Generating Conceptual Graphs From Functional Structures

by

Pierre Massicotte

B. Sc. (Honours), McGill University, 1985

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF
THE REQUIREMENTS FOR THE DEGREE OF
MASTER OF SCIENCE
in the School
of
Computing Science

SIMON FRASER UNIVERSITY
December 1988

ISBN 0-315-48805-0

Approval

Name :	Pierre Massicotte
Degree :	Master of Science
Title of Thesis :	Generating Conceptual Graphs from Functional Structures

Examining Committee:

Chairman: Dr. James P. Delgrande

Dr. Verónica Dahl
Senior Supervisor

Dr. Robert F. Hadley

Dr. Nick Cercone
External Examiner
School of Computing Science
Simon Fraser University

December 2nd, 1988
Date Approved

Title of Thesis/Project/Extended Essay

Generating Conceptual Graphs from Functional Structures

Author:

(signature)

Pierre Massicotte

(name)

Dec. 14, 1988

(date)

Abstract

Natural language analysis is the process of extracting the information carried by the various constructs of the language. This information can be used for various purposes, and depending on a particular purpose, can be appropriately represented by a specific formalism.

Analysis systems can be divided into many specialized tasks. This thesis is concerned with one of these tasks for natural language understanding. A semantic interpreter based upon a linguistic theory (*Lexical Functional Grammar*) is developed, which produces Conceptual Graph representations. Conceptual Graphs have been successfully used for other language processing tasks. Since there exist systems to generate language from this formalism [5, 4, 23], our interpreter could contribute to a machine translation system. Throughout this work, emphasis has been given to linguistic adequacy and implementation methodology rather than extensive language coverage: we have developed a methodology suitable to the two formalisms used. More possibilities offered by these formalisms could be added to this methodology to augment language coverage.

A representation for directed graphs in logic programming is also introduced. Its advantage lies in the possibility of verifying node accessibility very efficiently. The scope of the ideas presented for this representation goes beyond the particular application within which we exemplify it. For instance, the techniques developed for this verification operation can be used in any artificial intelligence application that represents knowledge through graphs.

This representation scheme contributes to the efficient implementation of an important semantic verification in our system. In a different framework, this representation scheme has been used to implement the enforcement of constraints expressed in terms of node domination in a syntactic structure.

Acknowledgements

This thesis would not be complete without including the expression of my gratitude towards many people who contributed time and effort. First, I would like to thank my senior supervisor, Dr. V. Dahl, not only for her guidance and generous help she provided throughout this work but also for the responsibilities and opportunities she offered me during my entire degree. Her constant support and encouragement have been an essential asset in bringing me to this point. I would also like to acknowledge my other supervisor, Dr. R. Hadley, for the useful comments and suggestions he brought to my attention. In addition, the understanding and care he has shown me at all times are very much appreciated. Dr. N. Cercone, my external examiner, also helped improving this document considerably. His positive and accurate suggestions transformed the fear of the defense into an instructive experience. Finally, Dr. P. Saint-Dizier contributed to this work through many conversations we had. It was he who introduced me to Lexical Functional Grammar while he was visiting Simon Fraser University.

The realization of such a lengthy document requires considerable knowledge of various text processing facilities. Steve Cumming's help is gratefully acknowledged in that respect.

I wish to express my deepest recognition to my colleague T. Pattabhiraman who has been very patient and resourceful in introducing me to computational linguistics. He has also been a very important friend at critical moments during my degree.

Finally, I want to thank all the other graduate students in the Computing Science Department who fostered a very pleasant work atmosphere.

À mes parents.

Table of Contents

Approval ii
Abstract iii
Acknowledgements iv
Table of Contents vi
List of Figures viii
1. Introduction 1
2. Preliminaries 5
2.1. The Problem 5
2.2. Initial Formalism 6
2.2.1. Introduction 7
2.2.2. Lexical Functional Grammar 7
2.2.2.1. Justification 7
2.2.2.2. Description of LFGs 9
2.2.2.3. Analysis Task with LFG 11
2.2.3. Functional Structures 11
2.2.3.1. General Format of Functional Structures 11
2.2.3.2. Verbal Information 13
2.2.3.3. Nouns 15
2.2.3.4. Pronouns 18
2.2.3.5. Modifiers 18
2.2.3.6. Adjuncts 20
2.2.4. Related Computational Work 21
2.3. Target Formalism 23
2.3.1. Justification 23
2.3.2. Description of Conceptual Graphs 23
2.3.3. Type Hierarchy 25
2.3.4. Related Work with Conceptual Graphs 26
3. A Graph Representation Scheme 27
3.1. Introductory Details 28
3.1.1. Definitions 28
3.1.2. Goal of the Representation 29
3.2. Usefulness to Our System 29
3.3. Related Work 30
3.3.1. An Integer Comparison Approach 30
3.3.2. A Unification Approach 30
3.4. A New Solution 31
3.4.1. Introduction 31
3.4.2. Description of the New Representation 32
3.4.3. Use of Difference Lists 34
3.4.4. Preprocessing 36

3.4.5. Comparison Mechanism 38
3.4.6. Analysis 38
3.5. Summary 40
4. Components of the System 42
4.1. Lexicon 42
4.1.1. Lexicon and LFG 43
4.1.2. Lexicon and CG 44
4.1.3. The Lexicon of Our System 45
4.1.3.1. θ-Roles 45
4.1.3.2. General Format of Lexical Entries 46
4.1.3.3. Consultation Mechanism 48
4.2. Type Hierarchy 50
4.2.1. Role of the Hierarchy 50
4.2.2. Implementation 51
4.3. Connection Between Both Components 52
5. Algorithms and Operations 53
5.1. General Strategy 53
5.2. Processing Subcategorizing Predicates 55
5.2.1. Subcategorizing for Grammatical Functions 55
5.2.2. Subcategorizing for a Clause 57
5.3. Processing Non-Subcategorizing Predicates 58
5.4. Processing Modifiers 58
5.5. Processing Relative Clauses 59
5.6. Processing Determiners 59
5.7. Processing Tense Information 60
6. Extensions and Conclusion 61
6.1. Summary of the Work Done 61
6.2. Extensions 62
Appendix A. Some Implementation Details 66
A.1. Representation of F-Structures 66
A.2. Representation of Conceptual Graphs 67
A.3. Important Predicates 68
Appendix B. Sample Runs 72
References 78

List of Figures

Figure 2-1: Machine translation paradigm assumed. 6
Figure 2-2: Example of a constituent structure. 9
Figure 2-3: Sample LFG rules. 10
Figure 2-4: General Format of F-Structure 11
Figure 2-5: F-Structure for *The girl handed the baby a toy.* 12
Figure 2-6: 14
Figure 2-7: Representation of sentences containing auxiliaries. 14
Figure 2-8: Example of a representation for a relative clause. 16
Figure 2-9: Representation of nouns subcategorizing for complements. 17
Figure 2-10: Representation of noun complements that are not subcategorized for. 18
Figure 2-11: Representation of modifiers. 19
Figure 2-12: Representation of a "modified" modifier. 20
Figure 3-1: Example of a graph structure. The type hierarchy used in CG has the same format. 32
Figure 3-2: Compiler generating the new graph representation. 37

Chapter 1
Introduction

Natural language analysis is the process of extracting the information carried by the various constructs of the language. Many applications for this process exist; but one of the most important is that of enhencing communications between humans and computers.

There is no doubt that a system that can "understand" what the requirements of its users are in a language more natural for them is a goal worth pursuing. Another challenging application for natural language analysis is in automatic machine translation. The objective here is to produce an appropriate representation for the meaning of a sentence that is suitable for use by a language generator which would produce a sentence expressing the same information in a different language. Many approaches for this task exist, some of which use more than one semantic representation in order to facilitate both language processing tasks.

This project is concerned with this latter application. We are interested in producing instances of a semantic representation scheme called *Conceptual Graphs* [31]. This project was mainly motivated by a previous project involving several researchers, including the author, which yielded a prototype for automatic language generation in which Conceptual Graphs were used as the semantic representation guiding the generation process. [4, 23, 6, 5] We have developed a framework to produce instances of this meaning representation that is based on a particular linguistic theory: Lexical Functional Grammar [22]. To our knowledge, this is the first attempt to unify these two contemporary theories. The ideas presented here have been implemented in Prolog.

There is no doubt that a good linguistic framework is essential to any system such as ours. Due to the intentions of its designers, this theory appears to be very suitable for work in natural language processing, and in particular language analysis.

In order to make our objective tractable, we concentrated on the generation of Conceptual Graphs from an intermediate representation which lies between syntax and semantics. This representation is an important part of the linguistic theory used and constitutes a very good representation from which a semantic representation can be derived. This intermediate representation, called *functional structures*, is not difficult to produce for a given sentence and implementations to generate it from various languages have successfully been developed. Therefore, we assume that any of these implementations could be combined to this work in order to produce a complete natural language analysis system.

Throughout this work, emphasis has been attributed to linguistic adequacy and implementation methodology, rather than extensive language coverage. We were mainly interested in developing a framework suitable to both formalisms used, on which extensions could be done modularly should broader cov~~erage be~~ sought. Nevertheless, the system we constructed is capable of handling an interesting subset of English. The set of target constructions originates from a collection of operating system error messages, which was used by the above mentioned language generator.

Aside from the main goal, an interesting result of this work is a new graph representation scheme in Logic Programming. We have devised a way of representing directed acyclic graphs that provides economical comparisons between nodes for accessibility. We show how this representation is useful to implement the type hierarchy of Conceptual Graphs, which is an important component of the system. This result originates in joint work with V. Dahl. The author's work stresses its application to the Conceptual Graph formalism [25], whereas V. Dahl has concentrated on applying it to represent and process syntactic structures on which the enforcement of constraints expressed in terms of node dominations has to be implemented [11, 15, 12, 13]. Such constraints are very important in another linguistic framework (Government and Binding [7]) and we elaborate on this application briefly in the last chapter.

The second chapter of this thesis will introduce the preliminary information relevant to the system developed. We begin this chapter by clearly stating the problem examined. An outline of the language coverage in included. The selection of this coverage was guided by that of the aforementioned generation system. An important motivation for the present system is automatic machine translation and we present the particular translation paradigm assumed. This paradigm is shared by both the present system and the existing language generation one, and therefore establishes a link between the two. Next, we describe each of the two formalisms used by the system. The linguistic framework underlying our system is introduced first. From this linguistic framework originates the initial formalism of this work. Before describing this initial formalism, we detail how it is obtained, as well as its position inside the overall linguistic theory. The general properties of the initial formalism are then presented. We also introduced some terminology that will be useful in the discussion of the subsequent chapters. Then we present actual representations of the formalism for various constructs of the speech. Most of these representations are exactly as dictated by the theory, but we included a few changes to better suit our needs. In doing these changes, we tried to stay in the philosophy of the theory. The last part of Chapter 2 is devoted to the presentation of the Conceptual Graph formalism, where the major aspects relevant to our system are briefly detailed. For each formalism, we also present some arguments to support their adoption in the system. Also, related computational work to each one is surveyed.

The next chapter will introduce the graph representation scheme. The set of structures it handles is formally identified. We specify the type of operations on those structures which we wish to perform efficiently. After having presented the scheme and a preprocessor to generate it automatically, we give an analysis of both the efficiency of the implementation of the operation of interest and the memory requirements needed. The efficiency gains are by and large due to the use of difference lists, on which many important operations can be reduced to unification. Two other approaches with similar goals are surveyed and some comparisons are drawn.

Chapter 4 will introduce the two major components of the system: the lexicon and the type hierarchy. We describe how the information needed by both formalisms can be combined and kept in a single unit. This contributes greatly to a simpler implementation of a system that uses both formalisms. The lexicon consultation mechanism contains interesting aspects from an implementational point of view. The discussion of the second component, the type hierarchy, mainly focuses on the role it plays within the system, as its structure and implementation will have been already introduced.

Chapter 5 is concerned with the algorithms and the various operations relevant to the production of conceptual graphs from functional structures. A single unit is responsible for handling all the parts of the input formalism and this contributes to the modularity of the approach. The most important parts are isolated and, for each one, the actual operations necessary to the correct processing are outlined.

The last chapter summarizes the work that has been done and points out directions for further research. An interesting approach to the problem of representing temporal information is mentioned. This approach is compatible with the theoretical basis of this work and could easily be incorporated in our system to provide a broader and more theoretical coverage of temporal adjuncts. The application of the graph representation scheme of Chapter 3 to the enforcement of linguistic constraints expressed in terms of node domination is also included.

Two appendices are included to cover some details relevant to the implementation. The first appendix begins with the description the Prolog representations used for both formalisms before listing the more important parts of the transformation program. The second appendix shows some sample runs of our implementation, illustrating some of the constructions handled by the system.

Chapter 2
Preliminaries

This chapter first describes explicitly the problem studied in this thesis. This description is followed by the introduction of the formalisms involved in the system, which is naturally divided into two parts. The first part is concerned with the linguistic theory where the initial formalism originates. The second part introduces the knowledge representation scheme generated by the system. In addition to a description, each section includes some justification for the choice of the corresponding formalism, as well as a survey of related work.

2.1. The Problem

The main aim of this thesis is to develop a semantic interpreter to produce conceptual graphs from functional structures. This interpreter takes as input a sentence represented in an intermediate formalism between syntax and semantic and produces a representation in the conceptual graph formalism.

The identification of the subset of the language covered was inspired from a collection of operating system error messages used in a previous project, in which we were involved [4, 23]. In addition to regular subject-verb-object sentences, the current system handles:

- arbitrarily embedded modifier constructions (e.g., *very large, not very pretty*),
- modal and auxiliary constructions, including passives (e.g., *A man is drinking wine.*)
- arbitrarily embedded relative clauses (e.g., *The students read a book that won a prize which deserves mention.*)
- some complex noun phrases containing prepositional phrases (e.g., the construction of house by the workers)

Throughout the development of this work, particular attention has been focussed on the generality of the process so that the techniques used could easily be extended to accommodate a larger subset of the language.

The framework of the previous project was machine translation. The translation paradigm that was assumed can be visualized by the diagram of figure 2-1.

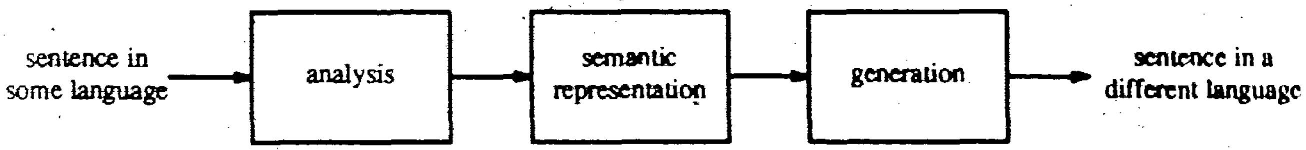

Figure 2-1: Machine translation paradigm assumed.

According to this approach, the translation task is divided into two independent components, *analysis* and *generation*, plus some overhead considerations to ensure compatibility between the two. The purpose of that project was to investigate the possibility of automatically generating (English) sentences describing typical operating system error messages. The generation of a particular sentence was guided by the encoding of its "meaning" in the Conceptual Graphs formalism. Material related to this project can be found in [4, 23, 6, 12, 15, 5]. Hence, the prototype developed in that project implemented the second component of the process shown in figure 2-1.

The present work adopts the translation paradigm depicted above[1] and materializes a step in the other component. However, this work does not constitute a complete analyser as the starting point is an abstract representation over sentences. We chose to overlook the production of this abstract representation, since there exists already a number of implementations producing such a representation (see section 2.2.4).

2.2. Initial Formalism

In our description of the problem, we mentioned the intermediate nature of the initial formalism. In this section, we describe this formalism in some detail; but first we introduce the theory in which it originates.

[1]We refer the reader to [30] for a survey of different approaches to machine translation.

2.2.1. Introduction

A solid linguistic background is, for obvious reasons, an essential requirement for any computational linguistic application. Linguistics is still an evolving discipline; there is no unanimously recognized theory that captures the phenomena of language soundly and completely. Although all linguistic theories aim at a similar goal (that of providing an adequate model for language) they diverge considerably in their respective approach, as each one is supported by a different set of assumptions.

Such distinctions among theories can occur at various levels. For instance, at the syntactic level, a distinction can be the use or not of transformational rules. In evaluating linguistic theories for computational uses, another important point to examine is the inclusion or not of semantic considerations in the syntactic component. In addition to their respective impact in linguistics, these and other similar aspects of theories can imply important consequences in a computational linguistic application. Considerations of this nature played an important role in selecting the linguistic basis of this work.

2.2.2. Lexical Functional Grammar

The linguistic formalism that we chose is called *Lexical Functional Grammar* (LFG) developed by Kaplan and Bresnan [22]. It has its roots in previous results in transformational grammar as well as in considerations from computer science and psychology. This grammar formalism requires only simple phrase structure rules, by transferring a substantial amount of syntactic information into the lexicon and by using multiple levels of representation.

2.2.2.1. Justification

The main attraction of this formalism is its modular approach to grammar. The different intermediate levels are obtained without any semantic considerations, which makes it very suitable for our purposes. The simplicity of the syntactic component reinforces this suitability to analysis. The exclusion of transformational rules is partly responsible for the simplicity of the grammar rules. Let us now examine this last statement in some detail.

Transformation rules are syntactic rules that do not introduce any new items in a syntactic representation but, instead, rearrange already generated items. They take a more abstract representation of a sentence (deep-structure) and produce one or many corresponding surface-structure(s), which account(s) for the particular word ordering of a given sentence. During analysis, such transformations must be reversed in order to retrieve the deep-structure of the sentence. Such rules are responsible, within their own grammatical environment, for constructions such as passives, wh-questions, and other complex constructions.

From a computational point of view, a grammar containing transformation rules provides some desirable aspects, as they help to keep the number of rules to a minimum by having rules that can apply to a collection of constructs, rather than having an individual production for each one. This reduction of the size of the grammar is usually achieved by sacrificing some efficiency: the application of transformations is guided by constraints whose implementations have traditionally been of limited efficiency. However, more recent work has shown some gain in the efficiency of such constraint implementations [4, 11, 12].

The most complete application in the aforementioned references mainly focuses on language generation, as opposed to analysis. However, the solutions they propose could be applied to analysis, but in using transformations in this latter process, an additional problem arises. In general, before a transformation can be done or undone, the item to be moved along with its destination must be identified. In the case of generation, the presence of specific nodes in the syntactic representation, which are natural destinations for movement, helps greatly to simplify the various possibilities. No such help is directly available during analysis. Therefore a grammar that does not include transformations can lead to a more efficient implementation of analysis.

LFGs avoid the complexity of the syntactic component usually encountered in non-transformational approaches by representing a substantial amount of syntactic information in the lexicon. However, for our purposes, this trade-off between syntactic simplicity and lexicon complexity is not as pronounced. The conceptual graph formalism also necessitates some information that is similar in some respect. A major hypothesis behind this work is that a lot of this information overlaps with what is needed by the LFG formalism and can also be represented in the lexicon.

2.2.2.2. Description of LFGs

LFG is probably the only contemporary linguistic theory that uses traditional grammatical relations (such as subject, object, etc.) as primitives. The validity of a sentence, according to this grammar, depends mainly on specifications stated in terms of these relations. In this theory, every sentence is assigned two pre-semantic representations: *constituent structure* (c-structure) and *functional structure* (f-structure). The first representation is basically a parse tree, that is a hierarchical structure indicating the various combinations of words into phrases inside a sentence. This structure is obtained by context-free-like rules (i.e., rules that have only one symbol on their left hand side) to which is associated some additional information, called *functional annotations*, used to construct the second representation. These functional annotations are dependent upon the the properties of the head of each category. Hence there is no one to one correspondence between the constituent and functional structures of a sentence. Figure 2-2 shows the c-structure of the sentence

The cat drank the milk

obtained using the rules shown in figure 2-3.

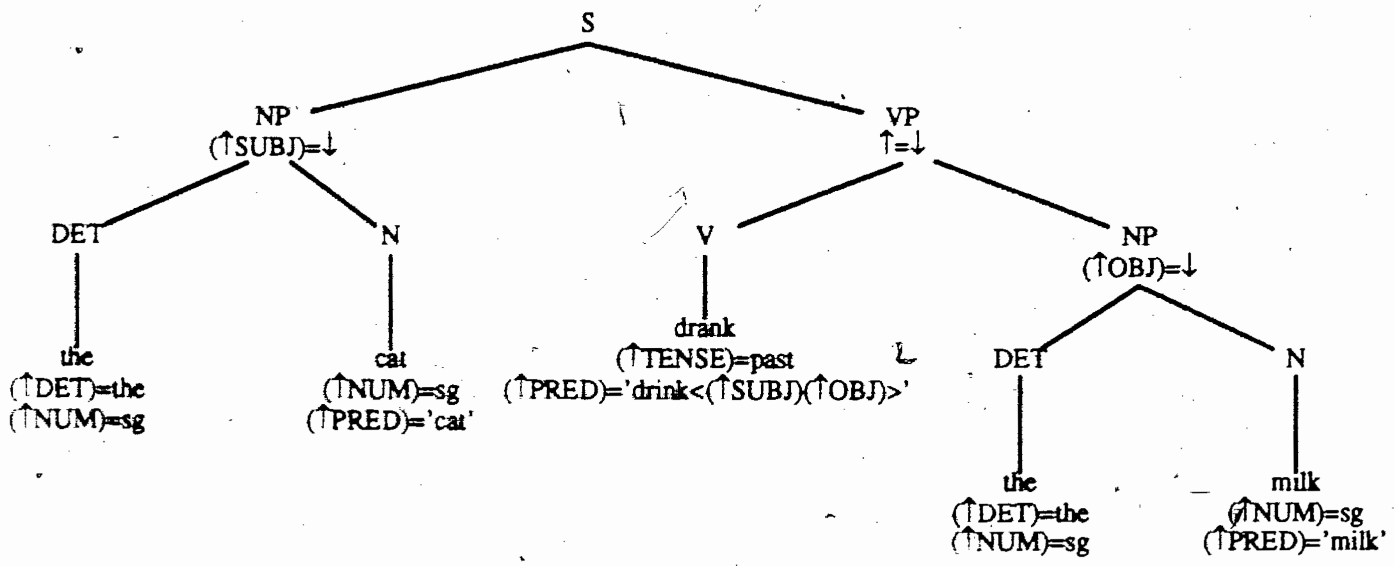

Figure 2-2: Example of a constituent structure.

The arrows included in the rules of figure 2-3 are variables referring to f-structures. The up-arrows ("↑") refer to the f-structure of the mother node and the down-arrows ("↓") refer to the one

$$\begin{array}{llcc}
S & \rightarrow & \underset{(\uparrow \text{SUBJ})=\downarrow}{NP} & \underset{\uparrow=\downarrow}{VP} \\
VP & \rightarrow & V & \underset{(\uparrow \text{OBJ})=\downarrow}{NP} \\
NP & \rightarrow & (DET) & N
\end{array}$$

Figure 2-3: Sample LFG rules.

of the current node. So for instance, the annotation under the NP node in the S-rule specifies that the f-structure of that node corresponds to the subject part of the f-structure of the sentence. Similarly, the annotation of the VP node ($\uparrow = \downarrow$) of the same rule specifies that the information about that node is directly included in its mother's f-structure. It is assumed that the annotation $\uparrow = \downarrow$ is associated with each pre-terminal node.

The information actually carried by these variables is obtained from the lexicon. To every lexical entry is associated some specific functional information. For instance, the followings could be possible entries for our sample sentence:

cat	N	($\uparrow$PRED) = 'cat'
milk	N	($\uparrow$PRED) = 'milk'
drank	V	($\uparrow$PRED) = 'drink<($\uparrow$SUBJ)($\uparrow$OBJ)>'
		($\uparrow$TENSE) = past

The "<($\uparrow$SUBJ)($\uparrow$OBJ)>" part in the verb entry is the subcategorization restriction specified by that verb. Section 2.2.3.1 will present actual format of f-structures for various constructs.

In this lexical approach to grammar, it is assumed that there is a different lexical entry for each use or form of a word. For instance, verbs that passivize will have a separate lexical entry for both their active and passive use. It is this strategy that is responsible for keeping the syntax rules needed to describe the language very simple, as most of the burden usually carried out by sophisticated rules in other frameworks is accomplished in this theory by an extended lexicon.

2.2.2.3. Analysis Task with LFG

Using this LFG formalism, the task of analysis naturally separates into independent components. First, a c-structure is derived for the sentence using the annotated context-free rules. Then, all the functional specifications are gathered to derive the f-structure. This collection of functional annotations, called the *functional description* of a sentence, can be considered as forming a set of equations expressing some property of the sentence. The solution to this set of equations is the f-structure of the sentence. Multiple solutions lead to syntactic ambiguity. Functional descriptions statements can be used to either directly generate an f-structure or to apply some constraints on existing structures, ensuring that they correspond to properties allowed by the grammar. An algorithm to solve these equations is given in [22]. The f-structures so produced then serve as input to the semantic interpreter.

2.2.3. Functional Structures

We saw briefly how a functional structure is derived from a sentence. These structures are the starting point of the work presented here. We now look in some details at the general format of these structures before examining representations for various constructs of the speech.

2.2.3.1. General Format of Functional Structures

An *f-structure* is a collection of attribute-value pairs expressing grammatical relations and other information relevant to the semantic interpreter. Its general format is shown by figure 2-4.

$$\begin{bmatrix} \text{Attribute}_1 & \text{Value}_1 \\ \text{Attribute}_2 & \text{Value}_2 \\ \vdots & \vdots \\ \text{Attribute}_n & \text{Value}_n \end{bmatrix}$$

Figure 2-4: General Format of F-Structure

The attributes can either be the name of a grammatical relation (e.g., subject, object, etc.), the name of a syntactic feature (e.g., tense, agreement, etc.)[2] or other syntactic information such as modifier, adjunct, relative. There are four types of values: symbols, semantic predicates, f-structures and sets. A *symbol* is the simplest type and can be thought as an atom in conventional logic formalisms. *Semantic predicates* originate from the lexicon and are to be handled by the semantic interpreter. Some of them take argument(s), referring to other parts in the structure. The concept here is similar to that of a logical predicate. The third type of value is an *f-(sub)structure*, that is a collection of attribute-value pairs as currently described, representing some information that form an entity, such as the information associated with a subject. Intuitively, a *set* here is a finite unordered collection of items. Figure 2-5 shows an example of a f-structure for the sentence *The girl handed the baby a toy*, possibly produced by Kaplan and Bresnan's grammar.

$$\begin{bmatrix} \text{SUBJ} & \begin{bmatrix} \text{DET} & \text{the} \\ \text{AGR} & \begin{bmatrix} \text{NUM} & \text{sg} \\ \text{PERS} & 3 \end{bmatrix} \\ \text{PRED} & \text{'GIRL'} \end{bmatrix} \\ \text{TENSE} & \text{past} \\ \text{PRED} & \text{'HAND<(}\uparrow\text{SUBJ)(}\uparrow\text{OBJ2)(}\uparrow\text{OBJ)>'} \\ \text{OBJ2} & \begin{bmatrix} \text{DET} & \text{the} \\ \text{AGR} & \begin{bmatrix} \text{NUM} & \text{sg} \\ \text{PERS} & 3 \end{bmatrix} \\ \text{PRED} & \text{'BABY'} \end{bmatrix} \\ \text{OBJ} & \begin{bmatrix} \text{DET} & \text{a} \\ \text{AGR} & \begin{bmatrix} \text{NUM} & \text{sg} \\ \text{PERS} & 3 \end{bmatrix} \\ \text{PRED} & \text{'TOY'} \end{bmatrix} \end{bmatrix}$$

Figure 2-5: F-Structure for *The girl handed the baby a toy.*

In this structure, SUBJ, TENSE, PRED are examples of attributes. PAST is a symbol value, 'HAND<(SUBJ) (OBJ2) (OBJ)>' and 'GIRL' are semantic predicates, and the value of SUBJ, OBJ, OBJ2 are f-structures.

[2]Appropriate abbreviations are usually used in the actual representations.

Three conditions control the well formedness of f-structures. The *uniqueness condition* ensures that every attribute has at most one value. The other two deal with the arguments of the semantic predicates. Together they impose a one to one correspondence between the arguments of a predicate and the parts of the structure that depend upon them. The *completeness condition* specifies that an f-structure must contain all the grammatical functions that appear in the argument list of the predicate. That is every argument of a predicate (e.g., SUBJ, OBJ2, OBJ in figure 2-5) must correspond to an attribute whose value represents the grammatical function. The term *government*[3] designates the relation between a predicate's argument(s) and the grammatical functions for which that subcategorizes. The *coherence condition* takes care of the counter part, stipulating that every grammatical function present in a f-structure must be governed by the arguments of the predicate of the sentence. An input string to which is associated an f-structure satisfying these three conditions is accepted as a grammatical sentence.

With these specifications, we can now examine actual f-structure representations for some parts handled by our system. In what follows, the mention of traditional grammatical categories is purely for exemplifying purposes. As it will be obvious shortly, there is no such distinction in the formalism.

In order to ease the following discussion, we introduce some terminology. Two entries in a f-structure are said to be *neighbours* (or alternatively in the same *neighbourhood*) if they are part of exactly the same f-structures. For example in figure 2-6, SUBJ and VCOMP, NUM and DET are in the same neighborhood but DET and VCOMP, and DET and TENSE are not. Two neighbouring entries are also said to be at the same *level*.

2.2.3.2. Verbal Information

In general, verbs are represented by a predicate containing arguments governing other parts. These predicates, together with the appropriate tense and agreement specifications, appear in the f-structure of their respective clause. However, auxiliaries[4] are understood as the main verb of a

[3]The present notion of government should not be confused with the one developed in Government and Binding theory.

[4]Under auxiliaries are included in this theory modal verbs, helping verbs (for progressive and perfect forms) and verbs used in passive constructions.

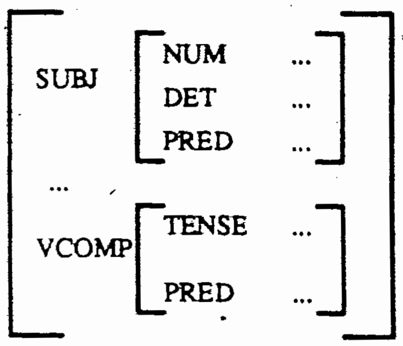

Figure 2-6:

clause, taking another clause (predicated by the regular verb with which the auxiliary is used) as complement. For instance, figure 2-7 shows the main lines of the representation of an English sentence having a progressive construction.

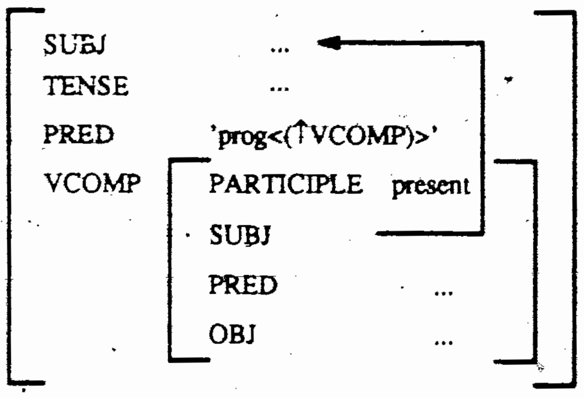

Figure 2-7: Representation of sentences containing auxiliaries.

In this representation, a pointer (represented by the arrow) is used to indicate that the value of its SUBJ attribute is exactly the same as the value pointed to. Such pointers arise in many constructions within the formalism. The representation of clauses without auxiliaries is easily deduced from figure 2-7.

2.2.3.3. Nouns

In contrast to verbs, nouns can play one of many grammatical relations These relations, as outlined above, appear in the f-structure of a sentence. As for verbs, the noun itself is represented by a semantic predicate that may or may not take arguments. This predicate is part of an f-structure always introduced by a grammatical relation (SUBJ, OBJ, etc.) attribute. The rest of the information related to the noun appears at the same level as the PRED attribute. This information includes the type of determiner, the agreement features, and possibly a relative clause, some modifiers, and a noun complement.

The type of determiner is represented by an entry having the attribute DET taking only a symbol as value. Determiners include articles and demonstratives. Hence possible values are *a, the, this, that, these, those*. In our application, we require this attribute-value pair to be present in the representation of every noun. The reason for this is that we make a distinction between a noun that occurs without a determiner and nouns that can never have any (e.g., proper nouns). The value in the former case is *none* and in the latter *n/a* (for non-applicable).

The agreement part has the attribute AGR to which is associated an f-structure value containing attribute-value pairs [NUM ...] (for the number) and [PERS ...] (for the person). Case and gender are not necessary for English but could be easily added in this structure (with a similar format as the number of the person) for other languages. The representation for the modifiers will be discussed in a forthcoming sub-section.

Relative clauses, when present, are represented by an f-structure being the value of the attribute REL. This attribute is a neighbor of the semantic predicate of the noun modified by the relative. The same parts as a main clause can appear inside a relative representation. However, there is an extra entry for the relative marker. This field has the attribute REL-MARK. It is kept even thought the relative marker usually has a grammatical function in the relative. Since the same value cannot have two attributes, the value of the attribute of the grammatical role played by the relative marker will be a pointer to the f-structure of the relative marker.

This format covers constructions such as

I read a book that won a prize.
The wine John bought is excellent[5].

where nouns are being modified by a clause. For example, the noun phrase *The book that John read* has the representation shown in figure 2-8

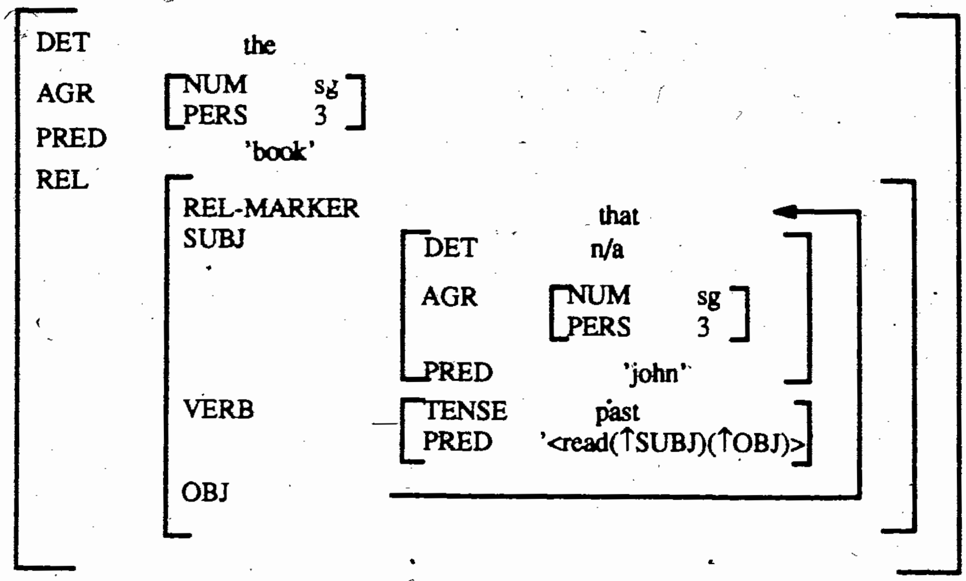

Figure 2-8: Example of a representation for a relative clause.

In addition to relative clauses, nouns can also be modified by other nouns. We distinguish two separate situations in this case, resulting in two different f-structure representations. The first one covers nouns that are always used with such modifiers. Consider for example:

The construction of the boat by the sailors.

In that use of the noun *construction*, there is always a *by-object* and a *of-object* implied, even when not explicitly stated. Such nouns are similar to verbs in that they imply and relate together other parts of the sentence. Hence it is natural to treat these nouns in a similar manner as verbs. Frey [16] uses semantic predicates taking arguments, corresponding to the predicate of the verb of the same root (e.g., construct, construction). The number of arguments is usually the same in both cases; but their names, however, may differ. Those arguments relates to attribute-value pairs at the same level as the predicate for the noun, just as in the case of verbs, as shown in figure 2-9. Hence, the Completeness and Coherence Conditions apply to these arguments.

[5]The relative marker is not realized in this example

$$\begin{bmatrix} \text{DET} & \text{the} \\ \text{AGR} & \begin{bmatrix}\text{NUM} & \text{sg}\\ \text{PERS} & 3\end{bmatrix} \\ \text{PRED} & \text{'construction<}(\uparrow\text{BY-OBJ})(\uparrow\text{OF-OBJ})\text{>'} \\ \text{OF-OBJ} & \begin{bmatrix} \text{DET} & \text{the} \\ \text{AGR} & \begin{bmatrix}\text{NUM} & \text{sg}\\ \text{PERS} & 3\end{bmatrix} \\ \text{PRED} & \text{'boat'} \end{bmatrix} \\ \text{BY-OBJ} & \begin{bmatrix} \text{DET} & \text{the} \\ \text{AGR} & \begin{bmatrix}\text{NUM} & \text{plur}\\ \text{PERS} & 3\end{bmatrix} \\ \text{PRED} & \text{'sailors'} \end{bmatrix} \end{bmatrix}$$

Figure 2-9: Representation of nouns subcategorizing for complements.

The second situation covers nouns to which precision not entailed by them is added. A good example of that is the possessive information as in *the book of John*. These nouns phrases are called *noun complements* and are represented in a similar way as relatives. Our representation follows the intuitive concept that such constructions can be easily paraphrased using a relative clause (e.g., the book of John → the book that belongs to John). Furthermore, as it is the case for adjuncts (which will be discussed shortly), the last two conditions on well-formedness cannot apply to such constructions. We use the attribute NCOMP and an f-structure value to represent the complement itself. Figure 2-10 is a good illustration of the possible nestedness that can occur in such a structure. The predicate *color* is, as expected, at the outmost level in that structure since it is the one that would carry the grammatical relation associated with that noun phrase in a sentence.

Some nouns can be used in either situation[6]. In this case, different semantic predicates are kept from the lexicon, reflecting a difference in the handling of the semantics of both cases.

[6]Notice that the case of nouns that always appear with modifiers is simply the regular case with an additional entry.

$$\begin{bmatrix}
\text{DET} & \text{the} \\
\text{AGR} & \begin{bmatrix}\text{NUM} & \text{sg} \\ \text{PERS} & 3\end{bmatrix} \\
\text{PRED} & \text{'color'} \\
\text{NCOMP} & \begin{bmatrix}
\text{DET} & \text{the} \\
\text{AGR} & \begin{bmatrix}\text{NUM} & \text{sg} \\ \text{PERS} & 3\end{bmatrix} \\
\text{PRED} & \text{'uniform'} \\
\text{NCOMP} & \begin{bmatrix}
\text{DET} & \text{the} \\
\text{AGR} & \begin{bmatrix}\text{NUM} & \text{sg} \\ \text{PERS} & 3\end{bmatrix} \\
\text{PRED} & \text{'team'} \\
\text{MOD} & \begin{bmatrix}\text{QUAL} & \text{visiting}\end{bmatrix}
\end{bmatrix}
\end{bmatrix}
\end{bmatrix}$$

Figure 2-10: Representation of noun complements that are not subcategorized for.

2.2.3.4. Pronouns

The representation of pronouns is simpler than the one of nouns. There is no need for the DET, REL and NCOMP fields. The PRED and AGR fields retain the same format[7]. There is an additional field for the antecedent of the pronoun. The value of such a field is a pointer to the f-structure of the antecedent. However, this field must be optional in the representation of pronouns, since the antecedent may not be explicitly stated. When not explicit in the sentence, the task of relating a pronoun to its antecedent is clearly part of the semantic interpretation.

2.2.3.5. Modifiers

Modifiers are not very much detailed in the literature. We developed a representation for them that follows the philosophy of the theory. Nouns and verbs can be modified by adjectives and adverbs (respectively). In turn, these modifiers can be modified by others as in *very tall*. Hence, we sought a representation for modifiers that would be general enough to be applied to many different constructions, regardless of their nature.

[7]There is never any argument in the predicate of a pronoun.

Each modifier of a noun or a verb has its own entry, of which its modifier(s), if any, are part. The modifiers are introduced by the attribute MOD and an f-structure value containing the pair [QUAL ...] and possibly the pair [OP ...] for negation. The collection of these [MOD ...] pairs becomes the value of the attribute MODATTR that appears at the same level as the PRED of the modified category. The same structure is used inside this one to handle the modifier to the one being represented in the first place.

Simple modifier(s) to a noun (or to a verb) are represented as in figure 2-11.

$$\begin{bmatrix} \text{DET} & \text{the} \\ \text{MODATTR} & \begin{bmatrix} \text{MOD} & \begin{bmatrix} \text{QUAL} & \text{old} \end{bmatrix} \\ \text{MOD} & \begin{bmatrix} \text{QUAL} & \text{rusty} \end{bmatrix} \\ \text{MOD} & \begin{bmatrix} \text{QUAL} & \text{yellow} \end{bmatrix} \end{bmatrix} \\ \text{PRED} & \text{'car'} \end{bmatrix}$$

Figure 2-11: Representation of modifiers.

The MOD attribute appear at the same level as the PRED of the modified category. When a modifier to another modifier is present, then a structural representation similar to the one of figure 2-12 is used. Again, the second MODATTR attribute appears at the same level than the category being modified (friendly). As for verbs, OP only appears when negation is employed. This is a modular representation scheme that allows to keep together all information related to each modifier.

$$\begin{bmatrix} \text{DET} & \text{the} \\ \text{AGR} & \begin{bmatrix}\text{NUM} & \text{sg}\\ \text{PERS} & 3\end{bmatrix} \\ \text{MODATTR} & \begin{bmatrix}\text{MOD} & \begin{bmatrix}\text{QUAL} & \text{friendly}\end{bmatrix}\\ \text{OP} & \text{not} \\ \text{MODATTR} & \begin{bmatrix}\text{DEG} & \text{very}\end{bmatrix}\end{bmatrix} \\ \text{PRED} & \text{'officier'}\end{bmatrix}$$

Figure 2-12: Representation of a "modified" modifier.

2.2.3.6. Adjuncts

We have seen how verbs and certain nouns relate other parts of the sentences in a similar way as a logical predicate by the use of arguments. However, a sentence can contain other elements that although modifying or complementing the predicate, are not syntactically related to it. This is the case, for instance, of the temporal or locative information in the sentence:

They met at three o'clock, in the park.

three o'clock and *in the park* serve as *adjunct* to the predicate *meet*. The information carried by these adjuncts is important for the semantic interpretation but cannot be associated with the predicate in the same manner as the subject or the object because predicates do not put any restriction on their adjuncts other than semantic ones. Hence adjuncts are not affected by the Completeness and Coherence Conditions.

Kaplan and Bresnan give an overview of how adjuncts can be treated. They chose to represent them as a set. This set is the value of the attribute ADJUNCT and its elements are f-structures, one for each particular adjunct. They give no detail about the internal representation of each adjunct.

We propose a more detailed structure for the adjuncts of a sentence. Each particular one has an f-structure being the value of the attribute describing the type of adjunct (e.g., location, time, cause, etc.). Each of these f-structures is basically the same as the one for nouns, except that there will be an additional field – having the attribute RELATION – to describe how the event represented by the predicate is affected by the adjuncts. This allows us to distinguish easily between things like *in the park*, *in front of the park*, etc. The f-structure representation for that is simply a field with the attribute RELATION and with an f-structure value. Such a value type is necessary here in order to handle possible modifiers as in *shortly after four o'clock*. The f-structures for all the adjuncts of a sentence are gathered into another f-structure which is the value of the attribute ADJUNCT, appearing at the same level as the predicate of the verb modified.

$$\left[\begin{array}{ll}
\text{LOCATION} & \left[\begin{array}{ll} \text{DET} & \text{the} \\ \text{PRED} & \text{'park'} \\ \text{RELATION} & \left[\begin{array}{ll}\text{TYPE} & \text{in}\end{array}\right] \end{array}\right] \\
\text{LOCATION} & \left[\begin{array}{ll} \text{DET} & \text{the} \\ \text{PRED} & \text{'fountain'} \\ \text{RELATION} & \left[\begin{array}{ll}\text{TYPE} & \text{in\_front\_of}\end{array}\right] \end{array}\right] \\
\text{TIME} & \left[\begin{array}{ll} \text{DET} & \text{n/a} \\ \text{PRED} & \text{'4-o'clock'} \\ \text{RELATION} & \left[\begin{array}{ll}\text{TYPE} & \text{after} \\ \text{MOD} & \left[\begin{array}{ll}\text{QUAL} & \text{shortly}\end{array}\right]\end{array}\right] \end{array}\right] \\
\text{CAUSE} & \left[\begin{array}{ll} \text{DET} & \text{a} \\ \text{PRED} & \text{'meeting'} \\ \text{RELATION} & \left[\begin{array}{ll}\text{TYPE} & \text{for}\end{array}\right] \end{array}\right]
\end{array}\right]$$

2.2.4. Related Computational Work

Frey and Reyle [17] completed a Prolog implementation of this relatively new linguistic theory. Their system inputs a sentence in German (##check##) and produces an f-structure. In accordance with the theory, they divide the task of parsing with this formalism into three major steps. First a parse tree is generated by the context-free-like rules. Then functional equations are instantiated. These equations are at the category level and their values are coming from the lexical items. Finally these equations are solved, producing f-structures. Their paper shows how to translate LFG into Definite Clause Grammar (DCG) and claims that the use of DCG and the procedural semantics of Prolog leads to an efficient implementation by having the above three steps realized simultaneously.

This work was later used as the parsing component in a knowledge base interface [18]. Frey et al.

developed a question-answering system capable of handling requests in a subset of German. The most important constructions with respect to such an application are covered, including conditional and relative clauses, universally and existentially quantified noun phrases, sentence and constituent negation, etc. The f-structures output associated with the input sentences are then passed to a semantic analyser. Discourse Representation Structures (DRS) developed by Kamp are the semantic formalism of this system. Finally, a data base is built from the DRS's using two possible methods: DRS's can be translated into Prolog clauses augmented with some deductive principles, or some inference rules can be derived to operate on these structures.

Another Prolog implementation of LFG was done by Yasukawa [37]. Although he aimed at comparable goals with Frey and Reyle's (use of DCG, direct constructions of f-structures during parsing), he focussed more on building a formal system to represent syntactic knowledge. As a result, he came up with good data structures for representing the various LFG primitives. These abstract data types are hidden from the user in order to keep the grammar rules closer to the original LFG theory. This is realized by having a macro notation that is similar to regular LFG rules, which are later translated into Prolog programs. This gives a less efficient implementation than the first one. However the LFG coverage is claimed to be wider in this second implementation.

Frey also investigated the details of a particular construction, the noun phrase, in the specific context of LFG [16]. He gives complete LFG treatment for constructions like comparatives, partitives, and noun complement. This includes defining appropriate phrase structure rules, along with associating these rules with functional schemata in order to produce f-structure.

Uehara et al. [35] used also LFG in their Integrated Parser. Their project integrated into a same module syntactic, semantic, and contextual analysers. Their goal aimed at text understanding as opposed to understanding sequences of individual sentences. Having these three components grouped together helps resolving some ambiguities that would be difficult to handle in a sequential process (i.e. syntax $\rightarrow$ semantics $\rightarrow$ context).

2.3. Target Formalism

The semantic representation that we used for this interpreter is the Conceptual Graphs (CG). These graphs were developed by John Sowa and are based on evidence from linguistics, philosophy, and artificial intelligence. In [31], Sowa first motivates conceptual graphs from different parts of cognitive science. Then he introduces the graphs formally in an axiomatic way. Finally he explains how conceptual graphs can be in some areas of Artificial Intelligence.

2.3.1. Justification

Conceptual graphs provide a good formalism to capture many aspects of natural language. The main motivation for using this formalism is to entertain the possibility of producing instances of the formalism used in the generation system, as mentioned above, in order to build a translator.

2.3.2. Description of Conceptual Graphs

The information in CGs is divided into two classes, each of which corresponding to a type of node. The concept nodes, represented by a box, carry the information about entities, attributes, states, and events. The relation nodes, always appearing in a circle, show how the concepts are related and the role that each one plays. These nodes can be thought of as names associated with every arc of the graph. These relations are directed and most of them are monadic or dyadic. However, there is nothing in Sowa's formalism that prevents having relations with a greater number of arcs. This type of nodes indicates the case relations, as well as the logical and causal links between various parts.

The concept nodes contain a type label and possibly a referent. A *type label* is a token describing the nature of a concept. The *referent*, separated from the label by a colon, specifies the extension of the concept referred to in the sentence. These referent include generic, individual, generic set, specified and partly specified sets. The following table illustrates the differences between each one.

Every generic concept is assumed to be existentially quantified. Variables can be used in generic concepts to indicate cross-reference: two concepts with the same label in a graph are assumed to represent distinct entities, unless they appear with the same referent (including variable name).

KIND OF REFERENT	GRAPH NOTATION	POSSIBLE READING
Generic	[MONKEY] or [MONKEY:*]	a monkey
Individual	[MONKEY:#29] or [MONKEY:#]	the monkey
Generic Set	[MONKEY: {*}]	monkeys
Named Individual	[MONKEY:Jocko]	Jocko
Specified Set	[MONKEY: {Jocko, Toto}]	Jocko and Toto
Partially Specified Set	[MONKEY: {Toto,*}]	Toto and others

All the labels are grouped into a partial ordered structure defined over the set of all the type labels, according to the level of generality. This hierarchy is introduced in the following section.

A particular CG is assumed to represent a unique proposition. It is possible to find a CG for any syntactically well formed sentence. However not all such sentences are meaningful. For instance, the sentence:

The tree ate a green dream.

is a syntactically valid English construction but conflicts with common sense knowledge. It is, for obvious reasons, desirable to prevent a semantic interpreter from finding representation for such sentences and to rule them out as nonsense, inconsistencies, errors, etc.

Sowa identifies the subset of conceptual graphs that represent actual or possible situations as *canonical graphs*. Initially, there is a set of CGs that form the *canonical basis* of the system. This basis associates a valid graph with every type in the hierarchy. Examples of such graphs are:

[EAT]–(AGENT)→[ANIMAL]
[PHYSOBJ]–(ATTR)→[COLOR][8]

The first one specifies that the agent of EAT must be of type ANIMAL. The second one ensures that COLOR is an attribute associated with a physical object. Such graphs can be used as semantic constraints to rule out sentences such as the one above, or to give patterns that are expected to be present with a particular concept. For instance, the concept GIVE is generally associated with someone giving, someone receiving, and something being transferred, so a canonical graph for this concept should include all these three items.

[8]CGs can also be represented in a linear form by using square brackets for boxes, and parentheses for circles.

There are four rules defined to form new canonical graphs out of existing ones. **Copy** creates an exact duplicate of a CG. **Restrict** replaces the type label of a concept by the label of a subtype; or if the concept is generic, change the referent to an individual (e.g., [DOG:*] $\rightarrow$ [DOG:SNOOPY]). In both cases, the conformity relation (to be introduced in the next section) must be satisfied before and after the modification. **Join** merges identical concepts found in two different graphs. The new graph is obtained by removing one instance of the concept and linking to the other one all the arcs originally connected to the first one. **Simplify** removes identical relations connecting the same concepts.

2.3.3. Type Hierarchy

Sowa defines a set (T) whose elements are *type labels*. Each of these type labels represents a set whose elements (if any) are assumed to satisfy a particular type. These elements are said ~~to be~~ *instances* of the given type. A unary operator (δ), called the *denotation operator*, yields, when applied to a type label, its corresponding set of instances. A function (called *type*) maps entities into T. This set of types is augmented by a partial ordering. This ordering relation, denoted "$\leq$", is reflexive, antisymmetric and transitive. It captures the notion of sub/supertype. Let T_1 and T_2 be two type labels, $T_1 \leq T_2$ is satisfied if $\delta T_1 \subseteq \delta T_2$.

To this structure are added two binary operators ($\cup$ and $\cap$), the first one returning the least upper bound and the second, the greatest lower bound of any two elements in the set T. These two bounds are also called *minimum common supertype* and *maximum common subtype*. With the addition of the these operators, the structure becomes a *lattice*. This lattice of types is bounded above by the universal types ($\top$), of which everything is a subtype, and below by the absurd type ($\bot$), of which nothing is a member. This bounded lattice forms the type hierarchy. An additional relation, called the *conformity relation* (denoted "::"), is defined on this hierarchy. It relates individuals markers to type labels and is satisfied when the particular individual is a member of the given type.

This hierarchy framework is first set up for concepts. However, it can be extended to include actions and properties. The function *type* can be extended to map conceptual relations to type labels. The same partial ordering of the structure can also be extended to type labels of conceptual relations. However, no type labels of concepts should have any common supertype with type labels of conceptual relations (except of course for the universal type).

2.3.4. Related Work with Conceptual Graphs

Conceptual graphs have been used in various areas of artificial intelligence. However, most of their applications are oriented towards natural language processing. In this section, we present some papers that focus on parsing languages with this semantic formalism.

Sowa and Way [32] developed a semantic interpreter covering a large subset of English. They generate CGs from a parse tree. The parse trees input to their system are generated by augmented phrase structure grammar rules. Only syntactic rules are used to produce the parse trees on which the semantic component operates. The idea of the production of graphs is as follows. A canonical graph is obtained from a canonical graph lexicon and is associated with every word. These graphs are then joined to form the semantic representation of the sentence. The joining process is guided by the parse tree, determining the order in which the joins should occur. This system is responsible also for detecting anomalous sentences (i.e. syntactically well formed but non meaningful), since no semantic information is used to generate the parse trees. They have implemented this system using the Programming Language for Natural Language Processing, a language with built-in facilities for parsing and graph operations.

Sowa also emphasizes the importance of including appropriate information in the lexicon in order to achieve good parsing. In [33], he presents a series of examples illustrating the semantic patterns that have to be represented in the lexicon. This includes thematic relations as well as other relations not explicitly stated by the syntax (e.g., complex noun phrases like *sound system technician*) and the fact that the meaning of more complex sentence can equally be carried by many simpler ones (e.g. *The janitor opened the door with an old key* and *The janitor opened the door. He used an old key.*). He then goes on to outline a general procedure to derive conceptual graphs from conventional syntax directed parser, without imposing any more restriction on the grammar rules. The implementation described above is based on this approach. The nature of his approach suggests some similarities with LFGs. Although they differ in goal, they both put emphasis on the lexicon and on head categories.

Chapter 3

A Graph Representation Scheme[9]

The use of graphs to represent knowledge originates at the earliest stage of artificial intelligence and has since then been very popular [26]. Graphs have been incorporated in numerous applications, which justify the large amount of attention they have received in the literature (e.g., [24, 36, 1]). However, the implementational details of even the most frequently performed operations have often been overlooked.

In the conceptual graph formalism, a directed acyclic graph serves to represent the type hierarchy.[10] In the original CG formalism, the hierarchy is assumed to represent the links among the various concepts. As we shall see later, in our system the type hierarchy is involved in a semantic verification process. For our purposes, we need to devote some attention to the implementation of the operations performed on the hierarchy, as their efficiency directly influences that of the entire system.

In this chapter, we introduce a method to represent directed acyclic graphs in Logic Programming. The main characteristic of this representation is that it allows frequently encountered graph operations to be performed very quickly, using features of Logic Programming. We also present a compiler that generates this representation from a more natural one. Section 6.2 will discuss some other applications for this graph representation scheme.

[9]The material described in this chapter has been presented at the Third Annual Workshop on Conceptual Graphs [25].

[10]Recall from Section 2.3.3 that the type hierarchy of conceptual graphs is a set of type labels on which a partial ordering (denoted "≤ ") is defined. Hence this structure can trivially be represented by a graph.

3.1. Introductory Details

Let us first introduce formally the type of structure we are considering and provide a few additional details about the nature of the problem. The class of structures on which this method applies is that of *directed acyclic graphs* which we present next, together with some of its characteristics.

3.1.1. Definitions

A *directed graph* (DG) is a 3-tuple (N, E, Γ). N is a finite non-empty set, called the *set of nodes*, E is a set of ordered pairs, called the *set of edges* and Γ is called the *incidence mapping* and maps E into $N \times N$. If $e \in E$, then $\Gamma(e) = (n,m)$, where n is called the *initial node* of e and m, the *terminal node* of e. We impose the following two restrictions on DG's:

- For any $n,m \in N$, there is at most one $e \in E$ such that $\Gamma(e) = (n,m)$ (i.e., there are no multiple edges in our DG's).
- For any $n \in N$, there is no $e \in E$ such that $\Gamma(e) = (n,n)$ (i.e., there is no loop in our DG's).

Let DG = (N, E, Γ) be a directed graph. A *root* r of DG is a node such that $r \in N$, and $\nexists e \in E$ such that $\Gamma(e) = (m,n)$. A *path* P in DG is a sequence

$$n_1, e_1, n_2, e_2, \ldots, n_t, e_t, n_{t+1}, \quad t \geq 0, \quad n_i \in N, \quad e_i \in E \quad and \quad \Gamma(e_i) = (n_i, n_{i+1}), \quad 1 \leq i \leq t.$$

For every n_i, $1 \leq i \leq t$, on P, we say that P *passes through* n_i. The above two restrictions on the structure of directed graphs enable the representation of a path by a sequence of nodes only. A path P is called a *simple path* if no node appears more than once in it. A node m is said to be *accessible* from another node n if there is a path containing both n and m in the DG. If there is at least one path that passes through every node of a DG then it is said to be *connected*. A DG is *acyclic* if all the paths it contains are simple. In what follows, we use the expression *directed acyclic graph* (DAG) to refer to a connected, directed and acyclic graph having a single root.

3.1.2. Goal of the Representation

A very important operation on the structure defined above is the one of accessibility verification: given two nodes, determine whether there exists a path from one to the other or not. In the case where the DAG represents a type hierarchy, this accessibility operation corresponds to sub/supertype verification. A node t_1 is a subtype of another one t_2 if there is a path from t_2 to t_1 in the graph representing the hierarchy. The representation proposed here aims at optimizing this node accessibility relation, in terms of which many hierarchy operations can be expressed.

An obvious method to determine whether a node N_1 is accessible from another node N_2 is to traverse the subgraph rooted at N_2, looking for N_1. However, depending on the representation of the DAG, this approach can be costly ($O(n^2)$). More efficient solutions can be obtained by taking advantage of the characteristics of the structure itself.

3.2. Usefulness to Our System

Although the representation introduced in this chapter has been used for implementing a particular approach to grammar than the one used in this work (see Section 6.2), the main motivation that lead us to develop the graph representation scheme described in this chapter was to devise a good implementation for the type hierarchy. Such a type hierarchy is an integral part of the Conceptual Graphs formalism and we adopted the same structure for the hierarchy of our system as the one defined by Sowa and introduced in 2.3.3.

As we shall explain in more details in the following chapter, the type hierarchy plays an important role in our system. In particular, a frequently performed operation, subtype verification, can easily be expressed in terms of node accessibility. We will see how this subtype operation is part of an important semantic verification. Due to the way in which the system operates, subtype is the dominant operation in this semantic verification. Therefore, a hierarchy representation that allows fast subtype verifications between nodes constitutes a crucial problem worth investigating.

3.3. Related Work

This problem of accessibility was examined inside the framework of type hierarchy elsewhere. The two papers we now survey take insights from the structure being searched to yield an efficient solution.

3.3.1. An Integer Comparison Approach

In [29], Schubert et al. present a method for type verification that reduces to integer interval inclusions. The various types present in their system are grouped in a tree structure. Each node of the tree represents a single type and is associated two numbers. The first one is the preorder traversal number of that node and the second one is the highest preorder number of its descendants. With this numbering scheme, a node will be identified as an ancestor of another one if the interval specified by its two numbers contains that of the other node. Clearly, this allows for quick verification of sub/supertype. However, the applicability of this numbering process solely rests upon the tree nature of the hierarchy, and hence cannot accommodate the hierarchy structure of Conceptual Graphs.

3.3.2. A Unification Approach

Dahl proposed in [8] a solution for the type verification problem based mostly on unification. There is no central structure representing the entire hierarchy. Instead, a partial hierarchy of types is included in every lexical entry. The partial hierarchy for a given entity contains only the types from the root to this entity. The comparison of two such partial hierarchies is done only via Prolog's unification of the two terms. This is the only operation needed to determine whether an entity is a subtype of another one.

3.4. A New Solution

As it was observed in the second approach surveyed, the use of unification yields not only a much simpler implementation than traversal, but also a more efficient one. The execution of any Prolog program necessitates some unifications. Since unification is a built-in part of any Prolog implementation, a program requiring only a fixed number of unifications for its execution is bound to be more efficient than another that requires some additional processing in addition to unification. Therefore, using unification as for as much processing as possible is clearly a desirable aspect which we sought while designing our representation.

3.4.1. Introduction

Our solution is based on the second method surveyed above. However, there are some aspects of this approach on which we wish to improve. First having a partial hierarchy to describe the type of a particular entry implies that a lot of redundancies are stored, in comparison to having a central structure and storing only one node per entry. This means that the common supertypes of closely related entities are stored for each one of these entities. This redundancy could be acceptable in the case of a tree structure. The number of redundancies in the representation is linearly dependent on the number of nodes in the structure. However, in the case of graphs, that relation becomes quadratic. Also, every time the hierarchy needs to be modified (e.g., to include more specific types) every entry has to be examined, since its partial hierarchy might be affected by the change. Furthermore, this approach is also limited to type relationships structured as trees. To befit the hierarchy structure of CG, we must allow more general relationship structures (i.e., graphs).

In Dahl's system, unification is used to compared nodes for accessibility; but in reality, the unification process takes place on paths from the root to particular nodes. For the reasons just mentioned, we do not want to associate a path to every node explicitly. The idea behind our approach to the accessibility problem is to have a representation allowing fast access to paths and then use unification on such paths for comparisons. This representation is introduced next.

3.4.2. Description of the New Representation

In order to minimize traversal during node comparisons, we digress from traditional graph representation and associate a node with an incomplete path from the root to that node as its representation. To simplify the discussion, we will always refer to the partial DAG of figure 3-1 in our examples.

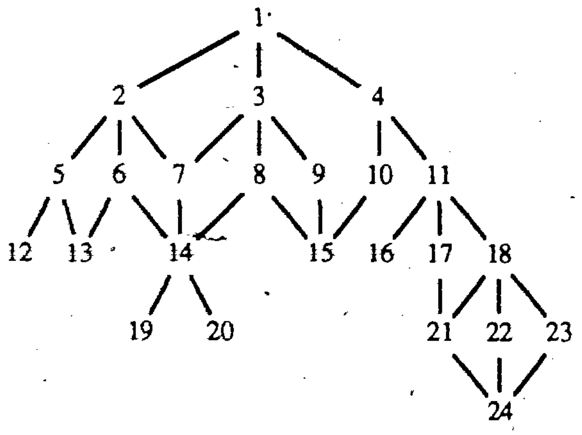

Figure 3-1: Example of a graph structure. The type hierarchy used in CG has the same format.

In general, graphs such as the one in figure 3-1 can contain paths between two nodes that are deterministic (i.e., unique). For instance, the path from **22** to **4** is deterministic; as opposed to, say, the one from **15** to **1**. Accordingly, we introduce the notion of *node determinism* to ease the discussion. We say that a node is *deterministic* if the path from the root to it is unique. Otherwise, the node is said to be *non-deterministic*. In figure 3-1, **22** is an example of a deterministic node, whereas **15** exemplifies the second kind of node.

The path(s) are associated to each node by the binary predicate **path(Node,Path)**. Let us consider the deterministic nodes first. The path from the root to such nodes is represented in our system by a difference list[11]. For example, here are some actual representations for nodes of this nature taken from figure 3-1.

[11] We will give more details on this data structure shortly

```
    path(2,[1,2|A]\A).
    path(16,[1,4,11,16|A]\A).
    path(22,[1,4,11,18,22|A]\A).
```

In this case, the **path** relations are stored as facts in a Prolog database.

The representation of nodes of the second kind is more complex. Because of their non-deterministic nature, storing their various paths explicitly can lead to combinatorial explosion. Hence, we must resort to some computation in order to represent the paths of such nodes.

The path representations will again be expressed in terms of the binary relation **path**. However, in this case, the predicate will be stored as a Prolog rule, as opposed to a fact. The part of the path from a node to its first non-deterministic ancestor is kept explicitly. Then a variable is introduced to represent the non-deterministic part. The role of the rule's body is to instantiate this variable to various paths through backtracking. For instance, node **20** in figure 3-1 would be represented as:

```
    path(20,A\B) :-
        path_to_parent(14,C),
        addend(C,14,A\[20|B]).
    path_to_parent(14,[1,2,6|A]\A).
    path_to_parent(14,A\B) :-
        path_to_parent(7,C),
        addend(C,7,A\B).
    path_to_parent(14,[1,3,8|A]\A).
```

The **path** rule is to be interpreted as follows. The path from the root to **20** is: **A** to which **20** is added at the end if the path from the root to a parent of **14** (which is the first non-deterministic ancestor of **20**) is **C** and **A** is obtained from **C** by adding **14** at the end. The binary predicate **path_to_parent** returns a path (as second argument) from the root to a parent of the node passed as its first argument. Such paths to parents are represented in the same way, that is, explicitly listed if unique, and by a recursive call to **path_to_parent** otherwise. The predicate **addend** returns as third argument a difference list formed by inserting the second argument at the end of the difference list passed as first argument.

During execution, a call such as

```
    ?- path(20,X).
```

would instantiate **X** to all possible paths to the node **20** successively, through Prolog's backtracking mechanism:

```
X = [1,2,6,14,20|Z]\Z ;
X = [1,2,7,14,20|Z]\Z ;
X = [1,3,7,14,20|Z]\Z ;
X = [1,3,8,14,20|Z]\Z ;
```

However, all the processing to find a particular value for **X** is done entirely by Prolog's unification.

3.4.3. Use of Difference Lists

An important data structure used in this algorithm is a difference list. This structure provides a good alternative to process sequences of elements in a simpler and more efficient way than regular lists. The basic concept is to use the difference of two lists in order to represent a certain list. For example, the list [a,b,c] can be represented as the difference of the lists L1 = [a,b,c,d,e,f] and L2 = [d,e,f]. L1 is called the *head* of the difference list and L2 the *tail*. We use the backslach character ("\") for the difference operator between the head and the tail. The improvements over regular list operations are achieved by a clever use of variables. A more complete presentation of difference lists appears in [34]. The following two procedures illustrate very well the kind of efficiency gained by the use of difference lists, compared to the equivalent procedures for regular lists.

```
addend_dl(H\[Elt|T],Elt,H\T).

append_dl(H1\T1,T1\T2,H1\T2).
```

The first predicate takes in a difference list (as first argument) and an element (as second argument) and returns the same difference list with that element inserted at its end. The other predicate returns as third argument the concatenation of the difference lists of its first two arguments. Its successful completion is conditional upon the possibility of unifying the tail of the first list with the head of the second one. The interesting feature about these two procedures is that everything is done through unification, which makes these two operations executable in constant time[12]. The equivalent predicates for regular lists have linear complexity in the size of the input list. These two procedures play a key role in the efficiency of our system.

This particular data structure was used to take care of the following situation. The only restriction to use unification to compare two paths is that they be both represented from the root to the nodes

[12]A formal account of efficiency is given in Section 3.4.6.

(top-down), rather than from the nodes to the root (bottom-up). But nodes have to be visited from bottom up during path construction in order to take advantage of possible tree-like structures. Intuitively, regular lists (or any other Prolog operators) could be used since they allow fast insertion in the first position, which is very suitable to the direction of the traversal. A problem originates with the variables that are possibly introduced for a path. Those variables get instantiated to a sub-path containing one or more nodes. If such a variable is inserted as the first element of a list, then it can be instantiated to only one term. For example, during the processing of the node **14** in figure 3-1 we would have the list **[X,14]** representing its path. Due to the way in which Prolog handles operators, the variable **X** can not be instantiated to more than one node as would be necessary, unless sublists are used[13]. To avoid this problem, we delay the addition of a node to a path until this path is fully instantiated. If paths are represented by a difference list, then insertions can be done at the end of the lists just as fast as to the beginning (i.e., the list does not have to be traversed), using the predicate **addend_dl**. This strategy is best illustrated with an example. Consider the actual predicate generated for the node **20**:

```
path(20,A\B) :-
    path_to_parent(14,C),
    addend(C,14,A\[20|B]).
```

First a path from the root to **14**, the first non-deterministic ancestor of **20**, is obtained. Then **14** is added at the end of that path to produce a new path itself terminated by **20**. It might be useful to reexamine the predicate **addend_dl** displayed above in order to fully understand this process.

Hence, during consultation of the new structure to find a path, the only processing done is in the case of nodes with multiple parents. In such cases, some kind of processing (recursion or iteration) is unavoidable. All the other operations, including the ones for the deterministic case, are handled by unification alone.

[13]This is not a problem inherent to lists, any defined operators would behave in the same way.

3.4.4. Preprocessing

In order to prevent potential users from being discouraged by the complexity of this representation, we developed a short compiler to generate the new graph representation automatically from a more intuitive one. Not only is such a compiler useful for the first generation of the representation, but it becomes extremely convenient when the hierarchy needs to be modified (to include more specific types for example). In such a case, one would normally have to examine the representation of all the nodes of the hierarchy, as their representation could be affected by such changes. Instead, the compiler can reprocess the new input representation, which can easily be updated.

The compiler assumes that the hierarchy to be processed is input by a binary predicate that describes for each node (first argument) the list of its parent(s) (second argument). These binary predicates are assumed to be stored as facts in a Prolog database. For example, a valid input representation for the graph of figure 3-1 could be:[14]

```
parent(2,[1]).                        parent(3,[1]).
parent(4,[1]).                        parent(5,[2]).
parent(6,[2]).                        parent(7,[2,3]).
parent(8,[3]).                        parent(9,[3]).
parent(10,[4]).                       parent(11,[4]).
parent(12,[5]).                       parent(13,[5,6]).
parent(14,[6,7,8]).                   parent(15,[8,9,10]).
parent(16,[11]).                      parent(17,[11]).
parent(18,[11]).                      parent(19,[14]).
parent(20,[14]).                      parent(21,[17,18]).
parent(22,[18]).                      parent(23,[18]).
parent(24,[21,22,23]).
```

This representation is reasonably simple, although perhaps not the most intuitive one, and also very concise: there is only one fact for each node and each edge is represented only once. Since the representation of non-deterministic nodes involves representations of its ancestors, the compiler requires that all the input predicates describing a node's ancestors appear before that node's predicate in the database.[15] The compiler is shown in figure 3-2.

[14]The name of the predicate used can be different, as it has to be specified by the user.

[15]A sorting process could be added to replace this restriction, should it be considered too severe in some particular case.

```
transform(UserPred) :-
      assert( path(1,[1|X]\X) ),
      Goal =.. [UserPred,Node,Ps],
      call(Goal),
      genrules(Node,Ps),
      fail.
transform(_).

genrules(A,[B]) :-
      clause(path(B,C),D),
      addend(C,A,E),
      assert( ( path(A,E) :- D ) ),
      !.
genrules(A,B) :-
      assert( ( path(A,C) :- path_to_parent(A,D), addend(D,A,C) ) ),
      lookparent(A,B).

lookparent(A,[]).
lookparent(A,[B|C]) :-
      clause(path(B,D\E),F),
      assert((path_to_parent(A,D\E):-F)),
      lookparent(A,C).
```

Figure 3-2: Compiler generating the new graph representation.

In the procedure **transform**, **UserPred** is the name of the predicate that represents the input graph (for example, **parent** in the above example). The first subgoal in the procedure simply initializes the system by specifying that the path from the root of the lattice is simply composed of that node. Then a fact **UserPred(Node,Ps)** is constructed and called to instantiate its arguments. The resolution mechanism of Prolog will instantiate this call with the first **UserPred** predicate in the data base. The call to **genrules** will generate all the facts and rules for that node in the new representation. It is divided into two disjoint possibilities. If **Node** only has a single parent then it is added to its parent's path and this newly formed path becomes the path of **Node** and the body of the parent's rule also becomes the body of the rule for **Node** to instantiate any variable that were originally on the parent's path. Otherwise, if the node has more than one parent, then a single variable is inserted for its path and a predicate (**path_to_parent**) to instantiate that variable is asserted. There are as many clauses asserted for such nodes as their number of parents. The generation of these clauses is done as follows. For each parent, add the node to the path of that parent and use the same body as in the parent's rule in the new one. The same process takes place for every fact describing the input graph.

3.4.5. Comparison Mechanism

The above described representation was designed to allow more flexibility in both the type of structure being processed and its input representation, while preserving the possibility of using unification to compare nodes for accessibility. We now examine the details of this operation in the framework of the subtype relation in type hierarchies.

Suppose we wish to determine whether T_1 is a subtype of T_2. Their respective paths are obtained by the calls **path(T_1,$Path_1$)** and **path(T_2,$Path_2$)**. For simplicity, let's examine only a successful case. The case in which it fails will be clear from this discussion. The subtype operation can be reduced to the one of determining whether $Path_2$ is the beginning of $Path_1$. $Path_1$ and $Path_2$ being two difference lists, the task can be thought of as trying to find a third (difference) list which appended to $Path_2$ would produce $Path_1$. The concatenation of two difference lists, done in constant time and only via unification, produces the desired result. The procedure for subtype is as follows:

```
subtype(T1,T2) :-
    path(T1,Path1),
    path(T2,Path2),
    prefix(Path2,Path1).

prefix(Path1,Path2) :- append1_dl(Path2,_,Path1).

append1_dl(Xs\Ys,Ys\Zs,Xs\Zs) :- var(Zs).
```

The call to builtin predicate **var** in **append_dl** is necessary in order to distinguish subtype from supertype, a distinction lost since all the arguments of **append1_dl** can either be instantiated when the predicate is called or not. The predicate to verify supertype would be exactly the same as the above one for subtype with the exception that the call to **var** would be replaced by one to **nonvar**. If only compatibility (i.e., either subtype or supertype) was desired, then no subgoals would be needed in **append_dl**.

3.4.6. Analysis

Before analysing the results obtained with the above representation, it is important to indicate the basis on which we judge efficiency. All our programming was developed in a logic programming framework, which differs substantially from any conventional languages. The nature of logic programs calls for a slightly different method of complexity measures, as opposed to the usual specification of complexity as a function of the size of the input. A good way to evaluate the complexity of a logic program is to examine the number of goals that need to be resolved in order to satisfy a given predicate [34].

Any successful computation of a logic program can be described by a proof tree. The nodes of such a tree correspond to the goals of the computations and the arcs show how they are invoked. The root of the tree is the initial goal of the computation. With such a representation, Shapiro [28] defines three complexity measures over logic program. In a proof **R**, the *length* of **R** is the number of nodes in the proof tree of **R**, the *depth* of **R** is the depth of the proof tree, and the *goal-size* is the maximum size of any node (goal) in the tree, the size of a goal being the number of symbols in its textual representation. The following measures are then introduced. Let **P** be a logic program,

- **P** is of *goal-size* complexity $G(n)$ if for any goal **A** in the meaning of **P** of size **n**, there is a proof of **A** from **P** of goal-size $\leq G(n)$.
- **P** is of *depth* complexity $D(n)$ if for any goal **A** in the meaning of **P** of size **n**, there is a proof of **A** from **P** of depth $\leq D(n)$.
- **P** is of *length* complexity $L(n)$ if for any goal **A** in the meaning of **P** of size **n**, there is a proof of **A** from **P** of length $\leq L(n)$.

With the above definitions, Shapiro shows that for any logic program **P** of depth complexity D(n), goal-size complexity G(n) and length complexity L(n), there exits an alternating Turing machine[16] (M) and a constant c uniform in **P** such that the set of strings accepted by M corresponds to the meaning of **P** and M operates in time $c{\cdot}D(n){\cdot}G(n)$.

We are aware that the results possibly obtained with this complexity treatment cannot be directly compared with ones obtained with the conventional treatment, due to major differences between both frameworks. Such a consideration would have to be taken into account if comparisons with implementations in other programming language were to be drawn. However, it is not the purpose of this work to compare implementations. We were merely interested in finding efficient solutions using the specific computational power offered by Prolog, since this language proved to suit our needs the best.

[16]An *alternating Turing machine* (ATM) is a generalized nondeterministic Turing machine (NDTM) distinguishing two types of states: existential and universal. In a existential state, an ATM behaves similarly as a NDTM, accepting an input string if and only if at least one of its next move lead to acceptance. In an universal state an ATM accepts a string if and only if all of its next moves lead to acceptance.

Using Shapiro's result, one can easily show that a goal that does not have any body (i.e., a fact) can be executed in constant time.[17] Hence the unification of the argument(s) of a given goal is bounded by a constant.[18] Clearly then, the time spent to compare two paths is constant since every operation is done through unification and a call to the Prolog built-in predicate **var**. The predicate **path** also returns deterministic paths in constant time because of similar considerations. Recursion happens only in cases of more than one path to the compared node(s). The non-determinism of such cases implies additional processing for any sequential algorithm.

For space considerations, our interest focuses on the number of rules that needs to be generated. Let n and e be respectively the number of nodes and edges in the graph being processed. As we saw, there are n predicates **path** generated. In addition, every node having more than one parent generates a **path_to_parent** predicate for each one of its parent. Let $p(n)$ represent the number of parent of node n. Hence there are $p(n)$ edges arriving at node n in the graph. We can define a function (say $r(n)$) mapping nodes to integers representing the number of rules they originate in the new representation as follows:

$$r(n) = \begin{cases} 1 & \text{if} \quad p(n)=1 \\ p+1 & \text{if} \quad p(n)=p, \; p>1 \end{cases}$$

Therefore, it is easy to see that the number of rules produced for the new representation is a linear function of the number of edges in the graph.

3.5. Summary

We presented a new method to represent DAG which allows quick node accessibility comparison. It can handle a wider range of structures than both systems surveyed. The possibility of quickly identifying tree structure and taking advantage or their determinism is allowed by specifying the input structure from bottom up. Furthermore, this way of representing the input does not affect the efficiency of the processing of multi-parent nodes. In such cases, no representation can avoid non-

[17]When we talk about "constant time", we refer to the number of resolution steps during the execution.

[18]This result holds only if the implementation of unification used does not include occur check, which is the case for most Prolog implementations.

determinism. We used this methodology to represent our type hierarchy, which is not restricted to a tree. However, in cases of tree structures, the efficiency of this representation is similar to the one of [29] and it becomes equivalent to the one described in [8]. This has been studied in depth in [11] with respect to parse tree representation and processing. The additional preprocessing, which is not necessary in this latter system, is easily compensated by the avoidance of the mentioned update problem. When the graph is modified, all that needs to be done is to run the program to generate the new representation on the updated input; nothing else is affected.

Chapter 4
Components of the System

There is a certain amount of information that is required about the various entities manipulated by both formalisms used in our system. Access to such information is required at various stages in our system. In order to represent this information efficiently, two major components are used: a *type hierarchy* and a *lexicon*. This fairly straightforward approach emphasizes the distinction between information about the entities and information about their relationship with respect to a common aspect of their associated information (type).

This chapter focuses on the description of these two components and their respective implementation. Both formalisms used in the system specify quite clearly the nature of the information they required and that constitutes the basis of the contents of the components of the present system. The design of these components has been guided by two factors: conformity with respect to the original formalism and efficiency of the operations accessing the information of the components.

4.1. Lexicon

The first major component of our system is the lexicon. Lexicons have always constituted an important component of natural language understanding systems, as they embrace information about the syntactic characteristics of the different words known to the system. It is becoming more and more common practice to also include semantic details about each entry. In our system, the lexicon establishes a link between the two formalisms (f-structure and Conceptual Graphs). Both of these formalisms require some specific information about words and concepts respectively, which can be kept in the lexicon.

Most of the information needed by one formalism complements that of the other as one is syntax oriented and the other is directed towards semantics. With regards to some aspects, there are similarities between the information used by each one. Our representation attempts to capture in a single component the details corresponding to both formalisms. Such a unification leads to the simplification of the entire system, as all the information about the concepts and words handled by the system are grouped in a unique location and accessed uniformly. The possibility of structuring the lexicon in such a way as to minimize the storing of redundancies without jeopardizing the efficiency of the mechanism that access such information constitute an important characteristics of our system.

Before examining the structure of our lexicon, we describe briefly the various details needed by each formalism and which have to be included in the system.

4.1.1. Lexicon and LFG

In LFG theory, the lexicon plays a very important role. As we mentioned before, the lexicon greatly contributes to the syntactic component, which is thus simplified. This is realized not by having overly sophisticated entries, but rather by having relatively simple entries and a different one for each use of a word. Hence, the size of an LFG lexicon tends to be much larger than that of other linguistic theories.

Basically, lexical entries in LFG contain the syntactic category of the word (e.g., noun, verb, adjective, etc.) together with some syntactic features such as, for instance, number, gender and case for nouns and adjectives, mood, tense, person, number for verbs, etc. They also include a semantic predicate which is of no use to the syntactic component; but is assumed to be treated by the semantic interpreter.

The basic idea behind the LFG approach is to take advantage of the information conveyed by the morphological structure of each word. This information will interact with the grammar rules through the annotations associated with each rule. Various rules will be selected or restricted according to the information originating from the lexicon. For instance, in the case of passive verbs, the difference in the construction would be attributed to a different lexical use of the verb, mapping different parts of the sentence to the semantic argument of the predicate. To illustrate this, consider the following two predicates that would normally explain the active and passive (respectively) uses of the verb *to scare*[19]:

scare: V, (↑pred) = 'scare<(↑SUBJ)(↑OBJ)>'
scared: V, (↑pred) = 'scare<(↑BY OBJ)(↑SUBJ)>'[20]

It is also possible to use lexical redundancy rules to reduce the number of entries of the lexicon not only in the case of the passive, but for other constructions as well [3].[21] However, whether such rules are used or not, the effect is the same: the syntactic component makes full use of the information provided by the morphology of each word of the sentence.

4.1.2. Lexicon and CG

The inclusion of semantic information in lexicons has abolished the exclusivity of their interaction with syntactic components only. Sowa suggests that the collection of canonical graphs known to a Conceptual Graph system be kept in a *conceptual catalog* [31]. This conceptual catalog contains a single entry for each concept which includes the word, its grammatical category and its semantic type. The formalism also includes a *canonical basis* which is a complete collection of canonical graphs that can be manipulated using the four formation rules (*join, copy, restrict, simplify*) in order to derive larger graphs. A canonical graph for a concept *c* specifies the type of concept(s) which can be linked to *c*. The relations used are also specified by a canonical graph. The nature of this canonical basis shows some similarities with that of lexicons. Although not explicitly states, it is easily conceivable that the conceptual catalog and the canonical basis be merged into a single unit (lexicon).

[19]We refer the reader to [2] for a more complete description of passive constructions in LFG.

[20]In the case of verbs that can subcategorize for a non-fixed number of argument(s) (e.g., to read), similar entries must be included for each use.

[21]The scope of such rules must be examined carefully as they should not apply to every possible entry (e.g., some verb do not passivize).

4.1.3. The Lexicon of Our System

The main objective of our lexicon is to group in the same location all the information required by both formalisms. In doing so, we tried to avoid imposing restrictions on either formalisms. In particular, the independence of LFG to any semantic formalism has been preserved. Although the semantic information included in each lexical entry happens to be very appropriate to CGs, we feel that it is general enough to suit other formalism just as well. Before examining the format of the lexical entries, we need to introduce an important kind of information kept in the lexicon: θ-role.

4.1.3.1. θ-Roles

F-structures express the relationships between various parts of a sentence mainly in terms of grammatical functions. However, in order to derive a semantic representation for a sentence, grammatical functions do not suffice, as more information is needed. A good illustration of this fact is obtained in examining the difference between corresponding active and passive sentences, such as:

John helps Paul.

Paul is helped by John.

These two sentences roughly provide the same information and therefore should be mapped to similar semantic representations. This implies that the relationships between *Paul* and *John* should be the same in the representation associated with either sentence. In order to capture that property, more information that what is included in the f-structures is necessary. The f-structure corresponding to these two sentences are quite different. In particular, *John* and *Paul* will not realize the same grammatical function in both f-structures. Such semantic similarities between syntactically different sentences are well explained by thematic roles.

Thematic roles (θ-roles) have now become a reasonably well accepted framework in linguistics as they have been included in many syntactic theories. They are intended to account for some semantic relations[22] between words of a sentence. Very informally, some parts of a sentence play an important role in its semantic structure. Such roles include the the theme, the agent and the

[22]Primarily and originally for the notion of *theme*, hence the name.

object of the action, as well as destination, goal, etc. We refer to these roles as *thematic roles*. Intuitively, some words presuppose or imply such roles. Consider for instance the verb *to drink*. Whenever this verb is used it automatically suggests that someone is doing the action of drinking (the agent θ-role) and that something is being drank (the object θ-role). These roles might not be realized in a sentence, but they are nevertheless understood. Such words are said to *assign* θ-roles. In contrast, words like *wine*, *cat*, *good* do not assign any θ-roles as they don't always occur in nor imply the same context. However, they can be assigned a θ-role by another word. Verbs and prepositions assign θ-roles most frequently. The reader is referred to [21] for a more formal and complete discussion of θ-roles.

The link between θ-roles and grammatical functions if done in the lexicon. We associate a θ-role to each grammatical function subcategorized for by a predicate. This approach is not new to the LFG formalism [2]. The various θ-roles will eventually be used as relation names between the concept corresponding to subcategorizing entities and their argument(s) in the graph, restriction such relations to be θ-roles.

Canonical graphs also include the nature of the relations linking two concepts. However, no formal account is given to identify the set of concepts that originates relations. The restriction of the use of θ-roles as relations between entities subcategorized and the entity that govern them is not incompatible with CG theory. A way of identifying such relations is essential to the formalism, and θ-roles constitute an adequate approach. They also provide a more solid theoretical basis to our system, and do not cause any loss of information in comparison with canonical graphs. In fact, canonical graphs could easily be built from the lexicon if they were needed in an augmented CG application.

4.1.3.2. General Format of Lexical Entries

The format used for our lexical entries allows a uniform consultation mechanism for the entire lexicon. Each lexical entry contains four fields, some of which might left unspecified. The first one is the token used in the f-structure formalism to refer to this entity. That is either a predicate name without its subcategorized argument appearing in the f-structure or the value of one of the attributes QUAL, DET, OP, DEG. Just like predicates, the value of these attributes corresponds (possibly indirectly) to a word of the sentence. The second one is the corresponding token in the conceptual graphs formalism, which may or may not be the same as the first field for some or all of the entries, depending on the application.[23] The two fields are independent and establish a mapping between f-structure entities and CG concepts.

The third field is the subcategorization information possibly prescribed by the lexical item. The actual representation for the arguments subcategorized for by a predicate is a list of pairs (being operands of the functor[24] "::"). Each pair contains the name of the grammatical function subcategorized for and the thematic role that the subcategorized argument is assumed to play in the sentence.

Finally, the last field of a lexical entry is also a list whose elements are understood to be in a one to one correspondence with the ones of the previous list. They specify a semantic type that must be respected by the predicate fulfilling the grammatical function prescribed by the corresponding element of the other list. The elements of this last list are taken from the type hierarchy. In order for a given argument to play a specific θ-role, the concept corresponding to that argument must satisfy a consistency condition. This is useful to detect syntactically correct but semantic ill sentences. We shall explain this process shortly.

All the entries are divided into disjoint classes. The four fields of a given entry are stored as arguments of a predicate named according to the grammatical category of the lexical entry. The choice of grammatical categories to distinguish the entries is arbitrarily, that is no advantage is taken from this fact by the access mechanism. The following examples of actual entries will hopefully clarify the representation of lexical entries.

```
adj(red,red,_,_).
adv(very,very,_,_).
det(a,unspec,_,_).
noun(cat,cat::_,_,_).
verb(drink,drink,[subj::agent,obj::theme],[animal,beverage]).
```

[23]The token of conceptual graphs are assumed to be language independent, so they do not have to correspond to the ones of any linguistically or otherwise oriented formalism.

[24]The term *functor* is used here in its standard Prolog meaning, that is the name or a relation having a fixed number or argument(s).

The first three entries exemplify entities that do not subcategorize. The last two fields of these entries are included, although left unspecified, to allow a uniform consultation mechanism for all the entries. In the case of determiners, the corresponding graph representation will be a referent appearing as a suffix to the appropriate concept. Therefore, the second field of the entries corresponding to determiners (e.g., the third entry above) should be interpreted as referents rather than concepts. A slot is included in graph tokens of concepts that generally appear with a referent (e.g., nouns), as shown in the forth entry above. This slot is separated with the actual token by the user defined functor "::". Such concepts always appear with some referent. Hence, this encoding strategy allows to associate a referent to a given concept using unification alone, which is an efficient and simple way of manipulating terms.

For simplicity, we assume that no lexical redundancy rules are used; therefore, if a word subcategorizes for more than one combination of arguments, an additional entry is required for each set as exemplified in the following:

```
    verb(give,give,[subj::agent,obj2::theme,obj::recipient],[human,entity,animal]).
    verb(give,give,[subj::agent,obj::theme,to-obj::recipient],[human,entity,animal]).
```

The first one would normally correspond to the di-transitive use of *to give* as in

John gave the cat some milk.

The second one corresponds to the use of *to give* with a prepositional phrase as in

John gave some milk to the cat.

Notice how the same semantic information is included in both entries, which correctly ensure the semantic equivalence between the two syntactically different uses of the verb.

4.1.3.3. Consultation Mechanism

Let us now introduce the manner in which the lexicon information is accessed. The approach used in our lexicon consultation is inspired from the well established technique of *hashing*.

The lexicon is consulted in order to retrieve the associated information of a particular f-structure element. At that point, the only information known about this element is the attribute of which it is the value in the f-structure. In particular, the grammatical category of the corresponding word is not included in the f-structure. Therefore, we must resort to an alternative way of directing the search.

We saw that the different lexical entries were divided into disjoint groups. The criteria by which this division was realized was the attribute under which a given entry was most likely be accessed from. In our system, we used grammatical categories to identify each group, but in theory, any other labeling would produce the same results. Then to each f-structure attribute which requires lexicon consultation is associated a collection of group names, listed in decreasing order of likeliness of finding a particular entry used inside this attribute under a particular group name in the lexicon. The search of the lexicon proceeds heuristically according to the order of the listing. The heuristic rules given from the lists are only useful to improve the efficiency of the search. Such lists are obtained from a binary predicate (**orderlist**) from which the following clauses are taken for sake of exemplification.

```
orderlist(fstr,[verb]).
orderlist(subj,[noun,pronoun,verb]).
orderlist(vcomp,[verb]).
orderlist(mod,[adj,adv]).
```

All the lexicon consultation is done through the predicate **lex** whose first four arguments correspond to the ones of the lexical entries and the last one correspond to the part of the f-structure that is being processed (e.g., SUBJ, OBJ, etc.)

```
lex(Ftoken,Gtoken,TR,Types,Part) :-
     findorder(Part,OrderL),
     searchlex(OrderL,Ftoken,Gtoken,TR,Types).

findorder(subj,[noun,pronoun,verb]).
findorder(obj,[noun,pronoun,verb]).
findorder(by_obj,[noun,pronoun,verb]).
findorder(of_obj,[noun,pronoun,verb]).
findorder(fstr,[verb]).
findorder(vcomp,[verb]).
findorder(det,[det]).
findorder(mod,[adj,adv]).

searchlex([Cat|_],Ftoken,Gtoken,TR,Types) :-
     Goal =.. [Cat,Ftoken,Gtoken,TR,Types],
     call(Goal), !.
searchlex([_|Rest],Ftoken,Gtoken,TR,Types) :-
     searchlex(Rest,Ftoken,Gtoken,TR,Types).
```

We are aware that this way of directing the search through the lexicon is dependent upon the language being processed and that in general the ordering cannot be axiomatized rigorously.

However, only limited linguistic knowledge is necessary in order to provide a good heuristic account for this task. The indexing facility of Prolog is highly responsible for the efficiency of this consultation mechanism. Further advantages can be gained with versions of Prolog that index with the first non variable argument in addition to the functor name.

4.2. Type Hierarchy

The second major component of the system is the type hierarchy. Such a structure has always been a faithful companion to graphical meaning representations, and Conceptual Graphs do not constitute an exception. Sowa defines and details carefully a hierarchy structure, together with some operations for the CG formalism [31], which we include in its entirety in our system. The main characteristics of this structure and some of its related operations have been presented in Section 2.3.3. In Section 3.4 we presented some arguments justifying the use of a independent component for the implementation of this structure.

4.2.1. Role of the Hierarchy

As was mentioned briefly before, a type hierarchy depicts all the relations that exist between the various concepts. In our system, it also carries out another important task. The type hierarchy provides a basis in order to reject syntactically well formed but semantically invalid sentences. Such sentences can very well be assigned an f-structure since they do not violate any rule of syntax. In particular, they respect the three conditions governing the well-formedness of f-structures. It is therefore a task of the semantic interpreter to block the analysis of such sentences.

Consider for example the erroneous sentence

The tree ate big dreams.

The f-structure predicate for *ate* presumably subcategorizes for two arguments: SUBJECT and OBJECT, to which should be associated the θ-roles AGENT and THEME respectively. In order to detect such anomalies, we associate a type that the concept fulfilling each argument must respect (fourth field of lexical entries). Careful specification of such type constraints is the key to this problem. For instance, the above sentence would be ruled out as semantically incorrect if we restrict the type of the concept corresponding to the AGENT θ-role to be a subtype of the type animal. Hence, the type hierarchy provides all the information needed for the enforcement of such semantic constraints during the generation of conceptual graphs.

This particular approach to semantic verification might be considered a little too severe, as it allows a unique interpretation for sentences according to a fixed framework. For the specific purpose of machine translation, it could be argued that such semantic verifications are unnecessary: a translation system could simply attempt to produce an equivalent sentence in a different language regardless of whether the original sentence is meaningful or not. However, we chose to include such semantic verification in our system because we want to keep the possibility of using our system as part of an analysis system independent from any possible enclosing application. In such analysis systems, the detection of semantically erroneous sentences is a very important task; one that cannot be excluded.

However, it might be desirable to loosen the fixed interpretation currently done by the system and allow the generation of a representation for sentences made about a different domain of reference than the simpler one (e.g., metaphoric sentences). The system does not presently handle such sentences. The approach used for semantic verification could be adapted to handle such sentences. For example, the same strategy as the one outlined above could be used on a different hierarchy, possibly with information coming from different lexical entries, to re-examine sentences ruled out as being semantically wrong by the existing verification mechanism to see if they could not be interpreted using this different scheme. This extension would not conflict at all with what is currently implemented.

4.2.2. Implementation

The graph representation scheme presented in the previous chapter was primarily designed to implement efficiently the type hierarchy of CGs. It is easy to verify that this hierarchy structure respects the definition of a directed acyclic graph stated in Section 3.1.1.

The most frequent operation on this hierarchy and therefore the one whose efficiency is of most concern is that of subtype verification. We saw in the previous chapter how such verifications can equivalently be expressed in terms of nodes accessibility in the graph. Hence, the representation introduced in the previous chapter is ideal for implementing the type hierarchy with respect to this operation.

We assumed that all the concepts known to the system ard structured appropriately by the type hierarchy. This collection of types could be input to the pre-processor generating the new graph representation using a two-place predicate as explained in 3.4.4. All the subtype verifications are performed exclusively on the new representation using the predicate **subtype** listed in 3.4.5.

4.3. Connection Between Both Components

In Sowa's formalism, the very important connection between the lexicon and the type hierarchy is easily established: all the type labels of a given canonical graph are assumed to correspond to elements in the type hierarchy. This applies to concepts as well as to the specification of the class of concepts that can be linked to that concept.

Our system does not make use of canonical graphs. However, that precious connection is preserved. Relations with the type hierarchy appear at two levels. Firstly, the conceptual graph tokens (second field of the lexical entries) are assumed to correspond to type labels. Hence they represent the most specific type that includes this entity. Secondly, the types specified by the fourth field and associated with the subcategorized arguments also correspond to type labels. They are assumed to be the most general type that an entity needs to respect in order to appear (in a meaningful way) as argument to this entity. Since the types of the entities satisfying the various θ-roles assigned by a lexical entry need not be the same, it is important to keep a mapping of what role calls for what type. This justifies the assumption of the one to one correspondence between the elements of the last ~~two~~ lists of the lexical entries.

Our method includes the same information as that conveyed by the canonical graphs. We chose a different medium to represent this information and hence digress slightly from the CG philosophy of building larger graphs strictly from smaller ones. In order to suit both formalisms, such a diversion was unavoidable. However, we feel that in doing so we don't loose any generality, as the graphs are produced using the same information, but represented and accessed in a different manner.

Chapter 5
Algorithms and Operations

The previous two chapters have presented the components containing necessary information to the system, together with their representation and all the operations that is needed during the generation of a graph. We are now in a position to describe the operations taking place in the production of a conceptual graph from an f-structure. By consideration to the reader, we will try to stay away from overly technical details, as we intend to keep this discussion at a higher level. (However, due to the nature of the material presented and the subjectivity of this aim, no guarantee for its fulfillment can be made!) The most relevant parts of the implementation of the process appears in Appendix A. Some sample graphs produced by the system are shown in Appendix B. In addition to these two appendices, the reader might find useful to consult some of the f-structure representations for various parts of the speech that are displayed in Chapter 2.

5.1. General Strategy

The generation of a conceptual graph from an f-structure naturally breaks down into three parts:

- extract some information from the f-structure,
- perform some operations in order to derive appropriate CG representation (including some verification of constraints),
- perform the necessary operations to include the derived information into the graph.

In order to achieve these steps, a single unit (predicate) identifies particular f-structure parts and takes the appropriate action depending upon the nature of those parts. These include lexicon consultation, some semantic verifications, and operations on a "working" data structure. Once the graph information is determined, it is inserted as concepts and relations.

This unit treats one single attribute-value pair at a time. It takes in an attribute-value pair, a (partial) CG and the external data structure. It returns an updated CG and a modified instance of the external data structure. These updates are done according to the information obtained from processing the pair. The general format of the predicate implementing this unit is:

translate(AttrVal,GraphIn,GraphOut,IntDataIn,IntDataOut)

The control flow is determined by another unit which isolates a single attribute-value pair and calls the first unit to process that pair. Basically, the pairs are processed in the order in which they appear and the system is, in general, independent of any particular order. If the value of a pair is another f-structure, then the processing of that f-(sub)structure takes precedence over the original one. This processing scheme reflects the uniformity of the composition of f-structures. Structurally, all attribute-value pairs of an f-structure are similar, hence, no discrimination can be done on this "external" aspect. It is therefore natural to abstract the examination of f-structures in a single unit that can handle any valid f-structure.

Since a single attribute-value pair is isolated from the rest of the f-structure during the processing, details concerning neighbouring parts become temporarily opaque. The consultation of neighbouring parts is often necessary, and therefore an accessing mechanism must be established. An obvious approach is to make the entire f-structure visible at all times. Other than being conceptually simple, this approach shows limited advantages. Not only is it inefficient, but the traversal and selection of specific information of an f-structure is also complex to realize. Instead, we identify the information that will potentially be necessary to the processing of a subsequent pairs, and collect it in an external structure that is input to and updated by the unit processing attribute-value pairs.

This external structure consists of three parts, some of which may not be necessary in processing a particular attribute-value pair. This structure is represented using the user-defined operator "#" and its general format is[25]:

FstrPart#ConceptList#RequiredParts

The first part (**FstrPart**) is the name of the enclosing grammatical function attribute, which is useful in the lexicon consultation. The second part (**ConceptList**) is a list of the concepts that have

[25]Although consisting of unrelated components, this external information is represented using a single structure, rendering uniform the processing in cases where some components are not required.

already been inserted in the graph. Finally, the last item (**RequiredParts**) this structure may include is a list of details relevant to a concept that has not yet been processed. For instance, a concept corresponding to a grammatical function subcategorized for might not be known at the time the subcategorizing concept is inserted, and yet it is required in order to complete a particular relation. In this case, the external structure is useful in order to remove any restriction on the ordering of the attribute-value pairs of a particular f-structure. When a concept is about to be inserted, it is verified whether that concept had been referred to (i.e., subcategorized for) previously. If so, then the proper operations that had to be postponed take place.

We now examine the particular operations done during the processing of the major f-structure attributes.

5.2. Processing Subcategorizing Predicates

We distinguish between two cases in processing f-structure predicates that subcategorize. The distinction is based upon the nature of the subcategorized information and upon the operations required in each case, which are quite different.

5.2.1. Subcategorizing for Grammatical Functions

The first case is the one in which a predicate subcategorizes for grammatical function(s). A call to the lexicon returns the corresponding concept:

lex(Ftoken,Gtoken,ArgL,Types,Part)

This call also returns the θ-role to be associated with each argument of the f-structure predicate through **ArgL**, along with the semantic types that each argument must respect. The concept is then inserted into the graph. The subcategorization aspect of this f-structure predicate will be reflected in the graph by having its corresponding concept instigating relations. These relations are named according to the θ-roles associated with the particular arguments. The destination concept[26] for each relation will be the concept associated with the predicate of the subcategorized grammatical function.

[26] Recall that the relations of a CG are directed.

If that predicate has already been inserted, then the relation can easily be completed. The part of the external structure containing the list of predicate in the graph can be used as a quick way to get to that concept from the grammatical function (otherwise, the f-structure predicate has to be identified and the lexicon needs to be consulted). The semantic type of the concept is compared with the type prescribed by the relation for compatibility (i.e., subtype) using the mechanism described in earlier chapters. If this verification is successful, then the relation is updated to include the new concept as destination. Otherwise, the generation of the CG is aborted, as the sentence is semantically ill formed. In the event where the concept corresponding to the predicate of the subcategorized grammatical function has not yet been inserted in the graph, then the relevant information about this relation is stored in the external structure. This information includes the concept originating the relation, the name of that relation, the grammatical function whose predicate is linked by the relation and the semantic type that the concept needs to respect in order to correctly be used in this relation. When processing a predicate for a grammatical function, the external structure is always examined in order to determine whether its concept had been required.

No mention is made here to grammatical categories. Although verbs most often subcategorize for grammatical functions, this is no assumed by the way in which we process such predicates. The same processing can be done on any predicate that subcategorizes for grammatical functions, independently of the category. The fast that some nouns (e.g., *construction*) should be treated in a similar way as verbs with respect to subcategorization is due to Frey [16]. Following this, our implementation treats verbs such as *to construct* and their corresponding noun (*construction*) similarly with respect to subcategorization. That is both words will be at the origin of a concept that will instigate two relations in a CG: agent and theme. This is illustrated in the following, using the linear representation of CGs:

$$[\cdots] \leftarrow (AGENT) \leftarrow [construct] - (THEME) \rightarrow [\cdots]$$

$$[\cdots] \leftarrow (AGENT) - [construction: \cdots] - (THEME) \rightarrow [\cdots]$$

5.2.2. Subcategorizing for a Clause

The second case of subcategorizing predicate is that of a verb taking a verbal complement[27]. The representation of sentences including such verbs requires more than one graph, which will be imbricated one within another. The reason for this multi-graph representation is that a single CG is assumed to represent a single proposition. Just as more complex propositions can be built from simpler ones, more complex graphs can be composed of smaller ones. For instance, according to the CG theory, the sentence

A man is drinking wine.

should be represented by two CGs. One to capture the relations between *man* and *drink*, and between *drink* and *wine*. This graph will appear inside the other one, which will include the temporal information (in this case progressive present) which operates on the inner graph. Some overhead operations on graphs are required in this case.

Since parts subcategorized for appear at a deeper level in an f-structure than the predicate that subcategorizes for them, the subcategorized part might not be processed when the subcategorizing predicate is processed. Therefore, its corresponding graph will not be completed yet. For that reason, a marker is generated to indicate that a subgraph that the current graph will include a subgraph. At that point the generation of the current graph is suspended until the subgraph is completed. Upon completion of the generation of the subgraph, the presence of the marker will indicate that this subgraph should be included inside the original one whose generation is then resumed. Arbitrarily long chains of subgraphs can be produced using this strategy.

The f-structures corresponding to such cases usually include a pointer[28] as the same part plays more than one role (e.g., the subject of the auxiliary and of the main verb). This is the reason why we always verify whether a concept corresponding to a particular entity has already been inserted in the graph. According to CG theory, two different instances of the same concept are assumed to represent different entities unless explicitly co-referenced. The part of the external structure containing the inserted predicates is very useful in this task.

[27]Included by this are helping verbs and auxiliaries.

[28]Some co-reference scheme can also be used.

5.3. Processing Non-Subcategorizing Predicates

The case of non-subcategorizing predicates is a little simpler. F-structure predicates are again mapped to corresponding concepts by the lexicon, whose consultation is as usual guided by the enclosing f-structure attribute. The concept so obtained is then inserted in the graph. If this concept was needed to complete a relation (i.e., a predicate subcategorizing for the f-structure attribute under which it appears has been processed already) then the information applicable to this concept is extracted from the external structure, using the attribute name to index the search. The semantic type that the relation requires this concept to satisfy is isolated and the subtype verification, using the process explained earlier, is applied. Depending upon the result of this verification, either the relation will be updated in the graph with this concept, or the generation of the graph will be stopped, due to a semantic type conflict.

5.4. Processing Modifiers

Modifiers can appear at various places in an f-structure and can be arbitrarily complex (e.g., *tall, very tall, not very tall*). A modifier phrase is identified in the f-structure by the attribute MODATTR. We introduced this special attribute to capture the scope of each modifier. This was illustrated by figures 2-11 and 2-12. The value of MODATTR is assumed to modify the predicate or the modifier appearing at the same level as MODATTR.

Correct processing of modifiers requires a consultation of the lexicon, to get the corresponding concept name, and the concept modified by this modifier (which we call the *victim*), which is passed through the external information structure. The operations include the generation of a concept in the graph together with the insertion of a relation (named according to the nature of the modifier) between the modified concept and its victim. The f-structure is then examined to see if there is any modifier to the current one (e.g., *very*, *not*, etc.) in which case the same process takes place with the current modifier as a victim. Otherwise the other modifiers the original victim are processed.

5.5. Processing Relative Clauses

Relative clauses are introduced in an f-structure by the attribute REL. They will be represented in a CG by a sub-graph attach to the noun they modify by the relation *rel-op*. The generation of this sub-graph is very similar to that of a regular graph, except for one point. The value of the attribute of one of the grammatical function subcategorized for in the relative may be a pointer to the relative marker, indicating that this marker fulfill that function. In this case, the semantic type corresponding to the concept being modified by the relation has to be retrieved and used in the subtype verification for the relative's relation. In the event of a successful verification, the concept introduced is co-indexed with the one to which the relative is attached.

5.6. Processing Determiners

The processing of determiners is relatively simple. Determiners are represented in CG as referents to concepts. The particular referent corresponding to a determiner is obtained from the lexicon through the call

```
    lex(Ftoken,Gtoken,_,_,det)
```

Then this referent is added to the appropriate concept, which is always the value of the PRED attribute appearing at the same level as the determiner. This predicate is obtained from the predicate list in the external structure. The rest of the processing is handled by the following two calls, executed after the call to the lexicon.

```
    member(Part::(Pred::_),PredL),
    member(Pred::Gtoken,CL)
```

Here, **member** is the standard list membership predicate. The first call to **member** retrieves the concept corresponding to the predicate of the f-structure part being processed, from the external structure. In general, this will be the first element in the predicate list. The second call to **member** is used to instantiate the unspecified variable kept in the lexical representation of the concept to the graph representation of the determiner. Because of our lexical representation for concepts used with a referent, the association of the referent to the concept can be done through unification only.

The processing of determiners is the only case where we impose a particular order in the attribute-value pairs. For simplicity, we assume that the predicate appear before the determiner in the f-structure.

5.7. Processing Tense Information

The tense information is kept at the same level as the predicate for the verb of the proposition. It can be used for two purposes. ~~The first one is fairly straight~~forward: generate the appropriate temporal information, which is represented as an operator on a (sub)graph.

The second one is to control the semantic validity of certain adjuncts constructions. For instance, the temporal information can be used to block the generation of erroneous sentences such as

I will leave yesterday.

In this case, *yesterday* is incompatible with the future tense of the verb.

Chapter 6
Extensions and Conclusion

6.1. Summary of the Work Done

The objective of this thesis was to investigate the possibility of producing Conceptual Graphs representing the meaning of sentences. We have achieved this goal within a working framework. This framework suggests that f-structures constitute an appropriate basis for semantic interpretation. For instance, the theory underlying this formalism would suggest similar representation for verbs and nouns which could then be easily treated similarly by the semantics. Hence a useful abstraction over grammatical category is easily obtained. Our system implements this approach.

Throughout this work, we tried to avoid ad hoc solutions by basing the various parts of the system on principled linguistic grounds. This is perhaps best illustrated by the restriction we imposed on relations originating from the concepts corresponding to a predicate and its subcategorized arguments. Those relations are very important in a conceptual graph and we chose to have them correspond to the θ-roles of the sentence. No such restriction exists on Conceptual Graphs. Other possibilities exist and could possibly have lead to similar results for this system. However, due to the possibilities offered by θ-theory and its strong support in Linguistics, we feel that its incorporation provides a solid basis to the system, and keeps the possibility of extensions to include other constructions open.

An interesting part of the system is the lexicon. In order to use each of the two formalisms in the system, a large amount of information about the various entities is necessary. A nice feature of our system is the inclusion of all that information into a single unit. This was made possible by re-expressing and representing the information used by Conceptual Graphs in a different way. There is nothing inherently wrong with the Conceptual Graph theory itself. However, we feel that our adaptation is more suitable to our needs and, in general, to any computational linguistic application that uses Conceptual Graphs.

With respect to the implementation of the system, the main characteristic is the use of unification to accomplish various tasks. The design has been purposely oriented to allow such an extensive use of this mechanism. Reducing many operations to unification is very attractive. It produces very efficient implementations, as this mechanism is inherent to the execution of all Prolog programs. Therefore a program with a lot of operations performed strictly by unification is bound to be more efficient than another one using other kind of manipulations.

This point is well supported by the graph representation scheme we developed. In this case, an operation that normally involves traversal of the graph (node accessibility) can be reduced to little more than unification. This scheme takes full advantage of partial tree structure often present in directed acyclic graph. The node accessibility problem is transformed to that of paths comparisons, which is entirely done through unification. In case of trees, the implementation of that operation is optimal with that scheme. In the general case of directed acyclic graphs, the efficiency is affected by the non-determinism of the structure. However, the advantages of partial tree structure(s) present in the graph are nevertheless preserved.

6.2. Extensions

As it is the case for most computational linguistic applications, the next sensible objective after this work would be to enlarge the class of constructions handled. The implementation of such constructs should not interfere with what is presently in place as we tried to keep possibilities for such additions open during the project's evolution. In particular, we have been careful to eliminate undesirable side effects after the processing of each part. We now point out some specific areas for extending this work.

A weak part of our system is the way in which adjuncts are handled. Such constructions usually carry some important semantic information about the circumstances surrounding an event or a state described by a sentence. A major characteristic of these constructs is that they add semantic information without being syntactically related to the rest of the sentence. Hence, they cannot be subcategorized for, as opposed to object complements for instance. It is therefore difficult to verify whether the semantic information of adjuncts is compatible with the rest of the sentence.

We mentioned a very simple case, handled by the system, about the relation concerning the tense of a clause and temporal adverbs. This approach is only useful for the simpler cases, as it cannot be nicely extended to cover more complex temporal constructions.

An interesting approach to the problem of representing temporal information is the one developed by Hornstein [20, 19]. He extends a general framework to represent temporal information which can, in particular, capture the interaction of tense with some temporal adverbs (like *yesterday*, *tomorrow*, etc.) and conjunctions (e.g., *when*, *before*, etc.). He uses an approach in which three entities are taken as basic: S – the moment of speech, R – a reference point, and E – the moment of the action. The idea is to represent tense by specifying these as points on an imaginary time line. In order to do so, two operations are introduced: *linearity* (denoted by "__") and *associativity* (denoted by ","). The first one indicates an order between the points (i.e., one happened "before" the other) whereas the second one is used to specify that two points are contemporaneous. A given tense is then represented by a unique configuration of these three entities with the two operations. For instance, the simple past tense of English corresponds to the configuration:

E,R__S

where both the time of the event and the point of reference occur before the moment of speech. The past perfect would correspond to

E__R__S

Temporal adverbs are assumed to modify either R of E.

The following process takes place in order to account for correct temporal constructions that use adjuncts. First a configuration is obtained from the tense of the verb. The various temporal adjuncts will be associated to either R or E, forcing a change on the initial configuration. For instance, the sentence

The cat ate tomorrow.

would initially be associated the configuration *E,R__S* from the tense of the verb. The adverb *tomorrow* forces the point to which it is attached to move after the point of speech. This is inherent in the meaning of *tomorrow*. The configuration *S__E,R* would then result.

In order to control the acceptance of derived configurations, the ***linearity condition*** is introduced. This condition specifies that the original linear order of a configuration cannot be changed by a derivation. [20]. The above example would violate this condition and hence be rejected as semantically ill formed.

This framework can also account for complex sentences in which the temporal information of some parts is expressed in relation to that of another part, as for instance in

The game was over when he got home at 10.

Both events will have a configuration from the tense of their respective verbs. Correct construction will be easily identified as follows. Both *S* points will be associated and the *R* point of the second one will be placed underneath the one of the first. If this can be done without affecting the linear order of either configuration, then the sentence will be assumed to be consistent with respect to the temporal information.

This approach is very suitable to our system. In order to treat temporal adjuncts this way, we would have to implement the correspondence between the particular tenses and their configuration in terms of the three entities. In the case of simple tenses, this is easily done, as the tense information is already present in the f-structure. For complex tenses (e.g., perfect), the exact tense configuration could be deduced from specific patterns in the f-structure. The manipulation of the adjuncts on a configuration could be coded in the lexicon, by possibly using one of the unused fields, as these adjuncts are always introduced by a non-subcategorizing entity. The inclusion of this approach would allow the system to handle more sophisticated constructions and would not interfere in any way with the processing of existing constructs.

Another important aspect of the system that could be enhanced is the treatment of quantification. Conceptual Graphs provide support for the simpler cases of quantification, as it was illustrated by the table shown on page 23. This treatment becomes inadequate when the scope of a particular quantifier ranges over more complex constructions, such as relatives clauses. Consider for instance the phrase

Books that have a hard cover costs ...

The concept representing *books* will have a generic referent. However, there is no easy way of extending the scope of this referent over the relative clause. The formalism does not provide any mechanism for capturing the modification of the referent by other concepts/sub-graphs. A solution to this problem would involve the development of a more detailed representation of referents, and a mechanism that would properly specify the range of each referent. Part of the problem is due to the fact that the scope may range over parts of a graphs which is hard to identify in terms of concepts and sub-graphs.

The graph representation scheme presented in Chapter 3, as we have said in the introduction, has been adapted to a very different framework than the current one [11]. The framework is called *Discontinuous Grammars*[29] [9, 14] and in particular, its Static Discontinuity family [10, 13, 15, 12] is attractive to implement transformational grammars, such as *Government and Binding* [7]. In such grammars, constraints on movement are usually expressed in terms of node domination. An example of such a constraint is *subjacency*, which stipulates that the movement of a node cannot cross two nodes with a particular characteristic (called *bounding nodes*).

The graph representation scheme of Chapter 3 can be used to represent the parsing history of a derivation. Then, when a transformation rule is about to be applied, the path to two bounding nodes are retrieved and the verification for the crossing over these two nodes can be done by unification alone, using predicates similar to the ones we gave in Chapter 3 for *subtype*. The structures represented in this case are always trees, therefore the comparison of nodes is always done in constant time.

[29]These grammars were first called *Gapping Grammars*.

Appendix A
Some Implementation Details

The major ideas presented in this thesis have been incorporated into an implementation in Prolog. Although not complete, the implementation covers the initial target language.

We now provide the reader with some implementational details that have purposely been omitted in the discussion for sake of clarity. This appendix is devoted the presentation of some implementational details of the transformation program, whose (English) description appears in Chapter 5. We have included the most important parts here.

In order to ease the discussion, we introduce the Prolog representation of f-structures and Conceptual Graphs.

A.1. Representation of F-Structures

The representation of f-structures is easily understood from its definition. Recall that an f-structure is composed of attribute-value pairs. We define the binary operator "/\" to represent an f-structure. Its first argument is an attribute and the second one is its associated value.

In cases where the value represent a predicate that subcategorizes for arguments, we include the argument(s) in a list which is joined to the value using the operator "::" as in:

```
    drink::[subj,obj]
```

This allows quick separation of the value and its argument(s) through unification.

The remainder of the representation of f-structures is reasonably straightforward, except in the case of coreference. Similar predicate values in an f-structure are assumed to refer to different entities, unless they are coindexed. In order to represent this co-reference, we define the operator "^", which joins a predicate to a subscript value, as in:

pred/\man^1

Two subscripted predicates will be assumed to refer to different entities, unless they have the same co-index. The following example shows an example of our representation for the sentence

A man is drinking wine.

```
fstr/\[subj/\[pred/\man^1, det/\a, agr/\[num/\sg, pers/\3]],
       tns/\present,
       pred/\prog::vcomp,
       vcomp/\[participle/\present,
               subj/\[pred/\man^1, det/\a, agr/\[num/\sg, pers/\3]],
               pred/\drink::[subj, obj], tns/\pres, agr/\[num/\sg, pers/\3],
               obj/\[pred/\wine, det/\a, agr/\[num/\sg, pers/\3]]
              ]
     ]
```

For consistency of the operations, we assume that the f-structure of a sentence appear as a value of the attribute *fstr*, as illustrated in the above example.

A.2. Representation of Conceptual Graphs

One of the main objective of this work was to complete a language translator as outlined in Section 2.1, In particular, we wanted to have the possibility of using the graphs produced by the present system as input to an existing language generator using CGs. Therefore, the representation for CGs that we chose has been strongly influenced by that of the other system, which is fully described in [27].

Following the above mentioned reference, we represent a Conceptual Graph by a three-place predicate *graph*, having the following format:

graph(Id,ConceptList,RelList).

Id is simply an integer identifier of the graph. *ConceptList* is a list of concepts/subgraphs of the graph, each one of which is associated a number. RelList is a list of the relations of the graph. The elments of that list are dyadic function terms describing for each concept originating relation(s) the list of concept(s) to which the originating concept is linked to, along with the name of each relations. The following are examples of graphs generated by the system

graph(2,[1~prog,2~g::1],[rel(1,[2::op])])
graph(1,[3~wine::unspec,2~drink,1~man::unspec],[rel(2,[3::theme,1::agent])])

Only minor syntactic variances have been adopted in comparison with the representation of [27], to improve readability.

A.3. Important Predicates

```
/*************************************************************************/
/* Translate for an f-str pred.                                          */
/*                                                                       */
/* 1st 3 arg.: F-structure, graph in and graph out                      */
/* last 2: part from when it is called (e.g. subj, verb, etc., for use  */
/*          by the lexicon)#list of f-str attr-graph token              */
/*          (e.g. [subj::1-cat_g,verb::2-drink_g,...])#list of things   */
/*          already referred to by relations but not processed yet.     */
/*      Format:                                                         */
/*          [GF::Cno::Name,...], where Cno is the concept originating a rel.*/
/*************************************************************************/
translate(_/\ptr::Dest,graph(Id,CL,RelL),graph(Id,CL,RelL),
                                Part#PredL#WantedLin,Part#PredL#WantedLout) :-
        member(Dest::Cno~Gtoken::_,PredL), !,
        seeifwanted(Part,RelL,(Cno~Gtoken)#WantedLin,WantedLout).

translate(pred/\Ftoken::_,G,G,A,A) :-
        auxiliary(Ftoken), !,
        assert(subgraph(Ftoken)).

translate(pred/\Ftoken::GFL,graph(Id,CLin,RelLin),graph(Id,CLout,RelLout),
                           Part#PredLin#WantedLin,Part#PredLout#WantedLout) :-
        addvar(GFL,ArgL),
        lex(Ftoken,Gtoken,ArgL,Types,Part),
        addconc(Gtoken,CLin,CLout,Cno), !,
        seeifwanted(Part,RelLin,(Cno~Gtoken)#WantedLin,WantedLtmp),
        insertTR(ArgL,Types,Cno,RelLin,RelLout,
                 [Part::Cno~Gtoken|PredLin]#WantedLtmp,PredLout#WantedLout).

translate(pred/\Ftoken,graph(Id,CLin,RelL),graph(Id,CLout,RelL),
     Part#PredLin#WantedLin,Part#[Part::N~Gtoken|PredLin]#WantedLout) :-
        lex(Ftoken,Gtoken,_,_,Part),
        addconc(Gtoken,CLin,CLout,N), !,
        seeifwanted(Part,RelL,(N~Gtoken)#WantedLin,WantedLout).

/*************************************************************************/
/* Translate for the det part.                                           */
/*                                                                       */
/* 1st 3 arg.: F-structure, graph in graph out                          */
/* last 2: AuxInfo as described for the pred part.                      */
/* Note: - assumes that the predicate has already been inserted in the  */
/*          graph i.e. it appears before in the f-str.                  */
/*************************************************************************/
translate(det/\Ftoken,graph(Id,CL,RelL),graph(Id,CL,RelL),Part#PredL#WL,
                                                        Part#PredL#WL) :-
        lex(Ftoken,Gtoken,_,_,det),
        member(Part::(_~Pred::_),PredL),
        member(_~Pred::Gtoken,CL).
```

```
/****************************************************************************/
/* Translate for modifier part.                                             */
/*                                                                          */
/* 1st 3 arg.: F-structure, graph in and graph out                          */
/* last 2: Part of the f-str whose predicate is modified#same 2 as above    */
/****************************************************************************/
translate(modattr/\ModL,Gin,Gout,Part#PredL#WL,Part#PredL#WL) :-
        member(Part::Cno~_,PredL),
        process(ModL,Gin,Gout,Cno#_,_).

/****************************************************************************/
/* Same argument as above except that Victim is the # of the conc modified */
/****************************************************************************/
translate(mod/\[Rel/\Ftoken|More],graph(ID,CLin,RelLin),
                graph(ID,CLout,RelLout),Victim#PredL#WL,Victim#PredL#WL) :-
        lex(Ftoken,Gtoken,_,_,mod),
        addconc(Gtoken,CLin,CLtmp,Cno),
        addrel(Victim,Cno::Rel,RelLin,RelLtmp),
        process(More,graph(Id,CLtmp,RelLtmp),graph(Id,CLout,RelLout),
                                                Cno#AuxI,Cno#AuxI).

/****************************************************************************/
/* Translate for other attributes. Simply call process on the value list   */
/****************************************************************************/
translate(Attr/\ValL,Gin,Gout,_#PredL#WantedL,AuxIout) :-
        member(Attr,[fstr,subj,obj,by_obj,of_obj,vcomp]),
        process(ValL,Gin,Gout,Attr#PredL#WantedL,AuxIout).

/****************************************************************************/
/* This predicate takes in a list of attr-value pairs and calls translate  */
/* for each element of that list.                                           */
/****************************************************************************/
process([],G,G,AuxI,AuxI).
process([Attr/\Val|Rest],Gin,Gout,Called#PredL#WantedL,AuxIout) :-
        ( member(Attr,[subj,obj,vcomp,by_obj,of_obj]) ->
        translate(Attr/\Val,Gin,Gtmp,Attr#PredL#WantedL,_#PredLtmp#WantedLtmp) |
            translate(Attr/\Val,Gin,Gtmp,Called#PredL#WantedL,
                                                _#PredLtmp#WantedLtmp) ),
        process(Rest,Gtmp,Gout,Called#PredLtmp#WantedLtmp,AuxIout).

/****************************************************************************/
/* seeifwanted: sees if a the predicate of part (an f-str attr having a    */
/*              pred field) is needed as a dest. in a relation. Is so, it  */
/*              removed from the wanted list (assume as many entries in the*/
/*              wanted list as necessary.                                   */
/* 1st arg.: part of the f-str whose predicate is checked                   */
/* 2nd arg.: relation diff list                                             */
/* 3rd & 4th arg.: Number of the concept being checked#wanted list as above*/
/****************************************************************************/
seeifwanted(Part,RelL,(Cno~CDest)#WantedLin,WantedLout) :-
        delete(Part::Ino::Name::Type,WantedLin,WantedLout), !,
        compatible(CDest,Type,Part),
        changerel(Ino,Cno::Name,RelL).
seeifwanted(_,_,_#WL,WL).
```

```
/**************************************************************************/
/* insertTR                                                               */
/*                                                                        */
/* 1st arg.: list of TR::GF obtained by convert (or o.w.)                 */
/* 2nd arg.: list of types associated with the subcategorized gf          */
/* 3rd arg.: concept # initiating relations                               */
/* 4th & 5th arg.: relation diff list (in and out)                        */
/* 6th & 7th arg.: last 2 elts of the aux info str as described in the    */
/*                 predicate part                                         */
/**************************************************************************/
insertTR([],_,_,RelL,RelL,AuxInfo,AuxInfo).
insertTR([GF::TR|Rest],[Type1|Types],Ino,RelLin,RelLout,
                                   PredL#WantedLin,PredL#WantedLout) :-
        member(GF::Cno~CDest,PredL), !,
        compatible(CDest,Type1,GF),
        addrel(Ino,Cno::TR,RelLin,RelLtmp),
        insertTR(Rest,Types,Ino,RelLtmp,RelLout,PredL#WantedLin,PredL#WantedLout).
insertTR([GF::TR|Rest],[Type1|Types],Ino,RelLin,RelLout,PredL#WantedLin,
                                                       PredL#WantedLout) :-
         addrel(Ino,_::TR,RelLin,RelLtmp),
         insertTR(Rest,Types,Ino,RelLtmp,RelLout,
                  PredL#[GF::Ino::TR::Type1|WantedLin],PredL#WantedLout).

/**************************************************************************/
/* Predicate that preforms the type verification between a concept (1st arg)*/
/* and a type (2nd arg). The type of the subset has to be a subtype of the  */
/* 2nd arg. The 3rd arg. is the gf being processed and is used only for     */
/* the error message in case of failure.                                    */
/**************************************************************************/
compatible(Concept::_,Type,_) :- subtype(Concept,Type), !. % special for ref.
compatible(Concept,Type,_) :- subtype(Concept,Type), !.
compatible(Concept,Type,GF) :-
        write('****'),nl,
        write('Generation of the graph stopped because '), nl,
        write(Concept), write(' is not a subtype of '), write(Type), nl,
        write('Error while processing the '), write(GF), nl, nl,
        fail.

/**************************************************************************/
/* Predicate to check whether a fstr token (without arg.) is an auxiliary  */
/**************************************************************************/
auxiliary(Ftoken) :- member(Ftoken,
                         [be,have,prog,perf,can,could,might,should,would]).

/**************************************************************************/
/* Predicate that takes care of the generation of sub-graphs               */
/**************************************************************************/
gensub([G1|Gs],[graph(NewId,[1~Name,2~g::Id],[rel(1,[2::op])]),G1|Gs]) :-
        retract( subgraph(Name) ), !,
        G1 = graph(Id,_,_),
        NewId is Id + 1.
gensub(LG,LG).
```

```
/*****************************************************************************/
/* Predicate that completes the generation of sub-graphs for relatives      */
/*****************************************************************************/
genrel(LG,NewLG) :-
        retract(crel(Cno,Gno)),!,    % Concept Cno of graph Gno has a relative
        Relgraph is Gno + 1,
        retract(graph(Relgraph,CL,RL)),
        retract(rel(Gtoken,Relgraph)),
        delete(graph(Gno,CL1,RL1),LG,LGd),
        addconc(g::Relgraph,CL1,CL2,Grelno),
        addrel(Cno,Grelno::Gtoken,RL1,RL2),
        genrel([graph(Relgraph,CL,RL),graph(Gno,CL2,RL2)|LGd],NewLG).
genrel(LG,LG).
```

Appendix B
Sample Runs

Here are some sample runs generated by the system for different constructions. In what follows, we use the symbol "%" to indicate that the rest of the line is a comment inserted by the author. Each graph is describe by the specification of its identifier, concepts and relations. The relations use the number associated with each concept in their representation.

```
% start is a predicate generating the hierarchy prepresentation and
% reminding the user of the format of particular query.

| ?- start.
Processing the type hierarchy...
Sample query:
fstr(F),translate(F,graph(1,[],[]),G,_#[]#[],_),gensub([G],Gs),
       genrel(Gs,Gs1),printLgraph(Gs).

The predicate "run" can also be used, the body of which is this sample query


yes
| ?- run.
f-str processed:
                    ===============> F-Structure <===============

fstr        subj        pred        construction::[by_obj,of_obj]
                        det         the
                        agr         num         sg
                                    pers        3
                        by_obj      pred        beaver
                                    det         the
                                    agr         num         sg
                                                pers        3
                        of_obj      pred        dam
                                    det         a
                                    agr         num         sg
                                                pers        3
            pred        help::[subj,obj]
            tns         past
            obj         pred        lake
                        det         the
                        agr         num         sg
                                    pers        3



                          Graph: 1

Concepts:
          Name                  #               Name              #
lake::spec                      5       help1                     4
dam::unspec                     3       beaver::spec              2
construction::spec              1

Relations:
from  to   Name          from  to   Name          from  to   Name
 1    3    theme          1    2    agent          4    5    theme
 4    1    agent

% This graph is associated with the sentence:
%    The construction of a dam by the beavers help the lake.
```

```
f-str processed:
                 ==============> F-Structure <==============

fstr        subj        pred        man
                        det         a
                        agr         num         sg
                                    pers        3
            tns         present
            pred        prog::vcomp
            vcomp       participle  present
                        subj        ptr::subj
                        pred        drink::[subj,obj]
                        tns         pres
                        agr         num         sg.
                                    pers        3
                        obj         pred        wine
                                    det         a
                                    agr         num         sg
                                                pers        3


                        Graph: 2

Concepts:
          Name                  #                        Name             #
prog                            1          graph 1                        2


Relations:
from  to   Name             from  to   Name             from  to   Name
 1     2   op


                        Graph: 1

Concepts:
          Name                  #                        Name             #
wine::unspec                    3          drink                          2
man::unspec                     1

Relations:
from  to   Name             from  to   Name             from  to   Name
 2     3   theme             2     1   agent

% These graphs represent the sentence:
%          A man is drinking wine.
```

```
f-str processed:
                    ==============> F-Structure <===============

fstr        subj        pred        man
                        det         a
                        agr         num         sg
                                    pers        3

            pred        drive::[subj,obj]
            tns         pres
            agr         num         sg
                        pers        3

            obj         pred        car
                        det         a
                        agr         num         sg
                                    pers        3
                        modattr     mod         qual        red
                                    mod         qual        good


                        Graph: 1

Concepts:
            Name                    #                               Name            #
good                                5           red                                 4
car::unspec                         3           drive                               2
man::unspec                         1

Relations:
from   to    Name            from   to    Name            from   to    Name
 2     3     theme            2     1     agent            3     5     qual
 3     4     qual

% This graph corresponds to the sentence:
%           A man drives a red, good car.
```

```
f-str processed:
              ================> F-Structure <================

fstr          subj          pred          cat
                            det           the
                            agr           num           sg
                                          pers          3
              pred          drink::[subj,obj]
              tns           pres
              agr           num           sg
                            pers          3
              obj           pred          milk
                            det           none
                            agr           num           sg
                                          pers          3
                            modattr       mod           qual          good
                                                        mod           deg           very
                                          mod           qual          hot

                             Graph: 1

Concepts:
          Name                   #                   Name           #
hot                              6          very                    5
good                             4          milk::unspec            3
drink                            2          cat::spec               1


Relations:
from  to   Name            from  to   Name          from  to   Name
 2     3   theme            2     1   agent          3     6   qual
 3     4   qual             4     5   deg

% This graph represents the sentence:
%        The cat drinks hot, very good milk.
```

```
f-str processed:
               ==============> F-Structure <==============

fstr        subj        pred        book
                        det         the
                        agr         num         sg
                                    pers        3
                        rel         relmark     that
                                    subj        pred        student
                                                det         the
                                                agr         num         pl
                                                            pers        3
                                    tns         past
                                    pred        read::[subj,obj]
                                    obj         ptr::relmark
            tns         past
            pred        win::[subj,obj]
            obj         pred        prize
                        det         a
                        agr         num         sg
                                    pers        3


                        Graph: 2

Concepts:
            Name                #                   Name            #
book::spec                      3           read                    2
student::spec                   1

Relations:
from   to   Name            from  to   Name            from   to   Name
 2     3    theme            2    1    agent


                        Graph: 1

Concepts:
            Name                #                   Name            #
graph 2                         4           prize::unspec           3
win                             2           book::spec              1

Relations:
from   to   Name            from  to   Name            from   to   Name
 1     4    relop            2    3    theme            2     1    agent

% These two graph represent the sentence:
%       The book that the students read won a prize.
%
% The concepts 'book' present in both graphs are assumed to represent the
% same instance.
```

References

[1] Ronald J. Brachman.
What IS-A Is and Isn't: An Analysis of Taxonomic Links in Semantic Networks.
IEEE Computer 16(10):30-36, October, 1983.

[2] J. Bresnan.
The Passive in Lexical Theory.
The Mental Representation of Grammatical Relations.
The MIT Press, Cambridge, Mass., 1982.

[3] J. Bresnan (editor).
The Mental Representation of Grammatical Relations.
The MIT Press, Cambridge, Mass., 1982.

[4] Brown, C., Dahl, C., Massam, D., Massicotte, P., Pattabhiraman, T.
Tailoring Government and Binding Theory for Use in Natural Language Translation.
Technical Report LCCR 86-4, Simon Fraser University,1986.

[5] Brown, C.
Generating Spanish Clitics using Static Discontinuity Grammar.
PhD thesis, Simon Fraser University, 1987.

[6] C. Brown, T. Pattabhiraman, P. Massicotte.
Towards a Theory of Natural Language Generation: The Connection between Syntax and Semantics.
Natural Language Understanding and Logic Programming II.
North-Holland, 1988.

[7] Noam Chomsky.
Lectures on Government and Binding, the Pisa Lectures, 2nd (revised) Edition.
Foris Publications, Holland, 1982.

[8] Verónica Dahl.
On Database Systems Development Through Logic.
ACM Transactions on Database Systems 7(1):102-123, March, 1982.

[9] Dahl, V.
More on Gapping Grammars.
In *Proceedings International Conference on V Generation Computer Systems.* Tokyo, 1984.

[10] Dahl, V.
Gramaticas discontinuas: una herramienta computacional con aplicaciones en la teoria de Reccion y Ligamiento.
Revista Argentina de Linguistica 2(2), 1986.

[11] Dahl, V. and Massicotte, P.
Meta-Programming for Discontinuous Grammars.
In *Proceedings of the Meta-Programming for Logic Programming Workshop*. University of Bristol, 1988.

[12] Dahl, V.
Representing Linguistic Knowledge through Logic Programming.
In *Fifth International Conference/Symposium on Logic Programming*. Seattle, August, 1988.

[13] V. Dahl.
Discontinuous Grammars.
Technical Report CSS/LCCR TR 88-26, Simon Fraser University,1988.

[14] Verónica Dahl and Harvey Abramson.
On Gapping Grammars.
In *Proceedings, Second International Logic Programming Conference, Uppsala, Sweden*, pages 77-88. Universitet Uppsala, 1984.

[15] Dahl,V.
Static Discontinuity Grammars for Government and Binding Theory.
In *Proc. Workshop 'Informatique and langue naturelle', Université de Nantes*. 1988.

[16] Werner Frey.
Noun Phrases in Lexical Functional Grammar.
Natural Language Understanding and Logic Programming.
North-Holland, 1985.

[17] Uwe Reyle and Werner Frey.
A Prolog Implementation of Lexical Functional Grammar.
In *Proceedings of the Eight International Conference on AI*, pages 693-695. IJCAI, Karlsruhe, 1983.

[18] Werner Frey, Uwe Reyle, and Christian Rohrer.
Automatic Construction of knowledge Base by Analysing Texts in Natural Language.
In *Proceedings of the Eight International Conference on AI*, pages 727-729. IJCAI, Karlsruhe, 1983.

[19] N. Hornstein.
Towards a Theory of Tense.
Linguistic Inquiry 8(3):521-557, 1977.

[20] N. Hornstein.
The Study of Meaning in Natural Language: Three Approaches to Tense.
Explanation in Linguistics.
Longman, 1981.

[21] R.S. Jackendoff.
Semantic Interpretation in Generative Grammar.
The MIT Press, Cambridge, Ma, 1972.

[22] R. Kaplan and J. Bresnan.
Lexical Functional Grammar: A Formal System for Grammatical Representation.
The Mental Representation of Grammatical Relations.
The MIT Press, Cambridge, Mass., 1982.

[23] C. Brown, T. Pattabhiraman, M. Boyer, D. Massam V. Dahl.
Tailoring Conceptual Graphs for Use in NL Translation.
Technical Report LCCR 86-14, Simon Fraser University,1986.

[24] A.S. Maida and S.C. Shapiro.
Intensional Concepts in Propositional Semantic Networks.
Cognitive Science 6(4), 1982.

[25] Massicotte, P. and Dahl, V.
Handling Concept-Type Hierarchies through Logic Programming.
In *Proc. Third Annual Workshop on Conceptual Graphs*. 1988.

[26] M. Minsky (editor).
Semantic Information Processing.
MIT Press, Cambridge, MA, 1968.

[27] T. Pattabhiraman, P. Massicotte, C. Brown.
User Manual for the MRI Translation Project
Laboratory for Computer and Communication Research, Simon Fraser University, 1987.

[28] Ehud H. Shapiro.
Alternation and the Computational Complexity of Logic Programs.
Journal of Logic Programming 1(1):19-33, June, 1984.

[29] L.K. Schubert, M.A. Papalaskaris, J. Taugher.
Determining Type, Part, Color, and Time Relationships.
IEEE Computer 16(10):53-60, October, 1983.

[30] L. Slocum (editor).
Machine Translation Systems.
Cambridge University Press, 1988.
First published as *Computational Linguistics* vol. II (1985) nos. 1-3.

[31] John F. Sowa.
Conceptual Structures: Information Processing in Mind and Machine.
Addison-Wesley Company, 1984.

[32] John F. Sowa.
Using a Lexicon of Conceptual Graphs in a Semantic Interpreter.
1987.

[33] John F. Sowa and Eileen C. Way.
Implementing a Semantic Interpreter using Conceptual Graphs.
IBM Journal of Research and Development 30(1), January, 1986.

[34] L. Sterling and E. Shapiro.
The Art of Prolog.
The MIT Press, Cambridge, Ma, 1986.

[35] K. Uehara, R. Ochitani, O. Mikami, J. Toyoda.
An Integrated Parser for Text Understanding: Viewing Parsing as Passing Messages Among Actors.
Natural Language Understanding and Logic Programming.
North-Holland, 1985.

[36] W.A. Woods.
What's in a link: Foundations for Semantic Networks.
Representation and Understanding: Studies in Cognitive Science.
Academic Press, New York, 1975.

[37] Hideki Yasukawa.
LFG system in Prolog.
In *Proceedings of COLING84, 10th International Conference on Computational Linguistic,* pages 358-361. Association for Computational Linguistic, Stanford University, July, 1984.