by the user, the interpreter tries to match it with the antecedent and consequent of every rule in order to determine if *modus ponens* or *modus tollens* may be applied. This process is recursively applied to every deduced fact. As soon as a fact (deduced or not) is about to be added, the database is searched for a more general one stating the opposite. If such a fact is found, the sentence must be incoherent with the previously stated premisses, forcing the interpreter to issue a message and to retract all the facts asserted during the deduction process.

*Interrogative sentences.* They may be simple questions (e.g. "What does the cat eat?") or of the form "*A* if *B*" stating hypotheses (e.g. "Does the mouse die if the cat eats?"). The graph representing the question will have the same form as for a declarative sentence, whereby wh-pronouns are simply dropped or substituted by generic concepts.

In the case of a simple question, looking for an answer consists in searching the database for a specialization of the graph representing the question. In the other case ("*A* if *B*"), the hypothesis *B* is temporarily put as a normal assertion in the database (i.e. it is tested for coherence with the known facts and all possible deductions are performed) and then *A* is treated like a simple question.

*Imperative sentences.* They are interpreted as instructions to the interpreter. In its actual version, commands consist of a single verb in the imperative form, e.g.

"stop!". The graph representing the command must be known to the interpreter, i.e. the program searches its internal command list for an exact copy of the graph.

Only three different commands are known in this version: "mostra!" (show), "apaga!" (erase), "para!" (stop). The first shows one by one on demand the current facts in the database, the second clears the database, and the third is used to quit the interpreter.

Some other aspects of the interpretation process are:

- The backtracking facility of Prolog is used to find alternative syntactic and semantic representations for the sentences and multiple answers for the questions.
- Fillmore's order of preference (agent, instrument, object) is used to join the verb and subject graphs.
- Relative clauses are translated into abstractions of the type corresponding to the noun they modify, like in [10] (an example is given in the appendix).
- Verb arguments and modifiers are distinguished in that the former restrict the concepts of the verb graph while the latter join new graphs, namely those of the modifiers. This means that the graph for the verb must already make provision for all possible arguments. Furthermore, the interpreter prevents arguments and modifiers from having the same semantic role.

Last, but not least, the interpreter can be used with or without CGE, the difference being how user input and program output is handled. In the former case, the CGE window is used to show the graphs, while dialog windows handle the user's input sentences. In the latter case, the linear notation and Prolog's basic I/O facilities are used.

The current state of the interpreter is not completely satisfactory as far as speed and flexibility are concerned. The coverage can also be much improved, but that wasn't the purpose of this application. The lack of speed is mainly due to the complexity of graph operations and to the constant copying of the intermediate graphs during the process to make backtracking possible. To increase flexibility and semantic coverage the interpreter could use schemata and the supertypes' canonical graphs.

## 5 Conclusions and Future Perspectives

As far as we know, GET is the first collection of tools to work easily with conceptual graphs (CG) in a logic programming environment with a graphical interface based on X-Windows. It is quite easy to build new types and relations with their associated background knowledge, especially using the CG Editor. The knowledge databases constructed in this manner could then be incorporated into other programs which would call the predicates provided by the CG Tools.

There were several advantages in using Prolog, in particular X-Prolog: an easy access to the X Toolkit C functions was possible, thus enabling the existence of a graphical editor; a linear notation parser and generator could be quickly built with a partially bidirectional DCG; and finally, the existing Portuguese extraposition grammar could be directly used for a toy semantic interpreter.

Unfortunately, X-Prolog is no longer supported. As such, we intend to port the Editor to APPEAL, an X-Windows based programming environment for SICStus Prolog. APPEAL also integrates the Widget Description Language and has the advantage of being supported by a company.

The main disadvantage of using Prolog is poor efficiency. The operations on conceptual graphs are extremely complex, mainly because of contexts (enabling the nesting of graphs) and coreference links (connecting nodes over arbitrary contexts), and the used data structures must be quite dynamic.

As far as it concerns the formalism itself, our overall feeling is that the conceptual structures' main source of expressiveness is also their main source of problems and fuzziness, namely the referents and coreference links. Therefore, some options had to be taken concerning some less clear points, others were deliberately postponed until the CG researching community agrees on them.

Nevertheless, the semantic interpreter showed that the mapping between natural language sentences and conceptual graphs is relatively straightforward. But there are problems such as anaphora resolution that require theoretically backgrounded treatment that we cannot find within the conceptual structures theory. Discourse Representation Theory [2] is the formalism we have chosen at the AI Centre (CRIA) of UNINOVA for handling some of those problems raised by text understanding or intentional participation in conversations by computers [3]. To concile the best of two worlds we envisage a Discourse Representation Structures (DRS) processor, as it already exists at CRIA, with anaphora resolution [5], etc., whereby the graphical visualization can be achieved using conceptual structures which are undoubtebly superior for expressing DRS conditions. The DRS—CG mapping could turn out to be easier than expected, as both formalisms have notions for contexts and referents.

## Acknowledgements

We would like to thank Irene Rodrigues for many fruitful discussions, Salvador Abreu for providing X-Prolog and Paulo Quaresma for providing his Graph Widget, both of which made CGE possible, and Claudia Ventura and Sabine Grüninger for reviewing this document.

## Appendix: An Example Session

The following is an excerpt of an actual session with the semantic interpreter without using CGE as the visual graph notation would take too much space. The examples illustrate mainly deduction and question answering. User input begins with '`|:`' and '`~`' stands for '`(NEG)->`'.

```
|: o gato joao come o rato se o rato nao fugir.
```

*(If the mouse doesn't run away, John the cat will eat it.)*

```
Syntactic analysis done! Semantic analysis done!
[PROPOSITION:
    [ESCAPE] -
        (AGNT) -> [MOUSE: #]
        (SRCE) -> [ENTITY]
] -
    (IMP) -> [PROPOSITION:
        [EAT] -
            (AGNT) -> [CAT] -> (NAME) -> ["Jo~ao"]
            (OBJ) -> [MOUSE: #]
    ]
    (NEG).
Another interpretation? (y/n) |: n

|: um rato nao foge se o rato come queijo.
```

*(If a mouse is eating cheese, it won't run away.)*

```
Syntactic analysis done! Semantic analysis done!
~[PROPOSITION:
    [ESCAPE] -
        (AGNT) -> [MOUSE]
        (SRCE) -> [ENTITY]
] <- (IMP) <- [PROPOSITION:
    [EAT] -
        (AGNT) -> [MOUSE: #]
        (OBJ) -> [CHEESE]
].
Another interpretation? (y/n) |: n

|: quem come o rato se o rato comer queijo branco?
```

*(Who eats the mouse if it eats white cheese?)*

```
Syntactic analysis done! Semantic analysis done!
[PROPOSITION:
    [EAT] -
        (AGNT) -> [MOUSE: #]
        (OBJ) -> [CHEESE] -> (ATTR) -> [WHITE]
] -> (IMP) -> [PROPOSITION:
    [EAT] -
        (AGNT) -> [ANIMATE]
        (OBJ) -> [MOUSE: #]
].
Another interpretation? (y/n) |: n
[PROPOSITION:
    [EAT] -
        (AGNT) -> [CAT] -> (NAME) -> ["Jo~ao"]
        (OBJ) -> [MOUSE: #]
```

```
].
Another answer? (y/n) |: y
I do not know.
```

```
|: os gatos que comem queijo nao comem ratos.
```
*(Cats that eat cheese don't eat mice.)*

```
Syntactic analysis done! Semantic analysis done!
~[PROPOSITION:
    [EAT] -
        (AGNT) -> [\a[CAT: *b = *a];
        [PROPOSITION:
            [EAT] -
                (AGNT) -> [CAT: *b]
                (OBJ) -> [CHEESE]
        ]]
        (OBJ) -> [MOUSE]
].
Another interpretation? (y/n) |: y
Sorry...
```

```
|: para!
```
*(Stop!)*

```
Syntactic analysis done! Semantic analysis done!
[PROPOSITION:
    [STOP] -> (AGNT) -> [ANIMATE]
].
Another interpretation? (y/n) |: n
Bye!
yes
|?-
```

## References


1. Salvador Pinto Abreu. *ALPES X-Prolog Programming Manual.* Centro de Inteligência Artificial, UNINOVA, 1989.
2. Hans Kamp and Uwe Reyle. *From Discourse to Logic: An Introduction to Modeltheoretic Semantics of Natural Language, Formal Logic and Discourse Representation Theory*, volume I. Kluwer, Dordrecht, 1991.
3. José Gabriel Lopes. Architecture for intentional participation of natural language interfaces in conversations. In C. Brown and G. Koch, editors, *Natural Language Understanding and Logic Programming III.* Elsevier Science Publishers, 1991.
4. José Gabriel Lopes and Irene Pimenta Rodrigues. Descrição parcial da sintaxe do Português. Technical report, CRIA/UNINOVA, June 1990.
5. José Gabriel Lopes and Irene Pimenta Rodrigues. Reasoning in resolution of temporal anaphores. Technical Note NT-1/91-CIUNL, Centro de Informática da UNL, January 1991.

6. Fernando Pereira and Stuart Shieber. *Prolog and Natural Language Analysis*, volume 10 of *CSLI Lecture Notes*. Center for the Study of Language and Information, 1987.
7. Fernando C. N. Pereira. Extraposition grammars. *American Journal of Computational Linguistics*, 7(4):243–255, 1981.
8. Paulo Quaresma. Graph widget: A tool for automatic data visualization. Technical Report RT-6/91-CIUNL, Centro de Informática da UNL, April 1991.
9. John F. Sowa. *Conceptual Structures: Information Processing in Mind and Machine*. The System Programming Series. Addison-Wesley Publishing Company, 1984.
10. John F. Sowa and Eileen C. Way. Implementing a semantic interpreter using conceptual graphs. *IBM Journal Res. Develop.*, 30(1):57–69, January 1986.
11. Richard Stallman. *GNU Emacs Manual*. Free Software Foundation, 1985.
12. Michel Wermelinger. GET: Graph Editor and Tools—the incomplete reference. Technical Report RT-3/91-CIUNL, Centro de Informática da Universidade Nova de Lisboa, January 1991.
13. Michel Wermelinger. GET—some notes on the implementation. Technical Report RT-4/91-CIUNL, Centro de Informática da Universidade Nova de Lisboa, January 1991.