



National Library
of Canada

Acquisitions and
Bibliographic Services Branch

395 Wellington Street
Ottawa, Ontario
K1A 0N4

Bibliothèque nationale
du Canada

Direction des acquisitions et
des services bibliographiques

395, rue Wellington
Ottawa (Ontario)
K1A 0N4

Your file Votre référence

Our file Notre référence

NOTICE

The quality of this microform is heavily dependent upon the quality of the original thesis submitted for microfilming. Every effort has been made to ensure the highest quality of reproduction possible.

If pages are missing, contact the university which granted the degree.

Some pages may have indistinct print especially if the original pages were typed with a poor typewriter ribbon or if the university sent us an inferior photocopy.

Reproduction in full or in part of this microform is governed by the Canadian Copyright Act, R.S.C. 1970, c. C-30, and subsequent amendments.

AVIS

La qualité de cette microforme dépend grandement de la qualité de la thèse soumise au microfilmage. Nous avons tout fait pour assurer une qualité supérieure de reproduction.

S'il manque des pages, veuillez communiquer avec l'université qui a conféré le grade.

La qualité d'impression de certaines pages peut laisser à désirer, surtout si les pages originales ont été dactylographiées à l'aide d'un ruban usé ou si l'université nous a fait parvenir une photocopie de qualité inférieure.

La reproduction, même partielle, de cette microforme est soumise à la Loi canadienne sur le droit d'auteur, SRC 1970, c. C-30, et ses amendements subséquents.

Canada

University of Ottawa
Ottawa, Ontario, Canada

**Text Processing without *a priori* Domain Knowledge:
Semi-Automatic Linguistic Analysis
for Incremental Knowledge Acquisition**

by

Sylvain Delisle, B.Sc.A., M.C.S.

Department of Computer Science
Faculty of Science

Joint Ph.D. Program of the Ottawa-Carleton Institute for Computer Science

A Thesis Submitted to
the School of Graduate Studies and Research
in Partial Fulfillment of the Requirements
for the Degree of Doctorate in Philosophy (Ph.D.)
in Computer Science

January 21, 1994

© Sylvain Delisle, 1994



National Library
of Canada

Bibliothèque nationale
du Canada

Acquisitions and
Bibliographic Services Branch

Direction des acquisitions et
des services bibliographiques

395 Wellington Street
Ottawa, Ontario
K1A 0N4

395, rue Wellington
Ottawa (Ontario)
K1A 0N4

Your file Votre référence

Our file Notre référence

The author has granted an irrevocable non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of his/her thesis by any means and in any form or format, making this thesis available to interested persons.

L'auteur a accordé une licence irrévocable et non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de sa thèse de quelque manière et sous quelque forme que ce soit pour mettre des exemplaires de cette thèse à la disposition des personnes intéressées.

The author retains ownership of the copyright in his/her thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without his/her permission.

L'auteur conserve la propriété du droit d'auteur qui protège sa thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

ISBN 0-315-93570-7

Canada



UNIVERSITÉ D'OTTAWA
UNIVERSITY OF OTTAWA

Abstract

Technical texts are an invaluable source of the domain-specific knowledge which plays a crucial role in advanced knowledge-based systems today. However, acquiring such knowledge has always been a major difficulty in the construction of these systems — this critical obstacle is sometimes referred to as the “knowledge acquisition bottleneck”. Typically, this acquisition phase is one of the knowledge engineer’s most demanding activities.

In order to lessen the burden on the knowledge engineer’s shoulders, several approaches have been proposed in the literature. A few of these suggest processing texts pertaining to the domain of interest in order to extract the knowledge they contain and thus facilitate the domain modelling. Unfortunately, these systems require an impracticable amount of complex *a priori* knowledge, especially domain-specific semantics. As a result, the initial knowledge engineer’s problem is replaced with another: how to acquire the bulk of this complicated semantic knowledge from text?

We herein propose a new approach to knowledge acquisition from texts; this approach is comprised of a new methodology and computational framework for the implementation of a linguistic processor which represents the central component of a system for the acquisition of knowledge from text. The system, named TANKA, is not given the complete domain model beforehand. It is designed to process technical texts in order to incrementally build a knowledge base containing a conceptual model of the domain. TANKA is an intelligent assistant to the knowledge engineer; when it cannot proceed entirely on its own, the user is asked to collaborate. In the process, the system acquires knowledge from text; it can be said to learn about the domain.

The originality of the research is due mainly to the fact that we do not assume significant *a priori* domain-specific (semantic) knowledge: this assumption represents a severe constraint on the natural language processor. The only external elements of knowledge we consider in the proposed framework are “off-the-shelf” publicly available and domain-independent repositories, such as a basic dictionary containing surface syntactic information (i.e. The Collins) and a lexical database (i.e. WordNet). Other components of the proposed framework are general-purpose.

The parser (DIPETT) is domain-independent with a large coverage of English: our approach relies on full syntactic analysis. The Case-based semantic analyzer (HAIKU) is semi-automatic: it interacts with the user in order to get his¹ approval of the analysis it has just proposed and negotiates refined elements of the analysis when necessary. The combined processing of DIPETT and HAIKU allows TANKA, the encompassing system², to acquire knowledge, based on the conceptual elements produced by HAIKU. The thesis also describes experiments that have been conducted on a Prolog implementation of both of these text analysis components.

The approach presented in the thesis is general and in principle portable to any domain in which suitable technical texts are available. The thesis presents theoretical considerations as well as engineering aspects of the many facets of this research work. We also provide a detailed discussion of many future work items that could be added to what has already been accomplished in order to make the framework even more productive. We claim that this new approach to the acquisition of knowledge through text processing helps us to better understand the nature of knowledge acquisition and text processing, and in particular the relationship between them. We also claim that it provides us with an appropriate technical framework for the efficient implementation of semi-automatic knowledge-acquisition assistants.

¹ In order to lighten the text, the terms 'he' and 'his' have been used generically to refer equally to persons of either sex. No discrimination is either implied or intended.

² DIPETT and HAIKU constitute a conceptual analyzer that can be used independently of TANKA or within a different encompassing system.

Sommaire

Les textes techniques représentent une inestimable source de connaissances spécifiques. Ces dernières jouent un rôle prépondérant dans les systèmes à base de connaissances les plus évolués. Cependant, l'acquisition de ces connaissances constitue toujours une difficulté majeure dans la construction de tels systèmes — on réfère parfois à cet obstacle comme le “goulot d'étranglement de l'acquisition des connaissances”. Cette phase d'acquisition est typiquement l'une des activités les plus exigeantes du cognicien³ (ou ingénieur de la connaissance).

Plusieurs méthodes ont été proposées pour faciliter la tâche du cognicien. Quelques unes suggèrent de traiter des textes qui portent sur le domaine d'intérêt dans le but d'en extraire la connaissance et de faciliter la modélisation du domaine. Malheureusement, les systèmes qui effectuent de tels traitements nécessitent *a priori* une quantité de connaissances complexes, particulièrement de nature sémantique, et spécifiques au domaine d'intérêt. Il en résulte que le problème initial du cognicien est remplacé par un nouveau problème: comment effectuer l'acquisition de l'essentiel de ces connaissances sémantiques complexes à partir du texte?

Nous proposons une nouvelle façon d'acquérir des connaissances à partir du texte, exprimée ici sous forme d'une nouvelle méthodologie et d'une structure informatique appropriée pour la réalisation d'un processeur linguistique qui constitue le coeur d'un système d'acquisition des connaissances. Le système, nommé TANKA, ne possède pas *a priori* le modèle complet du domaine visé. Il est conçu pour construire, de manière incrémentale, une base de connaissances représentant un modèle conceptuel du domaine, à partir du traitement informatisé de textes techniques. TANKA est un assistant “intelligent” du cognicien: lorsque le système ne peut procéder de façon entièrement autonome, l'usager est invité à collaborer. Ce faisant, le système acquiert des connaissances à partir du texte; on peut dire qu'il apprend sur le domaine.

L'originalité de notre recherche est due principalement au fait que nous n'assumons pas, *a priori*, de connaissances sémantiques, spécifiques au domaine d'intérêt: cela représente une sévère contrainte pour le module de traitement du langage naturel. Les

³ Dans le but de simplifier le texte de cette thèse, le genre masculin est utilisé au sens neutre et désigne aussi bien les femmes que les hommes, sans aucune discrimination.

seuls éléments externes de connaissances que nous utilisons dans le cadre de notre méthode sont des dictionnaires généraux et publics comme le dictionnaire Collins (pour ses informations syntaxiques de surface sur les mots de la langue anglaise) et la base de données lexicales WordNet. Les autres composantes du processeur linguistique sont tout à fait générales.

L'analyseur syntaxique DIPETT est indépendant du domaine d'application et possède une large couverture des phénomènes linguistiques de la langue anglaise. Notre méthode repose sur l'analyse syntaxique complète et détaillée. HAIKU, le processeur sémantique basé sur l'analyse de Cas, est semi-automatique: il compte sur l'utilisateur pour obtenir son approbation de l'analyse qu'il propose ou pour négocier certains éléments de celle-ci lorsque cela s'avère nécessaire. Le traitement combiné de DIPETT et HAIKU permet à TANKA, le système qui les englobe⁴, d'acquérir des connaissances à partir des éléments conceptuels produits par HAIKU. Cette thèse décrit aussi des expériences qui ont été conduites sur une implémentation en Prolog de ces deux modules d'analyse de texte.

La méthodologie proposée dans cette thèse est générale et en principe applicable à tout domaine pour lequel des textes techniques convenables sont disponibles. Cette thèse présente les considérations d'ordre théorique ainsi que les aspects d'ingénierie des multiples facettes de ce travail de recherche. Nous discutons aussi des ajouts futurs qui pourraient contribuer à rendre notre méthode, de même que sa structure informatique, encore plus productive. Nous prétendons que cette nouvelle méthode pour l'acquisition de connaissances à partir du texte nous aide à mieux comprendre la nature même de l'acquisition des connaissances et du traitement des textes et, en particulier, de la relation entre les deux. Nous prétendons également que cette méthode constitue un cadre technique approprié pour la réalisation efficace d'assistants semi-automatiques pour l'acquisition de connaissances.

⁴ DIPETT et HAIKU forment un analyseur conceptuel qui peut être utilisé indépendamment de TANKA ou au sein d'un autre système.

Table of Contents

1. INTRODUCTION: BACKGROUND AND OVERVIEW.....	1
1.1 The Need for Automated Knowledge Acquisition from Text: A Challenging Task.....	2
1.2 Text Processing and a priori Knowledge.....	4
1.2.1 Knowledge Requirements for Text Processing.....	4
1.2.2 A priori Knowledge.....	5
1.3 Text Processing for Knowledge Acquisition.....	7
1.3.1 Overview of the TANKA System.....	7
1.3.1.1 TANKA's Functionality.....	7
1.3.1.2 Architecture of the TANKA System.....	9
1.3.1.3 The Current Status of TANKA.....	11
1.3.2 Theoretical Foundations: Goals, Hypotheses and Assumptions.....	12
1.3.2.1 Goals.....	12
1.3.2.2 Hypotheses and Simplifying Assumptions.....	13
1.4 Main Results of the Research.....	18
1.5 Organization of the Thesis.....	19
 2. TEXT PROCESSING FOR KNOWLEDGE ACQUISITION: RELATED WORK.....	 21
2.1 Text Processing: A Quick Survey.....	22
2.1.1 Text Processing: Natural Language as Data.....	22
2.1.1.1 [Munch 86].....	22
2.1.1.2 [Jacobs & Rau 90].....	23
2.1.2 Natural Language Interfaces.....	24
2.1.3 Parsing-Related Issues.....	25
2.1.3.1 Broad-Coverage Parsing.....	25
2.1.3.2 Attachment.....	28
2.1.3.3 Conjunctions and Coordination.....	29
2.1.3.4 Discourse Processing.....	30
2.1.3.5 Fragments.....	32
2.1.3.6 Idioms.....	33
2.2 Acquiring Knowledge through Text Processing.....	35
2.2.1 Semantic and Domain-Related Knowledge.....	35
2.2.1.1 [Grishman <i>et al.</i> 82] and [Haas & Hendrix 83].....	35
2.2.1.2 [DeJong 82] and [Mooney 90].....	36
2.2.1.3 [Frey <i>et al.</i> 83].....	37
2.2.1.4 [Fum 85].....	38
2.2.1.5 [Nishida <i>et al.</i> 86].....	39

2.2.1.6 [Moller 88].....	39
2.2.1.7 [Silvestro 88], [Gomez 89], and [Reimer 90].....	40
2.2.1.8 [Regoczei & Hirst 89a], [Regoczei & Hirst 89b].....	42
2.2.1.9 [Rinaldo 89], [Gao & Salveter 91], [Moulin & Rousseau 90], [Moulin & Rousseau 91], [Cowie 83], and [Rousselot <i>et al.</i> 92].....	43
2.2.1.10 [Lang & Hirschman 88].....	44
2.2.1.11 [Zernik & Jacobs 90], [Smadja 91], [Jacobs 92a], [Brent 91a], [Basili <i>et al.</i> 92], and [Grefenstette 92b].....	46
2.2.1.12 [Ciravegna <i>et al.</i> 92], [Hauptmann 93], and [Kim & Moldovan 93].....	47
2.2.1.13 [Ballard 86], [Ayuso <i>et al.</i> 87], and [Grosz <i>et al.</i> 87].....	49
2.2.1.14 [Liu & Soo 93].....	50
2.2.2 Syntactic Knowledge.....	52
2.2.3 Lexical Knowledge	54

3. DIPETT: PARSING TECHNICAL TEXTS WITH A DOMAIN-INDEPENDENT, BROAD COVERAGE OF ENGLISH	58
3.1 DIPETT as Part of a Text Processing System.....	59
3.2 Linguistic Considerations.....	61
3.2.1 Parsing a Substantial Subset of a Natural Language.....	61
3.2.2 Determining the Linguistic Coverage of the Parser (or DIPETT as a computerized model of a subset of [Quirk <i>et al.</i> 85]).....	62
3.2.3 A Precise Characterization of the DIPETT Parser.....	63
3.2.3.1 Linguistic Theories and Parsing: Taking a Neutral Stand.....	63
3.2.3.2 Algorithmic Aspects of Parsing.....	64
3.2.3.3 A Typical Parse Tree.....	67
3.3 The Architecture and Components of DIPETT.....	69
3.3.1 The Driver and the User Interface.....	69
3.3.2 Lexicon and Dictionaries	70
3.3.2.1 DIPETT's Lexicon.....	70
3.3.2.2 The Collins Dictionary and the WordNet Lexical Database.....	74
3.3.3 The Scanner and the Lexical Analyzer	74
3.3.4 The Syntactic Module	77
3.3.4.1 Encoding English Grammar with DCG Rules.....	78
3.3.4.2 Definite Clause Grammars and Other Logic Grammar Formalisms.....	78
3.3.4.3 The Grammar's Main Non-terminals.....	81
3.3.4.4 The Influence of [Quirk <i>et al.</i> 85] on the Resulting Grammar.....	83
3.3.5 Subsidiary Components.....	84

3.3.5.1 The Memorizing Device and the Helper Mechanism.....	84
3.3.5.1.1 Notes.....	85
3.3.5.1.2 The Memorizer.....	85
3.3.5.1.3 The Helper.....	87
3.3.5.2 The Trace Mechanism.....	88
3.3.5.3 The Tagger.....	89
3.4 Syntactic Coverage: Difficult Issues.....	91
3.4.1 Structural Ambiguity.....	91
3.4.2 Time Constraints for Problematic Input Strings.....	92
3.4.3 Parsing Conjoined Structures.....	94
3.4.3.1 Conjoined Structures Handled by DIPETT.....	94
3.4.3.2 Top-Level Parsing Heuristics.....	95
3.4.3.2.1 Top-Level Rule Selection.....	96
3.4.3.2.2 Parsing Coordinated Clauses with Look-Ahead.....	98
3.4.4 Parsing Fragments.....	99
3.4.4.1 Fragments as Naturally Occurring Text Objects.....	99
3.4.4.2 Parsing an Input as a Series of Fragments.....	100
3.5 Simple Anaphora Resolution for Pronouns: the Pronoun Explicitizer.....	104
3.6 Testing the Syntactic Coverage with Actual Texts: Results and Discussion.....	108
3.6.1 Tuning the Parser with Hand-made Input Strings.....	108
3.6.1.1 Test Cases for Noun Phrases and Nominal Clauses.....	108
3.6.1.2 Test Cases for Verb Phrases and Sentences.....	109
3.6.1.3 Miscellaneous.....	109
3.6.2 The QUIZ User Manual.....	109
3.6.2.1 The Document.....	110
3.6.2.2 Results.....	111
3.6.3 The T4043 Tax Guide.....	111
3.6.4 Discussion.....	114
4. HAIKU: ACQUIRING CASE PATTERNS VIA INTERACTIVE SEMANTIC ANALYSIS.....	120
4.1 HAIKU as Part of a Text Processing System.....	122
4.2 Toward Comprehensive Case-based Semantic Analysis.....	127
4.2.1 Limitations of Simple Case-based Frameworks.....	127
4.2.1.1 One Case per Argument.....	127
4.2.1.2 Noun-Noun Relationships.....	128
4.2.1.3 The Need for an Extended Theory.....	128
4.2.2 A Comprehensive Framework for HAIKU.....	129
4.2.2.1 The Set of Cases.....	129
4.2.2.2 Notation.....	130
4.2.2.3. Details of the Proposed Framework.....	132

4.2.2.3.1. Complex NPs and embedded clauses.....	132
4.2.2.3.2. Stative and conjoined verbs.....	133
4.2.2.3.3. Structures other than simple declarative clauses.....	135
4.2.2.3.4. Conjoined structures (ref. 3.4.3.1).....	136
4.3 Interactive Case Analysis for Knowledge Acquisition: the Core of HAIKU.....	140
4.3.1 Data Structures.....	140
4.3.2 A Self-contained, Extended Example.....	142
4.3.3 The Process of Case Analysis and its Crucial Algorithms.....	145
4.3.3.1 From Multi-Clause Inputs to a Series of Individual but Interlinked Clauses.....	145
4.3.3.2 Case Analysis of Individual Clauses.....	148
4.4 Expressing Situations $\mathcal{S}_1$ to $\mathcal{S}_5$ in Terms of Three Parameters: the Verb, the Case-marker Pattern (CMP), and the Case Pattern (CP).....	151
4.4.1. Dealing with New CPs.....	151
4.4.2. Situation #1 ($\mathcal{S}_1$).....	151
4.4.3. Situation #2 ($\mathcal{S}_2$).....	152
4.4.4. Situation #3 ($\mathcal{S}_3$).....	153
4.4.5. Situation #4 ($\mathcal{S}_4$).....	154
4.4.6. Situation #5 ($\mathcal{S}_5$).....	155
4.5 Difficult Issues: Finding Suitable Example Sentences and Suggesting Relevant Case Patterns.....	157
4.5.1 Finding and Using Example Sentences.....	157
4.5.1.1 Avoiding Complete Rejection.....	157
4.5.1.2. Problems with the Closeness Metric.....	158
4.5.1.3. A Possible Scenario.....	158
4.5.2. Challenging HAIKU: a Difficult Example.....	159
4.6 HAIKU's Closeness Computation: Semantic Similarity Based on Syntactic Markers.....	162
4.6.1 General Definitions of Matching Case Marker Patterns.....	162
4.6.1.1 Definitions of Matching.....	163
4.6.2 Basic Computations for the Closeness Metric.....	165
4.6.2.1 Computing the Closeness Factor within pum.....	165
4.6.2.2 Computing the Comparison Coefficient within npum.....	165
4.6.2.3 Simple Examples of Closeness Calculations.....	167
4.6.2.4 Computational Details of the Closeness Metric.....	168
4.6.3 Examples of Closeness Calculations.....	170
4.6.4 Further Refinements of 'pum' and 'compute_CC'.....	172

4.6.5 Strategic and Algorithmic Aspects of the Closeness Computation.....	173
4.7 Testing the Semantic Analyzer with Actual Texts: Results and Discussion	176
4.7.1 An Experiment with HAIKU	176
4.7.2 General Observations.....	177
4.7.3 Results for Chapters 1 and 2 of the QUIZ Manual	178
4.7.4 Discussion	179
5. FUTURE WORK ON DIPETT AND HAIKU	182
5.1 A Parse Tree Editor with Learning Capabilities	183
5.1.1 The Editor.....	183
5.1.2 The Editor as a System that Learns.....	185
5.2 Refining HAIKU's CA: a Concept Ontology for Noun Semantics.....	187
5.3 Extending DIPETT's Coverage of English.....	189
5.4 Transforming DIPETT into a Flexible Expectation-Driven Parser	193
5.5 Text Summarization as Parametrizable Text Highlighting.....	196
5.5.1. Introduction.....	196
5.5.2. A Proposal Based on the Notion of Conceptual Distance.....	196
6. CONCLUSION.....	200
6.1 Significance of the Results.....	202
6.2 Current and Prospective Applications.....	206
6.2.1 DIPETT and HAIKU as MaLTe's Linguistic Module.....	206
6.2.2 A Tool for Assisting Conceptual Data Modelling	207
6.2.3 The Processing of Non-Technical Text.....	208
7. REFERENCES.....	211
 Appendix A: DIPETT's Driver and User Interface.....	234
A.1: The 'j' ("jump start") Option	235
A.2: The 'd' ("detailed menu") Option.....	238
A.3: The 'b' ("batch") Option.....	240
 Appendix B: DIPETT's English Grammar	242
B.1: DIPETT_6_PARSER_S_V.PL ("sentence & verb phrase rules").....	243

B.2: DIPETT_8_PARSER_N.PL ("noun phrase rules").....	273
B.3: A Simplified Prolog Description of DIPETT's Grammar.....	299
Appendix C: Parsing Statistics for the QUIZ Manual.....	324
Appendix D: Example Parse Trees.....	331
D.1: Example Parse Tree #1.....	332
D.2: Example Parse Tree #2.....	334
D.3: Example Parse Tree #3.....	337
D.4: Example Parse Tree #4.....	340
D.5: Example Parse Tree #5.....	342
Appendix E: HAIKU and Situations $\mathcal{S}_1$ to $\mathcal{S}_5$.....	346
E.1: Situation $\mathcal{S}_1$	347
E.2: Situation $\mathcal{S}_2$	349
E.3: Situation $\mathcal{S}_3$	351
E.4: Situation $\mathcal{S}_4$	354
E.5: Situation $\mathcal{S}_5$	356
Appendix F: HAIKU's CA Processor.....	359
F.1: The CA Driver	360
F.2: The Individual-clause CA Processor	373
Appendix G: Examples of Representative Semantic Analyses	388
G.1: Semantic Analysis #1.....	389
G.2: Semantic Analysis #2.....	393
G.3: Semantic Analysis #3.....	397
Appendix H: KB Accumulated by HAIKU in Chapters 1 and 2.....	403
H.1: MDict Entries.....	404
H.2: CMPDict Entries.....	408
H.3: CPDict Entries.....	410
Appendix I: HAIKU's Mini-Network Builder (MNB).....	412
I.1: From Augmented Parse Trees to Conceptual Network Frames	412
I.2: Clause Level Relationships.....	413

I.3: Case Relationships.....	414
I.4: Noun Modifier Relationships.....	415
I.5: Building Mini-Networks.....	415
INDEX.....	419

Symbols and Terminology

This thesis is about an artificial intelligence research project on text processing and knowledge acquisition. It focuses on the elements and components that are particularly relevant to the computational linguistic aspects of the project and which constitute the topic of the thesis. The intended reader is a relatively specialized person with a background in artificial intelligence and in natural language processing (or computational linguistics).

The use of specific symbols and terminology is characteristic of texts such as this. In order to avoid confusion about the interpretation that the reader is expected to give key symbols and terms in the following pages, we now briefly list their intended meaning.

Symbols

The following symbols are abbreviations for a number of terms frequently used throughout the present document. These symbols can be seen as “constants”: there is no distinction between the singular and the plural forms.

AI:	Artificial Intelligence	DCG:	Definite Clause Grammar
EBL:	Explanation-Based Learning	ES:	Expert System
KA:	Knowledge Acquisition	KB:	Knowledge Base
KBS:	Knowledge-Based System	KR:	Knowledge Representation
ML:	Machine Learning	MT:	Machine Translation
NL:	Natural Language	NL:	Natural Language Processing
NLU:	NL Understanding	TP:	Text Processing

Terminology

TP, NLP, and NLU: these three terms are often used interchangeably in the literature, and they certainly have many things in common. In particular, they share the concept of language processing (see [Allen 87], [Gazdar & Mellish 89], [Grishman 86], [Grosz *et al.* 86], [Harris 85], or [Winograd 83] for basic language-processing concepts and definitions). Here, they are also considered as roughly equivalent but with these qualifications. NLU emphasizes the KR and semantic aspects (i.e. meaning as opposed to

form) of NL analysis. TP emphasizes the processing of NL text (not speech, as could be implied by NLU or NLP) as a whole and not merely as a series of isolated or independent sentences. TP may also imply information retrieval. However, in this thesis, we do not consider statistical processing of text, such as concordance analyses, to be part of TP. It is also worth noting that we will not consider the generative aspect of language processing but only its analysis. Although analysis and generation have much in common, e.g. a grammatical theory, they differ on several important practical and technical aspects that we will not discuss here.

KA and ML: KA is “the transfer and transformation of knowledge from the forms in which it is available in the world into the forms that can be used by a knowledge system” [Gruber 89: p.3]. Thus, the term KA does not necessarily imply an automated (or semi-automated) process or method. For example, expert interviews and protocol interviews are manual KA methods. One major aspect of ML is the automation of KA — ML can be looked at as a specialization of KA. Explanation-based learning, for example, is based on the availability of a domain theory, training examples, and an inference engine which allows generalization from the first two elements (see [Michalski *et al.* 83] or [Ellman 89] for basic ML definitions and other approaches to ML). Contrary to KA, ML necessarily implies an *automated* process or method.

Remerciements

Je tiens ici à remercier les personnes qui ont le plus directement influencé mes travaux de recherche ou ma façon de penser et de voir les choses, et ce à différents degrés et de diverses façons, durant les quatre années de mes études doctorales à l'Université d'Ottawa.

En tout premier lieu, le professeur Stan Szpakowicz (Département d'Informatique de l'Université d'Ottawa) qui s'est avéré être un directeur de thèse d'une grande compétence et qui a aussi fait preuve de compréhension et de patience dans des situations parfois difficiles. J'ai apprécié son discernement et la qualité de ses commentaires sur les nombreux écrits qu'il a eu à réviser au cours des dernières années.

D'autres professeurs ont également droit à mes remerciements. Les professeurs Stan Matwin et Doug Skuce (Département d'Informatique de l'Université d'Ottawa), le professeur Jean-Marc Authier (Département de Linguistique de l'Université d'Ottawa), le professeur Emil Petriu (Département de Génie Électrique de l'Université d'Ottawa), et le professeur Franz Oppacher (École d'Informatique de l'Université de Carleton).

Je dois aussi souligner la contribution de Ken Barker, Terry Copeck, et plus récemment, de Jean-François Delannoy, tous impliqués dans les projets TANKA et MaLTe au Département d'Informatique de l'Université d'Ottawa. Que ce soit pour commenter des écrits, échanger des idées, ou mettre à l'épreuve les modules DIPETT et HAIKU, ces personnes ont fourni une précieuse collaboration. Particulièrement Terry Copeck qui, en tant que co-auteur dans quelques écrits techniques sur TANKA ou HAIKU, a participé au contenu et à la mise en forme de plusieurs passages qui apparaissent ici dans le chapitre 4. Terry a aussi réalisé deux modules utilitaires qui font partie de DIPETT: un module d'accès au dictionnaire Collins et un module de conversion des entrées lexicales du format Collins à celui de DIPETT.

Finalement, je voudrais remercier l'Université du Québec à Trois-Rivières, plus spécifiquement le Décanat des Études Avancées et de la Recherche ainsi que le Département de Mathématiques et d'Informatique, qui ont contribué au parachèvement de mon projet de doctorat en m'assurant les meilleures conditions matérielles possibles.

Special Thanks

I would like to thank the members of my thesis committee:

Dr. Harvey Abramson (external examiner),
Institute of Industrial Science, University of Tokyo;

Dr. Jean-Pierre Corriveau,
School of Computer Science, Carleton University;

Dr. Stan Matwin,
Department of Computer Science, University of Ottawa;

Dr. Ingrid Meyer,
School of Translation and Interpretation, University of Ottawa; and

Dr. Stan Szpakowicz (thesis supervisor),
Department of Computer Science, University of Ottawa.

Dédicace

À mon épouse Martine, mon fils Maxime, et ma fille Léanne:
pour avoir fait en sorte que mon univers de ces dernières années
ne se soit pas simplement résumé à celui de mon doctorat!

À mes parents:
pour le support inconditionnel.

CHAPTER 1

1. INTRODUCTION: BACKGROUND AND OVERVIEW.....	1
1.1 The Need for Automated Knowledge Acquisition from Text: A Challenging Task.....	2
1.2 Text Processing and a priori Knowledge.....	4
1.2.1 Knowledge Requirements for Text Processing.....	4
1.2.2 A priori Knowledge.....	5
1.3 Text Processing for Knowledge Acquisition	7
1.3.1 Overview of the TANKA System.....	7
1.3.1.1 TANKA's Functionality.....	7
1.3.1.2 Architecture of the TANKA System.....	9
1.3.1.3 The Current Status of TANKA.....	11
1.3.2 Theoretical Foundations: Goals, Hypotheses and Assumptions.....	12
1.3.2.1 Goals.....	12
1.3.2.2 Hypotheses and Simplifying Assumptions	13
1.4 Main Results of the Research.....	18
1.5 Organization of the Thesis.....	19

1. INTRODUCTION: BACKGROUND AND OVERVIEW

This chapter introduces the reader to the topic of our research by first justifying the goal of automating the acquisition of knowledge from text. Section 1.2 then identifies the three major types of knowledge that are required to perform advanced TP. Section 1.3 presents an overview of the approach we propose for acquiring knowledge *via* TP, as well as its theoretical foundations. Section 1.4 summarizes the main results of the research. We conclude chapter 1 with a brief description of the organization of the remainder of this thesis.

1.1 The Need for Automated Knowledge Acquisition from Text: A Challenging Task

Adaptability to new situations has long been recognized as one of the essential traits of human intelligence — see [Wegnez 87] for interesting definitions of natural and artificial intelligence. This feature reveals the critical role played by the KA and learning abilities of human beings in their impressive facility to adapt to new environments, circumstances, and conditions.

AI has also confirmed the utmost importance of adaptability for computer systems that could be qualified as (artificially) intelligent — a requirement explicitly mentioned in almost every recent textbook on AI. This is particularly evident since the late 70s and early 80s with the work on ES (see [Buchanan & Shortliffe 84] and [Farreny 85] on the MYCIN expert system). After a decade of experimentation with ES and KBS, one of the main conclusions was that the most stringent demand on the construction of KBS is the acquisition of the knowledge that is to be encoded in the KB. This problem has been christened the “KA bottleneck” [Lenat *et al.* 86] and has been the object of active research since 1985 (see for instance [Alexander *et al.* 88], [Diaper 89], and [Hickman *et al.* 89]; see also the special issue of SIGART, #108, on KA).

The initial methods developed for the acquisition of knowledge were essentially unautomated and typically involved a knowledge engineer and an expert whose knowledge was to be extracted and encoded in the ES (manually, by the knowledge engineer). As such methods proved to be quite tedious, error-prone, costly and time consuming, the need for better methods and tools to support KA became obvious. Other factors such as the scarcity of the expertise and the need to dynamically maintain the KB over a long period of time were also influential.

These better methods and tools for KA can be categorized into two main groups: those that are essentially manual (e.g. interview protocols [Kidd 87]), and those that are (to different degrees) automated (e.g. KA assistant [Gruber 89]). The latter group can be further subdivided: tools that support certain parts of interview methods (such as [Belkin & Brooks 88], [Diederich 87], [Diederich *et al.* 88], [Regoczei & Plantinga 88], and [Schmalhofer *et al.* 90]), and tools that are not related to interview methods but, rather, are based on NLP methods and take as input the user's (or expert's) text or excerpts from

textbooks or similar textual material. In the context of the present work, we are especially interested in this second subcategory.

It is our conviction that the automatic acquisition of knowledge from texts is one of the most promising avenues to the construction of truly intelligent systems. Why texts? We here borrow the words of [Nirenburg *et al.* 88: p.24-11] to justify this choice: "The vast majority of organized human knowledge exists in textual form, and human communication always resorts to natural language, regardless of the technological revolution. Therefore, we consider extraction of knowledge from texts to be of utmost importance.". Thus, we propose improving the productivity of the KA process through TP. The types of text that we are especially interested in are expository technical texts¹: we will come back to the processing of technical text in section 1.3.

¹ See [Copeck *et al.* 94] for a characterization of technical texts from a linguistic perspective.

1.2 Text Processing and *a priori* Knowledge

In its most general sense, TP requires phenomenal masses of knowledge. It can be viewed as drawing from three major types of knowledge²: linguistic, common-sense, and domain-specific knowledge. We now briefly examine these types of knowledge and provide a definition of what will be referred to as *a priori* knowledge throughout this document.

1.2.1 Knowledge Requirements for Text Processing

Linguistic knowledge pertains to the language of the text to be processed³. In modern linguistics, this knowledge is organized into several levels [Akmajian *et al.* 84]: morphology (i.e. the structure of words), syntax (i.e. the structure of sentences), semantics and pragmatics (i.e. meaning, language use and communication). Were we to consider spoken language, we would also have to include phonology (i.e. the structure and patterning of the sounds of language). In this document, linguistic knowledge includes morphology, syntax, and semantics. It is located in the DIPETT parser and the HAIKU Case-based semantic analyzer (see details in chapters 3 and 4).

Common-sense knowledge “is too commonsensical to be included in reference books” [Guha & Lenat 90: p.33], or, similarly, it is “the sort of knowledge that an encyclopedia would assume the reader knew without being told ... (e.g., an object can’t be in two places at once)” [Lenat *et al.* 86: p.65]. During the last decade, this “general knowledge” has proven to be important in helping KBS to overcome their inherent brittleness “when confronted by problems that were not foreseen by their builders” [Lenat *et al.* 86: p.65] — see also earlier work in [McCarthy & Hayes 69] and [Hayes 85]. This problem also preoccupies the NLP community, as exemplified by the work of [Barnett *et al.* 90] and [Dahlgren 88], amongst others. For reasons that will be explained in section 1.3, we will not consider this type of knowledge in the present research.

Domain-specific knowledge is specialized in the sense that it is particular to the domain, or world, considered by the text. For instance, in terms of computer applications, this knowledge is specific to the application domain. In a NL interface to a data base

² [Winograd 83: p.14] refers to these as knowledge of the language, knowledge of the world, and knowledge of the situation.

³ In this thesis, we will be concerned with English, although, in theory, it could be any natural language.

system on airplane engines. for example, knowledge about the parts that compose the engines is domain-specific. As discussed in the next subsection and in section 1.3.2, domain-specific knowledge plays a crucial role in this research.

As a final remark on the three knowledge types, it is important to realize that there are no absolute clear-cut boundaries between them, but instead, fuzzy delimitations that are meant to model and facilitate our own understanding of the role of knowledge in TP. In fact, these types of knowledge might have interconnected elements and might even have some elements in common. For example, in the stratified model of language (i.e. levels in the first paragraph above), it is clear that syntax influences semantics. Similarly, semantics and pragmatics both depend, probably to different degrees, on common-sense knowledge and domain-specific knowledge.

1.2.2 *A priori* Knowledge

A serious problem with many NLP systems is their dependency on the availability of complex *a priori* knowledge: knowledge that is domain-dependent and that must be acquired and encoded in the NLP system before it can be used productively. It can be argued that linguistic knowledge, especially morphological and syntactic knowledge, ought to be relatively standard (i.e. domain-independent) and thus should not be considered as real *a priori* knowledge⁴ — unfortunately, reality is not so simple; syntax, for instance, is always defined for a subset of a language and is never totally domain-independent in NLP (see [Grishman & Kittredge 86]).

Even if we suppose that morphological and syntactic knowledge should not be considered as *a priori*, we still face an important problem: semantic knowledge can be considered (at least partially) as domain-specific knowledge and thus should be part of the *a priori* knowledge of a NLP system. What is more important is that semantic knowledge is difficult to acquire, even by skilled users such as linguists or computer scientists. Consequently, this deficiency seriously restricts the adaptability and the portability of NLP systems. This problem has been clearly recognized only rather recently in the computational linguistic literature — see for instance [Ballard & Stumberger 86],

⁴ A subtle type of *a priori* knowledge in certain parsers (e.g. [Magerman & Marcus 91] and [Black *et al.* 92]) is their statistical basis: they often rely on statistical-based heuristics that have been derived during some significant pre-parsing phase which accumulated statistics during the processing of a corpus.

[Ballard 86, 88], [Grosz *et al.* 87], [Ayuso *et al.* 87], [Lang & Hirschman 88], and [Jacobs & Rau 90]).

As will be further explained and justified in section 1.3.2, one of the hypotheses that this research intends to demonstrate is the feasibility and the merits of an approach in which:

- morphological and syntactic knowledge are essentially domain-independent (i.e. not *a priori* as we understand it here); and
- semantic knowledge is incrementally acquired as the NLP system is being used and, thus, need not be considered as *a priori*.

1.3 Text Processing for Knowledge Acquisition

We now turn to the approach we put forward in this thesis. We view this approach as a new computational model-based methodology for incremental KA *via* TP. Amongst other things, we claim that this model:

- helps us better understand the nature of KA and TP, and in particular the relationship between them; and
- provides us with a computational framework for the efficient implementation of semi-automatic KA assistants.

In section 1.3.1, we first present an overview of the TANKA system, a partially implemented prototype of the ultimate model we envisage⁵. Then, in section 1.3.2, we establish the primary theoretical foundations of our approach and, by the same token, those of this research endeavor. Section 1.3.3 will discuss the issue of how to evaluate and measure success in such a research project. *We must emphasize that it is not TANKA as such which is the object of this thesis but, rather, all technical (engineering) and theoretical aspects of its central linguistic components: i.e. DIPETT and HAIKU.* But before getting into the details related to these two components, we must clearly present the context in which they appear.

1.3.1 Overview of the TANKA System

TANKA, which stands for Text Analysis for Knowledge Acquisition, is the name of the current prototype of our model⁶. We now describe its functionality, its architecture and its major modules. We also identify parts which have been implemented, those that are being implemented, and those which remain to be developed.

1.3.1.1 TANKA's Functionality

The TANKA system is designed to process certain types of technical texts in order to incrementally build a KB containing a conceptual model of the domain, i.e. the central

⁵ It must be pointed out that although TANKA is indeed partially implemented, the components we are concerned with in this thesis are fully implemented.

⁶ In some early documents, such as [Copeck *et al.* 90], [Delisle 90a], and [Delisle 91], the prototype's name was KATE instead.

domain covered by a text. We decided to concentrate on technical prose for three main reasons:

- it represents an extremely important type of prose [Kieras 85]: most textbooks and technical documentation (e.g. instruction manuals, maintenance manuals, scientific communications) are technical prose;
- curiously, research in NLP has invested considerably more effort in the study of stories (including dialogues and plans) than any other types of text⁷; and
- compared to other types of prose, technical prose offers several advantages for linguistic processing. In brief, technical prose can be interpreted more literally than other types of prose⁸.

In its normal mode of operation, TANKA is intended to behave as an intelligent assistant to a knowledge engineer who wants to build a KB from a technical text. More precisely, this means that TANKA represents a semi-automatic approach⁹ in which the user helps the system whenever the latter cannot proceed entirely on its own. Typically, this happens when the text fragment being processed:

- contains an unknown lexical item (see chapter 3); or
- is structurally ambiguous (see chapter 5); or
- uses a new or incomplete Case¹⁰ structure, or refers to previously unknown knowledge elements (see chapter 4).

In all these situations, the user must participate in the augmentation of the system's knowledge. Otherwise, he simply supervises the system and may override some of its default choices if the need arises.

When TANKA augments its KB, thanks to user intervention or supervision, the system actually "learns" new things as it incrementally extends its knowledge of the text domain by the dynamic acquisition of knowledge elements during the sequential TP of the input text: we assume that the text can be processed linearly (i.e. one sentence after

⁷ In the Schankian tradition, the Yale school of the '75-'85 period is certainly the best illustration of this observation. Story understanding is especially concerned with the representation of defaults and stereotypical courses of actions.

⁸ These types are also called "registers": see [Biber 89] and [Biber 93].

⁹ A fully automatic approach is currently beyond the state-of-the-art.

¹⁰ We will follow a well-established convention in linguistics which distinguishes 'Case' from 'case'. The first denotes markers for conceptual relations, while the second denotes English cases (i.e. the nominative case for subjects, the possessive case for ownership, the objective case, which includes the accusative and the dative cases, for objects of verbs and prepositions). Notice though that we will not need to refer to English 'case' in the rest of this document.

the other) from the beginning to the end (in other words, the text is so written that sequential reading suffices to understand it without much "jumping around").

We can even envisage a measure of the quantity of knowledge acquired by comparing, at a given moment in time, the final number of knowledge elements to the initial one. When this measure indicates that the rate of learning stabilizes, it probably means that TANKA has acquired a KB sufficiently large and complete to be useful to a KBS, or, more simply, that it cannot extract any new knowledge elements from this particular text.

The KB, which should be initially as small as possible at the very beginning (more on this in section 1.3.2), is represented in a KR formalism derived from Sowa's conceptual graphs [Sowa 84]. This formalism, called Conceptual Networks [Yang & Szpakowicz 90a, 90b]¹¹ and [Yang 93], has been defined for the appropriate representation of activities and objects. The concepts, which constitute the conceptual domain model built by TANKA, represent activities, objects, and properties. This allows the representation of generic knowledge, the type of knowledge we are interested in within the boundaries of TANKA.

Objects play various roles in activities. In a simple sentence with one verb, noun phrases are syntactically and semantically linked to the verb. A link, labelled by a Case (such as Agent, Object or Instrument) represents a role in the activity (such as "who", "what" and "with what"). The linguistic processing in TANKA begins with surface-syntactic analysis (in DIPETT), followed by Case analysis (in HAIKU) that results in a Case pattern of a sentence. The result of these steps is a semantic representation of the sentence. The semantic representation will be further transformed into a small self-contained semantic network, sometimes called a "mini-network", and matched against an ever-growing network representing the system's current understanding of the text.

1.3.1.2 Architecture of the TANKA System

The current architecture of the prototype is represented in Figure 1.3.1.2, below, which shows its main modules and its main data and knowledge bases.

¹¹ All aspects of TANKA that have to do with KR, Conceptual Networks, and the Knowledge Processing Module (see section 1.3.1.2) are the object of another Ph.D. thesis by Leixuan Yang. Therefore, the reader is referred to her publications for further details.

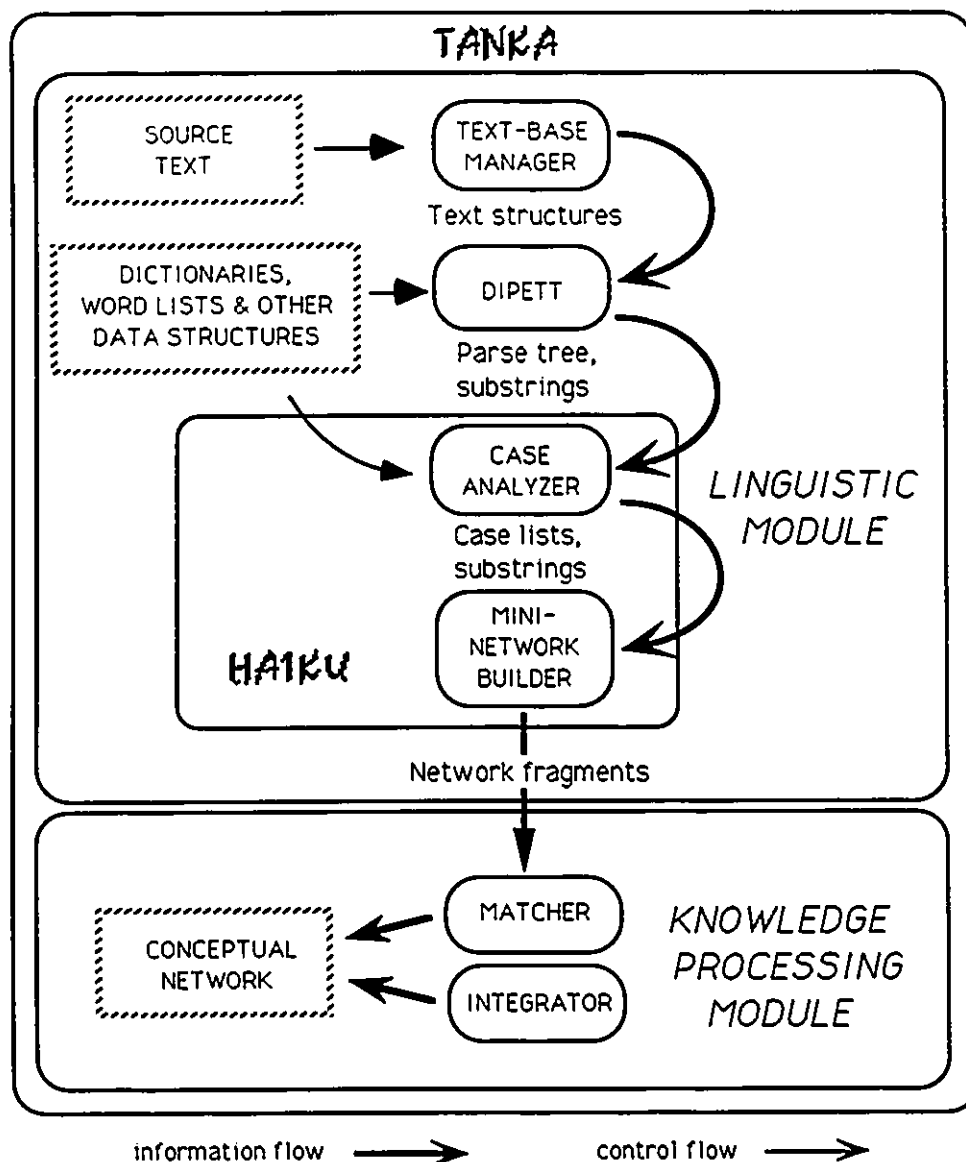


Figure 1.3.1.2. Overview of the TANKA System

The **Text Base Manager** can build a textbase from a text and thus provide flexible access to the text's chapters, sections, paragraphs, and sentences. The **DIPETT** parser covers a broad range of English syntax, including sentence fragments. It encapsulates all linguistic knowledge and related processing; it analyses (on-line or in batch) sentences of the textbase (or of any ASCII file) and produces parse trees that will be forwarded to **HAIKU** for interactive semantic processing. **HAIKU** is responsible for user-assisted se-

semantic interpretation. It extracts Case-marking phrases from DIPETT parse trees and uses its past processing history to select the most likely Case realizations of each phrase with very little *a priori* semantic knowledge — the user validates these selections.

HAIKU outputs Case structures that are then passed to the **Knowledge Processing Module** which will attempt to match, partially if not completely¹², the associated mini-network with the current conceptual network (i.e. KB). If new elements of knowledge are discovered, it will carry out their actual integration with the existing KB.

1.3.1.3 The Current Status of TANKA

The TANKA project is a long-term initiative of the Knowledge Acquisition Lab at the Department of Computer Science, University of Ottawa. Previous publications presented the project's overall design [Szpakowicz 90], discussed aspects of the knowledge representation used in TANKA [Szpakowicz & Koperczak 90], [Yang & Szpakowicz 90a, 90b, 91a, 91b], [Yang 93], and described the parser and the Case Analyzer — or early versions of these components — [Delisle 90a], [Delisle 91], [Delisle & Szpakowicz 91], [Delisle *et al.* 92], [Copeck *et al.* 90, 92].

At the time of writing, the current status of TANKA is as follows: the Text Base Manager and DIPETT are fully implemented; a second version of the Text Base Manager is expected soon. HAIKU's Case Analyzer is also fully implemented and the Mini-Network Builder is in development¹³. The Knowledge Processing Module is partially implemented and will undergo substantial development in the near future.

¹² A total (i.e. perfect) match of the mini-network with the KB indicates that the associated text fragment does not contain any new knowledge. In this case, there is simply no point in integrating this mini-network into the KB as it would not result in the system "learning" something new. On the contrary, a partial match should normally indicate that the associated text fragment does contain new knowledge elements because some of them could not be found in the current KB. This case calls for the integration (i.e. "learning") of these new elements into TANKA's KB. Otherwise, a partial match may indicate that the conceptual knowledge processor did not manage to properly "understand" the concepts in the mini-network and failed to relate them to the appropriate nodes in the KB. This situation calls for the user's intervention.

¹³ The author of this thesis was (and still is) responsible for DIPETT and HAIKU's Case Analyzer, while the Mini-Network Builder is being developed by another person, i.e. Ken Barker, a master's student. See Appendix I for further details on the MNB.

As far as major data and knowledge bases are concerned, their current status is the following. The complete textbase of the user manual of a fourth generation package¹⁴ has been built with the first version of the Text Base Manager. Data bases required for linguistic processing have been implemented or acquired. DIPETT's lexicon concentrates on closed classes while we rely on The Collins¹⁵ and WordNet [Miller *et al.* 90] for other classes or other types of information — more details on these two large repositories in chapter 3. Finally, a conceptual model of QUIZ (one of our domains of application) has been built manually, comprising some 300 concepts, thus representing TANKA's initial KB for this specific domain.

All software is implemented in Quintus Prolog on Sun workstations and some modules are also implemented in AAIS Prolog on Apple Macintoshes.

1.3.2 Theoretical Foundations: Goals, Hypotheses and Assumptions

This section discusses the theoretical foundation of this research. Chapters 3, 4, and 5 will reconsider several of these elements in greater detail, especially those that have to do with parsing (DIPETT) and Case analysis (HAIKU).

1.3.2.1 Goals

With respect to TANKA's intended functionality, as presented above in 1.3.1.1, the main goals of this research are the following:

- 1) Propose a methodology and a computational model of TP that is suitable for the semi-automatic and incremental acquisition of knowledge from technical texts.

¹⁴ This manual is a guide to Cognos' PowerHouse QUIZ report writer, a component of a fourth generation package. It contains seven chapters, which represent some 135 pages, plus three appendices, a glossary, and an index. The text base referred to here is that of the seven chapters plus the three appendices. For reasons that will be discussed in chapter 3, we have restricted ourselves to the processing of the first six chapters.

¹⁵ Our version of the Collins dictionary was extracted from PC-KIMMO [Karp *et al.* 92], which was itself derived from the 1979 edition of the Collins Dictionary of the English Language. It contains 90196 root forms and 317477 inflected forms distributed in these categories: pronoun, preposition, determiner, conjunction, adverb, noun, adjective, and verb.

- 2) Demonstrate the validity, usability, and potential of an approach to TP in which the linguistic module does not rely on extensive or complex *a priori* domain knowledge.
- 3) Assess the merits of the type of output produced by the DIPETT parser (i.e. detailed parse trees) and the HAIKU Case Analyzer (i.e. Case structures) as contributing elements to a KA system.
- 4) Propose an actual computational framework that implements the model in an efficient manner and which readily accommodates experimenting, testing, and future extensions. This goal emphasizes the importance of the engineering side of this research project.

We claim that the accomplishment of these goals will represent a significant step towards the solution of the “KA bottleneck” problem, as explained in section 1.1, by proposing a new approach to the construction of computer-based KA assistants.

1.3.2.2 Hypotheses and Simplifying Assumptions

The above goals imply a number of hypotheses that will have to be verified in this research. Some are specific to DIPETT and HAIKU; these will be reconsidered later in this document. Others are related to TANKA as a whole; these will be considered more conjecturally as they require a definitive prototype in order to undergo serious experimental validation. Let us now consider the key hypotheses.

- 1) TANKA’s linguistic module allows us to transform KB text understanding into KA: indeed, as parsing and Case-based semantic analysis proceeds, new semantic patterns (i.e. not already present in the KB) indicate the need to update the KB. In particular, this is only feasible with broad-coverage parsing since syntax seems to be the only support for meaning in the absence of a rich semantic model of the domain (see chapter 3)¹⁶. Our approach to text understanding is based on the basic principles of proposition-based analysis, in the sense of [Kintsch & van Dijk 78]: the meaning of a text is represented by means of a structure of propositions, where

¹⁶ This hypothesis also says that even for Case-based (or thematic) TP, syntax is important. As mentioned in [Tanenhaus *et al.* 89: p.232]: “We find little or no evidence suggesting that thematic analysis allows the processing system to bypass syntactic analysis”.

propositions are expressed in terms of a predicate, the clause's main verb in our case, and its (internal) arguments.

- 2) As TANKA acquires knowledge from texts, it should become a better (i.e. more intelligent or more knowledgeable) KA assistant. This improvement should be expressible as a function of the number of sentences successfully processed by TANKA. It should be measurable by the quality of TANKA's autonomous processing¹⁷ as well as by the diminishing number of user interventions. The system could then be considered as a minimally self-extending (semi-automatic) learning system.
- 3) Incomplete *a priori* domain knowledge can be equated with having a partial understanding of the domain and should allow us to identify the new knowledge to be integrated in the KB (i.e. to be "learned" by the system)¹⁸. This is quite similar to the manner in which humans seem to proceed when confronted with technical prose¹⁹.
- 4) Our findings for technical prose should also apply to expository prose in general — at least the fundamental elements of our approach. This seems to be justified by the fact that technical prose is simply one type of expository prose. As far as experimentation with technical prose is concerned, the implemented system has been tested on two texts, each on a different domain: one on a fourth generation package, the other on taxes (see details on these in chapters 3 and 4).
- 5) Morphological and syntactic knowledge should not be considered *a priori* since, in principle, it is domain-independent. We intend to show that semantic knowledge

¹⁷ TANKA's autonomous processing is everything that the system can do without requiring the user's intervention (except, perhaps, his approval).

¹⁸ This contrasts with typical approaches in which complete domain knowledge is assumed *a priori* and understanding is equated to the perfect and complete matching of the input's representation with the KB.

¹⁹ This hypothesis seems to be supported by [Kieras 85: p.93]: "... the major feature of technical prose is that it usually contains content that is novel to the reader, as in a textbook, and so the reader cannot apply content schemas to any great extent. This suggests that comprehending technical prose is not primarily a matter of matching the content to a previously known pattern, but rather involves dealing with the passage content at the level of the individual propositions conveying the content.". Also, [Kieras 85: p.103]: "... people can deal with complex technical prose material without full comprehension of it. Apparently they make use of the superficial characteristics of the semantics and the propositional structure. ... main ideas can be derived with only shallow semantic knowledge, and through the use of textual and sentence surface structure ...". Finally, as noted by Kieras, the relative unimportance of pre-stored schemas in technical prose, compared to story materials, is a distinctive feature.

does not necessarily need to be considered *a priori* as it is acquired dynamically by TANKA *via* its linguistic module.

- 6) As we are exclusively concerned with text, not handling non-textual material found in the original text is not a major drawback. Non-textual material refers to pictures, drawings, tables, and so on. This type of material is not made of normal and full text elements. In particular, the resulting KB accumulated by TANKA should be representative of the domain knowledge represented in the text and should not miss any significant part because non-textual material was not considered by TANKA.
- 7) Linguistic processing begins with parsing in DIPETT and proceeds with semantic analysis in HAIKU. This approach can be characterized as "syntax-first". It is based on the hypothesis that the semantics of an input can be extracted (in good part, since natural language is not entirely compositional²⁰) from its syntactic structure: the input can be decomposed, in a principled manner, in order to extract meaning from it. This is often called the principle of *compositionality* in linguistics²¹: "By COMPOSITIONALITY we mean that the meaning of the whole is a systematic function of the meaning of the parts." ([Hirst 87: p.27])²².

In attempting to meet the goals outlined in 1.3.2.1, as well as testing the above hypotheses, we have made several simplifying assumptions that we believe to be not too limiting. Some could simply be considered as hypotheses since only further work will allow us to fully evaluate what consequences they might have on the applicability of TANKA as a KA system.

- 1) As NL text represents knowledge and fully automatic interpretation is currently beyond the state-of-the-art in TP²³, the best we can realistically hope for is a semi-automatic approach.

²⁰ For instance, quantification and negation are notable exceptions.

²¹ This principle is also known as Frege's principle. The (too?) strong version of this principle says that there exists a homomorphism from syntax to semantics.

²² Jackendoff uses another name for the same concept: the principle of grammatical constraint ([Jackendoff 83]). See [Zadrozny 92] who challenges Hirst's definition of the principle of compositionality.

²³ In fact, as suggested in [Regoczei & Hirst 89a] and [Regoczei & Hirst 89b], fully automatic interpretation might simply be impossible to realize, at least in an efficient computerized framework.

- 2) The domain discussed in the text does not require significant expertise to be properly understood²⁴. In the principal example domain we consider, that of QUIZ, some basic knowledge of computer science should be sufficient²⁵. This assumption also implies that the user of TANKA is not expected to be a highly-skilled person. From a practical point of view, we assume that the user has a reasonable knowledge of the domain (i.e. the main concepts but not all the details) plus a reasonable knowledge of English grammar (e.g. what is a verb, a complement, a preposition, a Case).
- 3) The initial domain-specific KB (i.e. *a priori* knowledge) can be reduced to an absolute minimum without rendering the whole approach entirely manual and, thus, depriving it of any significant advantages over more “traditional” approaches to the acquisition of knowledge. That is to say, although we have good reason to aim for a “zero-minimum”, this is simply impossible since no learning can take place without some basic knowledge²⁶.
- 4) The source text is linguistically correct. Thus, no special processing (e.g. ill-formed input strings) will have to be considered. This assumption seems to be justified by the quality of typical didactically-oriented technical texts such as user manuals.
- 5) The source text can be processed sequentially, thus allowing us to incrementally construct the domain-specific KB. As is the case with the previous assumption, this seems to be justifiable by the type of text we consider.
- 6) The fact that TANKA’s KR does not consider (i.e. cannot represent) certain types of information produced by DIPETT²⁷ has no consequence for the validity of the

²⁴ Notice that the work of [Kieras 85] (see footnote 17) seems to justify this assumption.

²⁵ For a simple example of basic knowledge of computer science, see [Nirenburg & Raskin 87] for an elementary taxonomic model of computer science.

²⁶ This is a subtle issue: ‘minimum’ cannot be equated with ‘empty’ here. A fundamental constraint on learning machines is the following [Minsky & Paper 88: p.xiii]: “... one simply cannot learn enough by studying learning by itself; one also has to understand the nature of what one wants to learn. This can be expressed as a principle that applies not only to perceptrons but to every sort of learning machines: No machine can learn to recognize X unless it possesses, at least potentially, some scheme for representing X.”. In other words, no learning whatsoever is possible without some knowledge. In AI, this constraint has often been paraphrased as “You can only learn what you almost already know”.

²⁷ For instance, as already mentioned in 1.3.1.1, it is impossible to represent events other than activities. Or, as another example, it is difficult to represent relative clauses attached to objects (i.e. noun phrases). These apparent limitations are simply the result of design decisions related to TANKA’s KR and TANKA’s bias towards the representation of generic knowledge.

conceptual domain model constructed by TANKA. We base this assumption partly on [Voss 84: p.207]: "a major component of learning from text involves the construction of representations that take into account the most salient features of a text".

1.4 Main Results of the Research

We here list the main results of the research presented in this thesis. All details can be found in the following chapters.

- We provide a computational setting for the efficient implementation of semi-automatic assistants for KA from technical text. Much of the thesis concentrated on the engineering side of the proposal.
- We demonstrate the practicality of a NLP approach to KA in which the linguistic module does not rely on extensive or complex *a priori* domain knowledge, provided a competent user can supervise or guide the system whenever appropriate, especially during the initial “learning” phase since the system will behave more and more autonomously as it accumulates knowledge. In fact, if the system cannot “understand” an input, its KB lacks elements of knowledge that are worth acquiring.
- We demonstrate the appropriateness of a compositional approach using detailed syntactic analysis combined with Case-based semantic analysis: in this context, the surface form of the input text is quite naturally amenable to KA *via* NLP.
- We illustrate in a specific context how the individual problems of NLP and KA seem to share fundamental processing properties.

1.5 Organization of the Thesis

The rest of this thesis is organized into six chapters. The next chapter presents a fairly detailed review of related work. Whenever appropriate, we will be comparing the approach described here with those that have been proposed by other researchers.

Chapters 3 and 4 describe TANKA's main linguistic modules. These chapters present a detailed discussion of the DIPETT parser and the HAIKU Case-based semantic analyzer. Both modules have been designed, implemented, and tested.

Chapter 5 will discuss future work related to DIPETT and HAIKU.

Chapter 6 will conclude by evaluating the contribution of this research as well as its current and prospective applications.

Chapter 7 lists all references used in this document. Every reference is cross-referenced with the associated list of chapter numbers in which they are cited, for ease of consultation.

Finally, a series of appendices complete the thesis. For the most part, they contain logs showing various traces of execution of DIPETT or HAIKU, as well as listings of Prolog programs which are part of the implementation of these modules.

CHAPTER 2

2. TEXT PROCESSING FOR KNOWLEDGE ACQUISITION: RELATED WORK.....	21
2.1 Text Processing: A Quick Survey.....	22
2.1.1 Text Processing: Natural Language as Data.....	22
2.1.1.1 [Munch 86].....	22
2.1.1.2 [Jacobs & Rau 90].....	23
2.1.2 Natural Language Interfaces.....	24
2.1.3 Parsing-Related Issues.....	25
2.1.3.1 Broad-Coverage Parsing.....	25
2.1.3.2 Attachment.....	28
2.1.3.3 Conjunctions and Coordination.....	29
2.1.3.4 Discourse Processing.....	30
2.1.3.5 Fragments.....	32
2.1.3.6 Idioms.....	33
2.2 Acquiring Knowledge through Text Processing.....	35
2.2.1 Semantic and Domain-Related Knowledge.....	35
2.2.1.1 [Grishman <i>et al.</i> 82] and [Haas & Hendrix 83].....	35
2.2.1.2 [DeJong 82] and [Mooney 90].....	36
2.2.1.3 [Frey <i>et al.</i> 83].....	37
2.2.1.4 [Fum 85].....	38
2.2.1.5 [Nishida <i>et al.</i> 86].....	39
2.2.1.6 [Moller 88].....	39
2.2.1.7 [Silvestro 88], [Gomez 89], and [Reimer 90].....	40
2.2.1.8 [Regoczei & Hirst 89a], [Regoczei & Hirst 89b].....	42
2.2.1.9 [Rinaldo 89], [Gao & Salveter 91], [Moulin & Rousseau 90], [Moulin & Rousseau 91], [Cowie 83], and [Rousselot <i>et al.</i> 92].....	43
2.2.1.10 [Lang & Hirschman 88].....	44
2.2.1.11 [Zernik & Jacobs 90], [Smadja 91], [Jacobs 92a], [Brent 91a], [Basili <i>et al.</i> 92], and [Grefenstette 92b].....	46
2.2.1.12 [Ciravegna <i>et al.</i> 92], [Hauptmann 93], and [Kim & Moldovan 93].....	47
2.2.1.13 [Ballard 86], [Ayuso <i>et al.</i> 87], and [Grosz <i>et al.</i> 87].....	49
2.2.1.14 [Liu & Soo 93].....	50
2.2.2 Syntactic Knowledge.....	52
2.2.3 Lexical Knowledge.....	54

2. TEXT PROCESSING FOR KNOWLEDGE ACQUISITION: RELATED WORK

The goal of this chapter is to present a concise review of the work related to TP for KA. Before we commence the review, we wish to warn the reader of two important factors that have affected the following presentation:

- certain areas are covered briefly since they discuss work that is only remotely relevant to this thesis: nevertheless, they were included for the sake of completeness. Other areas are presented in more detail since they are directly relevant and they present some similarities with the objectives or technical aspects of TANKA, DIPETT, or HAIKU.
- our remarks or criticisms about other researchers' work should sometimes be taken with a grain of salt. Our bias is clearly that of practical TP for KA applied to what we refer to as "real texts". In other words, we are looking for KA approaches (or systems or tools) which can be applied (or used) in certain knowledge engineering contexts, using existing, unsimplified¹ technical texts. As well, we tend to be more enthusiastic about implemented systems *versus* high-level designs.

¹ An 'unsimplified' text is a text which has not been preprocessed in order to simplify it so that a given processor (e.g. a parser) can deal with it more easily or correctly.

2.1 Text Processing: A Quick Survey

In this subsection we review, in a rather cursory manner, work relevant to the main direction of this thesis. Typically, such work is considered as pertaining to the field of TP, NLP, NLU, or computational linguistics². We do not consider KA here; it will be discussed in section 2.2.

2.1.1 Text Processing: Natural Language as Data

Research in TP usually focuses on the processing of text, either individual sentences or entire texts (see [Barnett *et al.* 90], for instance). Regardless of whether certain projects are theoretically or practically inclined, most can be seen as describing formalisms, procedures and algorithms that are applicable to a particular type of data, natural language. The goals of TP systems³ often include the capacity to represent and manipulate the contents (the “meaning”) of a NL utterance in order to make it amenable to other types of processing, or to facilitate communication between humans and machines. Some TP system designs are even proposed as plausible hypotheses about the way human beings process NL; these sustain cognitive science claims.

Nowadays, there exist many books on the fundamental and historical aspects of TP research. [Gazdar & Mellish 89], [Allen 87], [Grishman 86], [Grosz *et al.* 86], and [Winograd 83] are well-known references. There are even books, such as [Gazdar *et al.* 87], that are bibliographic repositories of publications in the field. Let us examine two recent approaches that we see as typical in the field of modern TP while being related to the work proposed herein.

2.1.1.1 [Munch 86]

[Munch 86] describes a NLU system for a restricted domain of discourse (i.e. exercises in thermodynamics). Munch’s system transforms texts, which are written in a

² In order to simplify the presentation, we use ‘TP’ generically. It stands for TP itself, as well as for NLP, NLU or computational linguistics. Notice that we do not include the domain of information retrieval, which is considered fairly distinct, although it has recently been borrowing more and more from NLP techniques and principles. For an interesting reference on text processing “at large”, including the information retrieval aspects, see [Salton 89].

³ Representative categories of TP systems include NL interfaces to data bases (see section 2.1.2), machine-translation systems, text analysis systems, and speech understanding systems ([Allen 87]).

subset of Danish, into a representation language based on Cases. The system consists of a morphological processor, a syntactic processor, both domain-independent, and a semantic processor. The latter is language- and domain-dependent. It is controlled by a Case grammar. Case frames relate syntactic structures to a Case system and place semantic constraints on their constituents. A world KB is used to represent these constraints; knowledge is expressed in a *is-a* hierarchy which organizes all concepts in the domain of discourse. Semantic interpretation considers context by resolving some anaphora and ellipses.

In Munch's approach, proposition types are abstract concepts defined in the Case frames. Accordingly, Cases show semantic relationships between proposition types and abstract concepts. Cases are *ad hoc*, i.e. many are specific to the domain of thermodynamic exercises (e.g. 'force', 'stuff'). For each head verb and head nouns, as well as for some other heads such as prepositions, the Case grammar contains a Case frame which consists of the following elements: i) Cases, ii) selections (i.e. relate Case elements to syntactic constituents), iii) semantic constraints (i.e. expressed in terms of the *is-a* hierarchy mentioned above), iv) 'extracts' (i.e. elements from compound or complex semantic element), and v) 'presence' (i.e. specify whether constituents are mandatory, elliptic, or optional). The semantic analysis is performed in a mixed bottom up - top down way.

The output of the system is a list of propositions derived from the input phrase as well as the semantic element which the phrase denotes. This representation is then used by a problem solver in order to find a solution to the exercise — this type of approach to problem solving is also used in BEATRIX for the domain of naive physics ([Novak & Bulko 90], [Bulko 89]). Munch's system does not perform any kind of KA, not even lexical.

2.1.1.2 [Jacobs & Rau 90]

The SCISOR (System for Conceptual Information Summarization, Organization, and Retrieval) prototype system [Jacobs & Rau 90] performs text analysis and question answering in the domain of financial news. It selects and analyzes stories about corporate mergers and acquisitions from an on-line financial service. SCISOR's main tasks are i) the lexical analysis of the input, ii) the structuring of the raw news, iii) the topic analysis of the story, iv) the NL analysis of selected stories, and v) the storage and retrieval of

conceptual representations into and out of a KB. SCISOR can best be regarded as a text retrieval system augmented with a simple question-answering facility (see also [Barnett *et al.* 90], [Dick 92], and [Mauldin 91] for different approaches to text retrieval).

Of interest to us here are the topic analyzer and the NL analyzer. The first can be seen as a kind of filter which applies screening processes to weed out stories not related to the desired topic. After breaking the news into structures such as headline, story, and dateline, the topic analyzer looks for keywords in a pattern-analysis process of the input text and, if necessary, performs linguistics and conceptual analysis. The NL analyzer combines bottom-up, language-driven interpretation (using parse trees), and top-down, expectation-driven processing (using a few key conceptual expectations, i.e. key roles).

SCISOR can only acquire lexical knowledge. It uses a lexicon which, for each lexical entry, includes the linguistic properties of a word's root, and a set of senses which link to a hand-coded and predefined core concept hierarchy (i.e. an *is-a* knowledge base) of a thousand general conceptual categories. Several aspects of the system are not described very precisely in the paper, in particular, the user's participation (i.e. is SCISOR partially or fully automatic?), the parser's coverage (of both syntactic and semantic phenomena), and some issues related to knowledge representation (such as what is and what is not representable in the system's KB).

2.1.2 Natural Language Interfaces

Natural language interfaces (NLI) probably are the best known example of the applicability of NLP techniques and theories in the "real" world. Typically, a NLI is a NL module which acts as a front end to another piece of software such as a data base system (DBS) or an ES — see [Allgayer *et al.* 89] for an example of this latter type of application.

Although many formal query languages have been designed to simplify the problem of getting access to the correct data in the DBS, they often prove quite difficult to use, especially for casual users. Many of the shortcomings of formal query languages can be overcome by extending them and giving more intelligence to the user interface's interpreter. This allows the user to use a more natural syntax in the formatting of his queries. It also liberates him from having to know about the details of the DB. Intelligent

NLI can even tolerate slightly erroneous input strings or recognize certain types of logical inconsistencies.

Typically, a NLI to a DBS contains i) a parser (usually comprised of morphological, syntactic, and semantic modules), ii) a generator of formal queries, and iii) a group of DB access routines. Frequently, in order to improve performance, the grammatical analysis of a sentence is closely dependent upon its meaning, so the grammar is tailored to the system's knowledge structure. In many cases, calls on the data dictionary are written directly into the parser.

Topics of interest in the construction of NLI to DBS include the handling of (restricted) context, data modelling, user modelling, etc. NLI, especially those for DBS, are very well documented in the literature. [Cercone & McCalla 85] and [Wallace 84], for instance, both contain a large number of relevant references and both present a review of the literature. The master's thesis of the author of this thesis was on the topic of NLI, more precisely a NLI to an expert advisor system [Delisle 87]; it also includes a chapter on work related to NLI. We will not go into further details here.

2.1.3 Parsing-Related Issues

In the following subsections, we revise work of a technical nature (i.e. which has to do with implementation issues) that is complementary to several parsing-related issues mentioned elsewhere in this thesis. When appropriate, some of the references cited elsewhere are here presented in a more detailed fashion. Otherwise, we do not duplicate them even if they logically belong to one of the following subsections. Some elements of the work reviewed here are particularly important since they may influence the future enhancements of the DIPETT parser.

2.1.3.1 Broad-Coverage Parsing

As we have said in chapter 1, realistic and broad syntactic coverage in DIPETT is of the utmost importance for TANKA. The design of powerful parsers is a very difficult problem. Despite all the formalisms that have been proposed to describe and process syntax (e.g. generalized phrase structure grammars, lexical-functional grammars), the processing of language remains inherently complex, maybe so complex that we will never be able to come up with a truly efficient and general-purpose NL processor (see [Barton

et al. 87] and [Berwick & Weinberg 84]) — maybe this simply means that our current models are wrong.

Since the beginning of the 1980s, there has been an increased interest in the implementation of practical and efficient parsers; toy parsers could no longer be acceptable for “real-life” applications. Even parsers devised originally for theoretical goals, such as the Marcus parser ([Marcus 80]), have been revised to augment their coverage (see [Berwick 83] for an extended version of the Marcus parser). Some other projects, such as [Heidorn 72], the Alvey project ([Briscoe *et al.* 87], [Caroll *et al.* 91], [Grover *et al.* 93]) the LILOG project [Herzog & Rollinger 91], and [Jensen 89], proposed environments for the development of large grammars or parsers.

The Linguistic String Project [Sager 81] was one of the first large-scale NLP projects. It involved the construction of a fairly large-coverage parser (the Linguistic String Parser) as well as the definition of the Restriction Language⁴ for writing grammars. Their parser was very domain-specific (the domain was that of medical reports) and did not, for instance, handle conjoined phrases or sentences in any general fashion — a feature that we consider essential for a parser to be qualified as “broad-coverage”.

Although this work is certainly of historical importance, its first results later proved to be quite difficult to port to other domains. One (technical) reason was the fact that the original implementation was done in FORTRAN and the system’s design suffered from several important limitations related to that. Nevertheless, revised versions of the original parser and grammar have been (re)implemented with more modern tools and, as a result, some recent projects (e.g. [Lang & Hirschman 88], [Rinaldo 89]) still use NLP modules based on the original work of the Linguistic String Project. One major outcome of this line of work was the PUNDIT (Prolog UNDERstands Integrated Text) system at the Unisys Paoli Research Center, at the end of the 1980s⁵.

As a final remark here, let us simply note that the linguistic coverage of NLP systems is often vaguely described in the literature. It is one thing to claim that one has implemented a large-coverage parser, but quite another to explain what is meant by

⁴ A Restriction Language grammar consists of a context-free component, written in BNF, and a set of restrictions which are predicates on parse trees. See [Sager 81] and [Grishman 86].

⁵ For further references related to the PUNDIT system, see details in [Lang & Hirschman 88: p.50].

'large' and exactly what makes the coverage 'large'. The next chapter provides all the details to support our definition of 'large-coverage' for the DIPETT parser.

2.1.3.2 Attachment

The problem of attachment is best illustrated with a sentence like “The man saw the boy on the hill with a telescope”. What does the prepositional phrase⁶ (PP) ‘on the hill’ modify? What does the PP ‘with a telescope’ modify ? In the absence of semantic or pragmatic knowledge, one must resort to syntax using principles such as Right Association, Minimal Attachment ([Kimball 73]), or other heuristics. This is what we have done for DIPETT with the default rule that PPs should be attached to the main verb unless there are reliable syntactic clues indicating that they should instead be attached to a noun. We will come back to this in chapter 3.

In other situations, other solutions have been proposed. Similarly to what we propose in chapter 5, [Tomita 84] puts forward an approach in which grammatically ambiguous sentences are disambiguated by asking the user. It is based on what he calls Explanation List Comparison⁷ (ELC). It allows one to interact with the user without assuming that he is a linguist and without having to show him parse trees. For example, with the sentence “Mary saw a man with a telescope”, Tomita’s ELC-based algorithm generates the following interaction with the user in which he is asked to select either #1 or #2 as the correct interpretation:

- 1) The action (Mary saw a man) takes place (with a telescope), or
- 2) (a man) is (with a telescope)

[Jensen & Binot 87] suggest the use of definitions in on-line dictionaries to disambiguate PP attachments. They propose to use the definitions themselves, the example sentences (or phrases), the synonyms, and the usage notes and other comments attached to the dictionary entries. To that end, several tools are necessary (e.g. a parser), as well as heuristics and other pattern matching capabilities. Curiously, their paper is vague on one important potential problem: what happens if the dictionary definitions or the example sentences that are to be parsed are themselves plagued with PP attachment problems?

⁶ There are other sources of ambiguity than prepositional phrases (PP). In general, some (non-PP) phrases and clauses can also illustrate the problem of attachment.

⁷ This is a list of ‘explanations’ which are generated from explanation templates of all rules employed during the building of a parse tree. For instance, an explanation for the grammar rule ‘S → NP + VP’ is “(1) is a subject of the action (2)”, where (1) stands for NP and (2) stand for VP in that rule.

[Dahlgren *et al.* 89] propose a preference strategy for PP attachment which is based on ontological⁸, generic⁹, and syntactic knowledge. This strategy is expressed with rules that make use of information about the verb, the object of the verb, and the 'object' of the preposition. Some rules are 'global' (i.e. general), while others are specific to the prepositions. An example of a general rule is:

"comparative --> np_attach(PP): If there is a comparative construction, then NP-attach the PP".

In our view, the major restriction on the applicability of this approach is the mass of *a priori* knowledge it requires, in particular, ontological knowledge.

A similar remark, although to a lesser degree, can be made about the algorithms¹⁰ discussed in [Whittemore *et al.* 90]. In their empirical study, they evaluate a number of attachment 'predictors' (e.g. Right Association, Minimal Attachment, Referential Success) on a certain corpus of text. Based on the results of their evaluation, they define an algorithm which predicts PP attachment. Unfortunately, it includes tests such as "associate adjectives with locative (and possibly temporal) qualities with the nouns they modify". It seems quite difficult to describe, in computational terms, what constitutes a 'locative', a 'temporal', and a 'quality' without assuming extensive *a priori* (ontological, commonsense, world, or whatever) knowledge or even human intervention. Such discussions and proposals on the topic of PP attachment abound in the literature; recent ones are presented in [Agarwal 90], [Bogges *et al.* 91], [Hindle & Rooth 91], [Hobbs & Bear 90], [Jacobs 92b], and [Niv 92]. See also [Altman 88], [Schubert 84], and [Wilks *et al.* 85].

2.1.3.3 Conjunctions and Coordination

Recognized as one of the most difficult problems of NLP, the processing of conjoined elements has yet to receive a general and satisfactory solution. What is so complex about this phenomenon is that it encompasses several problems such as structural ambiguity,

⁸ Ontological knowledge is concerned with things that exist in a given universe and how they relate to one another.

⁹ Generic knowledge is concerned with characteristics, features, properties, etc., that can be associated with classes of individual objects (i.e. instances). This type of knowledge focuses on what is common to all instances of a given class, and not on what makes the instances different from each other.

¹⁰ The authors call them algorithms, but strictly speaking, these are heuristic rules.

elided or ellipsed elements, and fragments. One well-known example is Woods': *John drove the car through and completely demolished a window*. In this sentence, 'John' belongs to the second conjunct and 'a window' belongs to the first.

Generally speaking, the solutions that have been proposed in the literature can be categorized in two groups. First, those that are "grammar-based" (e.g. [Pereira 81], [Dahl & McCord 83], [Hirschman 86], [Dahl 89]). For the most part, these have been proposed in the framework of logic programming (and Prolog in particular): the approach used in DIPETT falls within this first category. Second, those that are "rule-based" (e.g. [Winograd 72], [Woods 73], [Bates 78], [Lesmo & Torasso 85]); many have been proposed in the ATN framework, often implemented in LISP. Both types of solution usually depend on an interrupt-driven mechanism (or exception-handling, sometimes involving "demons"), more or less transparent to the actual grammar rules.

A few solutions are more difficult to strictly classify in the above two groups. [Berwick 83] and [Kosy 86], for instance, each suggest an approach which is more of the "rule-based" type, but in the deterministic framework of the "wait-and-see" Marcus parser ([Marcus 80]). This contrasts with all the other solutions which are either strictly top-down or strictly bottom-up. There are also [Fong & Berwick 85] who suggest an approach of the "grammar-based" type, but which does not use a tree-structure representation. Instead, it uses reduced phrase markers (i.e. sets of monostings¹¹ and terminal strings satisfying certain predicates) and is reversible (i.e. can be used to generate sentences).

There are other variants. For example, [Lavelli & Stock 90] use an expectation-based algorithm that deals with intersentential ellipsis and coordination in a chart-based parsing framework. [Agarwal & Boggess 92] propose a deterministic and domain-independent algorithm for conjunct identification applicable to texts pre-labelled with syntactic and semantic tags, whereas [Kempen 91] presents an inheritance-based approach to the problem of clause-level coordination in Dutch, based on the psycholinguistic model of self-correction.

2.1.3.4 Discourse Processing

¹¹ A monosting corresponds to a symbol of the form T.NT.T where T is a sequence of zero or more terminal symbols and NT is a non-terminal symbol.

The pronoun “explicitization” mechanism — see chapter 3 — is essentially the only discourse processing performed by DIPETT. We could not dedicate more effort to discourse-related issues without departing from the main goal of this research. We now review work related to discourse processing.

Discourse processing is a rich and complex topic in itself. [Grosz 78] presents pioneering work on focus and ellipsis, while a section of [Grosz *et al.* 86] is dedicated to fundamental work on major aspects of discourse processing: reference (including Hobbs’ milestone paper on pronoun reference resolution ([Hobbs 78])), focus, and context. [Brady & Berwick 83] also present basic work ranging from focus and discourse entities to speech acts and intentions.

More recent work, which concentrates mainly on anaphora, is that of [Johnson & Klein 86] (proposes a framework based on Kamp’s Discourse Representation Theory¹²), [Carter 87] (suggests an approach which attempts to resolve anaphoric references with minimal non-linguistic knowledge), [Walker 89] (evaluates and compares two different approaches to the processing of anaphora), [Passonneau 89] (analyzes how discourse anaphoric uses of ‘it’ contrast with those of ‘that’), and [Lappin & McCord 90] (presents three algorithms for resolving anaphora in Slot Grammar).

Also of interest is [Landau 90], which describes a backward search algorithm operating on a LIFO stack of concepts and referents (established on Sowa’s conceptual graph formalism ([Sowa 84])), [Rolbert 90], which presents heuristic rules for identifying a discourse’s theme and finding pronoun antecedents in French, [Whittemore *et al.* 91] which treats “open” (i.e. unfilled) thematic roles as a type of anaphor in order to improve discourse understanding, especially at the intersentential level, and [Reiter & Dale 92] which introduces an algorithm for generating referring expressions in the context of the IDAS NL generation system. For a thorough review and discussion of the literature on anaphora, up to 1979, [Hirst 81] is a must.

Some other projects, which concentrate essentially on focus (or theme-rheme or topic-comment), are those of [Hajicova & Sgall 86] (proposes an approach to the representation of the hierarchy of topic-focus elements in the output of automatic text analysis systems), [Tucker *et al.* 86] (discusses the role of discourse in expository text), [Sgall *et al.* 86]

¹² See [Kamp 88] for a recent overview and discussion of this theory.

(discusses, among other things, the relation between focus and the structure of text — see also [Hajicova & Sgall 93]), and [Dahl 86] (discusses the use of focusing for pronoun resolution and other problems of reference resolution).

Other relevant work can also be found in [Kieras 85] (discusses the particular characteristics of reference in technical prose¹³), [Alshawhi 87a] (proposes a framework for solving interpretation problems related to context and focus), [Ayuso 89] (presents an extension of Webber's model for discourse entity generation, along with a partial implementation), and [Alshawhi 90] (presents an approach to the representation of many reference phenomena and their interpretation in context). See also the special issue of the journal THINK (Vol. 1, #3, December 1992) on anaphora.

2.1.3.5 Fragments

In our discussion of DIPETT in the next chapter, we justify the need to consider fragments and idioms (see 2.1.3). Fragments, for instance, are essential in fragmentary parsing as a last-resort parsing strategy: more details in chapter 3. Although these two phenomena have received more attention in the last couple of years, they are not very well covered in the literature, with the possible exception of ellipsis — a situation that one might find surprising when one considers the importance that both phenomena have in practical language processing (see [Jacobs & Rau 90], for instance). We wish to remind the reader that fragments, which do not frequently arise¹⁴ in well-written text, do appear in our texts when non-textual material (e.g. a table) is removed from the original input text before this text is forwarded to DIPETT. Fragments also appear in itemized lists.

In [Linebarger *et al.* 88], fragments are defined as “regular structures which are distinguished from full assertions by a missing element or elements which are normally syntactically obligatory. We distinguish them from errors on the basis of their regularity and consistency of interpretation, and because they appear to be generated intentionally”. The authors suggest that fragmentary sentences are regular structures which can be

¹³ The author makes interesting hypotheses and observations about the use of reference in technical prose, especially at the syntactic level. Here are a few: 1) many general concepts are usually referred to with plural nouns with no determiners, 2) indefinite noun phrases are relatively infrequent, and usually signal referents that are both textually new and knowledge-new, and 3) references to sets of items are frequent and convey very complicated semantic content as well.

¹⁴ This is not quite right if one considers that elision produces fragments. This is the position held by [Linebarger *et al.* 88] where fragments are defined as elisions.

processed by adding rules to the grammar of a NLP system. They identify several fragment types for which they define corresponding rules that allow parsing to take place.

For example, the type `'nstg_frag'` denotes an isolated noun phrase (noun-string fragment), e.g. *Loss of oil pump pressure*. Its corresponding rule says that “the syntactic parse tree for this fragment type contains no empty elements; it is a regular noun phrase ... The ISR¹⁵ transforms it into a VSO¹⁶ sequence. This is done by treating it as the subject of an element `empty_verb`; ...”. This amounts to saying that fragmentary sentences are parsed by inserting dummy elements into the ISR. In that way, they are no different from full assertions (i.e. non-fragmentary sentences). The authors have implemented this approach in the PUNDIT text-processing system (see also [Lang & Hirschman 88]).

Recently, the notions of fragment processing and “partial parsing” has become more and more important, especially as the basis of a “fallback” approach to the parsing of complicated input strings¹⁷. When an input cannot be analyzed in a reasonable period of time, the processor reconsiders the input as a series of fragments (or chunks) instead of a single sentence, thus producing a series of simpler parse trees instead of a much larger unique sentential parse¹⁸, if the input is grammatical: this is exactly what DIPETT does. In that sense, although each with its own variations, [Grishman *et al.* 92] talks about fitted parse, [Jacobs 90] and [Jacobs 92b] about text skimming and text segmentation, [Hobbs *et al.* 92] about fragments and terminal substring parsing, [McDonald 92] about partial parsing, and [Li *et al.* 90] about segment parsing.

2.1.3.6 Idioms

Idioms, expressions whose meaning is not given by those of its separate words¹⁹, represent a challenging problem to NLP processors. For example, we do not usually want

¹⁵ Intermediate Syntactic Representation (a regularized representation of syntactic structure).

¹⁶ Verb-Subject-Object.

¹⁷ The term “complicated input strings” includes ungrammatical strings (i.e. syntactically incorrect) and extra-grammatical strings (i.e. syntactically correct strings that are not recognized by the parser).

¹⁸ As stated in [Weischedel *et al.* 93: p.374]: “Instead of requiring complete parses, a forest is frequently produced, each tree in the forest representing a nonoverlapping fragment of the input.”

¹⁹ This definition of ‘idiom’ is used to cover figurative expressions (e.g. “She kicked the bucket”) as well as grouping of words functioning as single words (e.g. “even if” as a conjunction). In DIPETT, we consider only the second case, and for closed parts of speech: they may be looked at as “syntactic idioms”. We will sometimes refer to the latter as ‘groupings’. Finally, we note that nominal compounds represent a different phenomenon and that idioms should not be confused with collocations: see [van der Linden 92].

to consider (i.e. parse) “even if” as two separate parts of speech, but rather as a single one (as a conjunction)²⁰. The same applies to expressions such as “The Prime Minister kicked the bucket” where the sentence is usually meant to be interpreted in a figurative sense. These examples illustrate that idioms possess a particular semantic interpretation which sometimes affects the way they may be considered in the syntax: sometimes as locutions (‘groupings’), sometimes as figurative utterances (real idioms). We will not consider figurative utterances in this thesis.

[Dyer & Zernik 86] present an approach to the representation of phrase meanings and the process of acquiring such meanings from examples in context. Their proposal, which covers the problem illustrated above with the ‘kick-the-bucket’ example, is built on two major elements. A phrasal lexicon contains for every phrase a pattern, a situation, and a concept. A conceptual representation involves scripts, plans and goals, relationships, and even considers emotions and attitudes. From our standpoint, the major drawback of their approach is that it requires considerable amounts of *a priori* knowledge and requires the user to provide appropriate explanations (which are themselves in NL!). [Zernik 87] describes a related approach which “learns” idioms based on a ML-derived algorithm.

[Stock 87] presents a different strategy which is based on the principle of activation level and threshold in neural networks. The parser’s interpretation of a syntactic construction is literal unless a certain threshold is crossed by the weight of a particular idiom. In the latter situation, a new process is activated. A more detailed (and improved) version of the 1987 paper can be found in [Stock 89].

The recent special issue of Computational Intelligence (Vol. 8, #3) on “non-literal language” provides an interesting coverage of some of the key problems in this research topic.

²⁰ Notice that we do not want to prevent the parser from considering “even if” as separate parts of speech. In “They will end up even if each of them wins one more”, this would clearly be a mistake. We want the parser to consider potential idioms (i.e. ‘groupings’) where appropriate.

2.2 Acquiring Knowledge through Text Processing

This section reviews work relevant to the essence of this thesis. This is especially true of subsection 2.2.1, which covers references directly related to the main goals of TANKA's DIPETT and HAIKU modules. For the sake of completeness, we also include two shorter subsections on the acquisition of syntactic knowledge (2.2.2) and the acquisition of lexical knowledge (2.2.3). Occasionally, several papers will be grouped in the same subsection when they share important similarities. A final remark: the title of this section says "through TP". We will not elaborate here on pieces of work that do not seriously or realistically consider the problem of TP in its entirety. For instance, [Kaplan 89] and [Dick 92] both assume a manual encoding (or parsing) of the text and proceed directly with semantics. In our opinion, this is too big a simplification from a practical NLP point of view.

2.2.1 Semantic and Domain-Related Knowledge

2.2.1.1 [Grishman *et al.* 82] and [Haas & Hendrix 83]

Although the first paper is not concerned with KA as such, it nevertheless was one of the first to explicitly recognize the importance for NL interfaces (NLI) for the limiting of domain-specific information, and to propose an approach to meeting this objective. The main reason cited to limit domain-specific information was to facilitate portability to new domains. This work was part of the Linguistic String Project (see [Sager 81]).

The authors suggest that the design of the NLI's semantic component be based on a modelling of the structure of information in a domain. This model, which they call a domain information schema, specifies i) a set of semantic classes, ii) the words and phrases which belong to these classes, iii) the predicate-argument relationships among members of these classes which are meaningful in the domain of interest, and iv) some features of these relationships, such as functional dependencies between semantic classes. As they themselves observe, this domain information schema is quite similar to a data base schema.

The approach suggested by Grishman and his colleagues assumes the *a priori* availability of their domain information schema, i.e. it must be constructed before any TP

can take place. In TANKA we propose to go one step further with the semi-automatic acquisition of this model (or a similar one) during the processing of the input text. In that way, we recognize that building such a model is a complex task for which DIPETT and HAIKU will provide useful support.

[Haas & Hendrix 83] can also be looked at as pioneering work in the domain of KA *via* NLP. They suggested the idea of a KLAUS (Knowledge Learning And Using Systems) system, i.e. a system that could “hold a conversation with a user in English about his specific domain of interest, subsequently retrieve and display information conveyed by the user, and apply various types of external software systems to solve user problems.” ([Haas & Hendrix 83]: p. 405). In particular, the system’s aim was to collect and organize aggregations of individual facts for use in question-answering tasks. But their first prototype, called NANOKLAUS, had limited success: “NANOKLAUS is best described as a fragile, proof-of-concept system that was built to establish the feasibility of achieving the broader KLAUS goals.” ([Haas & Hendrix 83]: p.418). Nevertheless, their work laid the foundations of many approaches with analogous goals that were to be proposed afterwards, including our own.

2.2.1.2 [DeJong 82] and [Mooney 90]

DeJong’s work took place in the Yale AI Project and, in particular, was based on Schank’s view of KR for NLP. A major claim of his approach is that learning should be viewed as a fundamental part of the text understanding process. That is, understanding a situation for which there is no existing schema²¹ involves generalizing the new event into an evolving schema. The latter schema is then available to aid in future processing and can even be further refined through that processing. DeJong is really concerned with the automatic acquisition of knowledge structures which are not *a priori* available (more precisely, not directly available but *via* inferencing).

The author puts forward a generalization process, referred to as an explanatory schema acquisition process, which assumes that specialized schemata (i.e. those specific to a given situation) for an event can be generalized by a knowledge-based system to become a plausible and useful schema. When his FRUMP system processes a story, DeJong claims that the representation built during the “understanding” of the text can be

²¹ The term *schema* is used here in a generic sense. It may refer to scripts, frames, or anything that denotes a knowledge chunk.

viewed as an explanation of why the events are plausible. In a sense, his approach can be seen more like a ML approach, as in EBL, to the acquisition of knowledge from text.

In view of our objectives here, DeJong's approach suffers from several limitations. First, although he is interested in the acquisition of new schemata, he relies on the availability of considerable *a priori* knowledge: "background" knowledge and, in particular, all basic schemata (and some are already quite similar to the ones to be acquired) are given beforehand. Second, as is typical of Schankian approaches to NLP, the range of syntactic phenomena recognized by FRUMP is relatively limited; this is definitely a severe drawback for realistic TP since the system should be able to handle unsimplified text that contains a wide variety of syntactic forms in order to decode its semantics.

Mooney's GENESIS system [Mooney 90] shares several similarities with FRUMP. Also using a ML-derived approach, it is aimed at the understanding and "explanation" of short stories — it is not appropriate for the processing of expository texts. As stated by Mooney ([Mooney 90]: p.96): "EBL in GENESIS can therefore be viewed as the acquisition of schemata that allow the system to process narratives using efficient schema-based techniques which previously could only have been understood using inefficient, search-intensive, plan-based understanding.". Similarly to FRUMP, GENESIS depends on the availability of *a priori* knowledge in its "long term store": understanding is impossible if a narrative contains actions that do not suggest already known schemata. This is, in ML terminology, learning of the deductive type [Dietterich 89].

2.2.1.3 [Frey *et al.* 83]

Frey, Reyle, and Rohrer describe a system which translates German sentences into a database. The latter is used as the basis for a question-answering system. The input consists of a complete text, not only isolated sentences. Context and some discourse phenomena (e.g. anaphoric pronouns) are processed with Kamp's Discourse Representation Structures (DRS) (see [Kamp 88]). The system's parser is based on a lexical functional grammar and is implemented in Prolog using the DCG formalism — see 3.3 for more information about this formalism.

This paper has an attractive title, maybe too attractive for its contents if we consider our goal of conceptual KA within the context of TANKA. First, the range of syntactic

phenomena handled by the parser is narrow. Second, it is not clear what *a priori* knowledge is assumed, especially for the DRS-construction rules: those that map syntactic trees into DRS and those that translate DRS into Prolog clauses. Finally, the approach is reminiscent of those usually taken in the building of NL interfaces to data base systems; it is not always clear whether the authors aim at automating the construction of databases²² or KB (or what type of KA is really performed).

2.2.1.4 [Fum 85]

Fum presents the design of ALICE (Automatic Learning and Inference Computerized Engine), a system which should be capable of augmenting its KB by processing NL texts. ALICE has two different modes of operation. In the learning mode it is given a series of Italian sentences forming simple scientific passages. The information extracted from these sentences is integrated with that previously stored in the system's KB. The other mode is consultation: ALICE receives questions about the texts and returns the portion of the KB containing the information needed to answer it. This second mode involves an inference engine.

Fum's paper is extremely vague with regard to several aspects²³. For example, although he mentions that ALICE's linguistic coverage is simple, he does not explain what he means by that (even though the parser contains the following "specialists": morpholexical, syntactic, semantic, quantification, reference, time). It is also mentioned without any further details that the system comes with seed concepts and a seed vocabulary — in other words, the assumed *a priori* knowledge is not precisely identified. Finally, it is not clear at all how (if at all) the user is involved in the functioning of ALICE; there does not seem to exist any interaction once the original text has been given on input.

²² In that case we are more concerned with the representation and acquisition of facts than rules, general or generic knowledge. This distinction is important for text summarization systems (see [Hahn & Reimer 86] and [Nishida *et al.* 86], for instance).

²³ The remainder of section 2.2 will exclude such 'design' papers (see [Arinze 89] and [Nirenburg *et al.* 88], for instance) when they tend to be too general, too vague, or simply redundant in view of other (more detailed) work described here. They do not seem to make a significant contribution to the current state-of-the-art in TP, at least in terms of approaches that can be shown to be practical and realistic (i.e. approaches for which it is shown how they could be implemented).

However, an interesting feature of ALICE is that it is described as a cognitive model²⁴ of human TP with the parallels of KB vs. long term memory and working memory vs. short term memory amongst others. Although Fum's system has several high-level features that are similar to TANKA's (e.g. the augmentation of the KB *via* TP, the availability of only an incomplete KB at the beginning), it is difficult to evaluate it more precisely because the paper is of the high-level design type.

2.2.1.5 [Nishida *et al.* 86]

This paper briefly describes a method for the extraction of knowledge from expository texts based on "global text analysis". This method takes into account the information contained in the intersentential relations (e.g. 'detail', 'additional', 'serial', 'concurrent') and the headings of texts. For the latter, it is assumed that a title shows the main topic of a text and that section headings show local topics which constitute the main topic's attributes.

In this particular approach, text analysis is seen as the identification of the main sentential groups and their assignment of a 'standard' attribute name of a subframe in a specified field. The paper is silent on several major issues: parsing capabilities are not discussed; the KR used is not precisely explained; the *a priori* knowledge is not well identified (although it is clear that it is significant — it is expressed in the system's frames); the user's participation is left unspecified (it seems minimal); and the knowledge extracted by their approach is not precisely characterized (e.g. what can be generalized from the output representation?).

2.2.1.6 [Moller 88]

Moller says that automating text analysis is an important step toward automatic knowledge acquisition. He proposes to analyze book knowledge and structure it in a way that can serve as a basis for further KA. He suggests a method for the selection of relevant knowledge elements in a text, based on interpretation models. The models that Moller uses reflect the conceptual structure of an ES; they form a classification scheme for domain-specific knowledge elements.

²⁴ See [Jurafsky 92] for a discussion of three basic models of parsing or sentence interpretation: the linguistic, the computational, and the psychological paradigms.

Moller's approach consists of two processes: the text coding process (PROPOS) and the epistemological analysis process (EPISTOS). First, the text is semi-automatically transformed into a meaning representation which is expressed in terms of propositions (*via* propositional analysis). In other words, this approach does not rely on parsing. The system supports the user by providing grammatical information (on morphology, syntax, and semantics); the user makes the final decisions as to the most appropriate representation for a given input. Representations (of propositions) are then connected to each other; the final output is a tree-structure for a sentence or a network for multiple sentences.

Second, the output of PROPOS is used by EPISTOS to select particular linguistic concepts (words and statements) that represent knowledge elements and that are relevant to the modelling of domain knowledge. These concepts are attached to the structures of a corresponding interpretation model. The interpretation models in EPISTOS are not domain-dependent; they form generic models which, when filled with domain knowledge, lead to a model of a specific domain. Thus, the main task for EPISTOS is to relate parts of the propositional representation to particular components of an interpretation model. As in PROPOS, the process of epistemological analysis is "secured" by interaction; the final interpretative decision is left to the user.

It is not clear in Moller's paper what has been implemented exactly. Although there seems to be an implemented prototype for PROPOS, and none for EPISTOS, there is some ambiguity as to exactly what functionality has been automated or not²⁵. User's participation is left but vaguely specified. Moller's approach differs from ours in several important ways. There is no parsing of the input text; text analysis is based on propositional analysis, which is a rather difficult and time-consuming activity usually accessible only to highly specialized people. The same remark applies to epistemological analysis. In both cases, the user's participation is very important and quite specialized. Another major drawback in Moller's approach is that it assumes the *a priori* availability of the interpretation model for a given type of ES²⁶.

2.2.1.7 [Silvestro 88], [Gomez 89], and [Reimer 90]

²⁵ In fact, it seems that the number of activities that can be automated in Moller's approach is very limited.

²⁶ It is not clear in the paper whether the approach put forward by Moller is meant to be applicable only to ES applications.

Several approaches have been proposed to cope with classification tasks, which are typical of KB construction activities. We here review three papers that share this goal and use TP as a means of extracting more or less complex classification knowledge²⁷ from different types of textual material.

[Silvestro 88] presents a tool, which he refers to as a knowledge-base acquisition mechanism, that constructs KB by accepting English explanations (i.e. explanatory text) from the user. Silvestro's paper is vague on three major aspects:

- the linguistic coverage of the system's conceptual analyzer;
- the semantic representation output by this conceptual analyzer (although it seems to be based on Schank's conceptual dependencies); and
- the knowledge representation used in the system's KB (the author mentions 'tasks' and 'rules' without any further details).

In addition, Silvestro's approach seems to be restricted to taxonomic knowledge²⁸, based exclusively on the *is-a* relation (e.g. 'a course is an episode', 'a department is an organization'). As is true of many Schankian approaches to NLP, this one has a rather limited syntactic coverage and requires considerable *a priori* knowledge (especially semantic); this knowledge is hidden in the system's KB and the conceptual analyzer's lexicon. The paper does not clearly identify the user's participation once the input text has been entered.

Two somewhat similar systems are those of [Gomez 89] and [Reimer 90] which are also restricted to classification tasks, although the resulting knowledge representation is that of conceptual structures instead of production rules. Gomez's system has a relatively powerful linguistic processor although it seems to be biased towards clauses in which the main verb is 'be' (as they generally express *is-a* relations). Gomez makes a remark about *a priori* knowledge which is similar to that we made at the beginning of this document: "If the KA system is going to be domain-independent, then the system cannot have *a priori* knowledge about concepts which are domain specific". His approach allows the user to define missing (domain-related) concepts by relating these to the already existing ones using the '*is-a*' and '*part-of*' relations.

²⁷ Classification knowledge is concerned with the partitioning of a given universe's objects into a number of classes (or partitions) that are more or less naturally interrelated.

²⁸ Taxonomic knowledge is concerned with the classification of things according to their natural relationships.

Reimer's system, which uses a very simple parser, inductively generalizes the concepts learned from the texts to more generic ones with the help of domain-dependent background knowledge and the semantics of the representation constructs used (i.e. the semantics of the representation model used). This approach is more of the ML type. What is particular though is that he proposes to compute frequencies in order to avoid potential problems with overgeneralization. These frequencies are the relative frequencies with which a slot or a slot entry of a concept representation appears at its immediate subordinates; a slot (or slot entry) of a generalized concept is discarded when its frequency value gets below a certain cut-off value. As with Gomez's system, user interaction is a subordinate issue and is only roughly sketched.

2.2.1.8 [Regoczei & Hirst 89a], [Regoczei & Hirst 89b]

Although this work of Regoczei and Hirst seems to be essentially theoretical, we do consider it here since it gives a different perspective than other work reviewed in this chapter. In [Regoczei & Hirst 89a], the authors lay down the foundations of their approach to NLU and KA from text. They propose a theoretical system called LUKES that accepts English text and descriptions of concepts (with their associated names) on input, and produces a KB (or KB elements) on output. Typically, such a KB could be used by a KBS. The architecture of LUKES contains three levels of processing: verbal, conceptual, sub-conceptual.

Their discussion includes a comparison of several theories of knowledge in text. In particular, they contrast the theories of 'content' and 'representation'. In the first theory, text is seen as possessing a content and the knowledge derived from text is nothing more than the extraction of this content. In the second theory, text is seen as representing knowledge and does not intrinsically contain knowledge. The approach the authors favour is one in which text contains a set of perceivable clues that stimulate the reader to synthesize knowledge and attach it to the text. They refer to that as concept cluster attachment. Thus, LUKES' output (i.e. KB) can now be seen as text with concept clusters attached to it. Knowledge is captured both at the verbal level and the conceptual level.

They also propose an implementable²⁹ version of LUKES that they call LOGOS. This system contains several knowledge-based modules for which a huge *a priori* KB must be

²⁹ However, it is not clear how feasible this design really is. Neither is it clear what exactly is implementable and what can be automated.

acquired. To that end they suggest to use SORTAL, a KA tool to be used by a KA analyst to help set up a KB from a representative text. SORTAL is described in [Regoczei & Hirst 89b] where the problem of building conceptual models from NL input is discussed.

SORTAL's main function is to perform sortal analysis, i.e. to classify concepts, concept clusters, and conceptual structures that are talked about in a text. This classification is based on what they refer to as the "meaning triangle" which has three nodes: the text node relates to words, phrases, and sentences used by an agent to denote a concept; the concept node relates to the concepts internal to the agent and refers to the referent node; the latter relates to the "thing" in the external world (e.g. an object).

In SORTAL, the user is an intrinsic part of the system. SORTAL prepares frames (constructed on the basis of the meaning triangle) from the input text submitted by the user. These frames are then filled by the user. In other words, KA in SORTAL means acquiring new meaning-triangle diagrams associated with the input text³⁰. The resulting 'sortal' KB can then be passed to LOGOS to process documents about the domain and then, finally produce a KB directly usable by a KBS. As the reader can see, Regoczei and Hirst's approach is quite different from TANKA³¹.

2.2.1.9 [Rinaldo 89], [Gao & Salveter 91], [Moulin & Rousseau 90], [Moulin & Rousseau 91], [Cowie 83], and [Rousset et al. 92]

A different type of approach than those reviewed so far is exemplified by the work of [Rinaldo 89] and [Moulin & Rousseau 90] in which TP is reduced to its simplest form. Their systems behave as if they were simply looking for predetermined and fixed patterns (e.g. 'if', 'because', 'when') in the input sentences and use these key words to decompose the original sentences into representations that stand for production rules³² or KB elements. Output representations are still in natural language in Rinaldo's system, and in

³⁰ In fact, SORTAL constructs a special type of conceptual dictionary that will be used by LOGOS.

³¹ In particular, Regoczei and Hirst do not consider parsing in the two papers cited here. We are forced to conclude that their approach does not depend on parsing in any way. It appears that the equivalent processing is performed by the user.

³² This statement is not entirely fair to Rinaldo's system which uses the Linguistics String Parser (see [Sager 81]) and performs simple discourse processing. However, there remains no trace whatsoever of this information (especially syntax and semantics) in the output of his system. This is surprising when one considers that to make such rules amenable to knowledge-based processing, one would use elements of such information to transform the system's output into an appropriate KR.

a structured derivative form in that of Moulin and Rousseau. Such representations are meant to be used by rule-based systems (after some further processing).

Rinaldo's approach is not interactive and requires significant *a priori* knowledge³³ (e.g. all words are assumed to be known and are attributed a conceptual category beforehand). Its main goal is to test the feasibility of a design in which the system automatically generates rules from a scientific (medical) text. The user is expected to verify whether the output rules are correct and relevant. Another instance of this type of approach is the work of [Gao & Salveter 91]: they propose the Automated Knowledge Engineer system which is concerned with KA for classification expert systems.

Moulin and Rousseau's approach is, on the contrary, highly interactive and depends heavily on the user's participation; the user accomplishes all interpretation tasks. In fact, this system looks more like a simple support tool than an advanced NLP system. It assists a human user in deriving deontic rules from prescriptive texts by "semi-intelligently" carving the text, without any attempt to represent (understand) the text segments it manipulates. The system does not seem to perform any detailed parsing nor semantic analysis; it is restricted to stereotyped texts ("prescriptive" texts such as regulations).

Both approaches are useless for the acquisition of knowledge not expressed with rules in the surface form of the input text (i.e. directly in the input text).

Finally, [Cowie 83] and [Rousselot *et al.* 92] take a similar but ostensibly more general approach to the extraction of knowledge from scientific texts. They too aim at the construction of a KB from elements distilled from texts but without NL grammars, parsers, or analyzers. They simply assume that scientific texts are written with a very rigid and predictable syntax and thus propose a number of rules that perform knowledge extraction based on specific constructs and "key words". This is reminiscent of the approach used more than 25 years ago in ELIZA [Weizenbaum 67], and we know very well its intrinsic limitations — see chapter 7 of [Weizenbaum 76] for a stimulating discussion about ELIZA.

2.2.1.10 [Lang & Hirschman 88]

³³ To put it simply, this knowledge is used to determine whether a given input sentence appears to be meaningful (i.e. whether the system should try to extract a rule from it).

The authors describe the SPQR (Selectional Pattern Queries and Responses) module of the PUNDIT text-processing system. Although SPQR seems to be geared mainly towards disambiguation of parsing rather than conceptual domain modelling as in TANKA, the approach seems to have several high-level features in common with TANKA's. The authors too consider the following problems: the incremental acquisition of domain-specific knowledge from text, the use of broad-coverage grammars for TP³⁴, the elicitation of information from the user, and ease of portability to other domains.

Despite these similarities, Lang & Hirschman's approach is different from ours in several important aspects. First, as a consequence of their goal of acquiring domain knowledge to rule out semantically anomalous parses, SPQR essentially acquires selectional constraints. In TANKA, our goal is much more general: we aim at the acquisition of generic knowledge of which selectional constraints can be seen as a subset (although they are not enforced as such in TANKA). TANKA's output should be usable by KBS, whereas SPQR's output seems to be restricted to language parsing considerations. This means that user intervention in SPQR is about the admissibility of selectional constraints and not, as is true of TANKA, about the correctness of conceptual meanings extracted from text.

Second, the selectional constraints acquired by SPQR must be classified by the user as either allowable (i.e. good) or unallowable (i.e. bad). It is not at all obvious that such a dichotomic categorization is always possible or desirable. There might be cases where a mix of 'bad' and 'good' represents more faithfully a somewhat uncertain situation (which may be clarified later with more certainty). In TANKA (or more precisely, in HAIKU) there is no such subjective classification; if a new Case Pattern is found during parsing, the user only has to decide whether it is semantically correct and whether it should be kept in the KB. In HAIKU, if a Case-Marker Pattern already encountered before is found, the user can associate a new Case Pattern with it — see details in chapter 4. In SPQR, once a selectional constraint has been considered, the user does not seem to ever be allowed to reconsider it (even if this would appear to be justified).

Third, SPQR relies on the *a priori* availability of a complete *is-a* model (i.e. KB) of the domain for the generalization of selectional constraints into Case frames. This model

³⁴ They consider the parsing of telegraphic text elements whereas we do not. This probably explains the emphasis of their approach on parsing disambiguation. However, their approach does not seem to consider the problem of multi-sentence fragments (only single-sentence) whereas we do.

is not affected (e.g. updated) by the TP system or the user's participation; it is given beforehand and is purely static. In TANKA, we start with only the highest levels of the domain model and it is constantly updated as text is processed. This remark allows us to doubt that their approach could be applicable to large domains; it is a hybrid between lexically-oriented KA (see 2.2.1.13) and conceptually-oriented KA as we see it in TANKA.

2.2.1.11 [Zernik & Jacobs 90], [Smadja 91], [Jacobs 92a], [Brent 91a], [Basili *et al.* 92], and [Grefenstette 92b]

In recent years, there has been considerable interest in the extraction of information from corpora³⁵ (for instance, see the recent special issue of Computational Linguistics, Vol. 19, #1 and #2) and even dictionaries (for instance, see [Montemagni & Vanderwende 92] and [Dolan *et al.* 93]). One particularly active line of work is that of pattern extraction, be they of lexical, syntactic or semantic nature, depending on the assumed pre-processing (e.g. part-of-speech tagging, either manual or statistical — see [DeRose 88]) or on the availability of more or less significant *a priori* knowledge. Most of these approaches have a statistical basis: their results can only make sense if the number of occurrences of a given (stable) pattern is relatively large — the distribution of such patterns is also significant. And even then, in many cases, it is not obvious how to evaluate the reliability of the results they produce. Several questions remain to be satisfactorily answered: how sensitive are these methods to random error? how do they behave when we attempt to apply them to increasingly smaller texts? what are the criteria for constructing a valid corpus? We now briefly identify some of these approaches.

[Zernik & Jacobs 90] propose to collect word collocations — words that tend to co-occur together, especially around an input's main verb — to improve parsing accuracy and support the assignment of thematic roles in semantic analysis; [Zernik 92] applies this approach to the problem of tagging. [Smadja 91] discusses work similar to that of the two previous authors but suggests filtering raw collocations (which he calls “n-grams” when they contain more than two words) with robust parsing in order to eliminate invalid candidate n-grams. He applies this approach in the context of information retrieval. [Smadja & McKeown 91] show how the approach can be adapted to the problem of

³⁵ One relatively standard corpus is the Brown Corpus. It contains some 500 samples of English text and totals approximately one million words. Another corpus is the LOB (Lancaster-Oslo/Bergen) corpus which consists primarily of British English edited prose and is about the same size as that of the Brown corpus.

lexical selection in language generation. This idea of combining statistics with NLP to perform KA from raw texts is also considered in [Jacobs 92a] where it is said that “the statistical methods themselves must be an aid rather than a replacement for knowledge acquisition” ([Jacobs 92a]: p.182).

[Brent 91a] describes a program that analyzes untagged text and proposes a (partial) list of the verbs it contains accompanied by the (simple) subcategorization frames in which they occur — a similar approach is proposed in [Manning 93]. The algorithm used by Brent is expressed in terms of these three major phases: verb detection, subcategorization frame detection, and subcategorization frame decision. This last step is based on statistical models of the frequency distributions. [Brent 91b] presents a variant of that work which tackles the problem of automatically classifying verbs as either stative or active.

[Basili *et al.* 92] argue that a serious limitation of such approaches is that they “are based on the (strong) assumption that syntactic similarity in word patterns implies semantic similarity” ([Basili *et al.* 92]: p.96). They find that “the major problem with collocations is that reliable results are obtained only for a small subset of high-frequency words on very large corpora, otherwise the association ratio becomes unstable” ([Basili *et al.* 92]: p.97). In this paper, the authors are interested in “learning” selectional restrictions in order to provide support for the design of computational lexicons — see also [Grishman & Sterling 92] who have similar goals and propose a syntax-first strategy for studying word associations based on segmentation parsing and semantic hand-tagging.

A variant of the above proposals is exemplified by the system SEXTANT ([Grefenstette 92a] and [Grefenstette 92b]) which is proposed as an aid to information retrieval tasks. This system implements an extraction technique using segmentation parsing, some simple statistical measures, and no domain knowledge, to produce a list of words related (i.e. similar) to any words occurring in a given corpus. Not only may this output be valuable in an information retrieval situation, but it may also help the construction of lists of synonyms (or quasi-synonyms) for dictionary building tasks.

2.2.1.12 [Ciravegna *et al.* 92], [Hauptmann 93], and [Kim & Moldovan 93]

[Ciravegna *et al.* 92] present the SINTESI system. It extracts knowledge from short (4 or 5 sentences) descriptive diagnostic reports written in Italian, in order to summarize

their technical content and support the constructing of a KB on car faults. The information being sought by SINTESI in typical reports is the main fault, the chain of causes, the chain of effects, and the list of car parts involved. In brief, the system attempts to identify the significant objects (physical entities) mentioned in a report, and then represent the relationships between them. Generally speaking, the approach used in this system is one that is rather characteristic of sublanguage NLP strategies. The Italian syntax covered by SINTESI's parser is rather limited. However, the parser is able to cope with ill-formedness in a realistic manner. Parsing is essentially semantic-driven, looking for specific semantic markers that are stored beforehand in the lexicon, thus guiding the parsing process. All objects that may be of interest in a text are described in a KB that is available *a priori*. So the type of knowledge extraction performed by SINTESI is closer to text summarization than to KA as we see it in TANKA: there is no process of incremental KB augmentation as such.

[Hauptmann 93] tackles the problem of deriving a meaningful semantic representation (i.e. a frame) from its associated syntactic representation (i.e. a tree). The system he presents can function in two modes. In the interpretation mode, the system is given an input parse tree and, using its complete KB and a simple generalization strategy, produces the resulting frame representation. In the "learning" mode, the system is presented with the input parse tree plus the corresponding correct (and unambiguous) target frame representation³⁶ and produces the "mapping" rules that establish the relevant relationships between the two — of course, in the process, the system accesses its KB whenever appropriate. Hauptmann's work is relevant in the sense that his approach is syntax-first as is the case in our own work: we are also in the business of extracting meaning from structure. But the means and the end results are quite different. For one thing, Hauptmann's proposal relies on the *a priori* availability of a complete domain-specific KB. He also assumes that all essential selectional constraints are predefined as slot filler restrictions in the domain knowledge hierarchy: again, there is no KA of the kind we are interested in. It is really not clear how such an approach may be applicable to other domains, particularly to less contrived domains than the one used by Hauptmann, i.e. the doctor-patient domain. He says himself that his system was not designed as an easily portable or adaptable tool for any domain ([Hauptmann 93]: p.15). It may in fact be only applicable to sublanguage situations.

³⁶ One could seriously question whether the cost of creating these ideal meaning representations is not significantly higher than the one of creating the mapping rules themselves, at least for complex NL input strings.

[Kim & Moldovan 93] describe PALKA, a semi-automatic KA system designed to facilitate the construction of a large KB of semantic patterns accumulated from corpora. Broadly speaking, PALKA's goals are similar to those of TANKA, but the details are quite different. First, Kim and Moldovan's approach does not rely on full parsing — not even tentatively — but exclusively on text segmentation: the system cannot even exploit simple syntactic clues, such as which preposition introduces which prepositional phrase (playing a certain semantic role relative to the verb), for instance. Second, the user must select "relevant" input strings to be analyzed by PALKA, which uses a keyword strategy to verify whether the submitted passages appear indeed relevant. Third, once an input has been segmented, the system matches the input's phrasal elements with phrasal patterns found in PALKA's KB to "understand" the input. Fourth, PALKA requires a lot of *a priori* knowledge: a general and domain-specific concept hierarchy, frame definitions telling the system what to look for in a text (along with the relevant keywords), and a dictionary which establishes all the appropriate links between its entries and the hierarchical concepts. Finally, the system can acquire knowledge only on a one-event-at-a-time basis, i.e. all 'bombing' events, for instance, must be processed in a single phase (together) instead of incrementally as they are encountered in the text.

2.2.1.13 [Ballard 86], [Ayuso *et al.* 87], and [Grosz *et al.* 87]

The work described here is concerned with what we have called lexically-oriented KA in 2.2.1.10 — more on lexical knowledge in section 2.2.3. That is, these approaches differ from TANKA in two major ways: i) their main goal is not to acquire a semantico-conceptual modelling of a domain but, rather, to acquire lexical semantic information to facilitate the parsing of the text³⁷, and ii) the input accepted by these systems often consists of simple NL utterances typed in by the user (not text as in TANKA). We now briefly present typical examples of such systems.

[Ballard 86] and [Ballard & Stumberger 86] describe the TELI (Transportable English-Language Interface) database-oriented NL interface which is meant to be customizable by end users. The latter can use TELI's menu interface or its English-like specification facility to define or verify syntactic "Case frames" (i.e. templates) containing information on several types of allowable phrases (e.g. adjective phrases and

³⁷ A good proportion of the early systems of that type were in fact NL interfaces.

prepositional phrases). For example, if the user wants to find out what domain objects can be “in” a county, say, he would make the following English-like request: “a ? can be in a county”.

TEAM [Grosz *et al.* 87] is similar to TELI. It is also a database NL interface and has two modes: question-answering and acquisition. TEAM uses a menu-driven interaction through which the database expert provides information about files and fields in the database, and about the (database-related) conceptual content, the words and phrases used to refer to these concepts. The acquisition performed by TEAM affects the lexicon, the conceptual schema, and the database schema. Case frame acquisition for verbs is realized through the use of sample sentences about which TEAM asks canned questions. For every verb it must “discover” its number of arguments, their optionality, their argument-Case mappings, the kinds of prepositional phrases that can be used with the verb, and the particles that can be used with the verb.

IRACQ (Interpretation Rule ACQuisition) is presented in [Ayuso *et al.* 87]. It is a general NL interface system in which mappings (i.e. rules) between valid English constructs (nouns, verbs, adjectives, prepositions) and predicates of the domain may be defined by entering sample phrases³⁸. The rules are used to block a semantically anomalous phrase as soon as it is proposed by the parser. IRACQ depends on the *a priori* availability of a detailed domain model. This allows the distinction between conceptual information, which resides in the KB, and linguistic information which is used to characterize the subset of English that IRACQ handles.

As pointed out in [Jansen-Winkel 88], questioning the user is stereotypical in TELI and TEAM (see also the lexical acquisition system of KBNL [Barnett *et al.* 90]). Neither system can utilize additional user information and it is not possible to influence any aspect of the knowledge acquired. Compared to these two systems, IRACQ’s acquisition component is more powerful — the price to pay is to have access to a complete KB beforehand. The intended user of IRACQ is clearly more specialized than of the other two systems.

2.2.1.14 [Liu & Soo 93]

³⁸ Sample phrases must be given within the limits of the parser used by IRACQ. The same is true of TEAM. For example, TEAM’s NL processor does not accept verbs that require a sentential complement.

The work discussed in this subsection deserves a special mention because it is related to HAIKU's Case analysis, which is presented in chapter 4, rather than to TANKA's overall goals. The authors propose a method of acquiring domain-independent, thematic knowledge by exploiting syntactic clues from training sentences. The system's goal is to collect valuable information that is necessary to discriminate thematic roles (e.g. to propose the argument structure [Theme Goal] for the verb 'go') in input strings and to, ultimately, build an extensible thematic dictionary.

Syntactic ambiguities are resolved by the user (the "trainer") or simply avoided if the input comes from a syntactically pre-processed corpus. The syntactic properties of the thematic roles, e.g. whether a role can be associated with an animated argument, serve as a preliminary filter to reduce the hypothesis space of possible thematic roles of arguments in training sentences. Other heuristics are also used to further reduce the ambiguities. For instance, the "Imperative Heuristic" says that "imperatives are permissible only for Agent subjects".

Several elements of Liu and Soo's work are similar to some of HAIKU's features: the goal to acquire domain-independent knowledge, the reliance on detailed parse trees, the use of a fixed list of thematic roles, the use of a certain number of syntactic clues and other heuristic rules, and the definition of their thematic structures to be associated with verbs (which is similar to our Case Patterns, see 4.3.1). They also discuss the idea of using thematic expectations for correct and efficient parsing, as we do in section 5.4.

Their proposal also differs from our work in several important ways. First, the list of thematic roles contains only 13 roles. It is not clear whether this list is complete and what claims are being made about its universality. Second, the system does not seem to maintain the equivalent of what we call Case-marker Patterns (see 4.3.1): this is a serious disadvantage as such patterns would allow the system to be even more discriminating during the syntactic-clue phase. Third, their system is designed to find a verb's unique ("good") thematic structure, whereas we consider all different occurrences of thematic structures (Case Patterns). Since their system cannot determine whether an argument is optional or not, i.e. whether it is an adjunct or a required internal argument, this goal of determining only a verb's "good" thematic structure appears questionable: it may restrict the utility of this approach. On the other hand, their system is able to exploit corpora in order to find candidate argument structures (for verbs) that will be used to generate

sentences, which will be proposed to the trainer for further verification of their semantic validity. Finally, interaction with the trainer is well-designed and intuitive.

2.2.2 Syntactic Knowledge

The acquisition of syntactic knowledge is a difficult issue which seems to have received far less attention than the acquisition of lexical (section 2.2.3) or semantic knowledge (section 2.2.1). Typically, syntactic knowledge consists of rules or facts about the structure of acceptable sentences in a certain NL, i.e. grammar rules. We make a clear distinction between such knowledge and that of systems like TEAM or TELI (see section 2.2.1.13) which merely acquire fixed syntactic patterns without attempting to modify the system's grammar or deriving some sort of grammar rule from such patterns.

It seems that comprehensive automated approaches to the acquisition of syntactic knowledge have not yet been proposed, especially through TP — see [Langley & Carbonell 86] for a review from a ML viewpoint; see also the special issue of *Machine Learning* on "Machine Learning and Grammar Induction", Vol. 2, #1, 1987. We here review a few related projects that may show the way to more elaborated computational frameworks in the future. At present, the DIPETT parser does not have any means for (semi-)automatically updating its grammar, but this may change in the future, as we will discuss in section 5.1.

In [Shapiro & Neal 82], the authors describe a KBS that handles the acquisition, representation, and use of syntactic and semantic knowledge. The intended user is an expert in NLP. This system uses Shapiro's SNePS semantic network processing system. The NL subsystem initially "knows" about five basic categories such as 'L-CAT' which stands for 'lexical category' and 'VARIABLE' which represents the class of identifiers which the user will use as variables in his NL rules.

Before entering his rules into the system, the user must inform it of all members of L-CAT and VARIABLE categories which he will use. For example, "NOUN is an L-CAT". As we can see, the user's inputs are in NL. This implies that the system already has a kernel grammar which allows it to parse the user's NL input statements (such as "NOUN is an L-CAT"). Once parsed, the user's input is kept as a SNePS network structure in the system.

A sample rule that the user might enter is (user-defined categories are capitalized): "If a string consists of a G-DETERMINER followed by a NOUN called the MOD-NOUN followed by another NOUN called the HEAD-NOUN then the string is an NNP". The user's rule-statement is read by the system and processed by existing rules. When it has been completely analyzed, a translation of the user's rule is asserted in the form of a network rule structure. This rule is then available to analyze further user inputs. We observe that even if the user had to explicitly enter the new rule (instead of the system, say, "guessing" at it *via* TP), the latter is dynamically added to the previous rules thus augmenting the syntactic coverage of the system's grammar.

Another, completely different, approach is that of [Berwick 85] (in particular, chapter 3) where an acquisition model is hypothesized. Basically, the model says that syntactic expertise is acquired through the successive refinement of an initial parser, each new version being more elaborate than the previous one. Central to Berwick's model is an incremental and failure-driven acquisition procedure. When the system cannot parse an input with its current grammar rules, it stops and call upon the acquisition procedure. "The acquisition module tries to build a single new parsing rule that will bridge the current gap in the parser's knowledge. If it can succeed, then this new rule is added to the parser's data base, perhaps after generalization with existing rules. If it does not succeed, then the analysis of the sentence is abandoned for the moment. This process is then repeated for the next input sentence." ([Berwick 85: p.11]).

Although Berwick's model is very interesting from a cognitive science point of view, it is far less obvious how it could be used in a real KA application. In fact, since the publication of his book, we have not seen any practical follow-up on this approach. It is worth mentioning that the model was implemented in a computational framework based on transformational grammars and Marcus' parser ([Marcus 80]). The approach's initial state assumes the availability of a relatively important base of *a priori* knowledge, a part of which is semantic by nature (see [Berwick 85: pp.50-1]). For a knowledge-intensive approach with similar ultimate goals but which lies at the other end of the spectrum in terms of the generality of the knowledge acquired, see [Gomez 91] which attempts to acquire "verb patterns", amongst others, instead of general syntactic rules. See also [Simmons & Yu 92] where the authors introduce a new formalism called "context-dependent grammar" which is used to interactively acquire context-sensitive grammar rules for the parsing of a sublanguage.

2.2.3 Lexical Knowledge

In the last decade, the acquisition of lexical knowledge has become a topic of its own in computational linguistics. The following is necessarily partial and is only provided in order to give an overview of some of the most common approaches used in this field. Again, as in section 2.2.2, we do not reconsider systems such as TEAM and TELI (see section 2.2.1.13); in our view, they do not really perform automatic KA from text. As will be seen in the next chapter, DIPETT's simple lexical acquisition processor relies on The Collinsdictionary and functions semi-automatically (when parsing on-line) or automatically (when parsing in batch).

The special issue of Computational Linguistics on the lexicon (Vol. 13, #3-4) contains several papers on the acquisition of lexical knowledge from the (machine-readable) LDOCE tapes ([Longman 87]). [Alshawhi 87b] suggests a method to process word sense definitions from LDOCE in order to provide basic semantic information to a NL processor so that it can cope with occurrences of unknown words.

[Boguraev & Briscoe 87] discuss how they have restructured LDOCE into a LISP-readable format using the Longman grammar code system *via* a transformation program. The output of the latter can then be used to automatically derive entries which are appropriate for particular grammar formalisms (e.g. PATR-II).

[Byrd *et al.* 87] present a number of tools and methods for processing machine-readable dictionaries. They used LDOCE as well as several other ones (for English and Italian). DAM, one of the computerized tools they present, is a Dictionary Access Method which provides both random and sequential access to dictionary entries³⁹. Wilks and Slator have worked on a project similar to the ones above; they also use LDOCE (see [Slator 89]) to produce text-specific lexicons.

In the special issue of SIGART on KA, [Velardi *et al.* 89] present the PETRARCA system which acquires the definition of a word in the form of a catalogue of 'Cases' that they call surface semantic patterns. Their hypothesis is that these patterns reflect what humans commonsensically understand of a sentence. PETRARCA acquires such patterns

³⁹ One of the practical problems of using very large dictionaries for NLP is how to efficiently access their contents without assuming that the whole dictionary resides in main memory. This problem is discussed in [Boguraev & Briscoe 87].

by inductively looking at examples contained in a given NL corpus: to produce an entry in the semantic lexicon, their system analyzes a large sample of sentences including the new word to be acquired. We remark, though, that this type of lexical KA should not be confused with the problem of identifying unknown words during parsing (for instance, see [O'Shaughnessy 89] and [Erbach 90]) as it happens in DIPETT.

In order to perform this kind of inductive acquisition, PETRARCA needs two major sources of *a priori* knowledge: a set of rules that associate a possible semantic pattern (i.e. a conceptual relation) with each syntactic pattern, and a set of selectional restriction⁴⁰ rules on conceptual relations. Once the system has produced the list of surface semantic patterns that it inductively "generated", the user is expected to verify each of them and insert the acceptable ones in the lexicon. For a similar although more general and more knowledge-intensive approach, see [Jacobs & Zernik 88]; for a different approach based on a ML-derived strategy but also involving the user, see [Pustejovsky & Bergler 87].

Other relevant aspects of lexical acquisition and lexicon construction are presented in [Zernik 89], which describes a learning algorithm that exploits on-line texts and existing lexical categories; in [Velardi *et al.* 91], which proposes an algorithm to learn "syncategorematic" concepts (i.e. concepts' patterns of use) from text exemplars; in [Pustejovsky 91], which discusses several important aspects of lexical semantics and proposes some useful solutions for computational frameworks; in [Paquin & Dupuy 90], which presents the SATO (Système d'Analyse de Textes par Ordinateur) text-analysis system, used for constructing a concept dictionary for an expert system; and in [Cunningham & Veale 91], which argues for a unique representation formalism in the parser's lexicon and in the system's KB, thus making conceptual information available during the parsing process.

⁴⁰ On selectional restrictions: see [Fass 89b] for a review of related work and a discussion.

CHAPTER 3

3. DIPETT: PARSING TECHNICAL TEXTS WITH A DOMAIN-INDEPENDENT, BROAD COVERAGE OF ENGLISH	58
3.1 DIPETT as Part of a Text Processing System.....	59
3.2 Linguistic Considerations.....	61
3.2.1 Parsing a Substantial Subset of a Natural Language.....	61
3.2.2 Determining the Linguistic Coverage of the Parser (or DIPETT as a computerized model of a subset of [Quirk <i>et al.</i> 85]).....	62
3.2.3 A Precise Characterization of the DIPETT Parser.....	63
3.2.3.1 Linguistic Theories and Parsing: Taking a Neutral Stand.....	63
3.2.3.2 Algorithmic Aspects of Parsing.....	64
3.2.3.3 A Typical Parse Tree.....	67
3.3 The Architecture and Components of DIPETT.....	69
3.3.1 The Driver and the User Interface.....	69
3.3.2 Lexicon and Dictionaries	70
3.3.2.1 DIPETT's Lexicon.....	70
3.3.2.2 The Collins Dictionary and the WordNet Lexical Database.....	74
3.3.3 The Scanner and the Lexical Analyzer	74
3.3.4 The Syntactic Module	77
3.3.4.1 Encoding English Grammar with DCG Rules.....	78
3.3.4.2 Definite Clause Grammars and Other Logic Grammar Formalisms.....	78
3.3.4.3 The Grammar's Main Non-terminals.....	81
3.3.4.4 The Influence of [Quirk <i>et al.</i> 85] on the Resulting Grammar.....	83
3.3.5 Subsidiary Components.....	84
3.3.5.1 The Memorizing Device and the Helper Mechanism.....	84
3.3.5.1.1 Notes.....	85
3.3.5.1.2 The Memorizer.....	85
3.3.5.1.3 The Helper.....	87
3.3.5.2 The Trace Mechanism.....	88
3.3.5.3 The Tagger.....	89
3.4 Syntactic Coverage: Difficult Issues.....	91
3.4.1 Structural Ambiguity.....	91
3.4.2 Time Constraints for Problematic Input Strings	92
3.4.3 Parsing Conjoined Structures.....	94
3.4.3.1 Conjoined Structures Handled by DIPETT.....	94
3.4.3.2 Top-Level Parsing Heuristics.....	95

3.4.3.2.1 Top-Level Rule Selection.....	96
3.4.3.2.2 Parsing Coordinated Clauses with Look-Ahead	98
3.4.4 Parsing Fragments.....	99
3.4.4.1 Fragments as Naturally Occurring Text Objects.....	99
3.4.4.2 Parsing an Input as a Series of Fragments.....	100
3.5 Simple Anaphora Resolution for Pronouns: the Pronoun Explicitizer	104
3.6 Testing the Syntactic Coverage with Actual Texts: Results and Discussion	108
3.6.1 Tuning the Parser with Hand-made Input Strings.....	108
3.6.1.1 Test Cases for Noun Phrases and Nominal Clauses.....	108
3.6.1.2 Test Cases for Verb Phrases and Sentences.....	109
3.6.1.3 Miscellaneous.....	109
3.6.2 The QUIZ User Manual	109
3.6.2.1 The Document.....	110
3.6.2.2 Results.....	111
3.6.3 The T4043 Tax Guide.....	111
3.6.4 Discussion	114

3. DIPETT: PARSING TECHNICAL TEXTS WITH A DOMAIN-INDEPENDENT, BROAD COVERAGE OF ENGLISH

As explained in the first chapter, this research is concerned with the application of NLP to the task of acquiring knowledge from real¹ technical texts. The parser we describe here cannot rely on rich *a priori* domain-specific knowledge: it must be usable despite the scarcity of semantics. The handling of this serious constraint represents a contributing factor to the originality of the work presented in this thesis. We now present a detailed description of DIPETT (Domain-Independent Parser for English Technical Texts), TANKA's parser. Earlier descriptions of the parser, less detailed, can be found in [Delisle 90a, 91], [Delisle & Szpakowicz 91], and [Copeck *et al.* 92].

Before we get to section 3.1, we must clarify the meaning of the word 'parser' as it is understood in this thesis. The theory of compiling, in computer science, says that a compiler converts a string of characters into a string of bits, the object code ([Aho & Ullman 72: p.59]). In what follows, parsing includes the phases of lexical analysis and syntax analysis but excludes those of code generation and code optimization. DIPETT's task is to verify whether a given NL string is grammatical, according to its grammar of English, and, if so, produce the structure under which the input string is admitted by the grammar.

¹ As explained in chapter 2 (see also the first footnote in that chapter): 'real' means "existing and unsimplified".

3.1 DIPETT as Part of a Text Processing System

In order for TANKA to semi-automatically process technical text and incrementally build a conceptual model of the domain, its parser must rely on two important assumptions, as stated in chapter 1. First, the process of linear knowledge-based text understanding can be converted into incremental knowledge acquisition. Second, the text is linguistically correct and fully describes the domain.

The parser should accept most sentences found in the text. Without such a broad-coverage parser, automatic acquisition of knowledge from text simply cannot be realized or is bound to be very incomplete. This is because without a rich semantic model², syntax is the only support for meaning; the broader the parser's coverage, the better the representation of meaning³. DIPETT is a surface-syntactic and one-sentence-at-a-time⁴ parser with a large coverage of technical English⁵: the parser handles, fully or partially, approximately 90% of the sentences in our sample texts — a guide to a fourth generation language and a tax guide.

Discourse-related and semantic phenomena that we will not handle in TANKA are speech acts, beliefs, intentions, plans and goals, inference, and dialogue-related phenomena. Thus, DIPETT will have no provision for these. We justify this decision by two major observations. First, well-written expository technical texts are generally explicit and resorting to much implicit knowledge is neither necessary nor desirable. Second, as mentioned before, it is an important hypothesis of our approach to knowledge acquisition from text not to depend too importantly, if at all, on *a priori* domain knowledge; all the above phenomena do require considerable *a priori* knowledge or complex inference mechanisms to be handled appropriately. We do not consider dialogue-related phenomena because they do not normally arise in the kind of technical text we are interested in within this research project.

² It is essential for us *not* to assume any such *a priori* model. Indeed, we remind the reader that our goal is to (semi-automatically) construct that model!

³ This is not to say that syntax and semantics are the same. We only maintain that, in the absence of the domain's detailed semantic model, syntax is the next best way of getting at the meaning. See hypothesis #7 in section 1.3.2.2.

⁴ Other alternatives such as discourse-oriented strategies would go against our "no *a priori* domain knowledge" requirement. The only discourse-level phenomenon that we deal with is that of simple anaphora: see section 3.5.

⁵ More precisely, a large coverage of those subsets of technical English considered within the boundaries of this research.

As section 1.3 has already described TANKA and DIPETT's role within it, we will not go into further details at this point. DIPETT is also used as part of the MaLTe machine learning and text processing system: more on this in section 6.2.

3.2 Linguistic Considerations

3.2.1 Parsing a Substantial Subset of a Natural Language

In what follows, we talk about processing a *substantial* subset of a NL. This subjective characterization can apply informally to any subset which covers every fundamental syntactic structure [Winograd 83: p.468], and, for our purposes, most syntactic phenomena encountered in technical texts we considered in our project. Many NLP textbooks present only insubstantial examples: they would only be subsets of DIPETT's coverage.

Once a substantial subset of a desired NL has been determined (not a trivial task — see the next subsection), a parser must be designed to cover that subset in a manner that is computationally efficient and, as much as possible, easily understood by humans. Most known efficient parsing techniques for programming languages are insufficient for NL. (One exception is Tomita's extended LR parsing algorithm [Tomita 86], but good performance is not guaranteed for realistic texts.) Similarly, additions to context-free grammars required for describing NL phenomena drive the efficiency down (see [Barton *et al.* 87] and [Berwick & Weinberg 84]): the universal recognition problem for agreement grammars⁶ is *NP*-complete.

This result applies to any parser written with any specific grammar formalism for NL parsing, be it DCG, LFG, GPSG, or HPSG⁷, etc. This simply means that the task of parsing a substantial subset of any human language is a very difficult one: "... intractability can arise from a particularly deadly combination of agreement and ambiguity. Informally, human languages and the *NP*-complete SAT problem share two costly computational mechanisms: both enforce agreement among terminal symbols across unbounded distances and both allow lexical ambiguity. ... Thus, the linguistic mechanisms for agreement and ambiguity are exactly those needed to solve satisfiability

⁶ Agreement grammars (AG) are context-free grammars whose nonterminals are sets of feature-value pairs. DIPETT's DCG is a degenerate case of feature-value system in which the features correspond to argument positions ([Shieber 86: p.9]). AG can also be thought of as a restricted form of attribute grammars.

⁷ For a gentle introduction to Definite Clause Grammars (DCG), Lexical-Functional Grammars (LFG) and Generalized Phrase Structure Grammars (GPSG), and Head-driven Phrase Structure Grammars (HPSG), see, respectively, [Pereira & Shieber 87], [Sells 85] and [Shieber 86], and [Pollard & Sag 87].

— any linguistic theory that uses them, as any descriptively adequate theory must, will probably be computationally intractable.” ([Barton *et al.* 87: p.96])⁸.

The main agreement constraint in English is that of number: a subject must agree in number with the verb, a determiner must agree in number with the noun, and so on. If we were to parse French instead, we would have to account for gender agreement in addition to number agreement. As far as ambiguity is concerned, it abounds in NL. As discussed in [Landau 90], syntactic analyzers using no semantics, such as DIPETT, are confronted with many sources of ambiguities in a text: lexical, structural, anaphoric, elliptic, and even semantic for polysemous lemmas. We will see later how DIPETT deals with these difficult issues.

3.2.2 Determining the Linguistic Coverage of the Parser (or DIPETT as a computerized model of a subset of [Quirk *et al.* 85])

No clear, well-established methodology exists for determining the linguistic coverage of a NL parser. In the case of the TANKA system, our goal was to define a substantial domain-independent subset of English in order to cover expository technical texts. We had to start from scratch because grammars presented in the application-oriented NLP literature are all either too narrow, or described in an *ad hoc* fashion and difficult to understand and extend, or else too domain-dependent. This is a demanding enterprise: the author has invested approximately a year just to devise the parser’s core, including the definition of a large set of DCG rules to encode this knowledge, but excluding extensive testing and fine-tuning. This grammar design involved studying English grammars, especially [Quirk *et al.* 85], as well as modeling and formalizing a significant subset of this language.

We first defined the core of DIPETT’s grammar, using both general and NLP-oriented English grammar descriptions: in particular, [Attal 87], [Baker 89], [McCawley 88], [Stockwell *et al.* 73], [Winograd 83], and, most of all, [Quirk *et al.* 85]⁹. The resulting core grammar was compared with other grammars that have been proposed in

⁸ As is well known in computational linguistics, the deeper the linguistic analysis, the greater the decrease in performance and robustness! This is sometimes called the “performance paradox”.

⁹ Although this huge grammar of English — some 1800 pages — is not especially tailored to computational linguistics needs, it was nevertheless chosen as the main reference for two important reasons: 1) it is very exhaustive, comprehensive, and well organized, and, 2) in recent years, it has become a *de facto* standard in the field of computational linguistics.

the NLP literature (e.g. [McCord 82] and [Robinson 82]), in order to ensure that we had a superset of those (usually) simpler grammars. Finally, the syntactic phenomena found in our texts (i.e. the QUIZ manual and the Tax Guide) were analyzed, and compared with the coverage of our grammar. Discrepancies had to be explained and some resulted in the addition or modification of a grammar rule, especially when the actual testing took place.

The salient point is the importance we see in getting substantial domain independent coverage by considering published large grammars on which most of DIPETT's linguistic knowledge is based — this contrasts sharply with the approach in [Delisle 87] where the parser was strongly influenced by domain-related text samples. As a result, DIPETT's grammar represents a computerized model of a relatively large subset of [Quirk *et al.* 85]. See section 3.3.4 for more details.

3.2.3 A Precise Characterization of the DIPETT Parser

The purpose of this subsection is to characterize the DIPETT parser as succinctly as possible, before we plunge into the lower-level details in section 3.3. In order to do that, we examine three important facets of the parser. In 3.2.3.1, we state its position relative to modern linguistic theories. In 3.2.3.2, we classify it using a grid for NL parsers recently circulated in a "usenet group". We present a typical parse tree produced by DIPETT in 3.2.3.3 to conclude this characterization.

3.2.3.1 Linguistic Theories and Parsing: Taking a Neutral Stand

Linguistic theories of grammar, e.g. HPSG, use feature structures to encode linguistic objects: the form and content of these structures depend on the underlying theory. Most theories of this sort state universal principles that are assumed to hold for any NL, as well as language specific principles, such as word order, which hold for one or more languages¹⁰. In this sense, DIPETT is not committed to a particular existing linguistic theory. However, this does not mean that DIPETT has nothing in common with some of these theories, simply that our parser represents an implicit realization of an eclectic linguistic theory.

¹⁰ In modern linguistics, GB theory ([Radford 81] and [Chomsky 86, 87]) is a good example of such a broad theory. In fact, it is more a theory of language in general than "simply" of grammar. GB theory has significantly influenced the field of computational linguistics in recent years: see, for instance, [Stabler 86, 87], [Wehrli 88], and [LeBlanc *et al.* 93].

DCG ([Pereira & Warren 80]), the grammar formalism used in DIPETT, is theory-neutral — this is also true of other formalisms like FUG and PATR-II, for example¹¹. It does not impose any (linguistic-theory based) restrictions on the operations and manipulations of its data structures. Rather, DCG is seen as a very high-level programming language that is well adapted to the task of encoding a particular grammar, or even linguistic theory — see chapter 1 of [Herzog & Rollinger 91] for similar considerations. Most of the rules that encode DIPETT's knowledge of the English grammar are written in the form of DCG rules¹². If a parallel must be established, we would say that DIPETT may be considered as a functional grammar¹³, in the sense of [Kay 85], i.e. a grammar in which syntactic analysis is based on syntactic roles instead of only their strict position in the surface string.

In fact, it is the author's opinion that since the problem of parsing an NL is almost certainly *NP*-complete, and since no particular linguistic theory of grammar and no particular grammar formalism has been shown to clearly outperform its competitors in general, and, finally, since the context in which our parser evolves does not have to defend a particular linguistic theory, then there is simply no point in preferring one linguistic theory over others. For this reason, in accordance with Jurafsky's classification ([Jurafsky 92]), DIPETT clearly falls under the computational paradigm of parsing, as opposed to the linguistic paradigm or the psycholinguistic paradigm. For an example of the latter paradigm, see [Huyck & Lytinen 93].

3.2.3.2 Algorithmic Aspects of Parsing

Following a classification recently circulated in the 'comp.ai.nat-lang' newsgroup [Covington 93], we now describe the main technical and algorithmic facets of the DIPETT parser. Text in the Bookman typeface below was extracted from Covington's classification; bullets identify DIPETT's characteristics and the associated information is in the Times typeface.

A. Criteria relating to grammatical relations

¹¹ For an easily-understandable introduction to Functional Unification Grammars (FUG) and PATR-II, see [Shieber 86]. It is worth noting that the precursor of PATR, DIALOGIC, was introduced in [Robinson 82].

¹² Thus, DIPETT is not a lexicalist approach to NL parsing, i.e. it does not locate the most part of its syntactic (and semantic) knowledge in elaborate lexical structures.

¹³ Functional grammar is the precursor of Unification Grammar which is itself the precursor of Functional Unification Grammar (FUG).

(1) Constituency or dependency?

Is the goal of parsing (a) to segment the input string into constituents, or (b) to link heads to their dependents (arguments)? Parsers based on X-bar theory of course do a combination of the two.

- DIPETT: constituency

(2) Continuous or discontinuous phrases?

Does the parser require every constituent to be continuous? Some free-word-order parsers do not. (Applicable to dependency as well as constituency grammars: in dependency grammar, a constituent consists of any head plus all its dependents, their dependents, and so on recursively.)

- DIPETT: continuous

(3) Complete, partial, or no linear order?

Does the parser require fixed linear order of the elements of each constituent (or dependents of each head)? (E.g., Shieber's ID/LP parser, *Linguistics & Philosophy* 1984, can allow free order of the elements of a constituent.)

- DIPETT: complete order

B. Criteria relating to sequence of execution**(4) Rule invocation strategy: top-down, bottom-up, or mixed?**

Does the parser choose grammar rules on the basis of what it is looking for or on the basis of what it has found in the input? Left-corner parsing is an example of a mixed strategy (invoke a rule bottom-up and finish it top-down).

- DIPETT: top-down

(5) Completion: incremental or all-at-once?

Given a rule such as $A \rightarrow B C D$, does the parser complete the A constituent in a single step after parsing B, C, and D, or does it add B, C, and D to the A one by one? (The analogous question can be asked for dependency grammar.)

- DIPETT: all-at-once

(6) Access to the input string: left-to-right sequential, random access, or what?

Most parsers accept words sequentially from left to right. The discontinuous parser of Johnson (Proceedings, ACL '89) and the head-driven parser of Proudian and Pollard (same volume) accept them in an arbitrary order determined by the grammar. Eric Wehrli, in a talk at USC around 1984, described a parser that accepts English words backward (right-to-left).

- DIPETT: left-to-right

C. Criteria relating to nondeterminism

(7) Handling of nondeterminacy: backtracking? lookahead? concurrency?

For example, Prolog DCGs backtrack as needed; Marcus's parser uses lookahead; and Earley's algorithm pursues all alternatives concurrently.

- DIPETT: backtracking augmented with a few simple lookahead mechanisms (see 3.4)

(8) Limits on nondeterminacy, if any?

Marcus's parser has a strict limit on lookahead; most other parsers have no limits on (simulated) nondeterminism.

- DIPETT: no limits

(9) Mechanism to prevent duplication of work, if any?

The choices are: no chart; a passive chart that records only completed substructures; or an active chart that also records structures under construction.

- DIPETT: passive chart

(10) Other operations or tests during parsing?

Examples include feature unification or ATN register operations.

- DIPETT: Prolog's argument unification which can be equated to degenerated feature unification.

As far as DIPETT's central parsing algorithm is concerned, it relies on Prolog's built-in proof procedure which can be described as a resolution-based theorem prover with

unification (for variables). Thus, DIPETT's DCG rules are automatically converted into a standard logic program by Prolog, and its built-in proof procedure¹⁴ operates on the translated grammar as if Prolog's engine were a parser.

3.2.3.3 A Typical Parse Tree

We now show an actual parse tree produced by DIPETT. It can best be characterized as a simple functional syntactic representation, in the sense of functional grammars (see 3.2.3.1). DIPETT's parse trees are based on the input constituents' syntactic roles. This explains, for instance, why a passive sentence will see its surface subject moved to the complement slot in the resulting parse tree: its function is that of the complement — and *vice versa* for the surface complement. For other example parse trees, see Appendix D ("Example Parse Trees").

```
Unit #1: The new system file was copied by Tom , yesterday .
lexical analysis (cpu time) : 0.067 second(s).
First parse for input sentence:
...
parsing (cpu time) : 0.2 second(s).
```

```
parse_tree__decla_or_imper
  complex_sentence
    structure
      single_main_clause
        declarative
          statement
            subj
              pass_subj
                entity
                  proper_noun tom
                  number sg3
            predicate
              regular
                verb copy
                tense past_simple
                neg yes
                trans tr_intr
            voice passive
            complement
              svo
                entity
                  determinatives
                    deter the
                  attributes
                    attrs
                      adj new pos
                  head_noun
                    noun file
```

¹⁴ Prolog, as a theorem prover, is sound but incomplete.

```

        noun_modifiers
        pre_modif system
        number sg3
subordinate_clauses
    final
        subordinate
            adverbial
                simple_adv_clause yesterday
end_of_input period
```

3.3 The Architecture and Components of DIPETT

The DIPETT parser consists of these main components:

- the driver (see 3.3.1);
- the lexicon (see 3.3.2);
- the scanner and the lexical analyzer, which includes a simple morphological analyzer (see 3.3.3); and
- the syntactic analyzer, which can perform full or fragmentary analysis, on-line or in batch (see 3.3.4).

In addition, DIPETT contains several subsidiary non-standard components:

- a tagger (see 3.3.5);
- a dynamic dictionary expansion facility (see 3.3.3);
- a memorizing device, i.e. a well-formed substring table or passive chart (see 3.3.5);
- its *helper*, i.e. an error explanation mechanism (see 3.3.5); and
- an internal trace mechanism for debugging purposes (see 3.3.5).

The following subsections describe all these constituents. We wish to warn the reader that sections 3.3.2 and 3.3.3 are particularly technical. As should become clear as we progress through the remainder of this chapter, DIPETT is in fact a parsing environment which allows the user to select different options for different purposes — see [Erbach 91] for a different example of such an environment.

The author implemented DIPETT in Quintus Prolog — about 4500 lines of code, containing several hundred DCG rules — on SPARC workstations. An earlier version was ported to the Apple Macintosh in AAIS Prolog for a demonstration given at COLING-92 ([Copeck *et al.* 92]).

3.3.1 The Driver and the User Interface

DIPETT's driver represents the core of the interface, as far as user interaction is concerned. Its main function is to manage the interaction with the user and, in accordance with the user's choices, launch the appropriate processes and control their progress.

Normally, an input will be given, *via* the driver, to the lexical analyzer followed by the syntactic analyzer (and then by semantic analysis — see chapter 4).

For the time being, the user interface is visually rather elementary in that it does not exploit windows or other sophisticated tools — we are considering the possibility of using tools such as HyperNeWS in order to make the user interface more attractive.

Appendix A (“DIPETT’s Driver and User Interface”) shows an abridged, actual log of DIPETT’s driver in action. We urge the reader to take the time to go through this commented log: subsequent details on DIPETT will be easier to follow.

3.3.2 Lexicon and Dictionaries

3.3.2.1 DIPETT’s Lexicon

DIPETT’s surface-syntactic lexicon contains most English function words. As we will see later in 3.3.2.1, we essentially rely on an on-line version of The Collins dictionary for new entries belonging to open categories: nouns, verbs, adjectives, and adverbs. The lexicon’s entries are lexemes which fall into 26 categories (some of these are in fact subcategories). It also comprises a particle table that associates legal¹⁵ particles with each verb: together, they form phrasal verbs. This table is used during parsing to disambiguate particles and prepositions. A list of idiomatic constructions completes DIPETT’s lexicon.

The lexical entries are Prolog facts of the form

```
dict( Entry, Category, Info1, Info2 ).
```

where *Entry* is the root and *Category* is the root’s part of speech (more precisely, one of the 26 categories). Depending on the root’s part of speech, *Info1* is either the root’s number (singular or plural) or the root’s subcategory label. Again, depending on the root’s part of speech, *Info2* is either the root’s tense information, the root’s subcategory label, or the root’s basic form. A root has as many entries in the lexicon as it has parts of speech.

¹⁵ We use the term ‘legal’ as a synonym of ‘acceptable’, ‘allowable’, or ‘recognized’.

The list of legal parts of speech, based mostly on [Quirk *et al.* 85], and their associated numbers (for ease of reference) follows:

DETERMINERS:

predeterminers [1], central determiners [2], postdeterminers: cardinal numerals [3], ordinal numerals and general ordinals [4], and closed-class quantifiers [5];

ADJECTIVES:

adjectives [6];

NOUNS:

count nouns [7], mass nouns [8], proper nouns [9];

PRONOUNS:

personal [10], reflexive [11], possessive [12], relative [13], interrogative [14], and indefinite [15];

PREPOSITIONS:

prepositions [16];

CONJUNCTIONS:

conjunctions [17];

ADVERBS:

adverbs [18];

VERBS:

auxiliary [19], modal [20], stative [21], transitive and intransitive [22], transitive only [23], intransitive only [24];

ADVERBIAL PARTICLES:

adverbial particles [25];

SPECIAL:

words which do not easily fit into any of the above classes [26].

Here is an excerpt containing an instance of every legal category. DIPETT's lexicon contains approximately 2200 entries (roots), excluding the 'partic' and 'idiom' entries presented below.

```
[1] dict( all, predeter, _, indef).  
[2] dict( an, deter, sg3, indef).  
[3] dict( two, card_num, pl, indef).  
[4] dict( first, ord_num, sg3, def).  
[5] dict( few, quant, pl, indef).
```



```

[6] dict( actual, adj, pos, _ ).
[7] dict( abbreviation, countnoun, sg, _ ).
[8] dict( air, massnoun, _, _ ).
[9] dict( bob, propernoun, sg3, _ ).
[10] dict( her, pers_pron, sg, _ ).
[11] dict( herself, reflex_pron, sg, _ ).
[12] dict( hers, poss_pron, _, poss/3 ).
[13] dict( that, rel_pron, _, _ ).
[14] dict( how, interro_pron, _, indef ).
[15] dict( anybody, pron, sg3, indef ).
[16] dict( after, prep, _, _ ).
[17] dict( and, conj, _, _ ).
[18] dict( accurately, adv, _, _ ).
[19] dict( am, aux/be, sg, pres ).
[20] dict( can, modal_aux, _, can/pres ).
[21] dict( am, v/stat, sg, be/pres ).
[22] dict( begin, v/tr_intr, _, _/pres_inf_imp ).
[23] dict( append, v/tr, _, _/pres_inf_imp ).
[24] dict( die, v/intr, _, _/pres_inf_imp ).
[25] dict( by, adv_particle, _, _ ).
[26] dict( not, special_function_word, _, _ ).

```

The dictionary also contains two other sources of information. The first is a particle table which associates a list of legal particles with each verb¹⁶ already in DIPETT's lexicon. This information is used during parsing in order to help disambiguate particles and prepositions, since many words belong to both categories. Lists of particles are ordered according to their frequency of use¹⁷, from most frequent to least frequent. Our main reference for this table was [Quénelle & Hourquin 87]. Here is an excerpt from the particle table which contains 321 entries:

```

partic( access, [to] ).
partic( begin, [with, on, as] ).
partic( build, [up, out, in, on, over, ..., of] ).

```

¹⁶ We have not decided yet how this table is going to be expanded during TP if a missing entry is found.

¹⁷ This ordering is due to [Quénelle & Hourquin 87]. Although it might be considered as *a priori* knowledge, such ordering is in fact irrelevant since all the parser does is check whether or not a given token is a member of such a list (treated as a set).

```

partic( cancel, [out] ).
partic( check, [up, out, in, on, ..., through] ).
partic( clear, [up, out, with, off, away, of] ).
partic( design, [for] ).
partic( display, [] ).
partic( eliminate, [from] ).
partic( find, [out, in, for, against] ).
partic( finish, [up, in, with, off] ).
partic( give, [up, out, in, to, on, ..., forth] ).
partic( group, [about, around, round] ).
partic( include, [out, in] ).
partic( join, [up, in, with, to, on] ).
partic( link, [up, with, to] ).
partic( load, [up, with, down, into] ).
partic( print, [out, in, off] ).
partic( report, [out, to, on, for, back, upon] ).
partic( search, [out, for, through] ).

```

The second additional source of information is a table of idiomatic constructions¹⁸. It identifies short non-productive syntactic constructions that can usually be treated as single tokens: thus the term ‘groupings’ to refer to such constructions. DIPETT uses this table during scanning of the input string and suggests potential replacements to the user upon encountering such idiomatic items: see an example in 3.3.3. Here is an excerpt of this table which contains 409 entries:

```

idiom([not,only], not_only, adv, _, _).
idiom([not,so,much], not_so_much, conj, _, _).
idiom([not,to,say], not_to_say, conj, _, _).
idiom([now,and,again], now_and_again, adv, _, _).
idiom([now,and,then], now_and_then, adv, _, _).
idiom([now,that], now_that, conj, _, _).
idiom([nowhere,else], nowhere_else, adv, _, _).
idiom([of,course], of_course, adv, conj, _).
idiom([on,account,of], on_account_of, prep, _, _).

```

¹⁸ Such constructions are loosely referred to as “idioms” or ‘groupings’: see 2.1.3.5 and footnote 15 in that chapter.

```
idiom([[on,and,off], on_and_off], adv, _, _).
```

3.3.2.2 The Collins Dictionary and the WordNet Lexical Database

Two “off-the-shelf” dictionaries are used in TANKA. The first, The Collins dictionary, which we extracted from PC-KIMMO [Karp *et al.* 92], is used by DIPETT when new words (i.e. words not found in DIPETT’s lexicon) are encountered in the input text. It contains 90196 root forms and 317477 inflected forms distributed in these categories: pronoun, preposition, determiner, conjunction, adverb, noun, adjective, and verb. This dictionary, which may be seen as a general-purpose complement of DIPETT’s lexicon, is accessed by the lexical analyzer (see 3.3.3) during parsing or tagging (see 3.3.5).

The second dictionary, WordNet [Miller *et al.* 90], will probably be used by DIPETT in order to extract simple, domain-independent semantic categories to be associated with nouns — see 5.2. Currently, it is accessible through DIPETT’s main menu for consultation purposes only. WordNet is an on-line lexical database “whose design is inspired by current psycholinguistic theories of human lexical memory. English nouns, verbs, and adjectives are organized into synonym sets, each representing one underlying lexical concept. Different relations link the synonym sets.” ([Miller *et al.* 90: p.235]).

As a final remark, we note that both The Collins and the WordNet dictionaries are kept in secondary storage and are accessed with appropriate routines on an as-needed basis from within DIPETT.

3.3.3 The Scanner and the Lexical Analyzer

These modules performs two fundamental pre-parsing tasks: scanning and lexical processing of the input string, which includes basic morphological processing. First, the *scanner* is responsible for reading the input string, usually a sentence to be parsed, and constructing a list of tokens, i.e. words, non-alphabetic items, and ‘groupings’ (“idioms”). The scanner correctly handles strings in which there are numbers, small and capital letters¹⁹, contractions (e.g. can’t), abbreviations, dashes, and punctuation marks. As mentioned in 3.3.2.1, the scanner also detects potential ‘groupings’ and checks with the

¹⁹ Although at present it does not attach any special information to proper nouns, for example.

user whether a replacement may be desirable. An example of this appears in the example log below.

Second, *lexical analysis* is done on the list of tokens (obtained from the scanner) in order to construct a list of words (in their root form) using syntactic information found in DIPETT's lexicon. Many words have regularly inflected forms. In order to process these forms in a general and simple way, and to avoid repeating almost identical entries, we perform morphological analysis which converts tokens in the original list into their root forms, so that their associated entries can be accessed in the lexicon. For example, plural nouns are transformed into their corresponding singular (root) forms and regular verbs are transformed into their infinitive (root) forms. Of course, the information regarding the original form of a word (e.g. that it was originally written in its plural form) is kept for further syntactic processing.

Currently, the lexical analyzer does not perform any kind of (pre-parsing) lexical disambiguation. It simply collects all the potential parts of speech associated with every word (root) on input, according to DIPETT's lexicon (or The Collins dictionary if the word could not be found in that lexicon), and forwards this information to the parser. The latter then has to find a valid analysis (i.e. parse). Lexically ambiguous words can be said to be disambiguated once a resulting parse tree is obtained, an idea that is exploited in [Feng *et al.* 94].

If the input²⁰ happens to contain unknown words (i.e. those for which DIPETT's lexicon does not contain at least one entry²¹), the lexical analyzer allows the user to dynamically augment the lexicon by selecting or supplying the appropriate information, after having consulted The Collins, if necessary — see [Asker *et al.* 92] for an automatic lexical acquisition method based on EBL. These additions are temporary. They are lost at the end of the session²². However, they are automatically saved in a file so that the user can easily add them to DIPETT's permanent dictionary after the session if desired. Alternatively, the user could collect these new entries in a separate file and load them when necessary.

²⁰ Input strings are referred to as *units* by the parser. They usually are sentences but can also be fragments (see 3.4.4).

²¹ Currently, the parser cannot handle the situation in which there is only one entry which happens to be wrong.

²² *Session* means "parsing session". Currently, it is not a session with the whole TANKA system although it will be so in the future.

As the following example log of a session illustrates, the user can also skip the unknown word, or reenter it (e.g. if it has been mistyped), or simply abort the session if desired. In the following excerpt, the symbol "{ ... }" simply means that a portion of the actual session log has been cut out. This allows us to focus on the main point without having to go through a lengthy listing. This will apply to all logs presented in the remainder of this document. As before, the actual log is in the Courier typeface and comments were added in the Times typeface, in bold, and enclosed in braces.

{ ... }

What do you wish to do [0 @ 9, or e]? 1

Enter sentence: Tom asked Beth even if he didn't like her behavior.

IDIOMS FILTERING:

at least one potential idiom was found in this new input ...

"tom asked beth even if he did not like her behavior . "

> IDIOM: replace [even,if] with "even_if" {4,conj} [y/N]? y

{above: the scanner detected an 'idiom', starting at position #4 in the input string, that can function as a conjunction (conj) according to DIPETT's lexicon.}

{below: for this example, we asked DIPETT to show the scanner's output, in which 'cap' means capitalized and 'low' means lower case. Notice that "even if" is now considered as a single token ('even_if').}

DEBUG(scanner):

[word:tom/cap,word:asked/low,word:beth/cap,word:even_if/low,word:he/low,word:did/low,word:not/low,word:like/low,word:her/low,word:behavior/low,punctuation: . /end_of_sentence]

Unit #1: Tom asked Beth even if he didn't like her behavior .

"behavior" not in DIPETT's dictionary.

Abort, Skip, Re-enter, add to Dictionary [a/s/r/D)?

{below: the user is prompted for completing the information on this new lexical entry. To facilitate the user's task, the on-line Collins dictionary is consulted and the associated information is shown to the user.}

** Temporary addition

(but all info. is saved in file zzz.DIPETT-dict_add.pl) **

Identify category (part of speech) of "behavior" ;

>> Checking with KIMMO-COLLINS dictionary ...

>> Consulting Collins: do you wish to see the raw entry [y/N]? y

>> Collins suggests: behavior behaviour N sg

{PofS abbrevs: A/djective, Adv/erb, N/oun, V/erb, Pron/oun, Conj/unction, D/eterminer, Prep/osition}

Do you wish to see the list of other abbreviations [y/N)?

>> Here is the list of candidate entries extracted from Collins:

dict(behavior,countnoun,sg,_20003) (1) ;

please select one of these parts of speech;

enter a number between 1 and 1: 1

{above: The Collins entry is automatically converted into a DIPETT-compatible lexical entry. This simplifies the user's task: all he has to do then is select the appropriate part of speech. However, if for some reason he is not entirely satisfied with the system's suggestions, he may use the "manual mode" (below) which will allow him to construct a more appropriate lexical entry.}

```
++ You may switch to MANUAL MODE
    if you are not satisfied with your choice [y/N]
++ Enter another part of speech
    for "behavior" (independ. of current input) [y/N]?
```

{below: for this example, we asked DIPETT to show the lexical analyzer's output — this is useful essentially for debugging purposes.}

```
DEBUG(lexicalyser):
[dct(word:tom/cap,[d(propernoun,sg3,_7014)]),dct(word:asked/low,[d(v/
tr_intr,_7091,ask/perf),d(v/tr_intr,_7115,ask/past)]),dct(word:beth/c
ap,[d(propernoun,sg3,_7206)]),dct(word:even_if/low,[d(conj,_7283,_728
4)]),dct(word:he/low,[d(pers_pron,sg3,_7362)]),dct(word:did/low,[d(au
x/do,pl,past),d(aux/do,sg,past),d(aux/dc,sg3,past),d(v/tr_intr,_7439,
do/past)]),dct(word:not/low,[d(adv,_7622,_7623),d(special_function_wo
rd,_7604,_7605)]),dct(word:like/low,[d(adv_particle,_7706,_7707),d(co
nj,_7748,_7749),d(prepp,_7766,_7767),d(v/tr,_7724,like/pres),d(v/tr,_7
724,like/inf),d(v/tr,_7724,like/imp)]),dct(word:her/low,[d(deter,_790
7,def),d(pers_pron,sg,_7890)]),dct(word:behavior/low,[d(countnoun,sg,
_8110)]),dct(punctuation:./end_of_sentence,[])]
lexical analysis (cpu time) : 0.217 second(s).
First parse for input sentence:

{ ... }
```

As we can see, the file in which dictionary additions have been copied is explicitly identified for the user's convenience. In the above example, the file zzz.DIPETT-dict_add.pl has the following contents after the addition of the new entry 'behavior':

```
% zzz.DIPETT-dict_add.pl [date&time: 11/6/93 14:11:00]

:- multifile dict/4, idiom/4.
:- dynamic dict/4, idiom/4.

/* countnoun */    dict(behavior,countnoun,sg,_8110).

% # of new words (entries): 1
```

This Prolog-readable file can now be used to update the permanent dictionary, if necessary.

3.3.4 The Syntactic Module

The reader should not be misled by the brevity of this subsection which appears to contrast with its relative importance within this thesis: the syntactic module's description is in fact disseminated in parts of sections 3.2 and 3.3 and, most of all, in section 3.4, Appendix B, and section 5.3. The appendix is obviously the most complete and precise definition of DIPETT's grammar: it presents the two main Prolog files that constitute the grammar (see Appendices B.1 and B.2), followed by a simplified Prolog description of these two files (see Appendix B.3).

3.3.4.1 Encoding English Grammar with DCG Rules

The syntactic module is the central part of DIPETT. It encapsulates the system's knowledge about the syntax of English, expressed mostly as DCG rules. In terms of Prolog components, the syntactic module is organized into four separate parts: the DCG rules for the sentence- and verb phrase-related constituents, the DCG rules for the noun phrase-related constituents, the tense table which contains legal combinations of a verb's features (including its potential modals and auxiliaries), and the DCG rules for the question-related constituents.

Here is an example of a (simplified) rule packet taken from DIPETT's grammar. It deals with noun phrase post-modifiers and says, essentially, that such a modifier can occur (in the syntax) as a relative clause (`rel_clause`), as an appositive, as a restricted kind of comparison, or as a non-empty prepositional phrase. Other DCG rules define each of these non-terminals elsewhere in the grammar — see appendix B.

```
np_postmodif(_,rel,NpRel,X,Num)
    --> rel_clause(NpRel,X,Num) .
np_postmodif(POS,app,_,X,Num)
    --> appositive(POS,X,Num) .
np_postmodif(_,rc,_,X,_)
    --> restricted_comparison(X) .
np_postmodif(_,pp,_,X,_)
    --> required_prepphrases(X,_) .
```

3.3.4.2 Definite Clause Grammars and Other Logic Grammar Formalisms

For several reasons, we find DCG rules ([Pereira & Warren 80]) convenient for writing NL grammars:

- they allow us to implement the required degree of compositionality (see 1.3.2.2) naturally and efficiently;
- they have a straightforward interpretation;
- they are similar to phrase structure rules, which have been used extensively in linguistics; and
- they are neatly embedded in Prolog, the language chosen for TANKA.

Other logic grammar formalisms have been proposed in the literature²³. For the most part, they introduce additions to DCG rules. Some of them were designed specifically for the writing of NL parsers in order to handle certain linguistic phenomena more easily than with DCG rules alone. Three typical logic grammar formalisms are Extraposition Grammars (XGs), Modifier Structure Grammars (MSGs), and Discontinuous Grammars (DGs). These formalisms can be said to extend the capabilities of DCG rules in terms of linguistic expressiveness, but it should be noted that they are not computationally more powerful: the DCG grammar formalism is capable of describing type-0 languages. We now briefly present these three formalisms²⁴.

XGs ([Pereira 81]) were introduced in order to deal more conveniently with left extraposition (or relativization). It occurs when a constituent has been moved from another position in the sentence, for instance in a relative clause. In addition to DCG rules, XGs accept rules that have the form

NonTerminal ... Trace --> RHS

which says that `NonTerminal` can be expanded to `RHS` on the condition that `Trace` is given an empty realization later in the parse. `Trace` is put on the extraposition list, the mechanism used by XGs to deal with extraposition, conceptually equivalent to the ATN “hold” list ([Woods 70]).

We now show fragments of a simple XG of sentences that deals with relative clauses introduced by a relative pronoun. The last rule (which is not context-free) says that a relative pronoun (in a relative clause) is analyzed as a marker, which is followed by an unspecified number of constituents (“...”), which are followed by a trace. For instance, in “The mouse that the cat chased squeaks”, ‘that’ is the relative pronoun, which is analyzed as the relative marker. The trace stands for the extraposed noun phrase “The mouse” and

²³ For a general introduction to logic grammars, see [Abramson & Dahl 89].

²⁴ There are many others. For instance, Modular Logic Grammars ([McCord 85]), Restrictions Grammars ([Hirschman & Puder 82]), and Contextual Discontinuous Grammars ([Saint-Dizier 88]).

will be positioned between 'chased' and 'squeaks', i.e. "The mouse *relative_marker* the cat chased trace squeaks".

```

sentence --> noun_phrase, verb_phrase.
noun_phrase --> determiner, noun, relative_clause.
noun_phrase --> trace.
relative_clause --> [].
relative_clause --> relative_marker, sentence.
relative_marker ... trace --> relative_pronoun.

```

MSGs ([Dahl & McCord 83]), proposed for the treatment of coordination and other constructions in NL, constitute an extension of XGs. First, MSGs automate the process of building an analysis structure (parse tree) by making it implicit in the grammar. In XGs (and in DCG), the only way to build these structures in to augment non-terminals with appropriate arguments. Second, MSGs accept rules of the form

NonTerminalA:SemanticItem --> NonTerminalB

where NonTerminalA --> NonTerminalB is an XG rule and SemanticItem is a term of the form Operator-LogicalForm, which plays a role in the semantic interpretation of a phrase analyzed by application of the rule ([Dahl & McCord 83: p.72]). As with analysis structures, the treatment of several major types of coordination (and quantifier scoping) is implicit in MSGs rules through the use of coordination demons. "The interpreter has a general facility for treating certain words as "demons" (cf. Winograd [1972]), which trigger a backing up in the parse history that will help reconstruct ellisions and recognize the meaning of the coordinated sentence" ([Abramson & Dahl 89: p.92]).

DGs ([Dahl 89]) represent another generalization of XGs. DGs can deal with left and right extraposition, coordination, and can even be used to describe certain free-word-order phenomena. As stated in [Abramson & Dahl 89: p.143]: "They are basically metamorphosis grammars with the added flexibility that unidentified strings of constituents can be referred to (usually through a pseudo-symbol skip (X), where X stands for the skipped substring), to be repositioned, copied, or deleted at any position. They do not need to obey, as do XG's, any nesting constraints." — on metamorphosis grammars, the first type of logic grammars, see [Colmerauer 78].

Here are fragments of a simple DG of sentences, taken from [Abramson & Dahl 89: pp.145-6], which handle coordination with an elided object. The last rule elides an

expected object followed by a “skip” (i.e. a string of constituents of unknown length to be repositioned), which is copied unanalyzed, to be parsed later in its new location. This rule also reconstructs the elided object’s internal representation (P) *via* unification with the object in the second sentence. For instance, the input string “Mary saw and John heard the train” will be analyzed as:

and(saw(mary, the(X, train(X))), heard(john, the(X, train(X)))).

```

sentence(and(S1,S2)) --> sent(S1), and, sent(S2).
sent(S) --> name(K), verb(K,P,S), object(P).
object(P) --> determiner(X,P1,P), noun(X,P1).
object(P), and, skip(G), object(P) --> [and], skip(G), object(P).

```

Despite some obvious advantages of the above formalisms over DCG, the author of this thesis evaluated that the net gain in the expressiveness of DIPETT’s grammar would probably be small, and decided to use DCG. First, most of these other formalisms were designed to construct the semantic structure concurrently to its syntactic structure, during the analysis of the input string. Since our goal was to design a parser that would output syntactic trees, the use of such other formalisms would not exploit all their capabilities.

Second, the author remains unconvinced about the overall efficiency of several non-DCG formalisms, *for the parsing of real texts*. It is quite surprising that not a single comparative study of these formalisms has ever been published. For example, one could propose a non-trivial subset of a NL described as a context-free grammar. This subset would next be translated into each of the formalisms under consideration — several referees could be involved in order to make sure that every translation efficiently exploits each formalism’s expressiveness. Then, an experiment would consist of submitting a corpus (of real text) to each parser and collecting the results. In that situation, we would be looking for the parse trees (or graphs) as well as for the timing results in order to establish the foundation of a valid comparison.

To sum up, we see DCG as a very high-level programming language that is well adapted to our task of encoding a particular NL grammar, within the context of TANKA.

3.3.4.3 The Grammar’s Main Non-terminals

We now show the main non-terminals of the grammar by listing the major syntactic types recognized by the syntactic module. These types refer to syntactic units of English. Here, our goal is to discuss the surface-syntactic coverage of the parser from a linguistic perspective: we wish to avoid muddling the presentation with implementation-related aspects.

The list of non-terminals also illustrates the fact that we have striven for a natural approach to parsing in which grammar rules are, as much as possible, descriptively close to those that people learn when they study a language with a view to acquiring it.

- **Syntactic Units Related to Noun Phrase and Prepositional Phrase**

Adjective phrase.

Appendage²⁵: a constituent attached to the right of a noun phrase (NP): a prepositional phrase, an apposition, a relative clause, a past participle phrase, or a gerundial clause.

Apposition: a noun group as a nonrestrictive qualifier.

Classifier: nouns preceding the head noun of a NP, that classify the head noun.

Determiner sequence.

Gerundial clause.

Infinitive clause.

Nominal clause, functionally equivalent to a NP, e.g. report-, infinitive- and gerundial clause.

Noun phrase (simple and conjoined).

Post-classifier: number following the head noun of a NP (e.g. "page 12").

Post-determiner: group that follows the determiner.

Pre-determiner: group that precedes the determiner.

Prepositional phrase (simple and conjoined).

Relative clause.

Report clause.

- **Syntactic Units Related to Verb Phrase**

Adverb group.

²⁵ A rather eclectic unit. Its introduction was dictated by technical considerations.

Adverbial clause: one type of sentence modifier (e.g. “*After the food came*, we ate ...”).

Complete sentence (including sentence modifiers and conjoined elements).

Question.

Sentence modifier (e.g. adverbial clause, prepositional phrase, infinitive clause).

Sentence.

Verb phrase (simple and conjoined).

3.3.4.4 The Influence of [Quirk *et al.* 85] on the Resulting Grammar

As we discussed in section 3.2.2, [Quirk *et al.* 85] (or simply QUIRK in this subsection) was used as the main reference for English grammar. DIPETT’s grammar, i.e. its DCG rules as well as its non-terminals, can thus be said to reflect our view and understanding of the central elements of English grammar as presented in QUIRK. Even the Prolog implementation (Appendix B) contains numerous explicit references to QUIRK.

In this subsection, we wish to indicate more precisely what elements of QUIRK have been most influential on the resulting grammar. We here list the chapter numbers and titles, in QUIRK, for the lexical and syntactic aspects of DIPETT’s grammar.

- **for lexical aspects:**
 - chapter 2 (“A Survey of English grammar”),
 - chapter 3 (“Verbs and auxiliaries”),
 - chapter 5 (“Nouns and determiners”),
 - chapter 6 (“Pronouns and numerals”),
 - chapter 7 (“Adjectives and adverbs”),
 - chapter 8 (“The semantics and grammar of adverbials”),
 - chapter 9 (“Prepositions and prepositional phrases”),
 - chapter 13 (“Coordination”).
- **for syntactic aspects: (chapter titles not duplicated)**
 - chapter 3,
 - chapter 5,
 - chapter 7
 - chapter 8,

chapter 11 ("Sentence types and discourse functions"),
chapter 13,
chapter 14 ("The complex sentence"),
chapter 15 ("Syntactic and semantic functions of subordinate clauses"),
chapter 16 ("Complementation of verbs and adjectives"),
chapter 17 ("The noun phrase"),
chapter 18 ("Theme, focus, and information processing"),
Appendix III ("Punctuation").

The methodology used by the author for the KA task which consisted of extracting meaningful grammatical knowledge elements from QUIRK consisted of five basic phases:

- read a specific section or chapter in QUIRK with emphasis on elements that have an impact on (English) syntax;
- for syntactic elements not already covered by DIPETT's grammar and which seem necessary in order to parse realistic technical texts, formulate candidate additions to the parser using a simplified DCG notation;
- analyze the current state of the parser's implementation and decide how to best integrate the proposed additions, this time, expressed as legal DCG rules;
- test the resulting "new" parser, first with example input strings given in QUIRK, then with other similar examples;
- re-test the parser for rules that were part of the grammar before the latest additions in order to make sure that the resulting "new" parser is still behaving consistently.

3.3.5 Subsidiary Components

In this subsection, we describe DIPETT's non-standard mechanisms that contribute to making it a useful parsing environment. In order, we describe the parser's *memorizer* and *helper*, then its trace mechanism and, finally, its tagger.

3.3.5.1 The Memorizing Device and the Helper Mechanism

These features are presented together because they both depend on the same *notes* stored during analysis. The goal of the *memorizer* is to minimize the reparsing of syntactic substructures parsed earlier, which are next reconsidered due to a later failure.

The goal of the *helper* is to help the user identify why a given input has been rejected. Both features may be activated at the beginning of a parsing session.

3.3.5.1.1 Notes

Notes are assertions which record essential syntactic information about well-formed substrings. We only memorize the major building blocks of an English sentence: noun phrases, verb phrases, prepositional phrases. The corresponding DCG rules contain a Prolog assertion invoked if the user had activated the memorizer or the helper. We store the substring, its type and its syntactic structure produced by the parser. This resembles the well-formed substring table used in bottom-up and chart parsing, but we do not include the substring's length and position in the sentence, as happens in regular parsing [Pereira & Shieber 87] — another name for such a structure is a passive chart (see 3.2.3.2).

3.3.5.1.2 The Memorizer

The parser usually reconsiders substrings it has already parsed as part of a larger substring rejected for some reason (e.g. the parser may have selected a wrong rule for the input it is trying to parse). To minimize this redundancy, the memorizer allows the parser to check the state of its memory (*via* notes) whenever it has to parse a major syntactic substructure. If the candidate substring has been noted, it need not be reparsed. The notes are indexed on the first two parameters: the substring and its type.

An experiment was conducted to test the memorizer's efficiency and the results showed that, on average, it tends to speed up parsing. The parser was run on a set of 600 test sentences twice— see section 3.6 for details on this set. First, with the memorizer (and the helper) deactivated: this is the 'without' option. Second, with the memorizer (and the helper) activated: this is the 'with' option. We then compared the results of the 'without' parsing with the results of the 'with' parsing.

Of the 600 sentences, 140 (23.3%) caused DIPETT's grammar to "redundantly" re-parse major syntactic units, i.e. NPs, PPs, and VPs. This can be observed by looking at the memorizer's results (see the example below): these 140 input strings had at least one major syntactic unit fetched during parsing, instead of re-parsing it. Some had up to several hundreds. Of these 140, 120 (85.7%) were parsed faster in the 'with' test than in

the 'without' test. The remaining 20 (14.3%) were parsed slower in the 'with' test than in the 'without' test.

Among the 85.7% that were parsed faster, the average gain was 21.5%, with a standard deviation of 14.1. In comparison, among the 14.3% that were parsed slower, the average loss was 29.8%, with a standard deviation of 31.2. The combined average of those parsed faster with those parsed slower was a gain of 14.1%, with a standard deviation of 25.1. Finally, it is worth noting that in all other 460 input strings (600 - 140), the parsing time was essentially the same for the 'without' and the 'with' tests.

Here is an example illustrating the type of results the memorizer returns for an input that causes it to reparse major syntactic substructures:

```
Unit #9: You can find more information in the dictionary manual , but
right now we're going to produce a report .
Process this unit (Y/n), try another Version, or Exit [Y/n/v/e]?
```

```
lexical analysis (cpu time) : 0.267 second(s).
First parse for input sentence:
```

```
MEMORIZER results: # of units fetched (not re-parsed unnecessarily).
noun phrase(s)      : 32
prepositional phrase(s) : 20
verb phrase(s)      : 0
verbal clause(s) of type 1 : 0
verbal clause(s) of type 2 : 0
subordinate clause(s) : 5
```

```
parsing (cpu time) : 5.55 second(s).
```

```
parse_tree__decla_or_imper
  compound_sentence
    structure
      head_main_clause
        declarative
          statement
            subj
              entity
```

```
{ ... }
```

The memorizer's results are also useful for indicating to what extent an input is ambiguous relative to DIPETT's grammar: the more it is ambiguous, the more the memorizer's numbers will go high. For instance: if we remove the prepositional phrase "in the dictionary manual" from the above input, we instead get these results from the memorizer:

```

MEMORIZER results: # of units fetched (not re-parsed unnecessarily).
noun phrase(s)      : 4
prepositional phrase(s) : 3
verb phrase(s)      : 2
verbal clause(s) of type 1 : 0
verbal clause(s) of type 2 : 0
subordinate clause(s) : 0

```

This example illustrates the impact of a given syntactic substructure on the parsing of the string in which it appears. This information is useful for tuning the parser's grammar.

3.3.5.1.3 The Helper

One of the hardest tasks in the construction of realistic, useful parsers is giving the user meaningful information when an input is not parsable. We will later address the problem of input strings that seem to engage the parser in an endless computation (in section 3.4.2), but here we concentrate on input strings that are quickly rejected as not conforming to the parser's grammar.

The parser has a rudimentary helper mechanism that accesses notes in reverse chronological order (notes form a stack). The user has two options: a list, potentially long, of all successfully recognized major substrings, or a list of all major substrings *that have been effectively parsed* at a certain time, just before rejecting the whole input for some reason.

The second option, somewhat tricky, deserves a clarification. The processing involved here is based on the assumption that the most recent note corresponds to the most probable "almost parsed" major substring²⁶. Pattern matching between the original string (input) and the most recent memorized major substrings is done right-to-left. This can identify the problematic substring. The user may decide to resubmit a modified input. Here is an example illustrating the potential utility of the helper's second option: the incorrect input was "The user wants to print an".

(...)

²⁶ Since rules are tried top-down, and sequentially within groups of rules for the same unit, the most recent rule applied is no less informative than the previous ones, if any. That is because, in the worst case, this rule was rejected as were the previous ones; in the best case, the rule was correct and the parser did not have to try other rules. So, the previous rules tend to be less informative. To put it simply, if the helper cannot find a note that covers a certain substring, this substring is unparsable. Clearly, this is not a formal principle but, rather, a simple heuristic.


```

MOST PROBABLE "ALMOST PARSED" SYNTACTIC ELEMENTS:
TOKENS: [the,user,wants,to,print,an]
vp
[wants,to,print]
{ ... }
np_equiv
[the,user]
{ ... }
WORDS NOT PARSED: [an]

{ ... }

```

In fact, such input rejection includes two different phenomena. There are truly ungrammatical input strings, i.e. strings that are “bad” English according to a native speaker’s knowledge of the language (or some standard grammar such as [Quirk *et al.* 85]). Examples of ungrammatical input strings are “He has two office” (lack of number agreement) and “Theater to went John the” (incomprehensible word ordering). There also are extra-grammatical input strings, a much more difficult issue. The input could be perfectly acceptable to a native speaker but nevertheless rejected because the parser’s grammar did not have a rule that covered it. This problem has no easy solution. For a thorough discussion, see [Carbonell & Hayes 83], [Jensen *et al.* 83], [Kwasny 80], and [Mellish 89].

3.3.5.2 The Trace Mechanism

Another difficulty in implementing complex parsers is testing and fine-tuning. In Prolog, the implementer can try debugging facilities which, however, are cumbersome for logic grammars. It is often only interesting to know what rule is being examined by the parser, for which part of the input string, and what has been successfully recognized.

To satisfy these requirements, we have implemented our own trace mechanism. We insert a trace instruction (activated by a flag) in all principal rules — essentially those related to noun phrases, verb phrases, and prepositional phrases: see what the menu offers in Appendix A (“DIPETT’s Driver and User Interface”). The system allows the user to activate traces and control the trace mechanism. Here is a simple example in which we trace ‘sentence’, ‘nounphrase’, and ‘verbphrase’ (and ‘either_or__np’ which is automatically activated by ‘nounphrase’).

```

Unit #3: What you need is information .
Process this unit (Y/n), try another Version, or Exit [Y/n/v/e]?

```

```
lexical analysis (cpu time) : 0.033 second(s).
First parse for input sentence:
... trying sentence [what,you,need,is,information,..]
... trying verbphrase [you,need,is,information,..]
... trying either_or_np [you,need,is,information,..]
... trying nounphrase [you,need,is,information,..]
... trying verbphrase [need,is,information,..]
... trying either_or_np [is,information,..]
... trying nounphrase [is,information,..]
... trying verbphrase [is,information,..]
... trying either_or_np [information,..]
... trying nounphrase [information,..]

( ... )
```

3.3.5.3 The Tagger

The tagger's main function is to associate with every input token its list of potential parts of speech, according to DIPETT's lexicon or The Collins dictionary. We must emphasize that this tagger is quite simple. In particular, it is not a disambiguating tagger, i.e. it does not attempt to associate a unique part of speech (i.e. the "right" one) with every token. In that sense, it has nothing in common with more complex taggers which rely on statistical computations, frequencies data, and other corpora-collected sources of information (e.g. [DeRose 88]).

Tokens that cannot be found in this lexicon are looked up in The Collins dictionary and the accompanying information is kept for subsequent updating of DIPETT's lexicon, if necessary. As shown in section Appendix A ("DIPETT's Driver and User Interface"), the tagger can be activated either in batch or on-line and its results can be conveniently saved in a file for further processing. The tagger is built on top of DIPETT's scanner and, more importantly, its lexical analyzer (see 3.3.3).

One interesting application of the tagger is the construction of lexicons. In particular, when we want to analyze a new text (for which DIPETT's lexicon may be missing many nouns, verbs, adjectives, and adverbs), the tagger helps us in collecting the missing entries and their associated part of speech information. The way the tagger's results are presented also helps us in minimizing lexical ambiguity²⁷ by presenting the context (i.e. the original string) in which the tokens occurred. In that way, we have the possibility of

²⁷ For a knowledge-based approach to the problem of lexical disambiguation, see [McRoy 92].

adding to the lexicon only the relevant (relative to this particular text) parts of speech for these new entries.

We now present a simple example illustrating the tagger in action (all the 'sent' structures are saved in a Prolog readable file for further processing).

```
Unit #4: QUIZ turns the computer's data into information for you .
Process this unit (Y/n), try another Version, or Exit [Y/n/v/e]?

sent(4, '[quiz,turns,the,computer_s,data,into,information,for,you,.]',
[tag(quiz,[propernoun]),tag(turns,[countnoun,v/tr_intr]),tag(the,[deter]),tag(computer_s,[countnoun]),tag(data,[countnoun,massnoun]),tag(into,[adv_particle,prep]),tag(information,[countnoun,massnoun]),tag(for,[adv_particle,conj,prep]),tag(you,[pers_pron]),tag(.,[.])]).

lexical analysis (cpu time) : 0.1 second(s).

Unit #5: the new disk .
Process this unit (Y/n), try another Version, or Exit [Y/n/v/e]?
"disk" not in DIPETT's dictionary.
Abort, Skip, Re-enter, add to Dictionary [a/s/r/D]?
% NEW WORD ... According to COLLINS: disk disk N sg

sent(5, '[the,new,disk,.]', [tag(the,[deter]),tag(new,[adj]),tag(.,[.])]).

lexical analysis (cpu time) : 0.116 second(s).

{ ... }
```

A post-tagging filter has also been written²⁸ to alleviate the user's task in processing the tagger's output file by getting rid of "noisy" entries (e.g. closed class entries which are already in DIPETT's lexicon). Also, when a word can fill more than one part of speech, this filter uses the information recorded in its associated DIPETT parse tree in order to distinguish among them (e.g. 'pen': noun or verb) — as mentioned above in 3.3.3, this is one way to perform post-parsing lexical disambiguation.

²⁸ Mainly by Terry Copeck.

3.4 Syntactic Coverage: Difficult Issues

3.4.1 Structural Ambiguity

The current version of DIPETT always produces the first (and only the first) parse tree it can find. This does not mean that the first tree is always correct or the “best”: in [Paquet 93], it was found that between 60% and 70% of DIPETT’s (first) trees were correct. In that sense, DIPETT is a noncommittal surface-syntactic parser ([Jacobs & Rau 93]). Based on a few simple attachment heuristics (and no semantics at all), the parser finds the first reasonable parse tree, according to its grammar. For the well-known problem of prepositional phrase (PP) attachment, for example, DIPETT’s heuristics say that “if a PP cannot be attached with confidence to a preceding noun (e.g. an of-PP), then attach it to the preceding verb”²⁹. It is particularly important for Case-based semantic analysis that we minimize the number of PPs that may get incorrectly hidden in an embedded structure and thus be overlooked during the Case marker identification phase of HAIKU — more details in chapter 4. In other words, in case of doubt, they had better be attached to the verb than to the noun.

Despite such heuristics and however good they may be, misattachments caused by structural ambiguity are bound to occur — on the problem of attachment in parsing, see 2.1.4.2. Without a rich semantic model of the text to analyze, there is no way by which we can guarantee that DIPETT will always produce the best analysis. And even with such a model, some input strings are truly ambiguous (e.g. “She repaired the computer with the mouse”): only context or human intervention can help resolve the ambiguity. In order to cope with this problem in a realistic fashion, we are currently designing a post-parsing processor which will allow the user to easily revise DIPETT’s parse trees and correct misattachments when necessary — see section 5.1. This will help to ensure that the trees submitted to HAIKU for semantic analysis are (almost certainly) structurally correct.

It is worth noting that some researchers even question whether it makes good sense to always attempt to strive for a “perfect” parse tree (i.e. with no misattachments). Here is a quote from [Jacobs 92b: p.316]: “Yet phrase attachment does not always contribute very

²⁹ More precisely, DIPETT’s PP-attachment heuristics can be defined as Minimal Attachment (i.e. a constituent tends to attach to an existing nonterminal using the fewest additional nodes) unless there is a clue for Right Association (i.e. a constituent tends to attach to another constituent immediately to its right).

much to the results of text analysis, particularly in data extraction. Even worse, an otherwise inconsequential attachment decision can actually interfere with getting the correct semantic interpretation. ... The result is that a typical parser spends more time deliberating where to attach phrases whose attachment *doesn't* help semantic interpretation than it *does* on phrases whose attachment does matter, and parsing is fundamentally misdirected.”. We must say that DIPETT is guilty, to a certain degree, of that weakness. For certain types of ambiguity, DIPETT spends too much time figuring out where to attach certain constituents.

3.4.2 Time Constraints for Problematic Input Strings

When we discussed unparsable input strings and DIPETT's helper (section 3.3.5.1.3), we implicitly assumed the parser would give feedback in a reasonable time. Unfortunately, this cannot be guaranteed. Sooner or later, a *problematic input* will be submitted and the parser will be engaged in an excessively lengthy (even if finite) computation. Our experience shows that such problematic input strings are typically extra-grammatical or highly ambiguous (relative to DIPETT's grammar): in these situations, the parser is forced to explore a huge search space.

We must handle such extreme situations since our KA methodology requires finding a sensible parse tree for any input, as much as possible. We have decided to consider *time constraints*; in other words, the parser may be timed out³⁰. This subjective constraint is specified by the user at the beginning of a parsing session: see the example log in Appendix A (“DIPETT's Driver and User Interface”). It is simply an upper bound which represents the maximum period of time the parser will be allowed to spend on any input string. If it reaches that limit before finding a parse, it will abandon (full) parsing and, if the fragmentary parser is activated (see 3.4.4), ask the user whether it should attempt to parse the input as a series of fragments³¹, or simply abort processing of this particular string — for a different approach to parsing timing-out, see [Strzalkowski 92].

Following is an example of the ‘CPU time limit’ constraint mechanism in action with a preset limit of 15 seconds. In this example, the limit is reached both for full parsing and fragmentary parsing.

³⁰ This mechanism was motivated essentially by practical considerations. For a cognitive-science approach to text comprehension in which time plays, by design, a determining role, see [Corriveau 91].

³¹ Fragmentary parsing is subject to the same CPU time limit in order to avoid falling into the same trap.

```

Unit #7: Thus , when you tell QUIZ to report , say , a phone number ,
the dictionary " tells " QUIZ that this item is numeric , how many
numbers it contains , and how to format it .
Process this unit (Y/n), try another Version, or Exit [Y/n/v/e]?

lexical analysis (cpu time) : 0.25 second(s).
First parse for input sentence:

PARSING STOPPED for the following reason:
CPU TIME LIMIT EXCEEDED in ... verb_clause2 ;
> the current CPU time limit is: 15.0 second(s).
At that moment, the input string was:
[,a,phone,number,,,the,dictionary,"tells,"quiz,that,this,item,is,n
umeric,,,how,many,numbers,it,contains]

** Sorry dear user, UNABLE TO PARSE THE INPUT **

MEMORIZER results: # of units fetched (not re-parsed unnecessarily).
noun phrase(s)           : 22
prepositional phrase(s)  : 0
verb phrase(s)           : 1
verbal clause(s) of type 1 : 0
verbal clause(s) of type 2 : 98
subordinate clause(s)    : 8

** Reparse as a series of fragments [n/Y]?

PARSING STOPPED for the following reason:
CPU TIME LIMIT EXCEEDED in ... conj_np ;
> the current CPU time limit is: 15.0 second(s).
At that moment, the input string was:
[numeric,,,how,many,numbers,it,contains,,,and,how,to,format,it,..]
parsing (cpu time) : 30.05 second(s).
Error during parsing: end of analysis.

{ ... }

```

From a practical point of view, the combination of the CPU time limit and the batch parsing mode is very profitable. Since our KA methodology has no inflexible requirements as far as parsing time is concerned³², we have found this combination to be quite handy, especially for the processing of large volumes of text. Typically, one launches the parsing of a large body of text in batch, overnight, after having set a specific CPU time limit, then comes back the next morning with all the results saved in a file waiting to be further analyzed or processed with the semantic analyzer.

³² This does not mean that we do not care about the parser's performance, much to the contrary. But simply that for KA purposes, the question of whether DIPETT can parse any input in a few msec or not becomes secondary.

The timing out mechanism is nevertheless not infallible. It is implemented with a CPU time counter and a group of “check spots” in all the major DCG rules of DIPETT’s grammar. The function of these check spots is simply to compare the current total of CPU time spent so far with the preset limit; when the limit is reached, parsing is stopped. It is thus possible, although quite rare in practice, that a certain input leads the parser on a path in which the next check spot will not be reached before the preset limit, but only soon afterwards. In section 3.6, we will return to the issue of how the allotted CPU time affects the parser’s performance.

3.4.3 Parsing Conjoined Structures

As discussed in section 2.1.4.3, the parsing of conjoined constituents is a very difficult problem in practical parsing. A general and efficient solution remains to be found. We present conjoined structures recognized by DIPETT and explain how some of them are processed. In particular, we focus on conjoined clauses and verb phrases, the most difficult situations.

3.4.3.1 Conjoined Structures Handled by DIPETT

We list below the types of conjoined structures handled by the parser. The number of conjoined items is *unbounded*, i.e. the grammar does not impose any upper bound on the number of conjoined items. In what follows, we provide examples for every situation.

full (main) clauses:

multiple coordination (e.g. “The battery may be disconnected, the connection may be loose, or the bulb may be faulty”), correlative coordination (e.g. “Not only did Sam destroy his first document but he also destroyed all the subsequent ones”), and simple coordination (e.g. “David could not use his car yesterday because the new battery he had bought was dead”).

verb phrases:

e.g. “Lucy typed in her letter, printed it on the laser printer, and sent it to her mother”. (DIPETT’s parse tree explicitly represents the implied subject, ‘Lucy’ in the example here, of the second and following clauses.)

noun phrases and nominal clauses:

regular conjunction with 'and' and 'or' (e.g. "They accidentally deleted their system file, their new applications, and all their backup files") , plus the 'either-or' and 'neither-nor' bipartite constructions (e.g. "James hopes either to win the tournament or to finish in second place").

prepositional phrases:

same as for noun phrases and nominal clauses: e.g. "She sent an application form to the ACME company and to the EMCA corporation"; e.g. "Bob will go neither to the meeting nor to Madonna's party".

modifiers and subordinate clauses:

Modifiers of verb phrases and sentences can be joined by conjunctions 'and', 'or' (e.g. "From my perspective and from yours, we surprisingly came to the very same conclusion")³³, or form a sequence without conjunctions (e.g. "Amazingly, all of a sudden, they decided to launch the operation right away, without further delay").

adjectives:

The adjectives that precede the head noun of a noun phrase, possibly separated by noun classifiers (in nominal compounds), can be optionally joined by commas and conjunctions 'and' and 'or' (e.g. "We went to a long and boring presentation"). They can also form a sequence without conjunctions (e.g. "You now have the useful, new computer data file").

adverbs:

same as for adjectives above (e.g. "They proudly and patiently defended their title").

verbs:

same as for adjectives above (e.g. "Frank washed and dried the dishes").

3.4.3.2 Top-Level Parsing Heuristics

³³ DIPETT does not analyze the 'yours' ellipsis though.

When DIPETT is presented with an input string, it must first decide which top-level rules it should attempt to apply and in which order. Ideally, we would like to see the parser apply the rules in an optimal order that would minimize parsing time by first selecting the most promising rules. There are three major top-level rules for declarative statements in the grammar:

- 1) 'mult_coor': deals with multiple coordinated sentences;
- 2) 'compound': deals with compound sentences, i.e. correlative and simple coordination;
- 3) 'non_compound': simple and complex sentences, i.e. a single main clause with zero or more subordinate clauses.

A naive strategy would simply apply all three top-level rules in a pre-determined and immutable order — and hope for the best³⁴. We experimented with this strategy and found it quite inefficient. Because of the important number of situations dealing with conjunctions that the parser accepts (see 3.4.3.1), considerable time was wasted in juggling with rules doomed to fail right from the beginning. For example, there is no need to tentatively apply the 'mult_coor' and 'compound' rules when the input string contains only one verb. In order to deal with this problem, we devised a heuristic-based, top-level rule selection mechanism that we now describe.

3.4.3.2.1 Top-Level Rule Selection

The heuristic is based on three parameters, all of which are obtained after lexical analysis (i.e. before parsing): the number of potential³⁵ verbs in the input string, the presence (or absence) of potential coordinators in the input string, and the “verbal density” which, roughly speaking, indicates how potential verbs are distributed in the input string. This third parameter deserves further discussion as it is computed by a pre-parsing processor.

Inter-verbal intervals (substrings) are collected in order to evaluate the density of verbs in the current input string. An interval, noted 'int' here, is a substring located between two verbs. It is represented as 'int(POS_A, POS_B, Substring, Flag)' where

³⁴ It is the author's experience that real-life texts have a definite propensity to bring out the worst, not the best!

³⁵ Potential is used in the sense of 'candidate' for multi-part-of-speech tokens which can function as verbs: only parsing will tell us if they were in fact analyzed as such.

POS_A and POS_B mark the relative position (begin/end) of the substring in the current input string. The second parameter, 'Substring', contains all the substring's tokens. 'Flag' is a boolean which takes the value 'true' if all tokens in the substring (interval) are amongst the following tokens: ['and', 'or', 'but', ',', 'not']. In this situation, verbs are said to have a high density, which is typical of input strings containing conjoined verbs (e.g. We saved and deleted our files many times before"). We will say that such an interval is *dense*. Otherwise, 'Flag' is set to 'false'. We note that the computation takes into account the number of auxiliaries, but it is first-class verbs, i.e. those that cannot function as auxiliaries, that most influence the result.

Two special cases are the 'pre_int' and the 'post_int': the first goes from the very beginning of the input string to the first verb, and the second goes from the last verb to the end of the input string. Thus, for an input containing N potential verbs, we have the following situation:

[pre_int] Verb#1 [int#1] Verb#2 [int#2] ... Verb#N [post_int].

For example, the input "Beth and Ruth went to the mall this morning and they bought several new things for the house" has the following interval-based segmentation: [pre_int: "Beth and Ruth"], Verb#1: 'went', [int#1: "to the mall this morning and they"], Verb#2: 'bought', and [post_int: "several new things for the house"].

Let us now look at the algorithmic description of the top-level rule selection mechanism which bases its result on the analysis of the string's features we have just discussed. To simplify the presentation, we omit the word 'potential' and refer to "verbal intervals" simply as intervals.

IF the input contains 0 verbs,

THEN fail;

(i.e. no need to apply any of the 'mult_coor', 'compound', and 'non_compound' rules. If the input is a verbless fragment, it will be recognized by some other rule.)

ELSE, IF the input contains 0 sentential coordinators or only one verb,

THEN apply the 'non_compound' rule only;

(i.e. the input is either a simple or a complex sentence.)

ELSE, IF the input contains more than 2 verbs, and it contains correlative sentence coordinators, and all intervals ('int') have a 'false' boolean value, and the second coordinator is located in one of the middle intervals,

THEN first apply the 'compound' rule, followed by the 'non_compound' rule if the first fails;

(i.e. the computation returns two candidate rules to be applied in a precise order.)

ELSE, IF the input contains more than 2 verbs, and it contains at least one sentential coordinator, and this coordinator is located in one of the middle intervals,
THEN

IF all intervals are dense (i.e. have a 'true' Flag boolean value),
THEN first apply the 'non_compound' rule, followed by the 'compound' rule if the first fails;

ELSE, IF not all intervals are dense and not all intervals have a subject,
THEN first apply the 'compound' rule, followed by the 'non_compound' rule if the first fails;

ELSE, IF all intervals have a subject,
THEN first apply the 'mult_coord' rule, followed by the 'compound' rule if the first fails, followed by the 'non_compound' rule if the second fails;

ELSE, IF all intervals are dense,
THEN apply the 'non_compound' rule only; if it fails and if there is at least one sentential coordinator and if it is located in a middle interval then apply next the 'compound' rule;

ELSE, IF not all intervals are dense, and there is at least one potential sentential coordinator and if it is located in a middle interval,
THEN apply the 'compound' rule only;

ELSE, apply the 'non_compound' rule.

3.4.3.2.2 Parsing Coordinated Clauses with Look-Ahead

Conjoined sentences and conjoined verb phrases are usually most expensive to parse. We have devised two look-ahead mechanisms to improve this. The mechanisms simply check the lexical categories of tokens ahead of the parser's current position in the input string. This information is used by the parser (during parsing) to identify potential joining-points for conjoined sentences, and to avoid applying rules that cannot succeed.

When the rule for coordinated main clauses, simple or correlative, is invoked (i.e. the 'compound' rule in 3.4.3.2.1), the parser checks whether the input contains potential conjoined verb phrases. One look-ahead mechanism seeks a conjunction in a place that would separate the input into two sentences. Another look-ahead mechanism checks whether each of two parts contains at least one potential verb. If not, the joining-point suggested by the first look-ahead is rejected and another is sought. If none is found, the parser's attempt to recognize the input as conjoined sentences fails. If the second look-

ahead mechanism finds a potential verb in each part, then and only then does the parser actually attempt to parse conjoined sentences.

The parser also handles a phenomenon specific to conjoined verb phrases: implicit subjects and complements. (Arguably, this also happens in conjoined sentences, but its surface-syntactic manifestation is that of conjoined verb phrases.) In a typical situation, illustrated by the input “I have printed and read my messages”, the parser must recognize three facts: the (implicit) subject of the second verb phrase is ‘I’; the (implicit) complement of the first verb phrase is ‘my messages’; the verb sequence of the second verb phrase is ‘have read’ (with the auxiliary of the first verb phrase). Scoping of negation and adverbs in conjoined verbs is handled, too. For example “John *did not accidentally print and read* my personal messages” is analyzed as “[*John*] [*did not accidentally print*] and [*did not accidentally read*] [*my personal messages*]”. DIPETT handles all that and represents the missing constituents explicitly in the parse tree.

3.4.4 Parsing Fragments

3.4.4.1 Fragments as Naturally Occurring Text Objects

The ultimate goal of the TANKA system is the processing of free-form technical texts. One of the assumptions about our source texts is that all non-textual elements have been removed. In our domain, typical non-textual elements are tables or examples (e.g. data, programs, results)³⁶. When such elements are deleted, they leave behind “holes”. Most holes are located between sentences and do not affect the structure of the text. Some holes, however, cause *fragments* to appear in the text. Fragments are valid input strings that represent substructures of English sentences. Most fragments are “almost complete” sentences or verb phrases. Others are noun phrases, which also appear in enumerations, or prepositional phrases.

Here are several examples taken from the QUIZ text. The part of the original text that remains after the removal of non-textual elements has been underlined.

- 1) For example, > SORT ON DATEJOINED D.

³⁶ See [Delisle *et al.* 91] and [Matwin & Szpakowicz 92] for a machine-learning approach to the representation and acquisition of knowledge from textual examples in technical text.

2) To find out which items in your files are keys, simply access the file and ask QUIZ to show you the items in it, as in

> ACCESS EMPLOYEES
> SHOW ITEMS

3) For instance, if part of your input is

> SORT ...
> HEADING ...
then QUIZ prints the report-group ...

4) SET REPORT COPIES 2 allows you to specify the number of ...

Since our semantic analysis is based on Cases, only fragments with verbs, i.e. “almost complete” sentences and verb phrases, seem to be of interest to KA. In our domain, the most frequent non-textual elements are QUIZ code and printed reports. The holes they leave usually play the role of noun phrases. One strategy that was investigated was to fill the hole with a dummy noun phrase marker which indicates to the parser that normally there should be a legal noun phrase at this location. The input becomes parsable after this substitution. The net effect is to discard the conceptually weak fragments (without verbs) and rehabilitate those that seem conceptually meaningful.

But the situation with these apparently weak fragments is in fact more complicated than this. In particular, if we manage to construct a powerful re-attachment facility (as mentioned in 3.4.1), some verbless fragments might be “discardable” or “re-attachable” (for example, to a valid verb phrase), by the user. In that sense, they may not be as problematic or useless as we have just said.

3.4.4.2 Parsing an Input as a Series of Fragments

Fragments are also important for other practical considerations. As we saw in 2.1.4.5., the notion of “fragmentary”, “partial”, or “segmentation” parsing has recently become relatively important in NLP. Most of the time, the reason is that full parsing may get too costly, especially in terms of time, for certain types of “real” input strings, i.e. texts that have not been simplified or edited before processing, as in TANKA.

In DIPETT, fragmentary parsing is used essentially as a “fallback” or “last resort” parsing strategy, especially in combination with the CPU time limit discussed in 3.4.2. The main argument behind this is the following: if the parser cannot find a complete analysis for a given input, either because it could not do so in a certain allotted time or

because the input is simply extra-grammatical, then a fragmentary analysis is still better than no analysis at all. This is also perfectly justifiable from a KA perspective: acquiring fragments of knowledge is better than no acquisition at all (although it could be complicated for “tricky” fragments).

More precisely, fragmentary parsing in DIPETT is parsing an input as a series of (concatenated) major substructures (substrings). Here is how the core of DIPETT’s fragmentary parser is defined:

```
last_resort_parsing(parse_tree__frag_series(FRAGS,Tag))
--> frag_series(FRAGS), terminal(Tag).
```



```
frag_series([Frag_H|Frag_R])
--> frag_element(Frag_H), frag_series(Frag_R).
frag_series([]) --> {}.
```



```
frag_element(sentence_fragment(FE))
--> sentence_core_s(FE).
```

{above: a fragment may be a simple sentence, NounPhrase + VerbPhrase. In that case, the original string was probably a complicated sentence.}


```
frag_element(sentence_fragment(FE))
--> verbphrase(2,_,_,nil,[],FE,nil).
```

{above: a fragment may be a simple verb phrase.}


```
frag_element(FE)
--> sentence_frag(FE).
```

{above: a fragment may be one of these ‘sentence_frag’ (defined elsewhere in the grammar): a noun phrase, a prepositional phrase, an adjectival phrase, or an adverbial phrase.}


```
frag_element(_)
--> terminal(_), {!, fail}.
```

{above: a terminal punctuation mark cannot be parsed as a valid fragment.}


```
frag_element(skipped_token(ST))
--> [dct(_:ST/_,_)].
```

{above: at this point, the first token of the current list of input tokens is simply skipped and fragmentary parsing continues with the remaining list of tokens.}

We now provide an example of fragmentary parsing in action, with a preset CPU time limit of 5 seconds. As can be observed, the fragmentary parser lists useful information on the analysis: the CPU time required, the number of fragments, the number of tokens that had to be skipped, and the percentage of the input string recognized. In the following example, the input was analyzed as a series of these five substructures: a prepositional

phrase ('pp'), a simple sentence ('statement'), an adjective ('attrs'), a nominal clause ('nominal_clause'), and an adverb ('adv').

Unit #2: In the next few pages you can learn how to ask the right questions even if you've never used a computer before .

Process this unit (Y/n), try another Version, or Exit [Y/n/v/e]?

lexical analysis (cpu time) : 0.35 second(s).

First parse for input sentence:

PARSING STOPPED for the following reason:

CPU TIME LIMIT EXCEEDED in ... verb_clause2 ;

> the current CPU time limit is: 5.0 second(s).

At that moment, the input string was:

[ask,the,right,questions,even,if,you,have,never,used,a,computer,befor
e..]

**** Sorry dear user, UNABLE TO PARSE THE INPUT ****

MEMORIZER results: # of units fetched (not re-parsed unnecessarily).

noun phrase(s)	: 2
prepositional phrase(s)	: 0
verb phrase(s)	: 8
verbal clause(s) of type 1	: 0
verbal clause(s) of type 2	: 10
subordinate clause(s)	: 0

**** Reparse as a series of fragments [n/Y]?**

fragmentary parsing (cpu time) : 0.4 second(s).

**** Number of fragments : 5**

**** Number of tokens not parsed : 0**

**** % of input string recognized : 100.0%**

parsing (cpu time) : 5.75 second(s).

parse_tree_frag_series

sentence_fragment

pp in

{ ... }

sentence_fragment

statement

{ ... }

sentence_fragment

attrs

{ ... }

sentence_fragment

nominal_clause

{ ... }

sentence_fragment

adv

{ ... }

end_of_input period

One final remark on fragmentary parsing. It is our experience that this strategy almost always parses any input quite rapidly, i.e. at most a few seconds. But the parser cannot

guarantee that this will always be the case. There are two reasons for this. First, the CPU time limit could be set to such a small value that even fragmentary parsing would not have time to do much. Second, despite the fact that fragmentary parsing is less constraining for the parser, there is still the possibility that it deceives the timing mechanism. And the explanation for this is simple: the fragmentary parser attempts to recognize the longest possible substrings first, thus the possibility of falling into a trap is still present, although in practice rather small.

3.5 Simple Anaphora Resolution for Pronouns: the Pronoun Explicitizer

Another aspect of TP we discussed in section 2.1.4.4, on discourse processing, is the resolution of anaphora. What matters most for our KA needs in the context of TANKA is the resolution of the major simple types of anaphora. Otherwise, if such references to discourse entities remain unresolved (i.e. non-explicit), then semantic analysis will produce elements containing conceptual unknowns that, in the end, will be integrated in the KB, thus representing holes in the acquired domain model.

Because of the approach used in the rest of the system, and because solving anaphora seems essentially as demanding as solving the NLU problem itself³⁷, we again opted for an interactive semi-automatic approach to the resolution of pronoun anaphora³⁸ in DIPETT. Pronoun resolution is performed between parsing and semantic analysis by a separate processor which operates on parse trees. In fact, the purpose of this processor is not so much anaphora resolution as finding non-pronominal counterparts (i.e. referents³⁹) of pronouns.

The main cases that we decided to handle are the following ones: personal pronouns (including the generic ‘I’, ‘you’, and ‘we’), the ‘it’ pronoun (including the empty ‘it’), and the ‘one’ pronoun (including the generic case)⁴⁰. The strategy we use is that of “explicitization”. Our goal is to resolve pronoun anaphora by replacing their occurrences in the parse tree by the entity to which they refer, thus making explicit the intended discourse entity. When such an explicitization operation occurs, the updated parse tree records the information that the new entity was obtained *via* this process. This may be useful for further semantic processing.

Further details are provided below where we show a commented log of the pronoun “explicitizer” in action for the input “Bob copied a file and he didn’t know what it contained”.

³⁷ This is, in a nutshell, one of the main results of Hobbs’ pioneering work on this topic (see [Hobbs 78]).

³⁸ More precisely, the explicitization process described here works for both cases of endophora: anaphora (i.e. backward reference) as well as for a restricted case of cataphora (i.e. forward reference). For the first, the referent may be found in the current or the previous input. For the second, the referent may be found only in the current input.

³⁹ See [Merlo 93] which proposes an algorithm for the incremental computation of possible and impossible intrasentential coreferents.

⁴⁰ Relative pronoun resolution and verb phrase ellipsis resolution have a lot in common with the resolution of these pronouns: see , respectively, [Cardie 92a, 92b] and [Hardt 92a, 92b].

Unit #1: Bob copied a file and he didn't know what it contained .
lexical analysis (cpu time) : 0.117 second(s).
First parse for input sentence:

{ ... }

parsing (cpu time) : 0.233 second(s).

parse_tree__decla_or_imper
compound_sentence
structure

{ ... }

HAIKU: initiating Case-based semantic analysis ...

Proceed with "pronoun explicitization" [n/Y]?
ANAPHORA "EXPLICITIZATION" PRE-PROCESSOR ...

{below: 'ENTITIES' are explicit entities already introduced in the current discourse, normally *via* noun phrases, whose scope covers the current input and the previous one. They represent the list of potential referents⁴¹ for the 'PRONOUNS' that must be "explicitized".}

ENTITIES IN CURRENT PARSE TREE (OR CLAUSE):
[entity(proper_noun(bob),number(sg3),
tokens([bob])),
entity(determinatives(deter(a)),head_noun(noun(file)),number(sg3),
tokens([a,file]))]
PRONOUNS IN CURRENT PARSE TREE (OR CLAUSE):
[entity(ref(pers_pron(he)),number(sg3),
tokens([he])),
entity(ref(pron(it)),number(sg3),
tokens([it]))]

Trying to find a referent for:
entity(ref(pers_pron(he)),number(sg3),
tokens([he]))

List of candidate entities in this input:
entity(proper_noun(bob),number(sg3),
tokens([bob])) (1) ;
entity(determinatives(deter(a)),head_noun(noun(file)),number(sg3),
tokens([a,file])) (2) ;

{below: by selecting '1', the user tells the explicitizer that 'he' refers to 'Bob'.}

if appropriate, please select one of these referents;
enter a number between 1 and 2 (or 0 if none is appropriate): 1

⁴¹ At present, referents are filtered through the simple test of number agreement with that of the pronoun: the referents that pass this test are called "potential referents". When a conceptual ontology will be applicable to nouns (see 5.2), a more discriminating filtering will then be possible. In the current example, this would avoid the situation in which the system proposes 'a file' as the potential referent for the pronoun 'he'. We also note that gender information associated with pronouns in The Collins dictionary could be used profitably in this situation.

{below: the “new functor” (in the resulting updated parse tree) keeps track of the fact that this entity was obtained through pronoun explicitization.}

```
>> anaphora resolved: new functor is entity_pro

Trying to find a referent for:
entity(ref(pron(it)),number(sg3),
      tokens([it]))
List of candidate entities in this input:
entity(proper_noun(bob),number(sg3),
      tokens([bob])) (1) ;
entity(determinatives(deter(a)),head_noun(noun(file)),number(sg3),
      tokens([a,file])) (2) ;
```

{ below: by selecting ‘2’, the user tells the explicitizer that ‘it’ refers to “a file”.}

```
if appropriate, please select one of these referents;
enter a number between 1 and 2 (or 0 if none is appropriate): 2

>> anaphora resolved: new functor is entity_pro

UPDATED PARSE TREE(parse_tree__decla_or_imper(compound_sentence
{ ... }
```

Here is another example showing other aspects of the pronoun explicitizer with the input “We agree that one cannot always win every battle”.

```
Unit #2: We agree that one cannot always win every battle .
lexical analysis (cpu time) : 0.117 second(s).
First parse for input sentence:

{ ... }

parsing (cpu time) : 0.166 second(s).

parse_tree__decla_or_imper
  simple_sentence
    structure

{ ... }

HAIKU: initiating Case-based semantic analysis ...

Proceed with "pronoun explicitization" [n/Y]?
ANAPHORA "EXPLICITIZATION" PRE-PROCESSOR ...

ENTITIES IN CURRENT PARSE TREE (OR CLAUSE):
[entity(determinatives(deter(every)),head_noun(noun(battle)),number(sg3),
      tokens([every,battle]))]
PRONOUNS IN CURRENT PARSE TREE (OR CLAUSE):
[entity(ref(pers_pron(we)),number(pl),
      tokens([we])),
 entity(ref(pron(one)),number(sg3),
      tokens([one]))]
```

```

Trying to find a referent for:
entity(ref(pers_pron(we)),number(pl),tokens([we]))
> FOR THIS PRONOUN, SUGGESTING "generic", OK [n/Y]? n

```

```

Trying to find a referent for:
entity(ref(pers_pron(we)),number(pl),tokens([we]))
List of candidate entities in this input:
[]

```

{below: the user did not want to give a generic interpretation to the ‘we’ in this input. Instead, he enters a new noun phrase which provides the valid referent for this occurrence of ‘we’.}

```

Looking for the referent of:
entity(ref(pers_pron(we)),number(pl),tokens([we]))
Please, select one of these possibilities:
(0) check entities in the previous parse tree;
(1) empty subject (e.g. "IT is warm today.");
(2) generic (e.g. "ONE would think that ...");
(3) associate the referent with a new NP;
(4) unknown or unable to determine referent.
Enter your choice [0/1/2/3/4]: 3
> enter a simple NP (e.g. "the data files."): [the,authors,..]

```

```

lexical analysis (cpu time) : 0.033 second(s).
NP_parsing (cpu time) : 0.017 second(s).

```

```

entity(determinatives(deter(the)),head_noun(noun(author)),number(pl),
tokens([the,authors]))

```

```
>> anaphora resolved: new functor is entity_pro_new_NP
```

{below: however, for ‘one’, the generic interpretation is selected.}

```

Trying to find a referent for:
entity(ref(pron(one)),number(sg3),tokens([one]))
> FOR THIS PRONOUN, SUGGESTING "generic", OK [n/Y]?

>> anaphora resolved: new functor is entity_pro_generic

```

```

UPDATED PARSE TREE(parse_tree__decla_or_imper(simple_sentence
{ ... }

```

Resolution of two other types of pronoun reference would also need to be accomplished: deictics⁴² (e.g. “these expenses”) and definite references (“the children”). However, at the time of writing, we have not decided yet if these references should be resolved within the explicitizer or within the MNB.

⁴² To refer deictically means to refer in a pointing manner, e.g. “Pick that up and put it over there” ([Hirst 81: p.7]).

3.6 Testing the Syntactic Coverage with Actual Texts: Results and Discussion

It is all too common in the literature to find unsupported claims about broad-coverage parsers or free-form text parsers. Often, it is not explicitly stated what is meant by “broad-coverage”, or no serious testing was performed and the reader has no idea of the percentage of the intended input that can in fact be successfully analyzed.

This section summarizes the series of tests that have been performed on the DIPETT parser, and the main observations we derived from these. We think the results support our claim that DIPETT is indeed a broad-coverage parser of an important subset of technical English: at least, it should give the reader a good idea of what we understand by this notion.

3.6.1 Tuning the Parser with Hand-made Input Strings

During the implementation and testing phases of DIPETT’s grammar, every DCG rule was tested with specific test cases. For the whole grammar, a total of 600 test cases were assembled in a series of files: they range from 3-word to 26-word long (correct) input strings. It goes without saying that all of them are parsed successfully by DIPETT: the parser finds at least one parse tree for all of them. For 90% of the test cases, the first parse tree is the “best” tree (according to our knowledge of the language).

We outline the nature of these various test cases. Roughly speaking, the order is bottom-up with increasing complexity. It must be clear that these tests were prepared with the objective of testing every rule in the parser but, obviously, not all possible combinations. Each test case is described in terms of what syntactic elements it non-exclusively emphasizes, in the input sentences it contains.

As these test cases are useful to re-test the parser after an update of the grammar — and as such they represent a kind of maintenance documentation — we also include the file names for every test case, enclosed in brackets. The goal here is to show what kinds of tests have been performed on the DIPETT parser.

3.6.1.1 Test Cases for Noun Phrases and Nominal Clauses

Noun Phrase:

determinatives (pre-, central, post-) [test_np_a_1];
noun types (common, proper, mass, substitutes) [test_np_a_2];
adjective phrases, plus adverbs as adjectives [test_np_a_3];
noun pre-modifiers and post-modifiers [test_np_a_4];
noun phrase core, including genitives, and its various equivalents [test_np_a_5];
noun phrase postmodifiers and conjoined prepositional phrases [test_np_a_6];
conjoined noun phrases and, plus either-or and neither-nor structures [test_np_a_7];

Nominal Clause:

various types of nominal clauses [test_np_b_1];
conjoined nominal clauses, plus either-or and neither-nor structures [test_np_b_2];

3.6.1.2 Test Cases for Verb Phrases and Sentences

declarative, plus Here+Be, There-exist, and imperative constructions [test_s_v_a_1];
verb sequences, plus all tenses and negation forms [test_s_v_a_2];
verb sequences, plus adverbs and conjoined verbs [test_s_v_a_3];
regular active and passive statements, plus statives [test_s_v_a_4];
various complements for actives, passives, and statives [test_s_v_a_5];
conjoined verb phrases, including elided modals or auxiliaries [test_s_v_a_6];
subordinate clauses, for initial, medial, final positions [test_s_v_a_7];
special contextual markers and QUIZ examples [test_s_v_a_8];
coordinated clauses, including correlatives and multi-coordinated [test_s_v_a_9];
sentences with (ending) tags [test_s_v_a_10];
various complex and ambiguous conjoined units [test_s_v_a_11];

3.6.1.3 Miscellaneous

QUIZ code examples and various strings functioning as noun phrases [test_np_c_1];
various fragments [test_s_v_b_1];
title or caption as sentence equivalents and parenthetical units [test_s_v_c_1];
simple questions [test_q_a_1].

3.6.2 The QUIZ User Manual

3.6.2.1 The Document

The QUIZ manual is the text on which we performed the most extensive and detailed testing. The reason is that it represented well the typical text we envisaged at the commencement of the TANKA project, especially as a representative of the class of (technical) expository texts on fourth generation software — see [Black *et al.* 92] for related work applied to the domain of computer manuals.

This manual is a guide to Cognos' PowerHouse QUIZ report writer, a component of a fourth generation package. The 1986 version, of which we had an on-line copy, contains seven chapters, which represent some 135 pages, plus three appendices, a glossary, and an index. We have restricted ourselves to the testing and processing of the first six chapters, i.e. the seventh and the three appendices simply contained too much QUIZ code or other non-textual material and, thus, were not appropriate for English parsing. The resulting text contains 10863 words and an average of 14.2 words per sentence.

We have submitted the full first six chapters to DIPETT in order to test and evaluate its coverage and performance. Before we present the detailed statistics, we now provide a brief summarization of these six chapters, with their respective title, number of pages, and number of "units" (i.e. full sentences or fragments).

CHAPTER	TITLE	# pages	# units
1	Simple QUIZ Reports	11	171
2	Creating More Complex Reports	7	100
3	Customizing Reports	14	162
4	Summaries and Planned Reports	11	111
5	Defined Items	11	111
6	Production Reports	13	112

To give the reader the flavour of this text, we conclude the presentation of the QUIZ manual by showing an excerpt from chapter 1.

"In the next few pages you can learn how to ask the right questions even if you've never used a computer before. The key to the simplicity of QUIZ lies in its

relationship with the data dictionary. In the dictionary, the application designer sets down the basic rules and standards. In it are described the records you'll be reading with QUIZ. (Elements are named and given features that QUIZ uses when building your reports.)

Thus, when you tell QUIZ to report, say, a phone number, the dictionary "tells" QUIZ that this item is numeric, how many numbers it contains, and how to format it. The dictionary also contains the information QUIZ needs to provide you with automatic page and column headings, and many other features.

You can find more information in the dictionary manual, but right now we're going to produce a report. For the purpose of this sample report, let's say you work for a company that has offices in five U.S. and Canadian cities. You need a list that tells you who works in each of those offices, their employee numbers, and the date each joined the company. And you need it now."

3.6.2.2 Results

As we mentioned above, we wanted to test DIPETT's coverage and performance when faced with "real" texts. We also wanted to study the influence of parsing time on the parser's performance. This explains why statistics were computed for five different CPU time limits, i.e. 30, 60, 90, 120, and 300 seconds⁴³. For all input strings, DIPETT first attempted to get a full analysis in the allotted time; if it failed to do so, it would then attempt to find a fragmentary analysis (see 3.4.4.2), still with the same time limit. Appendix C ("Parsing Statistics for the QUIZ Manual") lists the statistics for each chapter, then for the whole document. Observations and comments are reported in 3.6.4.

3.6.3 The T4043 Tax Guide

The 1990 Canadian T4043E Tax Guide is a supplementary document on child care expenses, as a complement to the general Tax Guide. It has some 15 pages and contains three chapters. We discarded the second chapter since it contains mostly boxed text presented in a non-linear fashion with lots of visual pointers. The remaining text contains 305 units, totaling 5773 words and an average of 18.9 words per sentence (Vs 14.2 for the

⁴³ These timings are for a SPARC-2 workstation. Roughly speaking, the equivalent for a SPARC-10 is 50% smaller.

QUIZ manual). As we did for the QUIZ manual, we here show a brief excerpt from the first chapter.

"For purposes of child care expenses, an eligible child could be your child, or your spouse's child, or a child for whom you have claimed a personal amount and who was, at any time during the year under 14 years of age (born in 1976 or later), or 14 or over (born before 1976), dependent on you or your spouse, and mentally or physically infirm.

Note: The "personal amount" referred to in this Guide is the non-refundable tax credit amount claimed for a dependent child. Refer to lines 304 and 305 in the 1990 General Tax Guide for more information on these non-refundable tax credits.

Who is a supporting person? A supporting person of an eligible child is the child's parent, your spouse, or any other person claiming a personal amount for the eligible child, if the parent, spouse, or other person claiming the personal amount, lived with you at any time in 1990, and at any time within the first 60 days of 1991. If you married in 1990, your spouse becomes a supporting person for all of 1990. If you separated in 1990, and did not reconcile by March 1, 1991, your spouse is not a supporting person.

Example:

Miguel was a single parent with a 5-year-old son. He worked full time in 1990 and left his son at a day care center. Miguel is the "supporting person". However, in November 1990, Miguel married Rosa and they have lived together since then. They both work. Miguel's income is higher than Rosa's."

For this text, we did not compute the same statistics as we did for the QUIZ manual, except of course for the major ones which are automatically provided by DIPETT. The experiment consisted of running DIPETT both on the QUIZ manual and on the T4043 text with "extreme" CPU time limits of 5 minutes and 5 seconds⁴⁴, and then comparing the differences in coverage.

⁴⁴ These are for a SPARC-10 workstation. Roughly speaking, they would translate into 10 minutes and 10 seconds, respectively, on a SPARC-2.

Unlike the tests described in 3.6.2, which were all performed under user supervision, the ones here were performed using DIPETT's batch parsing mode (see 3.3.4). This explains why the overall coverage appears slightly weaker here: when the parser stumbled upon a 'grouping' ("idiom"), or an unknown word (the lexicon was missing a few entries), it could not get the expected assistance from the user. However, the impact of this on the interpretation of the results is negligible: all the following results were obtained *via* this batch mode, under the same constraints and conditions.

Here are the results as produced by DIPETT. Comments follow in section 3.6.4.

5 MINUTES

THE T4043 TAX GUIDE

- 0- cpu time limit: 5.0 minute(s).
- 1- Total nb. of units processed: 305
- 2- Total nb. of units parsed as fragments: 131 (42.9508%)
- 3- Total nb. of units not parsed: 27 (8.85246%)
- 4- Total nb. of units skipped: 16 (5.2459%)
- 5- Total nb. of units fully parsed: 131 (42.9508%)
- 6- % of success. parses, full or frag. (excluding #4): 90.6574%
- 7- Total nb. of dictionary additions: 0
- 8- Total cpu time (parsing & other tasks): 9.72691 hour(s).
- 9- Average cpu time per input processed⁴⁵: 1.91349 minute(s).

THE QUIZ MANUAL

- 0- cpu time limit: 5.0 minute(s).
- 1- Total nb. of units processed: 767
- 2- Total nb. of units parsed as fragments: 334 (43.5463%)
- 3- Total nb. of units not parsed: 34 (4.43286%)
- 4- Total nb. of units skipped: 34 (4.43286%)
- 5- Total nb. of units fully parsed: 365 (47.588%)
- 6- % of success. parses, full or frag. (excluding #4): 95.3615%
- 7- Total nb. of dictionary additions: 0
- 8- Total cpu time (parsing & other tasks): 18.2669 hour(s).
- 9- Average cpu time per input processed: 1.42896 minute(s).

5 SECONDS

THE T4043 TAX GUIDE

- 0- cpu time limit: 5.0 second(s).
- 1- Total nb. of units processed: 305
- 2- Total nb. of units parsed as fragments: 122 (40.0%)
- 3- Total nb. of units not parsed: 69 (22.623%)
- 4- Total nb. of units skipped: 16 (5.2459%)
- 5- Total nb. of units fully parsed: 98 (32.1311%)
- 6- % of success. parses, full or frag. (excluding #4): 76.1246%
- 7- Total nb. of dictionary additions: 0
- 8- Total cpu time (parsing & other tasks): 27.0489 minute(s).
- 9- Average cpu time per input processed: 5.32109 second(s).

THE QUIZ MANUAL

- 0- cpu time limit: 5.0 second(s).
- 1- Total nb. of units processed: 767

⁴⁵ "per input processed" means "per individual input string submitted on input to DIPETT".

```

2- Total nb. of units parsed as fragments: 308 (40.1565%)
3- Total nb. of units not parsed: 126 (16.4276%)
4- Total nb. of units skipped: 34 (4.43286%)
5- Total nb. of units fully parsed: 299 (38.9831%)
6- % of success. parses, full or frag. (excluding #4): 82.8104%
7- Total nb. of dictionary additions: 0
8- Total cpu time (parsing & other tasks): 56.1394 minute(s).
9- Average cpu time per input processed: 4.39161 second(s).

```

3.6.4 Discussion

Sections 3.6.2 and 3.6.3 presented the results of extensive tests that have been conducted on the DIPETT parser. We now draw the main conclusions from these results and make several observations.

First, the salient points of the results listed in section 3.6.2 — timings here are for a SPARC-2 workstation:

- we observe a certain variance in the nature of the text, across chapters. If we compare chapters 1 and 4 (for a CPU time limit of 5 minutes), for instance, we see that chapter 1 is fully analyzed in a proportion of 53.2%, and fully or partially analyzed in a proportion of 91.8%. Chapter 4 gets 46.8% and 100% for the same items.
- the biggest gain in terms of successful coverage, for full or fragmentary analysis (see item (10) in the tables), is in the 30-60 seconds interval. After that, from 60 to 90, from 90 to 120, and from 120 to 300, the gains are rather meagre. This suggests that there is no point in setting a very high CPU time limit: if the parser cannot find an analysis, or reject the input, in 60 seconds or less, then chances are it will not do much better even with a longer time limit.
- full parsing of the manual ranges from 45% to 50%. Full or fragmentary goes from 90% to 95%. Although full parsing is moderate, we think it compares very well to other broad-coverage parsers described in the literature⁴⁶, especially if we

⁴⁶ A recent comparative study of six large-coverage parsers of French reports success rates averaging 57% for a hand-coded corpus that, in our opinion, has nothing in common with the difficulties DIPETT has to handle in our texts — see details in T.A. Informations (Revue Internationale du Traitement Automatique des Langues), 32(2), 1991. Also, [Black *et al.* 92] reports that in a recent effort to “gauge the state of the art in parsing”, they submitted a corpus of 35 sentences, averaging 13 words per sentence, to four of the major large-coverage parsers for general English. It was found that “the best-performing parser was correct for 60% of the sentences and the remaining parsers were below 40%”. Their evaluation required

consider that we deal with real texts. When augmented with fragmentary parsing, the parser covers up to 95% of the text, which translates into the possibility of extracting more or less partial knowledge from essentially the whole text.

- the fragmentary parser is quick and productive: when it is given an input, it finds a fragmentary analysis 90% of the time under the allotted time limit and basically never rejects an input as extra-grammatical. In another analysis of DIPETT's fragmentary analyses for the QUIZ manual, it was found that the average number of fragments is 3.92 (per unit) and the average percentage of the input string recognized is 85.19% (per unit).

Second, the salient points of the results listed in section 3.6.3 — as above, timings are for a SPARC-10 workstation (i.e. half of those for a SPARC-2):

- no matter what the CPU time limit, the coverage of the QUIZ manual is better than that of the Tax Guide. The difference, approximately 5%, is primarily for full parses. This comes as no surprise when we compare the average number of words per sentence: it is 30% higher for the Tax Guide. A close examination of the Tax Guide also shows that this text is particularly "hostile" to automatic parsing as it contains numerous enumerations as well as involved conjoined and embedded structures.
- a somewhat surprising result is the relatively good performance of DIPETT under a severe constraint of only 5 seconds for its CPU time limit. The results show that, for both texts, overall coverage (i.e. full or fragmentary) goes down about 13%. The main decrease is for the fully analyzed units, which support 75% of that drop. But still, for the QUIZ manual, the parser managed to fully analyze 39% of the units and covered, either fully or by fragments, 83% of the units. This experiment tends to confirm the observation we made above which says that there is no need to set a very generous CPU time limit. In that vein, we list below the results for the three even more severe time constraints of 2, 1, and 0.5 seconds, for the QUIZ manual only.

0- cpu time limit: 2.0 second(s).
1- Total nb. of units processed: 767

that the parses accurately delimit and identify the major constituents and assign roughly correct parts of speech, i.e. they were not looking for "perfect" parses.

```

2- Total nb. of units parsed as fragments: 288 (37.5489%)
3- Total nb. of units not parsed: 175 (22.8162%)
4- Total nb. of units skipped: 34 (4.43286%)
5- Total nb. of units fully parsed: 270 (35.2021%)
6- % of success. parses, full or frag. (excluding #4): 76.1255%
7- Total nb. of dictionary additions: 0
8- Total cpu time (parsing & other tasks): 30.8437 minute(s).
9- Average cpu time per input processed: 2.41281 second(s).

```

```

0- cpu time limit: 1.0 second(s).
1- Total nb. of units processed: 767
2- Total nb. of units parsed as fragments: 231 (30.1173%)
3- Total nb. of units not parsed: 252 (32.8553%)
4- Total nb. of units skipped: 34 (4.43286%)
5- Total nb. of units fully parsed: 250 (32.5945%)
6- % of success. parses, full or frag. (excluding #4): 65.6207%
7- Total nb. of dictionary additions: 0
8- Total cpu time (parsing & other tasks): 19.8497 minute(s).
9- Average cpu time per input processed: 1.55278 second(s).

```

```

0- cpu time limit: 0.5 second(s).
1- Total nb. of units processed: 767
2- Total nb. of units parsed as fragments: 140 (18.2529%)
3- Total nb. of units not parsed: 378 (49.2829%)
4- Total nb. of units skipped: 34 (4.43286%)
5- Total nb. of units fully parsed: 215 (28.0313%)
6- % of success. parses, full or frag. (excluding #4): 48.4311%
7- Total nb. of dictionary additions: 0
8- Total cpu time (parsing & other tasks): 12.3487 minute(s).
9- Average cpu time per input processed: 0.965999 second(s).

```

In our opinion, rigorous testing of DIPETT shows that it is indeed a large-coverage parser of (a significant subset of) technical English, as long as we are concerned with obtaining a first reasonable, not necessarily perfect, full or fragmentary analysis of its input. Assuming that post-parsing processes will take care of potential inadequacies in the resulting parse trees (see 5.1), with the user's assistance when appropriate, we claim that DIPETT assumes very well its role in the context of a semi-automatic approach to knowledge acquisition from texts.

This said, extensive testing also revealed a number of areas in which the grammar was weak: punctuation (especially commas), complicated combinations of simple substructures, certain types of conjoined structures, enumerations, certain types of complement, etc. We recently compiled a list of explanatory analyses on the QUIZ manual units that were rejected in order to identify the main limitations ([Paquet 93]). This will guide us in future updates of DIPETT's grammar, especially in focusing on the

improvements that could be most beneficial to the parser's coverage and performance. We will get back to this in section 5.3.

For instance, one of the major observations of [Paquet 93] was DIPETT's inability to deal correctly with singular noun phrases containing an empty determiner. Thus, the parser would reject "He went to school" simply because it expected a determiner in front of 'school'. In a recent experiment, we updated DIPETT's grammar in order to handle this phenomenon and reparsed the entire QUIZ manual. We obtained a significant increase of 5% for full analyses.

A general trend that seems to prevail is that DIPETT's grammar is idealistic: it is particularly good at parsing text that is written in accordance with Quirk *et al.*'s definitions and recommendations of well-formedness. However, when the input is extra-grammatical, relative to such an ideal grammar, the parser becomes rather slow and often rejects input strings for unimportant reasons (e.g. a misplaced comma).

CHAPTER 4

4. HAIKU: ACQUIRING CASE PATTERNS VIA INTERACTIVE SEMANTIC ANALYSIS.....	120
4.1 HAIKU as Part of a Text Processing System.....	122
4.2 Toward Comprehensive Case-based Semantic Analysis.....	127
4.2.1 Limitations of Simple Case-based Frameworks.....	127
4.2.1.1 One Case per Argument.....	127
4.2.1.2 Noun-Noun Relationships.....	128
4.2.1.3 The Need for an Extended Theory.....	128
4.2.2 A Comprehensive Framework for HAIKU.....	129
4.2.2.1 The Set of Cases.....	129
4.2.2.2. Notation.....	130
4.2.2.3. Details of the Proposed Framework.....	132
4.2.2.3.1. Complex NPs and embedded clauses.....	132
4.2.2.3.2. Stative and conjoined verbs.....	133
4.2.2.3.3. Structures other than simple declarative clauses.....	135
4.2.2.3.4. Conjoined structures (ref. 3.4.3.1).....	136
4.3 Interactive Case Analysis for Knowledge Acquisition: the Core of HAIKU.....	140
4.3.1 Data Structures.....	140
4.3.2 A Self-contained, Extended Example.....	142
4.3.3 The Process of Case Analysis and its Crucial Algorithms.....	145
4.3.3.1 From Multi-Clause Inputs to a Series of Individual but Interlinked Clauses.....	146
4.3.3.2 Case Analysis of Individual Clauses.....	148
4.4 Expressing Situations $\mathcal{A}_1$ to $\mathcal{A}_5$ in Terms of Three Parameters: the Verb, the Case-marker Pattern (CMP), and the Case Pattern (CP).....	151
4.4.1. Dealing with New CPs.....	151
4.4.2. Situation #1 ($\mathcal{A}_1$).....	151
4.4.3. Situation #2 ($\mathcal{A}_2$).....	152
4.4.4. Situation #3 ($\mathcal{A}_3$).....	153
4.4.5. Situation #4 ($\mathcal{A}_4$).....	154
4.4.6. Situation #5 ($\mathcal{A}_5$).....	155
4.5 Difficult Issues: Finding Suitable Example Sentences and Suggesting Relevant Case Patterns.....	157
4.5.1 Finding and Using Example Sentences.....	157
4.5.1.1 Avoiding Complete Rejection.....	157
4.5.1.2. Problems with the Closeness Metric.....	158

4.5.1.3. A Possible Scenario.....	158
4.5.2. Challenging HAIKU: a Difficult Example.....	159
4.6 HAIKU's Closeness Computation: Semantic Similarity Based on Syntactic Markers.....	162
4.6.1 General Definitions of Matching Case Marker Patterns.....	162
4.6.1.1 Definitions of Matching.....	163
4.6.2 Basic Computations for the Closeness Metric.....	165
4.6.2.1 Computing the Closeness Factor within pum.....	165
4.6.2.2 Computing the Comparison Coefficient within npum.....	165
4.6.2.3 Simple Examples of Closeness Calculations.....	167
4.6.2.4 Computational Details of the Closeness Metric.....	168
4.6.3 Examples of Closeness Calculations.....	170
4.6.4 Further Refinements of 'pum' and 'compute_CC'.....	172
4.6.5 Strategic and Algorithmic Aspects of the Closeness Computation.....	173
4.7 Testing the Semantic Analyzer with Actual Texts: Results and Discussion.....	176
4.7.1 An Experiment with HAIKU.....	176
4.7.2 General Observations.....	177
4.7.3 Results for Chapters 1 and 2 of the QUIZ Manual.....	178
4.7.4 Discussion.....	179

4. HAIKU: ACQUIRING CASE PATTERNS VIA INTERACTIVE SEMANTIC ANALYSIS

The preceding chapter presented the elements and features of the DIPETT parser. The present chapter describes the details of the HAIKU semantic processor which is considered as a separate, post-parsing module¹. Together, the DIPETT and HAIKU modules make up the heart of TANKA's TP subsystem — HAIKU is sometimes called the Conceptual Analyzer² in this chapter. Earlier descriptions of the semantic analyzer's design can be found in [Copeck *et al.* 90, 92], [Delisle 91], and [Delisle *et al.* 92]. Before we get inside of HAIKU, we first provide a brief overview of Case Theory³, the linguistic theory behind the semantic module.

In this thesis, our approach to semantic analysis can be looked at as a variant of Fillmore's Case Theory as described in [Fillmore 68]⁴. Our proposal is made in the context of KA: it can be characterized as Case-based text understanding for KA, based on full syntactic analysis⁵. Basically, Case Theory views the clause⁶ as the most important syntactic element in the sentence and focuses on the verb within the clause. Cases express

¹ Semantic analysis is performed after syntactic analysis. This "pipeline" model differs from others based on incremental approaches to semantic interpretation in which meaning is computed on a word-by-word basis, as the sentence is "read" from left to right — for instance, see [Haddock 89].

² The Conceptual Analyzer described here has not much in common with those developed at the Yale AI Project during the 1975-1980 period (see [Schank & Riesbeck 81] and [Riesbeck 83]). They differ mainly in the following ways: i) we perform conceptual analysis after full syntactic analysis whereas they perform conceptual analysis using semantic and limited syntactic knowledge in parallel, ii) we base our semantics on Case grammars whereas they base theirs on Conceptual Dependencies ([Schank 75]), iii) we perform syntax-based parsing whereas they perform expectation-based parsing (or one of its derivatives, such as script-driven parsing, interest-driven parsing, goal-driven parsing [Riesbeck 82]), and iv) we do not rely on significant *a priori* knowledge to perform conceptual analysis whereas they rely on tremendous amounts of *a priori* knowledge (mostly semantic but also pragmatic to a certain extent) which is hidden in Conceptual Dependency representations.

³ As already mentioned in a footnote of chapter 1, 'Case' denotes markers for conceptual relations.

⁴ It is interesting to note that although [Fillmore 68] is always cited as the first description of Fillmore's Case Theory, his theory originates in an earlier paper, i.e. [Fillmore 66]. Also, in his 1966 paper, Fillmore cites [Tesnière 59] as his "source of inspiration". This simply means that the modern school of Case Theory, of which Fillmore is the originator, is rooted in Valency theory, of which Tesnière is the originator. For further details, see [Somers 87].

⁵ See [Grishman & Sterling 90] for another approach to information extraction from texts based on full syntactic analysis. They described an application which creates a data base from US Navy messages describing naval encounters.

⁶ Clause is used here in the standard linguistic sense, i.e. a group of words containing a subject and a finite verb, forming a sentence or part of a sentence.

the meanings of the other major syntactic constituents⁷ in the clause in terms of their relationships with the verb — they are often considered as “arguments” of the verb. Each of these relationships is a Case, i.e. a semantic role, and each Case is unique⁸ in the clause.

Cases in sentences are denoted by Case markers. Cases are semantic abstractions and Case markers are their lexico-syntactic materializations. Case markers are realized in two different ways: lexically (e.g. a preposition that introduces a prepositional phrase), and positionally⁹ (i.e. subject or ‘psubj’, direct object or ‘pobj’, indirect object or ‘piobj’). For instance, in the following sentence, taken from Fillmore’s well-known 1968 paper, we have:

John opened the door with the key.

Here, the noun phrase ‘John’ denotes the (positional) Agent Case, the noun phrase ‘the door’ denotes the (positional) Object Case, and the prepositional phrase ‘with the key’ denotes the (lexical, introduced by ‘with’) Instrument Case. In Fillmore’s early analysis, according to [Winograd 83: p.317], the Agent is the instigator of an event, the Object is the entity that moves or changes or whose position or existence is in consideration, and the Instrument is the stimulus or immediate physical cause of an event.

⁷ Traditionally, Case Theory focuses on semantic roles (Cases) that noun phrases play with verbs. In order to handle complex sentences, noun equivalents and nominal clauses of any sorts can be the object of Cases.

⁸ Except, maybe, for locations. However, it could be argued that repeated locations are in fact the materialization of a fragmented, complex location Case.

⁹ Strictly speaking, because of DIPETT’s functional nature, these “positional” constituents are not merely surface positional. We refer to them as such to distinguish them from Cases introduced by lexical markers (e.g. prepositional phrases).

4.1 HAIKU as Part of a Text Processing System

The knowledge-processing part of TANKA assists the user in building a conceptual model of the domain. Concepts derived from linguistic structures are added to a growing conceptual network that represents knowledge about the domain. The linguistic processing in TANKA begins with surface-syntactic analysis (performed by DIPETT), followed by Case-based semantic analysis (performed by HAIKU) that results in a Case pattern of a sentence. The result of these steps is a semantic representation of the sentence. The semantic representation is further transformed into a small self-contained semantic network, and matched against an ever-growing network representing the system's current understanding of the text.

In what follows, we present the design of HAIKU and the details related to its current implementation. HAIKU's analysis is centred around the predicate-argument structure of sentences, denoted by Case markers and the associated Case fillers. In the rest of the chapter, we discuss the algorithms and the data structures that underlie the interactive process, guided by the system but supervised by the user.

We now show a partial, actual log (in Courier typeface) of HAIKU in action. It illustrates the simple user interface dialogue and gives an idea of the highest level algorithms involved. As in chapter 3, explanations were added in Times typeface, bold, and enclosed in braces. As well, the original log was slightly reorganized to improve the presentation. Subsequent details on HAIKU should be easier to follow after having gone through the following commented log. Some details, e.g. CP and CMP, will be provided later.

```
Unit #1: Bob printed a QUIZ data file on the new printer from his
office .
lexical analysis (cpu time) : 0.1 second(s).
First parse for input sentence:
```

```
{ ... }
```

{below: first, the input string is parsed by DIPETT.}

```
parsing (cpu time) : 7.85 second(s).
```

```
parse_tree__decla_or_imper
  simple_sentence
    structure
      single_main_clause
        declarative
```

```
{ ... }
```

{below: then, HAIKU proceeds with semantic analysis.}

HAIKU: initiating Case-based semantic analysis ...

```
UPDATED PARSE TREE(
  parse_tree__decla_or_imper(simple_sentence(structure(single_main_clause(
    declarative(*statement1*))),
    tokens([bob,printed,a,quiz,data,file,on,the,new,printer,from,his,office])),
  end_of_input(period))
```

{above: every 'statement' (main clause) in the parse tree produced by DIPETT is replaced by a starred symbolic identifier which will be further developed below. This shows how the initial input is decomposed for semantic analysis: the process operates on individual clauses.}

LOGICAL CLAUSAL STRUCTURE

```
top_structure
  [*statement1*]
```

{above: the top-level logical structure shows how the clauses are hierarchically organized and inter-connected *via* conjunctions. In this example, the input is a simple stand-alone clause.}

SYMBOLIC A-LISTS

```
coordinators
clauses
```

```
[*statement1*,
statement(subj(entity(proper_noun(bob),number(sg3),tokens([bob]))),pr
  edicate(regular(verb(print,tense([past_simple]),neg(yes),trans(tr_
    intr))),voice(active),complement(svoo_svoc_svoa([entity(determinat
      ives(deter(a)),head_noun(noun(file,noun_modifiers(pre_modif(conj_n
        oun_pre_mod(implicit_and,[quiz,data])))),number(sg3),tokens([a,qu
          iz,data,file])),[conj_pps(implicit_and,[pp(on,entity(determinative
            s(deter(the)),attributes([attrs(adj(new,pos)),tokens([new]))],head
              _noun(noun(printer)),number(sg),tokens([the,new,printer])),tokens(
                [on,the,new,printer])),p_asyndetic_conj_pp(pp(from,entity(determin
                  atives(deter(his)),head_noun(noun(office)),number(sg3),tokens([his
                    ,office])),tokens([from,his,office])))),tokens([on,the,new,printe
                      r,from,his,office])))),tokens([printed,a,quiz,data,file,on,the,n
                        ew,printer,from,his,office]))]
```

{above: detailed structure of the only main clause found in the input string. The 'SYMBOLIC A-LISTS' structure is a simple (association) list which associates every starred symbol with the sub-structure it stands for (i.e. replaces) in the previous output structures above.}

INPUT

```
STRING([bob,printed,a,quiz,data,file,on,the,new,printer,from,his,office,
ce,.] )
```

{above: the current input string.}

HAIKU: Case Analysis of current input's clauses ...

{below: decomposition of the current clause into its main functional constituents, i.e. subject, verb (main activity), complement, and modifiers.}

CURRENT CLAUSE: *statement1*

```

CURRENT SUBJ   :
subj(entity(proper_noun(bob), number(sg3), tokens([bob])))
CURRENT VERB   : print
CURRENT COMPL  :
complement(svoo_svoc_svoa([entity(determinatives(deter(a)), head_noun(
  noun(file, noun_modifiers(pre_modif(conj_noun_pre_mod(implicit_and,
    [quiz, data])))), number(sg3), tokens([a, quiz, data, file])), [conj_pps
    (implicit_and, [pp(on, entity(determinatives(deter(the)), attributes(
      [attrs(adj(new, pos)), tokens([new]))], head_noun(noun(printer)), numb
        er(sg), tokens([the, new, printer])), tokens([on, the, new, printer])), p_
        asyndetic_conj_pp(pp(from, entity(determinatives(deter(his)), head_n
          oun(noun(office)), number(sg3), tokens([his, office])), tokens([from, h
            is, office])))), tokens([on, the, new, printer, from, his, office]))))
CURRENT MODIFS : [main_verb_modif(nil), clause_modif(nil)]

```

{below: Case analysis begins. HAIKU first shows the Case-marker Pattern (CMP) it found in the current clause: a positional subject, a positional object, a prepositional phrase introduced by 'on', and a prepositional phrase introduced by 'from'. The user is allowed to overrule the CMP found by HAIKU.}

```

CMP found by HAIKU: psubj-pobj-on-from
> Do you wish to overrule this CMP
  (enter d to discard this clause, or
   a to discard the entire input) [y/N/d/a]?

```

{below: HAIKU consults its current KB and finds that the current clause is typical of a situation known as "situation #3" in which an entirely new CMP was found with a known verb (i.e. a verb already encountered before). The associated Case Pattern (CP) will either be new or not.}

S3: entirely new CMP with known verb; CP is known or new.

{below: because the CMP has never before been encountered, HAIKU cannot propose a very likely candidate to the user for his approval. However, it attempts to propose "close" candidates. Note: all the intermediate results returned by the metric are only shown here as a trace of execution; what matters to the user is the closest CP proposed by HAIKU.}

```

entirely new CMP; no obvious candidate CPs to propose
list of close CMPs proposed by the "closeness metric":
[ THE INPUT CMP PROPERLY UNDERMATCHES: ,
  THE INPUT CMP NON-PROPERLY UNDERMATCHES:
  ,match(2.8125, [psubj, pobj, with, from]), match(2.8125, [psubj, pobj, on, adv
  ]),
  THE INPUT CMP PROPERLY OVERMATCHES:
  ,match(2.5, [psubj, pobj, on]), match(2.5, [psubj, pobj, from]), match(0.8333
  33, [psubj, pobj]), match(0.25, [pobj]), match(0.166667, [psubj]),
  THE INPUT CMP NON-PROPERLY OVERMATCHES:
  ,match(0.777778, [psubj, on, in]), match(0.555556, [psubj, pobj, to]), match(
  0.555556, [psubj, pobj, piobj]), match(0.555556, [psubj, pobj, adv]), match(0
  .125, [pobj, adv]), match(0.0833333, [psubj, with]), match(0.0833333, [psubj
  , to]), match(0.0833333, [psubj, for]), match(0.0833333, [psubj, adv])]

```

{below: of all the above candidates, HAIKU shows an example sentence and its associated CP for the closest candidate.}

Candidate CPs with example sentences for the closest CMP (psubj-pobj-with-from):

```

CP: agt-obj-inst-lfrm
Ex: [this, letter, was, prepared, by, tom, with, the, new, computer, from, the
, other, room]
current clause's main verb (infinitive): print
current clause's CMP (or a permutation): psubj-pobj-on-from
most probable or closest CPs, which are of compatible length:
agt-obj-inst-lfrm (1) ;

```

{below: the user is asked to approve this candidate CP, if appropriate. Otherwise, HAIKU will display all the relevant information to facilitate the user's task in entering the correct CP, i.e. the list of valid Cases in HAIKU, as well as the knowledge it has on the two prepositions ('on' and 'from') used to introduce the two prepositional phrases.}

```

if appropriate, please select one of these CPs:
enter a number between 1 and 1 (or 0 if none is appropriate): 0

```

NEW CP: list of 28 legal Cases in HAIKU

```

agt:agent; benf:beneficiary; expr:experiencer; inst:instrument;
obj:object; recp:recipient; caus:cause; manr:manner; val:value;
eff:effect; purp:purpose; freq:frequency; tat:time_at;
tfrm:time_from; tto:time_to; ttru:time_through; dir:direction;
lat:location_at; lfrm:location_from; lto:location_to; ord:order;
ltr:location_through; ornt:orientation; cont:content;
excl:exclusion; acmp:accompaniment; matr:material; meas:measure.
REM: A CASE MAY BE SUFFIXED WITH A '?' TO DENOTE UNCERTAINTY
(e.g. lat?).

```

```

here is what the cmDict knows about this Case marker: on
(format: Case(COMmonness,Example); CO=1:common, CO=2:uncommon)
cont(1,She was reading ON ECONOMICS.)
lat(1,We were sitting ON THE SOFA.)
tat(1,We returned ON MONDAY.)
purp(1,We spent the money ON BOOKS.)
caus(1,We acted ON HIS ADVICE.)
inst(1,I spoke to her ON THE TELEPHONE.)
ornt(1,The bottle lay ON ITS SIDE.)

```

```

here is what the cmDict knows about this Case marker: from
(format: Case(COMmonness,Example); CO=1:common, CO=2:uncommon)
lfrm(1,They came down FROM THE HILLS.)
tfrm(1,We work FROM NINE to five.)
matr(1,We make them FROM POTATOES.)
caus(1,He fainted FROM LACK OF FOOD.)

```

{below: the user entered 'agt-obj-lat-lfrm' as the CP he judged most fitted. Then HAIKU checks whether the elements of the CMP and the CP are correctly paired.}

```

please enter the new CP (e.g. agt-obj-tat),
or enter 'h' to see the current input string and CMP,
or CR to abort ["new CP"/h/CR]? agt-obj-lat-lfrm
CMP & CP will be paired as follows: psubj/agt pobj/obj on/lat
from/lfrm ; Correct [n/Y]?

```

{below: HAIKU now updates its KB according to the new elements of knowledge it just acquired.}

```

UPDATING mDict for print ...
=> print [pobj,psubj-pobj,psubj-pobj-adv,psubj-pobj-on,psubj-pobj-
on-adv,psubj-pobj-on-from,psubj-pobj-to]
[[adv,[manr]], [from,[lfrm]], [on,[lto,lat]], [pobj,[obj]], [psubj,[agt]]
,[to,[lto]]]

```

```

UPDATING cmpDict for psubj-pobj-on-from ...
=> psubj-pobj-on-from [[agt-obj-lat-
lfrm,'[bob,printed,a,quiz,data,file,on,the,new,printer,from,his,offic
e,.]']]

```

```

UPDATING cpDict for agt-obj-lat-lfrm ...
=> agt-obj-lat-lfrm [print]

```

{below: finally, the Case structure is produced. It contains, for each clause, its main verb, its CMP and its CP, then its subject filler plus its complement fillers, and its modifiers.}

```

CASE STRUCTURES
  clauses
    [*statement1*,
      case_structure(
        verb(print,tense([past_simple]),neg(yes),trans(tr_intr)),
        psubj-pobj-on-from,
        agt-obj-lat-lfrm,
        entity(proper_noun(bob),number(sg3),tokens([bob])),
        [entity(determinatives(deter(a)),head_noun(noun(file,noun_modif
        iers(pre_modif(conj_noun_pre_mod(implicit_and,[quiz,data])))
        )),number(sg3),tokens([a,quiz,data,file])),[conj_pps(implici
        t_and,
          [pp(on,entity(determinatives(deter(the)),attributes([attrs(adj
          (new,pos)),tokens([new])),head_noun(noun(printer)),number(s
          g),tokens([the,new,printer])),tokens([on,the,new,printer])),
          p_asyndetic_conj_pp(pp(from,entity(determinatives(deter(his)
          ),head_noun(noun(office)),number(sg3),tokens([his,office])),
          tokens([from,his,office])))),tokens([on,the,new,printer,fro
          m,his,office]))],
          [main_verb_modif(nil),clause_modif(nil)])]]

```

HAIKU semantic analysis (cpu time) : 3.383 second(s).

{ ... }

We note that 'case_structure' is a regular structure defined as

```
case_structure(MV, CMP, CP, Subject, Complements, Modifiers)
```

where 'mv' is the clause's main verb, 'cmp' is the clause's Case-marker Pattern, 'cp' is the clause's Case Pattern, 'subject' is the clause's (syntactic) subject, 'Complements' is the clause's (syntactic) complement(s), and 'Modifiers' is the clause's (syntactic) adverbial modifier(s).

4.2 Toward Comprehensive Case-based Semantic Analysis

In this section we explain precisely what we mean by Case analysis (CA) and list the sorts of clauses and phrases DIPETT can parse and HAIKU must therefore deal with. We then present an original framework that constitutes HAIKU's foundation for semantic analysis geared towards KA *via* TP. This framework is implemented in Prolog in the current version of HAIKU: see 4.3 for high-level algorithmic aspects of the Case analysis process.

4.2.1 Limitations of Simple Case-based Frameworks

In all fairness, the above introduction to Case Theory¹⁰ would not be complete without a few words on its limitations¹¹. First, the definition of a set of Cases is a delicate issue. This includes the Case names themselves as well as the set of semantic relations that one wants to consider: we will come back to this in section 4.2.2.1. Second, the one-Case-per-argument constraint¹² seems to be violated for certain verbs (e.g. it can be argued that the verb 'sell' has its Cases Agent and Source filled by the very same argument). Third, Case systems¹³ ignore relationships such as possession (e.g. the baby's pyjamas) which operate directly between nouns and do not involve the verb.

In the next two subsections, we look at how we intend to deal with the last two issues in HAIKU (and TANKA). Then in 4.2.1.3, we explain why an extended theory is seriously needed.

4.2.1.1 One Case per Argument

We do not consider the violation of the one-Case-per-argument constraint as a significant problem in technical texts, or even in general. This for two main reasons: i) this violation happens infrequently, and ii) several solutions have been proposed in the

¹⁰ We do not want to elaborate unduly on the basics of Case Theory as they have been the object of numerous papers and textbooks since their inception. In particular, for a thorough coverage of the topic, we strongly recommend [Somers 87] to the interested reader.

¹¹ Interestingly enough, these limitations have historically been more important to linguists than to researchers in computational linguistics or AI.

¹² As expressed in [Fillmore 68], this constraint says that each argument is associated with just one Case, so that there is a one-to-one matching of Cases and arguments in any proposition.

¹³ We refer to Case systems as computational-linguistic paradigms based on Case Theory or Case grammars.

literature — see for instance [Somers 87: chapter 8] — which are amenable to our needs, should we decide to consider this as a special case in the future.

4.2.1.2 Noun-Noun Relationships

About the third limitation mentioned above, we consider it as negligible. We argue that sufficient linguistic meaning is organized around the verb to permit a system of reasonable power to be based solely on it¹⁴. In addition, semantic relationships deemed important are often elevated to the centre of the clause and expressed through verb constructions: “the baby has pyjamas” (*versus* “the baby’s pyjamas”).

4.2.1.3 The Need for an Extended Theory

Case systems have always been controversial — see [Somers 87: chapters 7-9] for a discussion of their pros and cons. We have opted for Case analysis because it closely matches the surface syntactic form of text fragments¹⁵. This form is all there is in the knowledge-scant acquisition task we have set ourselves.

We wish to formulate a Case-based semantic theory to account for realistic sentences. Such sentences tend to be more complicated than those typically used to illustrate Case systems, in which a single main verb is related in a straightforward way to a few simple noun phrases (NPs) and prepositional phrases (PPs). Furthermore, we want to represent the meaning of entire texts, and for that we need an *extended* theory that will deal with phenomena usually neglected by simple Case systems.

HAIKU will encounter a variety of syntactic phenomena in its input. For many of these, traditional Case theory has little to say. Here is a list of syntactic structures, some of which are relatively complex, that establishes the scope of the semantic analysis required to meet TANKA’s objectives. Section 4.2.2.3 will describe how these structures will be interpreted in the context of the framework we propose.

¹⁴ This argument is in fact related to our approach to the representation of knowledge in technical domains which is based on the idea of associating verbs with domain activities and nouns with domain objects.

¹⁵ See [Wu 93] where a thematic role system which does not closely match the surface syntactic form is proposed. More precisely: “The proposed representation depends more heavily on nested structures than ordinary case systems, thereby reducing the number of thematic roles. A side effect of this is that surface cases do not map straightforwardly onto thematic roles in a one-to-one fashion. For instance, the sentence (5) John moved his furniture from San Francisco to Berkeley is analyzed as an outer *volitive* frame whose EFFECT role is filled by an inner *locative*.” ([Wu 93: p.327]).

- **complex NPs and embedded clauses:**
NPs with post-modifiers, e.g. relative clause, PP, or appositive (“Bob, *the owner*, ...”);
nominal clauses, e.g. *to_infinitive_clause*, *ing_clause*, or *that_clause* (“*That Bob did it* is clear”);
comparative clauses, e.g. “Bob is *more intelligent than your sister*”.
- **stative and conjoined verbs:**
stative verbs, e.g. “Bob is *on holiday*”;
conjoined main verbs, e.g. “Bob *washed and dried* the dishes”.
- **structures beyond simple clauses:**
complex and compound sentences, e.g. “*When Bob saw it, he immediately spoke up*”;
sentences with structures other than Subject-Verb-Object and declarative clauses, e.g. *Here+be* and existential-*There* sentences, imperatives (“*There are three people in Bob's family*”);
structures without a main verb, e.g. a title, caption or fragment (“*The World According to Garp*”)
- **conjoined structures:**
inter-sentential conjunctions ;
intra-sentential conjunctions: NPs and nominal clauses, PPs, verb phrases (VPs), subordinate clauses (sentence-level modifiers), adverbial phrases, and adjectival phrases.

4.2.2 A Comprehensive Framework for HAIKU

4.2.2.1 The Set of Cases

We have defined a set of 28 Cases¹⁶, together with their abbreviations, that are used by the Conceptual Analyzer — see Table 1 below. This list of Cases is the result of several phases of work by the TANKA team ([Barker *et al.* 93]). At one point or another, we

¹⁶ Case names themselves are not so important. What really matters is that TANKA and its users refer to them in a consistent fashion. For a proposal on how to handle Cases without using names (i.e. labels), see [Delisle 90b]. A related topic is that of semantic primitives: for a review of related work and an interesting discussion, see [Fass 89a].

considered, analyzed, and criticized the various Case lists proposed in [Bruce 75], [Celce-Murcia 75], [Cook 79], [Fillmore 68], [Grimes 75], [Larson 84], [Loewen 88], [Somers 87], and [Sparck Jones & Boguraev 87]¹⁷.

Category	Number	Case	Abbreviation
<i>PARTICIPANT</i>	1	Agent	AGT
	2	Beneficiary	BENF
	3	Experiencer	EXPR
	4	Instrument	INST
	5	Object	OBJ
	6	Recipient	RECP
<i>CAUSALITY</i>	7	Cause	CAUS
	8	Effect	EFF
	9	Opposition	OPP
	10	Purpose	PURP
<i>TIME</i>	11	Frequency	FREQ
	12	Time_at	TAT
	13	Time_from	TFRM
	14	Time_to	TTO
	15	Time_through	TTRU
<i>SPACE</i>	16	Direction	DIR
	17	Location_at	LAT
	18	Location_from	LFRM
	19	Location_to	LTO
	20	Location_through	LTRU
<i>QUALITY</i>	21	Orientation	ORNT
	22	Accompaniment	ACMP
	23	Content	CONT
	24	Exclusion	EXCL
	25	Manner	MAN
	26	Material	MATR
	27	Measure	MEAS
	28	Order	ORD

Table 1. List of Cases Used in HAIKU (TANKA)

4.2.2.2. Notation

The presentation uses the following notational conventions:

¹⁷ Another valuable reference on Cases and semantic roles is [Longacre 83]. See also [Dick 92] for a discussion of Somers' Cases and the application of his grid in a knowledge representation framework based on conceptual graphs.

i) major syntactic groups are denoted by their common abbreviations (e.g. NP for noun phrase, NC for nominal clause) or by the mnemonic names used in DIPETT's grammar: subscripts denote the syntactic roles of groups (e.g. NP_{subj} for a NP serving as subject, PP_{compl} for a PP serving as complement).

ii) Cases associated with a verb are shown in the form:

$$\begin{array}{c} \mathcal{CA}(S) \\ \Rightarrow \\ [\text{Case}_1 : \text{LABEL}_1(\text{sgr}_1 : \text{string}_1, \text{mv}), \dots, \text{Case}_n : \text{LABEL}_n(\text{sgr}_n : \text{string}_n, \text{mv})] \end{array}$$

where " $\mathcal{CA}(S) \Rightarrow \dots$ " means "Case analysis of sentence S produces this set of (unordered) Cases ...". The notation " $\text{Case}_i : \text{LABEL}_i(\text{sgr}_i : \text{string}_i, \text{mv})$ " is interpreted as "The i-th Case found by $\mathcal{CA}$ in sentence S for the main verb mv is LABEL_i, which is realized by the substring string_i making up the syntactic group sgr_i".

EXAMPLE:

$$\begin{array}{c} \mathcal{CA}(\text{"John broke the window with a hammer"}) \\ \Rightarrow \\ [\text{Case}_1 : \text{AGT}(\text{NP}_{\text{subj}} : \text{"John"}, \text{break}), \\ \text{Case}_2 : \text{OBJ}(\text{NP}_{\text{compl}} : \text{"the window"}, \text{break}), \\ \text{Case}_3 : \text{INST}(\text{PP}_{\text{compl}} : \text{"with a hammer"}, \text{break})]. \end{array}$$

iii) we now extend this notation to cover more complex sentences. Transformation of an input sentence into a collection of simple clauses is indicated by ' $\leftrightarrow$ ', which is interpreted as equivalence. Conjunction (and disjunction) in the transformed sentence are indicated by parentheses and the symbols ' $\wedge$ ' (and ' $\vee$ '). Thus, a sentence containing two main verbs joined by the conjunction 'and' would be analyzed with the following rule, which can easily be generalized for N main verbs (see 4.2.2.3.2):

$$\mathcal{CA}(\text{two-'anded'-main verbs-Sentence}) \leftrightarrow \mathcal{CA}(\text{Clause}) \wedge \mathcal{CA}(\text{Clause}).$$

EXAMPLE:

$$\begin{array}{c} \mathcal{CA}(\text{"John washed and dried the dishes"}) \\ \leftrightarrow \\ \mathcal{CA}(\text{"John washed the dishes"}) \wedge \mathcal{CA}(\text{"John dried the dishes"}). \end{array}$$

4.2.2.3. Details of the Proposed Framework

The following *semantic interpretation rules* express in simple terms the principles underlying HAIKU's CA process. The rules intentionally focus on more complicated linguistic constructions.

4.2.2.3.1. Complex NPs and embedded clauses

- NPs with post-modifiers:

Ex.1: "John knows the man *who wrote that letter*" (relative clause)

Ex.2: "John knows the man *with the long hair*" (PP)

Ex.3: "John knows Santa Claus, *our federal representative*" (appositive)

INTERPRETATION RULE NPPM. NP post-modifiers do not mark a Case at the sentence's main verb level. Their purpose is to specify the individual(s) to which the statement refers. The same is true of pre- or post-modifiers within a noun phrase (e.g. John's car, data file). However, a NP post-modifier containing a verbal clause requires its own (recursively applied) Case analysis, whose result is attached to the related NP. In the following example, Case₂₁ and Case₂₂ are Cases within Case₂. Curly brackets enclose a Case structure with embedded sub-structures.

EXAMPLE:

CA("John knows the man who wrote that letter")

~

```
[ Case1 : AGT( NPsubj : "John", know ),
  { Case2 : OBJ( NPcompl : "the man who wrote that letter", know ),
    [ Case21 : AGT( NPsubj : "the man", write ),
      Case22 : OBJ( NPcompl : "that letter", write ) ] } ] .
```

- Nominal clauses:

Ex.4: "John wants *to go to New York*" (to_infinite_clause)

Ex.5: "John finished *writing his last letter last night*" (ing_clause)

Ex.6: "John knows *that she has copied his homework*" (that_clause)

INTERPRETATION RULE NC. Nominal clauses mark Cases because they are complements of the sentence's main verb. They are syntactically and semantically more complicated than NPs but play the same role in the sentence. For example, it must be inferred that John is the syntactic subject of *going* in Ex.4 and of *writing* in Ex.5 (whereas "John wants *you* to go to New York" is different). Just as verbal NP post-modifiers do, embedded clauses require separate analysis.

EXAMPLE:

$\mathcal{CA}$ ("John knows that she has copied his homework")

→

[Case₁ : AGT(NP_{subj} : "John", know),
 { Case₂ : OBJ(NC_{compl} : "that she has copied his homework", know),
 [Case₂₁ : AGT(NP_{subj} : "she", copy),
 Case₂₂ : OBJ(NP_{compl} : "his homework", copy)] }].

- Comparative clauses:

Ex.7: "Bob is *more intelligent than your sister*"

Ex.8: "Bob works *faster than his colleagues*"

INTERPRETATION RULE CPT. Comparatives mark a Case_x usually MANR (see Case #25 in Table 1, in 4.2.2.1).

EXAMPLE:

$\mathcal{CA}$ ("Bob works faster than his colleagues")

→

[Case₁ : AGT(NP_{subj} : "Bob", work),
 Case₂ : MANR(COMPAR_CL : "faster than his colleagues", work)].

4.2.2.3.2. Stative and conjoined verbs

- Stative verbs:

Ex.9: "A data file *is* a computer science concept"

Ex.10: "This file *seems* empty"

INTERPRETATION RULE ST. Most sentences with a stative main verb can be seen as hierarchical knowledge-base assertions of varying modality: “This file is empty”; “This file seems empty”. Such assertions introduce new concepts into the current discourse, or associate properties with known concepts. Although they do not mark Cases, they communicate semantic information about inheritance, attributes, act-act relationships and so on, and are retained by HAIKU for further processing by the Mini-Net Builder — work on this is also in progress (see Appendix I).

- Conjoined main verbs:

Ex.11: “Bob *washed and dried* the dishes”

A significant difficulty with conjoined verbs is to establish whether they convey a temporal¹⁸ ordering. Consider “Bob *washed and dried* the dishes” and “Bob *dried and washed* the dishes”. We usually understand the former as “Bob washed the dishes *and then* dried them”. The dried-and-washed example is less obvious since it may sound slightly odd to some people. In “Bob washed the dishes and sang” on the other hand, the acts would probably be interpreted as simultaneous.

INTERPRETATION RULE CMV. Ignore the temporal factor and produce conjoined verb phrases or sentences.

EXAMPLE:

$$\begin{array}{c} \mathcal{CA}(\text{“Bob washed and dried the dishes”}) \\ \leftrightarrow \\ \mathcal{CA}(\text{“Bob washed the dishes”}) \wedge \mathcal{CA}(\text{“Bob dried the dishes”}). \end{array}$$

We also considered two other candidate interpretation rules, of which the first is the one that most accurately represents the intended meaning by “colouring” the conjunction of the two acts.

¹⁸ Conjoined verbs can also convey a causal relationship as in “Bob twisted and broke his ankle” meaning “Bob twisted his ankle causing him to break it”. As with the implicit potential temporal relationship, implicit causal relationships are currently ignored.

ALTERNATE RULE #1: keep a *temporal link* ('later' in this example) between the semantic structures of the two acts.

EXAMPLE:

$CA("Bob \text{ washed and dried the dishes} ")$

$\leftrightarrow$

$CA("Bob \text{ washed the dishes} ") \text{ later:} \wedge CA("Bob \text{ dried the dishes} ")$.

ALTERNATE RULE #2: convert the conjoined verbs into a neologism (e.g. washed-*and-dried*.) which is interpreted as a composite act blending the constituent verbs.

EXAMPLE:

$CA("Bob \text{ washed and dried the dishes} ") \leftrightarrow CA("Bob \text{ washed-and-dried the dishes} ")$.

4.2.2.3.3. Structures other than simple declarative clauses

- Other types of sentences:

Ex.12: "Here is the solution to Bob's problem" (Here+*be*)

Ex.13: "There is more than one solution to this problem" (existential-There)

Ex.14: "See the solution in the preceding chapter" (imperative)

INTERPRETATION RULE HBET. In a Here+*be* or an existential-There sentence, the main verb is usually a form of *be*. These sentences are viewed as statements containing stative verbs (see 4.2.2.3.2), which do not mark Cases. Their purpose is to introduce concepts into the discourse. They may have implications for temporal ordering (e.g. Here+*be*) or existential assertion.

INTERPRETATION RULE IMP. Syntactically, an imperative statement is a declarative sentence without a subject¹⁹. Imperatives, however, are often directed to the reader (who is then the implicit Agent) and do not convey new information. Only certain kinds of imperatives mark Cases, in particular those indicating the sequence of steps to perform a task. These statements require more study. At the time of writing, we have yet to decide exactly what kind of particular processing such statements deserve.

¹⁹ Imperatives in technical texts share several properties with purpose clauses in the context of instruction understanding. see [Di Eugenio 92].

- Fragments:

Ex.15: "Report 1.1: Accessing a single file" (title or caption)

Ex.16: "Control breaks" (fragment)

INTERPRETATION RULE TC. A title or a caption is a meta-textual element intended to facilitate the reader's task of understanding the text. Although some have a structure that seems to merit normal semantic analysis, they rarely contain new information. Consequently, titles and captions do not mark Cases but might deserve treatment as inter-sentential discourse operators.

INTERPRETATION RULE VLF. Verbless fragments do not mark Cases: some may occasionally function as Here+*be* statements. Other fragments that do contain verbs may mark a Case. In particular, fragments obtained from fragmentary parsing and which contain a verb deserve the normal Case analysis: they often contain valuable knowledge elements.

4.2.2.3.4. Conjoined structures (ref. 3.4.3.1)

- Inter-sentential conjunctions:

Ex.17: "Bob has been practicing medicine for more than ten years.

Therefore he is very competent."

Ex.18: "Tom went to Mary's place.

However, he did not know that her boyfriend was there."

INTERPRETATION RULE ISCJ. A conjunction marks a discourse relation rather than a Case. This implies that the conjunctive clause or sentence is semantically linked with one or more preceding statements. Such discourse relations will be handled by the Mini-Network Builder — see Appendix I ("HAIKU's Mini-Network Builder (MNB)").

EXAMPLE: (for Ex.17)

CA("However, he is not very competent")

↔

connector/inter_sentential:however **CA**("he is not very competent").

- Intra-sentential conjunctions:

Complex and compound sentences

Ex.19: "Although I admire her reasoning, I reject her conclusions" (complex)

Ex.20: "I admire her reasoning but I reject her conclusions" (compound)

INTERPRETATION RULE CCS. Coordinated clauses are analyzed independently and their corresponding semantic structures are linked at the discourse level. Discourse relations do not mark Cases, but HAIKU must retain the link between the semantic structures of such clauses: they may share concepts or relations²⁰.

EXAMPLES: (for Ex.19 and Ex.20)

CA ("Although I admire her reasoning, I reject her conclusions")

$\leftrightarrow$

CA ("I reject her conclusions") connector:although CA ("I admire her reasoning");

CA ("I admire her reasoning but I reject her conclusions")

$\leftrightarrow$

CA ("I admire her reasoning") connector:but CA ("I reject her conclusions").

Conjoined verb phrases

Ex.21: "Bob wrote the letter and sent it to his friend"

INTERPRETATION RULE CVP. A sentence with conjoined VPs can be broken into two or more individual clauses in each of which the associated constituents are explicitly represented: Case analysis is then applied to the individual clauses. The resulting structures is linked by a meta-connector, which does not itself mark a Case. (Note: for conjoined verbs, see 4.2.2.3.2 above.)

²⁰ At the time of writing, several aspects dealing with discourse, such as relations and connectors, are currently under development within the boundaries of the Mini-Network Builder (MNB). This work is being conducted by Ken Barker, a member of the TANKA team. As a consequence, the semantic interpretation process described in this chapter is responsible for ensuring that the subsequent modules get the proper information for elements under its jurisdiction.

EXAMPLE:

CA ("Bob wrote the letter and sent it to his friend")

↔

CA ("Bob wrote the letter")

connector/intra_sentential:and

CA ("Bob sent it to his friend").

Conjoined noun phrases and nominal clauses

Ex.22: "Bob broke *the window and the drinking glass*" (conjoined NPs)

Ex.23: "Bob wants *to write a letter and to send it today*" (conjoined nominal clauses)

INTERPRETATION RULE CNPNC. Similar to conjoined main verbs, these conjoined elements could be analyzed as: i) a single composite object to which one Case is assigned; ii) a set of co-objects, with one Case or co-Case attributed to each element; or iii) a set of independent objects translated into corresponding individual clauses that are linked by meta-connectors. Since we think that options ii and iii would tend to unnecessarily complicate the CA process — not to mention the supplementary burden this unwarranted complexity would impose on the user's shoulders — we favour the first option, i.e. only one Case is assigned to the composite object.

Conjoined prepositional phrases

Ex.24: "Bob sent his file *to the A printer and to the B printer*" (the same preposition)

Ex.25: "Bob sent his diskette *in a prepaid mailer and with an urgent stamp*" (different prepositions)

INTERPRETATION RULE CPP. If the prepositions are identical, only one Case is marked. If prepositions differ, a different Case should be attributed to each PP.

Conjoined subordinate clauses and adverbials (sentence modifiers)

Ex.26: "*Slowly but steadily*, Bob learned to drive a school bus"

INTERPRETATION RULE CSCA. Adverbials usually mark MANR. Whether conjoined or not, they mark one Case. A subtle point is that the more adverbials there are in a

sentence, the more complex is the MANR Case. In that regard, adverbials are special: they can be accumulated while they leave the distribution of other Cases unaffected. The general rule is to process adverbials as modifiers of the main verb. An exception are a few discourse-level “comment” adverbs such as *obviously* in “*Obviously*, Bob is not hungry”; they are also called *emphasizers* (see [Quirk *et al.* 85: pp.583-589]) or *meta-comments* (see [Zukerman 91]).

Conjoined adjectivals

Ex.27: “Bob has found a *very large and important* set of data”

INTERPRETATION RULE CADJ. Adjectives, conjoined or not, do not mark Cases.

4.3 Interactive Case Analysis for Knowledge Acquisition: the Core of HAIKU

This section provides a detailed description of HAIKU's Case-based semantic analysis process, which is especially designed for KA from English technical texts. In brief, HAIKU begins its analysis with a parse tree produced by DIPETT, and semi-automatically extracts the semantic pattern which best characterizes its interpretation in terms of elements of the set of semantic relations (i.e. Cases) used in TANKA — we described the set of Cases in 4.2.2.1.

HAIKU accumulates these semantic patterns in its KB and refers to it when it processes new inputs. Inputs that have little in common with this KB do, typically, introduce new elements of knowledge: they deserve to be integrated into this incremental KB²¹, with the user's collaboration²². Otherwise, HAIKU simply asks the user to confirm its proposed interpretation since it managed to "understand" the input by recognizing a known semantic pattern, i.e. a pattern that it had already encountered and "memorized" before. In that sense, HAIKU is instrumental to TANKA's goal of turning knowledge-based text understanding into KA. We claim that the knowledge acquired during this operation constitutes a conceptual model of the text's domain.

We refer the reader to Appendix G ("Examples of Representative Semantic Analyses") for an example log illustrating how Case analysis is realized within HAIKU and, in particular, how the system interacts with the user during the KA process.

4.3.1 Data Structures

Several data structures will be referred to in this section: we now describe them precisely. What we called "HAIKU's KB" above is in fact the combination of the three following dictionaries: the 'MDict' (Meaning dictionary), the 'CMPDict' (Case-marker Pattern dictionary), and the 'CPDict' (Case Pattern dictionary). It must be noted that these may be empty when the system is first used on a new text: HAIKU does not expect a

²¹ In fact, HAIKU can be said to perform a kind of Case-based reasoning, an analogical problem solving paradigm well-known in ML.

²² More precisely, this collaboration follows the principles of Sergot's query-the-user facility ([Sergot 83]) in which the user is regarded as an extension of the system's data base (or KB). In this situation, the job of HAIKU's equivalent to Sergot's facility is to inform the user what information is needed, and to convert his response into Prolog assertions which the Case analysis process can use. According to [Sergot 83], such an approach can be characterized as a form of metaprogramming. However, the work presented in this thesis does not explicitly exploit this concept of metaprogramming.

minimal amount of knowledge to work properly or to be useful. The price to pay is that this initial phase will be more tedious for the user, but it will soon improve as HAIKU processes more and more sentences from the input text and, consequently, acquires more and more knowledge from it.

- **Case-marker Pattern (CMP):** an ordered list of Case markers, linked by hyphens. The resulting expression represents the Case markers appearing in a clause. Notation: '→' is a symbol associating a clause with its CMP. In the context of HAIKU, a clause has only one CMP, i.e. only one syntactic analysis from which a unique CMP is derived.

EXAMPLE: "the program ran at midnight" → psubj-at

("the program" is the subject, the prepositional phrase "at midnight" is the complement).

- **Case Pattern (CP):** an ordered list of Cases, linked by hyphens. The resulting expression represents the Cases occurring in a clause. Notation: '⇒' is a symbol associating a clause with a CP. In the context of HAIKU, a clause normally has only one CP, although it may have more than one if the clause is semantically ambiguous.

EXAMPLE: "the program ran at midnight" ⇒ AGT-TAT

("the program" denotes the agent, "at midnight" denotes the time of the activity).

- **Meaning Dictionary (MDict):** a dictionary of entries for individual words. Verb entries contain a list of CMPs found with the verb, and a list of the Cases associated with the Case markers in these CMPs. Notation: '→' is a symbol which associates a CM (i.e. Case Marker) with the list of Cases it has realized (i.e. marked) so far.

The Meaning Dictionary also contains entries for preposition and adverb Case markers. Each lists the Cases its marker can realize — these lists are fixed and domain-independent: see details in [Barker *et al.* 93].

```
EXAMPLE: mDict(run, [ psubj-at, psubj-pobj-at ],
               [ at → [LAT, TAT],
                 pobj → [OBJ],
                 psubj → [AGT] ] ).
```

- **Case-marker Pattern Dictionary (CMPDict):** a dictionary of entries indexed on CMPs. An entry contains a list of the CPs associated with a CMP in the text. Each CP is illustrated by an example sentence. This dictionary may be initialized with entries for a number of common CMPs. Notation: '⇒' is a symbol associating a CP with a clause. In the context of HAIKU, a CP may be associated with one or more clauses since many syntactically different clauses can have the same semantic pattern.

EXAMPLE: `cmpDict(psubj-at,
 [AGT-LAT ⇒ "The program ran at the office"
 AGT-TAT ⇒ "Updates run at midnight"]).`

- **Case Pattern Dictionary (CPDict):** a dictionary of CP entries that contains a list of verbs associated with the CP in the text.

EXAMPLE: `cpDict( AGT-OBJ-TAT, [ print, run ] ).`

All three dictionaries are updated continuously as processing of text progresses. As mentioned above, a parsable clause should normally be associated with a unique CP. To accomplish this, HAIKU first searches its dictionaries for the CP (*target CP*) that matches the CP of the input sentence most closely, if not perfectly. Next, an example sentence to illustrate this CP is fetched from the dictionary. If the CP and example sentence are not acceptable, the system asks for the user's help: more details later.

4.3.2 A Self-contained, Extended Example

In order to provide an illustration of the algorithm and the contents and organization of the data structures, let us assume that HAIKU has processed the following fragment of a technical manual:

"The daily update executes at 7 a.m.; this program knows Word Perfect file format. We print hourly, while the system prints all reports at noon. You can run jobs at any time.

The operating system executes print jobs at this site; it also runs a report generator at each local site by datalink. Transaction process-

ing runs at all locations; as do user programs. Although the various operating systems run programs, each process owns resources."

HAIKU will break these sentences into eleven simple clauses before performing CA. We have sorted them on their verbs to make it easier to look them up in the dictionaries that follow:

The daily update executes at 7 a.m.
The operating system executes print jobs at this site.
This program knows Word Perfect file format.
Each process owns sufficient resources.
We print hourly.
The system prints all reports at noon.
User programs run at all locations.
The various operating systems run programs.
You can run jobs at any time.
Transaction processing runs at all locations.
The operating system also runs a report generator at each local site by datalink.

After processing from a virgin state, HAIKU's dictionaries have the following contents. Note that not all input sentences appear as examples in the CMP dictionary. Similarly, duplicate CMPs or CPs are represented by a single item.

Meaning Dictionary

```
mDict( execute, [ psubj-at, psubj-pobj-at ],
      [ at → [LAT, TAT], pobj → [OBJ], psubj → [AGT] ] ).

mDict( know, [ psubj-pobj ],
      [ pobj → [OBJ], psubj → [AGT] ] ).

mDict( own, [ psubj-pobj ],
      [ pobj → [OBJ], psubj → [AGT] ] ).

mDict( print, [ psubj-adv, psubj-pobj-at ],
      [ adv → [FREQ], at → [TAT], pobj → [OBJ],
```



```

    psubj → [AGT] ] ).

mDict( run, [ psubj-at, psubj-pobj, psubj-pobj-at, psubj-pobj-at-by ],
    [ at → [LAT, TAT], by → [INST], pobj → [OBJ],
    psubj → [AGT] ] ).

```

Case-marker Pattern Dictionary

```

cmpDict( psubj-adv,
    [ AGT-FREQ ⇔ "We print hourly" ] ).

cmpDict( psubj-at,
    [ AGT-LAT ⇔ "User programs run at all sites",
    AGT-TAT ⇔ "The daily update executes at 7 a.m." ] ).

cmpDict( psubj-pobj,
    [ AGT-OBJ ⇔ "Each process owns sufficient resources" ] ).

cmpDict( psubj-pobj-at,
    [ AGT-OBJ-LAT ⇔ "The system executes print jobs at this site",
    AGT-OBJ-TAT ⇔ "The system prints all reports at noon" ] ).

cmpDict( psubj-pobj-at-by,
    [ AGT-OBJ-LAT-INST ⇔ "The operating system also runs a report
    generator at each local site by datalink" ] ).

```

Case Pattern Dictionary

```

cpDict( AGT-FREQ,      [ print ] ).
cpDict( AGT-LAT,       [ run ] ).
cpDict( AGT-OBJ,       [ know, own, run ] ).
cpDict( AGT-OBJ-INST-LAT, [ run ] ).
cpDict( AGT-OBJ-LAT,   [ execute, run ] ).
cpDict( AGT-OBJ-TAT,   [ print, run ] ).
cpDict( AGT-TAT,       [ execute ] ).

```

4.3.3 The Process of Case Analysis and its Crucial Algorithms

This section provides a precise description of the framework defined above in 4.2.2. If we look at that description as the “what and why” of our proposal for semantic analysis, the following can be said to represent the “how”, i.e. the algorithmic definition of the CA process. As a matter of fact, this very definition was implemented in Prolog and is at the heart of HAIKU as a piece of software: Appendix F (“HAIKU’s CA Processor”) presents the implementation.

4.3.3.1 From Multi-Clause Inputs to a Series of Individual but Interlinked Clauses

HAIKU’s Case-based semantic analysis operates on single clauses. Complex sentences are decomposed into simple clauses by a tree-transforming algorithm, called the CA driver (see Appendix F.1), that is relatively straightforward, assuming correct attachment of Cases to clauses. As mentioned before, since attachment is a difficult, knowledge-intensive linguistic problem, the system will employ a single attachment strategy that yields correct results reasonably often. We are currently working on a simple parse tree editor that will allow the user to modify any unsatisfactory attachments interactively, right after parsing and just before CA (see 5.1).

We now describe the gist of the CA driver’s main algorithm for the systematic decomposition of an arbitrary English input, sentence or fragment, into a stream of interlinked clauses to which CA will subsequently be applied. For fragments, only substructures containing verbs are considered for Case analysis.

```

IF the input parse tree contains conjoined verb phrases or conjoined verbs,
    THEN first decompose it according to the rules of interpretation of sections
    4.2.2.3.4 and 4.2.2.3.2, respectively.

EXTRACT all clauses from the input parse tree (possibly updated by the above step)
and collect them in a list, as well as their logical organization if there are any clausal
connectors (i.e. conjunctions). Individual clauses are recognized in the input by these
syntactic substructures which are produced by DIPETT: certain relative clauses, to-

```

infinitive clauses, 'that' clauses, regular and genitive 'ing' clauses, 'wh'-interrogative declarative clauses (e.g. whether/if clauses), and statements²³.

We now show an example log illustrating the steps involved in this pre-CA phase.

```
{ ... }
```

Unit #1: Tom removed all his files and Beth wrote and printed a new letter .
lexical analysis (cpu time) : 0.15 second(s).
First parse for input sentence:

```
{ ... }
```

parsing (cpu time) : 0.167 second(s).

```
parse_tree__decla_or_imper
  compound_sentence
    structure
      head_main_clause
        declarative
          { ... }
            next_main_clause
              coord and
              declarative
                { ... }
                  end_of_input period
```

HAIKU: initiating Case-based semantic analysis ...

{below: HAIKU detected conjoined verbs and explains to the user how they are interpreted, in accordance with its CMV interpretation rule (see 4.2.2.3.2).}

WARNING! The input contains 2 DISTINCT CONJOINED VERB(S):
conj_verbs(and,
 [verb(write,tense([past_simple]),neg(yes),trans(tr_intr)),
 verb(print,tense([past_simple]),neg(yes),trans(tr_intr))])

Semantic Analysis will proceed according to this inference rule:
SUBJ [Verb1 & Verb2 & ... & VerbN] COMPL =>
SUBJ Verb1 COMPL, SUBJ Verb2 COMPL, ... , SUBJ VerbN COMPL.
An input containing N (distinct, conjoined verbs will generate N clauses for HAIKU.

```
>> CONJOINED VERBS: clause #1 ...
verb(write,tense([past_simple]),neg(yes),trans(tr_intr))
```

UPDATED PARSE
TREE(parse_tree__decla_or_imper(compound_sentence(structure(head_main_clause(declarative(*statement1*)),next_main_clause(*coord1*,declarative(simple_sentence(structure(single_main_clause(declarative(*stateme

²³ A 'statement' is a regular sentence with a subject, a main verb, and a complement. This structure is also used in the definition of other structures that do not explicitly appear here, such as the that-clause (also called the report clause).

```
nt2*))),tokens([beth,wrote,and,printed,a,new,letter])))),tokens([tom
,removed,all,his,files,and,beth,wrote,and,printed,a,new,letter])),end
_of_input(period))
```

{below: the top-level structure is two coordinated main clauses.}

```
LOGICAL CLAUSAL STRUCTURE
top_structure
  [*statement1*,*coord1*,*statement2*]
```

{below: and these clauses are joined by the coordinator 'and'.}

```
SYMBOLIC A-LISTS
coordinators
  [*coord1*,coord(and,nil)]
clauses

[*statement1*,statement(subj(entity(proper_noun(tom),number(sg3),tok
ns([tom]))),predicate(regular(verb(remove,tense([past_simple]),neg(ye
s),trans(tr))),voice(active),complement(svo(entity(determinatives(pre
deter(all),deter(his)),head_noun(noun(file)),number(pl),tokens([all,h
is,files])))),tokens([removed,all,his,files])))]

[*statement2*,statement(subj(entity(proper_noun(beth),number(sg3),tok
ens([beth]))),predicate(regular(verb(write,tense([past_simple]),neg(y
es),trans(tr_intr))),voice(active),complement(svo(entity(determinativ
es(deter(a)),attributes([attrs(adj(new,pos)),tokens([new]))),head_nou
n(noun(letter)),number(sg3),tokens([a,new,letter])))),tokens([wrote,
and,printed,a,new,letter])))]

INPUT
STRING([tom,removed,all,his,files,and,beth,wrote,and,printed,a,new,le
tter..])
```

HAIKU: Case Analysis of current input's clauses ...

{below: all set for Case analysis; details in 4.3.3.2.}

```
CURRENT CLAUSE: *statement1*
CURRENT SUBJ   :
subj(entity(proper_noun(tom),number(sg3),tokens([tom])))
CURRENT VERB   : remove
CURRENT COMPL  :
complement(svo(entity(determinatives(predeter(all),deter(his)),head_n
oun(noun(file)),number(pl),tokens([all,his,files]))))
CURRENT MODIFS : [main_verb_modif(nil),clause_modif(nil)]

CMP found by HAIKU: psubj-pobj

{ ... }
```

4.3.3.2 Case Analysis of Individual Clauses

This section discusses Case analysis of a single clause with one main verb: this is what the CA driver has ensured before we get to the analysis of individual clauses. The algorithm can be characterized as a search in the space of CMPs currently known to the system — this space grows as accumulation of Case patterns progresses. The search uses a heuristic based on the formal similarity between the input CMP and the CMPs in the dictionary. If a perfectly matching pattern cannot be found, linguistically informed partial matching is attempted (see details in 4.6).

Before we present the details of the CA process, we must mention that there is some further decomposition involved here. Just as the CA driver takes care of inter-clausal decomposition, another processor takes care of intra-clausal decomposition²⁴. The latter extracts, for every individual clause, its subject, its main verb, its complement, and its adverbial modifiers. Once HAIKU has identified these constituents, it can easily determine the CMP that should be associated with the current clause.

The CA algorithm (see Figure 4.3.3.2, page 150) uses the following terminology:

- i) *input verb* refers to the main verb of the input; *input CMP* refers to the CMP of the input;
- ii) a *new* pattern is one not yet encountered by HAIKU with the input verb, while an *entirely new* pattern has not yet been encountered with any verb.

There are five meaningful combinations of input verb, CMP and CP, depending on whether or not HAIKU has encountered each before. The algorithm dealing with each situation uses the labels $\mathcal{A}_i$, $i = 1, \dots, 5$. This code finds one or more target CPs and identifies suitable example sentences used in the interaction with the user. Following the algorithm, in section 4.4, each situation is discussed and illustrated with examples. We refer the reader to Appendix F.2 for the Prolog implementation of the individual-clause CA processor.

²⁴ This supports well the claim made in chapter 1 that our approach is fundamentally compositional.

If a clause is parsable, it should be associated with a unique CP. To accomplish this, HAIKU first searches its dictionaries for the target CP that matches the CP of the input sentence most closely. Next, an example sentence to illustrate this CP is fetched from the dictionary.

Procedure `find_close` identifies CMPs close to input CMP, according to a closeness metric defined later in 4.6.

```

IF input verb is in MDict                -- known verb V
THEN
    IF input CMP is among CMPs in MDict entry for input verb
                                           -- known CMP already associated with V
    THEN                                  -- known CP or new CP with known CMP
        ⌘1:      dialogue about CPs in CMPDict entry for input CMP;
                update dictionaries (1);
    ELSIF input CMP is in CMPDict         -- known CMP not yet associated with V
    THEN                                  -- new or known CP
        ⌘2:      dialogue about CPs in CMPDict entry for input CMP;
                update dictionaries (2);
    ELSE                                  -- entirely new CMP
        find_close(input CMP, close CMPs);
        get the user to select closest CMP from among close CMPs;
        ⌘3:      dialogue about CPs in CMPDict entry for closest CMP;
                                           -- new or known CP
                update dictionaries (3);
    END IF;
ELSE                                      -- new verb
    IF input CMP is in CMPDict            -- known CMP
    THEN                                  -- new CP
        ⌘4:      dialogue about CPs in CMPDict entry for input CMP;
                update dictionaries (4);
    ELSE                                  -- entirely new CMP
        find_close(input CMP, close CMPs);
        get the user to select closest CMP from among close CMPs;
        ⌘5:      dialogue about CPs in CMPDict entry for closest CMP;
                                           -- new CP
                update dictionaries (5);
    END IF;
END IF.

```

Figure 4.3.3.2. The CA Algorithm

4.4 Expressing Situations $\mathcal{A}_1$ to $\mathcal{A}_5$ in Terms of Three Parameters: the Verb, the Case-marker Pattern (CMP), and the Case Pattern (CP)

4.4.1. Dealing with New CPs

The target CP (see 4.3.3.2) is first sought in one entry of the CMPDict. CPs and example sentences that illustrate their semantics are presented to the user until one is selected whose CP most closely matches that of the current input. If no stored CP is acceptable, HAIKU reverts to a different strategy. It works with the user to build the CP from individual CM-Case mappings. This is summarized in the following procedure.

`new_CP_dialogue:`

None of the proposed CPs has been accepted by the user. There are two avenues down which this dialogue can proceed.

FIRST, search the MDict entry for each lexical marker. The user is shown a list of Cases covered by this marker. When he selects one, a new CP is produced by adding the selection to the pattern for the rest of the input. This CP will be stored in the input CMP's entry. The current sentence will be an example sentence for this new pattern.

SECOND, use procedure `find_close` to identify the CMPs closest to the input CMP, and continue as if this new CMP were on input.

Finding an example sentence is easy if target CP is identical to a stored CP: HAIKU uses the sentence which was stored as a result of previous processing. Otherwise, the example sentence could be derived from that stored with the chosen CP. However, because derived sentences cannot be guaranteed to be well-formed, only actual input sentences are stored with new CPs at present.

4.4.2. Situation #1 ($\mathcal{A}_1$): known verb & known CMP & known or new CP

Fetch the CPs from the CMPDict entry for the input CMP²⁵. Present the CPs and their example sentences to the user. Usually, one is selected; otherwise call `new_CP_dialogue`

²⁵ The presentation in sections 4.4.2 to 4.4.6 uses a style which is more imperative (i.e. command-like) than elsewhere: this is suitable for the CA-algorithm description at this point in chapter 4.

(see 4.4.1): we have encountered a new CP. An example log illustrating how HAIKU deals with $\mathcal{A}_1$ is presented in Appendix E.1.

- **analyze Cases (1)**

Assume the input sentence is "User programs can print reports at 5 ppm" (where 'ppm' stands for "page per minute"). The input verb is print and the input CMP is psubj-pobj-at. A search of MDict for this verb finds an entry, and within that entry, a match for the input CMP. This success sends HAIKU to the CMPDict where it retrieves the CPs associated with psubj-pobj-at. There are two: AGT-OBJ-LAT and AGT-OBJ-TAT. While the AGT and OBJ associations are the same as in the retrieved CP, at 5 ppm expresses the MEAS Case (Measure), indicating a new CP.

- **update dictionaries (1)**

Because both the input verb and input CMP are already in HAIKU's databases, there is a possibility that the Case Pattern may also already be known. In that event the only updating that might occur is if the user chooses to replace the CP's current example with the input sentence.

MDict: any Case not associated with an input Case marker is added to its list. In our example, this means that MEAS is added to the list for at, producing at $\rightarrow$ [MEAS, LAT, TAT].

CMPDict: if the CP is new, add it and the input sentence to the list associated with the known CMP. For our example that means adding AGT-OBJ-MEAS $\leftrightarrow$ "User programs can print reports at 5 ppm" to the entry for psubj-pobj-at.

CPDict: look for an entry for the CP. If one exists, add the input verb to its list; otherwise add a new entry for the CP containing the input verb. Since our example sentence realizes a new CP, the entry (AGT-OBJ-MEAS, [print]) is added to CPDict.

4.4.3. Situation #2 ($\mathcal{A}_2$): known verb & new CMP & new or known CP

The CMP has been encountered with other verbs. A list of candidate target CPs is retrieved from the CMPDict. These are proposed to the user. If none is accepted, initiate

`new_CP_dialogue` (see 4.4.1). An example log illustrating how HAIKU deals with $\mathcal{A}_2$ is presented in Appendix E.2.

- **analyze Cases (2)**

Assume the input sentence is "Programs run quickly". The input verb is `run` and the input CMP is `psubj-adv`. A search of MDict finds that this pattern has not been associated with this verb before. However, HAIKU finds the following CMPDict entry for this pattern²⁶: [AGT-FREQ $\Rightarrow$ "We print hourly"]. This single item is shown to the user, who rejects the assignment of `FREQ` and instead associates `quickly` with `MANR`.

- **update dictionaries (2)**

MDict: add the new CMP to the input verb's list. If a new CP has been found, add new Cases to the lists of existing Case markers and associations for any new (Case marker $\rightarrow$ Case) mappings. For our example that means adding `psubj-adv` to `run`'s entry, producing [`psubj-at`, `psubj-adv`, `psubj-pobj-at`, `psubj-pobj-at-by`]. The example also adds a single association for the mapping `adv  $\rightarrow$  [MANR]`.

CMPDict: the CMP is known, so it already has an entry. If CA processing has produced a new CP, add it and the input sentence to the entry's list of CPs.

CPDict: look for an entry for the CP. If one exists, add the input verb; otherwise add a new entry containing the input verb. Our example involves the new CP `AGT-FREQ`, so the entry (`AGT-FREQ`, [`print`]) is added to CPDict.

4.4.4. Situation #3 ($\mathcal{A}_3$): known verb & entirely new CMP & new or known CP

The input CMP has never been encountered before. One or more neighbouring candidates must first be found using the closeness metric (see 4.6). When the best candidate has been found, HAIKU proceeds as in situation $\mathcal{A}_2$. An example log illustrating how HAIKU deals with $\mathcal{A}_3$ is presented in Appendix E.3.

²⁶ More precisely, HAIKU finds this CMP (in its CMPDict) or a permutation of it. Order does not matter with patterns: ultimately, it is only their semantic interpretation that counts. Thus, `subj-obj-at-by` is equivalent to `subj-obj-by-at`, and, similarly, `AGT-OBJ-LAT-TAT` is equivalent to `AGT-OBJ-TAT-LAT`.

- **analyze Cases (3)**

Assume the input sentence is "The typical session runs by dialup". The input verb is *run* and the input CMP is *psubj-by*. A search of MDict finds that this pattern has not been realized by this verb before; a search of CMPDict shows that no verb whatsoever has employed it previously. Lacking any exactly matching CPs to show the user, HAIKU must search for likely candidates in the CMPDict. It calls *find_close* to do this. The list of CMPs which this procedure returns is headed by *psubj-pobj-at-by*. The single CP in this entry is shown to the user, who accepts its mapping of *by* → *INST* as correct.

- **update dictionaries (3)**

MDict: add the new CMP to the input verb's list. If a new CP has been found, add new Cases to the lists of existing Case markers and associations for any new (Case marker → Case) mappings. Because the three CM-Case associations in our example already appear in *run*'s entry, no updating is necessary.

CMPDict: add a new entry to this dictionary, indexed on the new CMP. Use the input sentence as the example for its single CP.

CPDict: look for an entry for the CP. If one exists, add the input verb to its list; otherwise add a new entry containing the input verb. Our example sentence realizes the new CP *AGT-INST*, so the entry (*AGT-INST*, [*run*]) is added to CPDict for it.

4.4.5. Situation #4 (♣₄): new verb & known CMP & new CP

The input contains a new verb with a known CMP. HAIKU looks for candidate CPs in the CMPDict and suggests these to the user. Since we have encountered a new CP, the next step is to call *new_CP_dialogue* (see 4.4.1). An example log illustrating how HAIKU deals with ♣₄ is presented in Appendix E.4.

- **analyze Cases (4)**

Assume the input sentence is "Sessions process concurrently". The input verb is `process` and the input CMP is `psubj-adv`. Searching MDict establishes that HAIKU has never encountered this verb before; however, a query to CMPDict retrieves the input CMP. This provides a likely CP to show to the user: AGT-FREQ. The user edits the candidate to replace `FREQ` with `MANR`.

- **update dictionaries (4)**

MDict: add a new entry to this dictionary, using the input CMP and the approved CP as its contents. For "Sessions process concurrently" this means adding

```
process, [ psubj-adv ], [ adv → [MANR], psubj → [AGT] ].
```

CMPDict: an entry always exists for the input CMP in this situation. If the approved CP is new, add it to the entry and use the input sentence as its example. If the CP is already associated with the CMP, give the user a choice between the current example and the input sentence.

CPDict: look for an entry for the new CP. If one exists, add the input verb to its list; otherwise add a new entry containing the input verb. Our example AGT-MANR is a new CP, so the entry (AGT-MANR, [`process`]) is added to CPDict.

4.4.6. Situation #5 ($\mathcal{A}_5$): new verb & entirely new CMP & new CP

The input contains a new verb with an entirely new CMP. One or more neighbouring candidates must first be found using the closeness metric. When the closest CMP has been found, HAIKU suggests the associated CP. Since we have encountered a new CP, the next step is to call `new_CP_dialogue` (see 4.4.1). An example log illustrating how HAIKU deals with $\mathcal{A}_5$ is presented in Appendix E.5.

- **analyze Cases (5)**

Assume the new sentence is "Many users require backups for audits". The input verb is `require` and the input CMP is `psub-pobj-for`. Neither item is found in its dictionary, so HAIKU must search for the CMP closest to `psubj-pobj-for`. `find_close` returns the not-very-helpful `psubj-pobj`. Lacking any recommended

association for the preposition *for*, the user might well choose to initiate constructive definition, selecting *PURP* as the Case marked by *for*.

- **update dictionaries (5)**

MDict: add a new entry for the input verb; add the new **CMP** to its list. Add associations for all markers and their (single) Cases. For *require*, *psubj-pobj-for* is the **CMP** and the three associations are [*for* → [PURP], *pobj* → [OBJ], *psubj* → [AGT]].

CMPDict: add a new entry for the input **CMP**. Store the input sentence as the approved CP's example sentence. In our case, this means adding *psubj-pobj-for*, [AGT-OBJ-PURP ⇔ "Many users require backups for audits"]. If the CP is new, add the verb to the list of verbs which realize the CP. Select one example as in situation $\mathcal{A}_1$.

CPDict: look for an entry for the new CP. If one exists, add the input verb to it; otherwise add a new entry with the input verb. *AGT-OBJ-PURP* is a new CP, so an entry for *require* is added to this dictionary.

4.5 Difficult Issues: Finding Suitable Example Sentences and Suggesting Relevant Case Patterns

We here discuss two aspects that cause problems with the current CA process. This discussion complements the above presentation by focusing on some limiting facets of HAIKU, as it stands at the time of writing. Even though such considerations could be reported in the context of the “future work” part of this thesis (i.e. chapter 5), we deliberately include them here since this discussion will help the reader to get a better overall understanding of HAIKU.

4.5.1 Finding and Using Example Sentences

4.5.1.1 Avoiding Complete Rejection

The CA procedure requires user cooperation. At present, if an example sentence is inappropriate, the user has no way to partially exploit this example: it is rejected *en bloc*. But since the system learns nothing from a total rejection, we can envisage another strategy in which the user could identify the constituents at fault and, building on the elements of this example, more easily determine a suitable example sentence (as well as a valid CP for the current input, of course).

Consider the example sentence “They can execute the program at will”. Suppose *execute* is a new verb but the CMP `psubj-pobj-at` has already been encountered (situation 84). Suppose the CMPDict shows that `psubj-pobj-at` may correspond to AGT-OBJ-TAT or AGT-OBJ-LAT, and that both these patterns previously occurred with *run* (this would be recorded in the CPDict). The CMPDict provides two example sentences “The system prints all reports at noon” and “The system executes print jobs at this site”.

The system would present the examples to the user. Both would be rejected because neither expresses the Case expressed by *at* in “They can execute the program at will”. With no more stored examples to propose, HAIKU might try to generate new ones, but this would involve tackling the difficult problem of reliably coming up with meaningful sentences. Instead, a future extension of HAIKU could ask the user to identify incorrect elements in as simple a way as possible — for example, see 4.5.1.3.

The dialogue would let the user identify *at will* as the problem spot. The mappings *psubj*-AGT and *pobj*-OBJ would be considered satisfactory because they are equivalent to their counterparts in the example sentences. HAIKU would then suggest the alternative Cases *at will* can realize, among them the correct MANR. It is up to the user to recognize this, providing information which the system will learn. "They can execute the program at will" is then recorded as an example of the newly-learned mapping *psubj-pobj-at*/AGT-OBJ-MANR.

4.5.1.2. Problems with the Closeness Metric

Similarly, patterns obtained by using the closeness metric (see 4.6) in situations $\mathcal{A}_3$ and $\mathcal{A}_5$ pose additional problems for the production of apt example sentences. Suppose the example sentences for AGT-OBJ (pattern P1) and AGT-OBJ-LAT-MANR (pattern P2) do not satisfy the user, and HAIKU must base another on the example stored for AGT-OBJ-LAT, "Bob can run the software at home". To use this example for P1 we need to discard LAT and its corresponding marker, converting the sentence into "Bob can run the software". Conversely, to use it for P2 we must add MANR and a corresponding Case marker (e.g. "Bob can run the software at home *with ease*").

The question is whether adding or removing Case markers (and their accompanying Case fillers) in example sentences will produce reasonably well-formed and meaningful results. Removing Case realizations appears to be less troublesome than adding them: in fact, this is one of the basic assumptions underlying the definition of the closeness metric (in section 4.6). In any event the operations of the previous paragraph seem straightforward, given a record of the mappings between the elements in CMPs and CPs for each example sentence. It remains to be seen whether this ease of implementation will be matched by usefulness.

4.5.1.3. A Possible Scenario

HAIKU's dialogues are designed to extract a maximum of information with a minimum of effort. A single interaction could identify one or more incorrect elements in the semantic analogy between the input and the example sentences. Depending on the situation, the user could simply approve the analogy (option 0), or he could adjust the suggested semantic pattern (option 1), or he could prefer to look at other example

sentences, if there are others (option 2), or he could decide to build a new CP from scratch (option 3). Figure 4.5.1.3 shows a screen that illustrates the hypothetical interaction with the user. (For an interactive semantic disambiguation module, called an 'augmentor', that deals with multiple parses, see [Brown & Nirenburg 90].)

Does the sentence (current input)

"The man put the parcel in the car yesterday"

resemble (example sentence)

"Every day, dogs chase frisbees in the park" ?

where

a) the man	->	dogs	(agent)
b) the parcel	->	frisbees	(object)
c) in the car	->	in the park	(location at)
d) yesterday	->	every day	(frequency)

0) OK.

1) Adjust this pattern ...

2) 3 other patterns exist. Look at an example sentence for one of them.

3) Build a completely new pattern.

Enter a choice:

Figure 4.5.1.3. A Hypothetical Interaction with the User During Semantic Analysis

4.5.2. Challenging HAIKU: a Difficult Example

One of the original questions TANKA sought to investigate is the strength of association between individual verbs and CPs. Is LAT (Location_at) more likely to occur with run than with *think* ? This unresolved issue has a direct impact on the design of HAIKU. An example will illustrate:

Let *take* be the input verb and psubj-pobj-from-to the input CMP. Assume the dictionary CMP psubj-pobj-to is associated with *take* and has AGT-OBJ-BENF mapped to it. Assume further that the dictionary also contains psubj-pobj-from-to

associated with the verb *throw* and AGT-OBJ-LFRM-LTO. What association should be chosen for the preposition *from* in the input: the BENF in the close pattern found with the same verb, or the LFRM in the exactly matching pattern found with a different verb? This is the situation:

INPUT	take: psubj-pobj-from-to	
DICTIONARIES	take: psubj-pobj-to	AGT-OBJ-BENF
	throw: psubj-pobj-from-to	AGT-OBJ-LFRM-LTO

The current version of the CA algorithm would work as follows:

- a) *take* is a known verb — this is one of the situations $\mathcal{S}_1$, $\mathcal{S}_2$, or $\mathcal{S}_3$;
- b) it is not situation $\mathcal{S}_1$ because psubj-pobj-from-to has never occurred with *take* before;
- c) it is situation $\mathcal{S}_2$ because psubj-pobj-from-to is a known pattern not associated with *take*.

The algorithm would then suggest: target CP is AGT-OBJ-LFRM-LTO.

Would a hybrid pattern AGT-OBJ-BENF-LFRM be superior? Or should HAIKU begin with AGT-OBJ-BENF and negotiate with the user on the Case to be mapped to *from*? A hybrid is not guaranteed to be semantically consistent, but using LFRM-LTO rather than BENF would often misdirect the user. We conjecture that verb preference — that is, using patterns found with the input verb — should override almost all other considerations.

It can be argued that HAIKU should try to find a close pattern with the same verb before switching to a different one: this would result in target CP is AGT-OBJ-BENF. We believe HAIKU should use the closeness metric only after all possible perfect matchings with the same and then with different verbs have been considered. This minimizes use of the metric and the difficult task of generating good example sentences for hybrid CPs.

In summary, the CA algorithm currently gives highest priority to CMPs already encountered with the input verb, borrowing the CPs of closely matching CMPs associated with other verbs if necessary.

4.6 HAIKU's Closeness Computation: Semantic Similarity Based on Syntactic Markers

One mechanism essential to HAIKU is a procedure to produce syntactic Case-marker patterns close to the pattern in the input sentence. It is invoked whenever the input does not (perfectly) match any previously processed pattern. This happens in situations $\mathcal{A}3$ and $\mathcal{A}5$ in the CA algorithm (see 4.4).

In this section we propose a conceptually and technically neat method to identify known semantic patterns close to a novel pattern. This occurs when HAIKU requires the user to respond to specific questions about the current sentence. The questions are prepared from linguistic elements previously encountered in the text similar to elements in the new sentence. We present a metric to characterize the similarity between semantic Case patterns. The computation is based on syntactic indicators of semantic relations: it is defined in terms of a few simple heuristics and uses symbolic pattern matching. A shorter version of this section appeared in [Delisle *et al.* 93].

4.6.1 General Definitions of Matching Case Marker Patterns

The Case Analyzer begins by searching its KB (i.e. dictionaries) for the CP that exactly or most closely matches the CP of the input clause. It then looks for an example sentence to illustrate this matching CP. As we said before, only one CP should be associated with a given parsable input after user interaction and approval.

In certain situations the Case Analyzer must rank partially-matching CMPs in terms of their closeness to the input CMP. The algorithm to compute the closeness of two patterns is based on notions of undermatching and overmatching among CMPs. It is similar in spirit to some methods used in pattern recognition for computing distances between patterns ([Belaïd & Belaïd 92]). A CMP is a set, represented here as a Prolog list e.g. [psubj, pobj, at]. Intuitively, an undermatch occurs when a given CMP is completely or partially contained in another CMP. The computation of closeness is described in Prolog: the code is short and simple enough to be listed throughout the explanations without having the reader sent back and forth to an appendix for small pieces of code. Predicates not defined here come from the Quintus Prolog library ([QP-3.1 91]).

4.6.1.1 Definitions of Matching

Pattern P1 can match pattern P2 in three ways, listed here in the order of decreasing discrimination from the perspective of Case analysis: it can match perfectly (PM), as a proper subset (PUM), and *via* a non-empty intersection set (NPUM). Disjoint patterns are never considered as matching.

- **PM: perfect match**

Two patterns have identical membership: $P1 \equiv P2$.

EXAMPLES:

If $P1 = [\text{psubj}, \text{pobj}, \text{at}]$ and $P2 = [\text{psubj}, \text{pobj}, \text{at}]$, then P1 and P2 match perfectly.

If $P1 = [\text{psubj}, \text{pobj}]$ and $P2 = [\text{psubj}, \text{pobj}, \text{at}]$, then P1 and P2 do not match perfectly.

Prolog implementation:

```
% seteq: set equality.
pm(P1, P2) :- seteq(P1, P2).
```

- **PUM: proper undermatch**

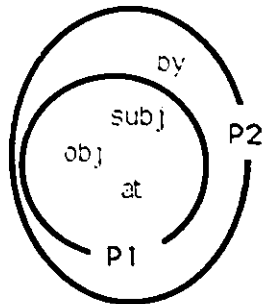
P2 is larger than P1. We say that P1 properly undermatches P2, or that P2 properly overmatches P1.

EXAMPLES:

If $P1 = [\text{psubj}, \text{pobj}, \text{at}]$ and $P2 = [\text{psubj}, \text{pobj}, \text{at}]$, then proper undermatch fails.

If $P1 = [\text{psubj}, \text{pobj}, \text{at}]$ and $P2 = [\text{psubj}, \text{pobj}, \text{at}, \text{by}]$, then proper undermatch succeeds. See figure below.

If $P1 = [\text{psubj}, \text{pobj}, \text{at}, \text{by}]$ and $P2 = [\text{psubj}, \text{pobj}, \text{by}, \text{from}]$, then proper undermatch fails.

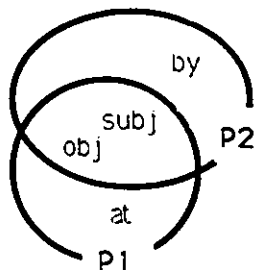


Prolog implementation:

```
% subset(P1, P2): P1 is a subset of P2.
pum(P1, P2) :- \+ seteq(P1, P2), subset(P1, P2).
```

- **NPUM: non-proper undermatch**

The patterns are different, but there is a non-empty intersection between them. Cardinality (technically, the length of a list) is used as a criterion to emphasize specificity: a higher cardinality indicates a more discriminating set. Notice that the npum relation is symmetric if the lengths of the patterns are the same.



EXAMPLES:

If $P1 = [psubj, pobj, at]$ and $P2 = [psubj, pobj, by]$, then $P1$ non-properly undermatches $P2$. See figure above.

If $P1 = [psubj, adv]$ and $P2 = [pobj, by, from]$, then non-proper undermatch fails.

If $P1 = [psubj, pobj, at, from]$ and $P2 = [pobj, at, adv]$, then non-proper undermatch fails since $P2$ is shorter than $P1$.

Prolog implementation:

```
npum(P1, P2) :-
    \+ subset(P1, P2), length(P1, LP1), length(P2, LP2), LP1 =< LP2,
    intersection(P1, P2, ITST), ITST \== [].
```

4.6.2 Basic Computations for the Closeness Metric

The following procedures refine those outlined in the previous section. Additional parameters return numerical values of closeness factors. The new versions supply zeroes instead of failing when a match is not found.

4.6.2.1 Computing the Closeness Factor within pum

The closeness factor CF measures how much P1 differs from P2. It allows us to compare any two patterns P1 and P2 quantitatively: *the higher the CF, the closer the patterns are to one another.*

Prolog implementation:

```
pum(P1, P2, CF) :-
    \+ seteq(P1, P2), subset(P1, P2), !,
    length(P1, LP1), length(P2, LP2), CF is LP1/LP2.
pum(_P1, _P2, 0).
```

EXAMPLES:

If P1 = [psubj, pobj] and P2 = [psubj, pobj, at], then CF = 2 / 3.

If P1 = [psubj, pobj] and P2 = [psubj, pobj, at, from], then CF = 2 / 4.

If P1 = [psubj, pobj, at] and P2 = [psubj, pobj, at, from], then CF = 3 / 4.

4.6.2.2 Computing the Comparison Coefficient within npum

In this definition, CF₁₂ and CF₂₁ are the closeness factors between P1 and P2 calculated from the perspective of P1 and P2 respectively. CC, the Comparison Coefficient between them, is computed separately.

Prolog implementation:

```
npum(P1, P2, CF12, CF21, CC) :-
    \+ subset(P1, P2), intersection(P1, P2, ITST), ITST \= [],
    length(P1, LP1), length(P2, LP2), LP1 =< LP2, !,
    length(ITST, N), CF12 is N/LP1, CF21 is N/LP2,
    compute_CC(ITST, CF12, CF21, CC).
npum(_P1, _P2, 0, 0, 0).
```

The greater CF_{ij} , the closer the intersection $ITST$ is to P_i . Since P_2 can be no smaller than P_1 ($LP_1 \leq LP_2$)²⁷, CF_{21} must always be equal to or smaller than CF_{12} , which can never equal 1. Also, the nature of CMs in the intersection is significant. More importance should be attached to $ITST = [pobj]$ than to $ITST = [adv]$: for the vast majority of English verbs, an object is semantically more informative than any adverb. The calculation of the Comparison Coefficient takes these facts into account.

The Comparison Coefficient represents the syntactico-semantic similarity between two situations expressed in two more or less similar clauses. CC is the product of CF_{12} , CF_{21} and V , a number that depends on the specific nature of the markers in the intersection: *the more significant these markers are, in terms of Case analysis, the higher V should be.* CF_{12} and CF_{21} are multiplied to obtain the highest ranking for patterns closest to the input pattern. Addition might be used instead of multiplication. Multiplying numbers from the interval $[0, 1]$ resembles finding the joint probability of independent events. It is a matter of further experiment to see whether using addition would drastically change the behavior of the `compute_CC` algorithm. CC is computed as follows.

Prolog implementation:

```
compute_CC(ITST, CF12, CF21, CC) :-
    get_ind_values(ITST, List_of_Values),
    sumlist(List_of_Values, Sum),
    CC is (CF12 * CF21 * Sum).

get_ind_values([], []) :- !.
get_ind_values([H | Hs], [V | Vs]) :-
    get_ind_value(H, V), get_ind_values(Hs, Vs).
```

The predicate `get_ind_value` supplies a numerical value for a Case marker reflecting its semantic specificity in Case-based semantic analysis. The pattern $[psubj, pobj]$ is less

²⁷ In fact, `npum` has two versions. One, `npum_u`, is used by `best_under_match_npum` and uses the test $LS1 \leq LS2$. The other, `npum_o`, is used by `best_over_match_npum` and uses $LS1 < LS2$.

specific than [psubj, pobj, at, from] because it occurs with more verbs and so is less discriminating. A perfectly specific CMP would occur with one and only one verb.

Our table of CM values reflects a belief that subjects and objects are less semantically significant than prepositions but more significant than adverbs. The values are subjective but can be tailored to the type of text at hand or modified according to assumptions one may wish to make and verify on certain texts. We will refer to this table as the *CM-ranking*.

Prolog implementation:

```
get_ind_value(Prep, 2.5) :- dict(Prep,prep,_,_), !.
get_ind_value(piobj, 2.0).
get_ind_value(pobj, 1.5).
get_ind_value(psubj, 1.0).
get_ind_value(adv, 0.5).
```

4.6.2.3 Simple Examples of Closeness Calculations

The four following examples assume that attempts to match by pm(P1, P2) and pum(P1, P2, _) fail, but that non-proper undermatching invoked by npum(P1, P2, CF12, CF21, CC) succeeds.

- a) Let $P1 = [\text{psubj}, \text{pobj}, \text{at}]$ and $P2 = [\text{psubj}, \text{pobj}, \text{from}]$, so that $ITST = [\text{psubj}, \text{pobj}]$; we have that $CF12 = 2/3$, $CF21 = 2/3$, and $CC = (1 + 1.5) * (2/3) * (2/3) = 1.11$
- b) Let $P1 = [\text{psubj}, \text{pobj}, \text{at}]$ and $P2 = [\text{psubj}, \text{pobj}, \text{from}, \text{adv}]$, so that $ITST = [\text{psubj}, \text{pobj}]$; we have that $CF12 = 2/3$, $CF21 = 2/4$, and $CC = (1 + 1.5) * (2/3) * (2/4) = 0.83$

According to the CM-ranking, P1 and P2 are closer to one another in a) than in b). Also [psubj, pobj, at] is intuitively closer to [psubj, pobj, from] than to [psubj, pobj, from, adv]: the added adverb in the second pattern diminishes the closeness of the two sets because it is not matched in the first set.

- c) Let $P1 = [\text{pobj}, \text{by}, \text{adv}]$ and $P2 = [\text{psubj}, \text{pobj}, \text{adv}]$, so that $ITST = [\text{pobj}, \text{adv}]$; we have that $CF12 = 2/3$, $CF21 = 2/3$, and $CC = (1.5 + 0.5) * (2/3) * (2/4) = 0.67$

- d) Let $P1 = [pobj, by, adv]$ and $P2 = [psubj, pobj, by]$, so that $ITST = [pobj, by]$; we have that $CF12 = 2 / 3$, $CF21 = 2 / 3$, and $CC = (1.5 + 2.5) * (2 / 3) * (2 / 3) = 1.78$

According to the CM-ranking, $P1$ and $P2$ are closer to one another in d) than in c). Two patterns which match on a particular preposition are more similar than those which match on an unspecified adverb.

4.6.2.4 Computational Details of the Closeness Metric

In this subsection, `In_CMP` refers to the pattern of the clause currently being processed by the system. `L_in` is the list of known CMPs in HAIKU's dictionaries against which `In_CMP` will be matched. The closeness metric is only calculated, based on some heuristics that we mention below, when a perfect match between the input CMP and known CMPs cannot be found.

- **Top-level procedure to compute closeness**

The procedure begins with a call to `find_closer_patterns`. This predicate takes `In_CMP` and `L_in` as inputs and returns four ordered lists of CMPs, `L_out_1`, `L_out_2`, `L_out_3`, `L_out_4`, that properly undermatch, non-properly undermatch, properly overmatch and non-properly overmatch `In_CMP`.

The order of these arguments in `find_closer_patterns` reflects HAIKU's preference in matches: proper undermatching, non-proper undermatching, proper overmatching, non-proper overmatching. Undermatching is preferable to overmatching. Undermatching provides the system with candidate patterns that express known meaning and subsume the input CMP. Overmatching introduces a greater uncertainty: candidate patterns here correspond to a potentially large class of underspecified situations that may represent semantically more distant candidate CMPs.

Suppose the input CMP is `[psubj, pobj, at]`, and the CMP dictionary contains `[psubj, pobj]` and `[psubj, pobj, at, from]`. The input CMP is a proper undermatch of `[psubj, pobj, at, from]` and a proper overmatch of `[psubj, pobj]`. The former is a much more discriminating candidate than `[psubj, pobj]`. If the user approves `[psubj, pobj, at, from]` the system can focus on a relatively small set of

Case Patterns as candidates for the current input. Giving priority to [psubj, pobj] would force the user to deal with a much larger set of CPs and render the knowledge acquired interactively less certain.

Proper matching is preferred to non-proper matching. A proper match is more likely to represent a valid situation, whereas a non-proper match may correspond to a situation which could be semantically less fitting. Each of the four output lists is ordered by decreasing specificity: longest matches are preferred to shortest ones because they correspond to closer neighbours of the input CMP in the space of known CMPs. The closer the neighbour, the more likely it is that it will lead to a valid candidate semantic pattern.

Prolog implementation:

```
% Given a list L_in of already known CMPs, this predicate finds the
% lists (in decreasing order of 'closeness') of patterns in L_in that
% are close to In_CMP, the input CMP.
```

```
find_closer_patterns(In_CMP, L_in, L1, L2, L3, L4) :-
    best_Under_Match(In_CMP, L_in, L1, L2),
    best_Over_Match(In_CMP, L_in, L3, L4).
```

- **A procedure to find best undermatches**

We now present the computational details of finding the “best neighbours” for a given input CMP (In_CMP). These candidate patterns are found in terms of best proper undermatches and best non-proper undermatches with the input CMP. The procedures are defined using ‘pum’ and ‘npum’ which were described earlier.

Prolog implementation:

```
% L_in : list of patterns against which In_CMP is tentatively
% matched.
% L_out: sorted list of pum/npum patterns, in decreasing order.
% Elements of L_out have the form 'match(Closeness, Pattern)'.

best_Under_Match(In_CMP, L_in, L_out_pum, L_out_npum) :-
    best_under_match_pum(In_CMP, L_in, [], L_out_pum),
    best_under_match_npum(In_CMP, L_in, [], L_out_npum).

% PROPER UNDERMATCHING
best_under_match_pum(_S, [], L_in, L_out) :-
    sort(L_in, L_out_temp), reverse(L_out_temp, L_out).
best_under_match_pum(S, [S_H | T], L_in, L_out) :-
    pum(S, S_H, CF_H), CF_H \== 0, !,
    best_under_match_pum(S, T, [match(CF_H, S_H) | L_in], L_out).
best_under_match_pum(S, [_S_H | T], L_in, L_out) :-
    best_under_match_pum(S, T, L_in, L_out).

% NON-PROPER UNDERMATCHING
best_under_match_npum(_S, [], L_in, L_out) :-
    sort(L_in, L_out_temp), reverse(L_out_temp, L_out).
best_under_match_npum(S, [S_H | T], L_in, L_out) :-
    npum(S, S_H, _, _, CC_H), CC_H \== 0, !,
    best_under_match_npum(S, T, [match(CC_H, S_H) | L_in], L_out).
best_under_match_npum(S, [_S_H | T], L_in, L_out) :-
    best_under_match_npum(S, T, L_in, L_out).
```

- **A procedure to find best over-matches**

The definitions of `best_over_match_pum` and `best_over_match_npum` are analogous to those of `best_under_match_pum` and `best_under_match_npum` with the first two parameters (`S` and `S_H`) inverted. Note that `npum` has been defined with the condition " $LP1 \leq LP2$ ". Substituting " $LP1 < LP2$ " in the 'best_over_match' versions eliminates patterns that would appear in both overmatching and undermatching sets.

Prolog implementation:

```
best_Over_Match(In_CMP, L_in, L_out_pum, L_out_npum) :-
    best_over_match_pum(In_CMP, L_in, [], L_out_pum),
    best_over_match_npum(In_CMP, L_in, [], L_out_npum).
```

4.6.3 Examples of Closeness Calculations

We have already discussed the application of the closeness metric within HAIKU in sections 4.4.4 (§3) and 4.4.6 (§5). We here present two examples that illustrate the

computation of the closeness metric, based on two different sentences. Suppose Case analysis has found a new CMP with a known verb. Assume perfect match and proper undermatch each fail in both examples.

- **Example #1: “A system runs a datalink from Toronto before midnight.”**

CMP: psubj-pobj-before-from

The CMP is represented by $P1=[psubj, pobj, before, from]$.

`find_closer_patterns(P1, Set_of_all_CMPs, L1, L2, L3, L4)` finds:

$L1 = [[psubj, pobj, before, from, adv]]$ (properly overmatching CMPs);
 $L2 = [[psubj, pobj, at, before], [psubj, pobj, at, by]]$ (non-properly overmatching CMPs);
 $L3 = []$ (properly undermatching CMPs);
 $L4 = []$ (non-properly undermatching CMPs);

The list of candidate CMPs sorted in descending order of closeness is: $[psubj-pobj-before-from-adv, psubj-pobj-at-before, psubj-pobj-at-by]$. $psubj-pobj-before-from-adv$ would be the first CMP HAIKU suggests to the user. If it is rejected, $psubj-pobj-at-before$ would then be suggested, and so on.

Examples of answers for $L3$ and $L4$ would be:

$L3 = [[psubj, pobj, before], [psubj, pobj, from], ...]$
 $L4 = [[psubj, pobj, at], [psubj, pobj, in], [psubj, pobj, adv], ...]$

- **Example #2: “The system runs a datalink by 7 midnight.”**

CMP: psubj-pobj-by

The CMP is represented by $P1=[psubj, pobj, by]$

`find_closer_patterns(P1, Set_of_all_CMPs, L1, L2, L3, L4)` finds:

$L1 = [[psubj, pobj, at, by]]$ (properly overmatching CMPs);

```
L2 = [ [psubj, pobj, by, from], [psubj, pobj, from], [psubj, pobj,
      adv] ] (non-properly overmatching CMPs);
L3 = [ [psubj, pobj] ] (properly undermatching CMPs);
L4 = [ ] (non-properly overmatching CMPs);
```

The list of candidate CMPs sorted in descending order of closeness is: [psubj-pobj-at-by, psubj-pobj-by-from, psubj-pobj-from, psubj-pobj-adv, psubj-pobj]. psubj-pobj-at-by would be suggested first.

4.6.4 Further Refinements of 'pum' and 'compute_cc'

Extensive testing of the metric on all kinds of hypothetical CMPs revealed certain problems with the interpretation of the metric's output. This output is organized into four separate (ranked) lists of candidate patterns (i.e. properly overmatching CMPs, non-properly overmatching CMPs, properly undermatching CMPs, and non-properly overmatching CMPs) which are individually sorted.

From the user's point of view, it is rather tempting to compare any pairs of candidate patterns, and their associated comparison coefficients (CCs), without paying too much attention to the fact that the four lists have different "weights". In doing so, some of the CCs involved in such a comparison may seem counter-intuitive, at least for certain border cases.

For example, for $P1 = [psubj, pobj, to]$, $P2 = [psubj, pobj, on]$, and $P3 = [psubj, pobj, from, to]$, we have that $P3$ is a proper overmatching CMP of $P1$, with a CC of 0.75, and that $P3$ is a non-proper overmatching CMP of $P2$, with a CC of 0.83. If we just look at the CCs, we may be tempted to choose $P2$ as the closest pattern to $P3$ since $0.83 > 0.75$. But this is not a good choice. $P1$, with a CC of 0.75, is a better alternative. $P1$ is a proper overmatch, which is preferable to a non-proper-overmatch (i.e. $P2$).

In order to reduce potential confusion in the interpretation of the metric's results by allowing the comparison of any pair of CCs, *regardless of the specific sublist they come from*, we now redefine 'pum' and 'compute_cc' as follows. First, 'pum': it is now redefined similarly to 'npum', except that $P1$ must be a subset of $P2$. Notice that this version has a new first parameter, 'X', which is used for distinguishing the comparison

test for undermatching ($X='u'$) from the one for overmatching ($X='o'$). It is assumed that all previous calls to this predicate would be updated correspondingly. This applies as well to 'compute_cc' below.

Prolog implementation:

```
pum(X,P1,P2,CF) :-
    \+ seteq(P1,P2), subset(P1,P2),
    intersection(P1,P2,ITST), ITST \== [],
    length(P1,LP1), length(P2,LP2),
    ((X == u, LP1 <= LP2) ; (X == o, LP1 < LP2)), !,
    length(ITST,N), CF12 is N/LP1, CF21 is N/LP2,
    compute_cc(X,ITST,CF12,CF21,CF).
pum(_,_P1,_P2,0).
```

Second, 'compute_cc': it is now redefined with an explicit weight for distinguishing undermatchings from overmatchings. This weight was defined empirically as $2/3$, i.e. an overmatching CMP is worth two thirds of a comparable (equivalent) undermatching CMP. This heuristic now ensures the desired readability in the metric's results: it is these final versions of 'pum' and 'compute_cc' which are implemented in the current version of HAIKU.

Prolog implementation:

```
compute_cc(X,ITST,CF12,CF21,CC) :-
    get_ind_values(ITST,List_of_Values),
    sumlist(List_of_Values,Total),
    CC_0 is (CF12 * CF21 * Total),
    ((X == u, CC = CC_0) ; (X == o, CC is CC_0 * (2/3))).
```

4.6.5 Strategic and Algorithmic Aspects of the Closeness Computation

The distinction between undermatching and overmatching is not important to the user when he looks at candidates. `find_closer_patterns` will return a correctly ordered list of potential candidates from both populations. Wasting time on unlikely candidate patterns could be avoided if CA uses a threshold for values of CF (for pum) and of CC (for npum), below which candidates are discarded. Alternatively, an upper limit could be set on the number of candidates to be considered: then if `find_closer_patterns` returns more than one candidate, HAIKU might never go past the second or third.

The dictionary of Case-marker Patterns is represented in Figure 4.6.5 (next page) as a graph into which the heavily outlined input pattern has been inserted in the appropriate location. To simplify things non-proper overmatches, the least preferred type, are not shown. The closeness algorithm can be summarized as follows, assuming a given CM-ranking:

- First find any undermatching patterns. Sort the list of results to have at its head matches with the most matching CMs and fewest unmatched CMs. Sort matches of the same length and number of matching CMs to favour the most heavily-weighted CMs.
- Next find any overmatching patterns. Sort the list of results to have at the head matches with the fewest additional CMs. Sort matches of the same length to bring forward those with the most heavily-weighted CMs.

`find_closer_patterns` currently returns the CMP closest to the input CMP according to the following priorities:

```

properly undermatching CMPs
>
non-properly undermatching CMPs
>
properly overmatching CMPs
>
non-properly overmatching CMPs.
```

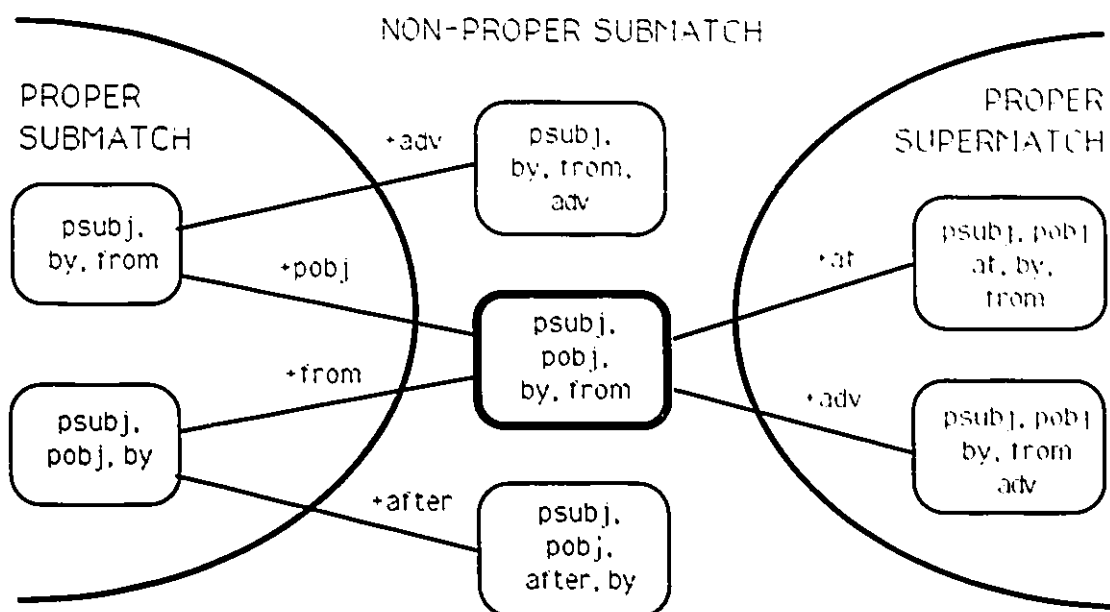


Figure 4.6.5. Relationships between Case-marker Patterns (CMPs)

In summary, we have translated the difficult problem of finding the best approximation of a given semantic patterns into the computationally feasible task of finding the best match for a pattern of symbols which denote basic semantic relations. Linguistically, we propose to define semantic similarity *via* syntactic congruence and compute it from syntactic markers (i.e. the Case-marker Patterns). The metric we propose allows our system to suggest to the user valid candidate semantic patterns for the input at hand, based on the system's knowledge accumulated so far. The metric can be used in any setting in which such a substitution of syntactic for semantic phenomena can be justified.

4.7 Testing the Semantic Analyzer with Actual Texts: Results and Discussion

Ultimate testing of HAIKU is not possible at the time of writing: it would require the MNB to be fully operational²⁸; the same is true of TANKA for which we require the Conceptual Knowledge Processor. As a consequence, we will not be in a position to show experimental support for some of the broad claims of chapter 1. However, we here discuss the most significant trends.

Extensive stand-alone testing of HAIKU was performed. First on the set of 600 test inputs described in 3.6.1 in order to ensure that HAIKU's Case Analyzer was able to handle all different types of DIPETT output (with the user's help whenever appropriate). Second, on numerous segments of the QUIZ manual and the T4043 Tax Guide (see sections 3.6.2 and 3.6.3, respectively) to fine-tune the CA process and to verify how little help was necessary for HAIKU to function as a useful KA tool — we will come back to the latter point in chapter 6. Finally, a separate lab experiment was conducted on the QUIZ manual: this is what we discuss below. At the time of writing, we are organizing a large-scale testing of (DIPETT and) HAIKU on two new texts, namely the MUC-4 text and the 1991 Ontario Building Code.

4.7.1 An Experiment with HAIKU

In this lab experiment, we used HAIKU on the first two chapters of the QUIZ manual. In the first phase of the experiment, we parsed the chapters with DIPETT (in batch mode with a CPU time limit of 60 seconds) and saved the parse trees in two files, one for each chapter. Out of the 272 input strings in both chapters, approximately 92% of them resulted in a parse tree, producing either full or fragmentary analysis. In the second phase of the experiment, we ran HAIKU on both files of parse trees starting with an empty set of dictionaries²⁹, i.e. the MDict, the CMPDict, and the CPDict were all empty when initiating the CA process — all HAIKU had was its list of (domain-independent) prepositions and the Cases they realize ([Barker *et al.* 93]). We now list the results.

empty (HAIKU) dictionaries

²⁸ Most probably, this will not be possible before a year: something over which the author has absolutely no control. But see Appendix I for a brief description of the current design for the MNB.

²⁹ We remind the reader that this is one of HAIKU's strengths: it need not to be fed with a minimal KB to be usable.

4.7.2 General Observations

- The price to pay for starting with an empty KB was a temporarily more demanding and tedious user interaction: since HAIKU had not accumulated much knowledge yet, it could not make useful suggestions to the user. The system relied on the user to accumulate enough knowledge before becoming more productive.
- But this “minimal learning phase” faded out after HAIKU had processed some 50 input strings. At that point, it started more and more often to make useful suggestions to the user, thus making the interaction easier and significantly more fruitful³⁰. Thus, there seems to be a learning curve which represents the minimal knowledge the system must acquire in order to “know” about the text’s salient conceptual patterns. Hypothetically, this curve could also be looked at as the adaptation effort required to port from one domain to another. From the user’s point of view, the accumulation of this “minimal knowledge” translates into his participation becoming less of the knowledge-intensive type and more of the approval or selection type.
- The speed of semantic analysis was quite reasonable: it took the author half a day (i.e. approximately 4 hours) to analyze chapters 1 and 2 with HAIKU — a less experienced user would probably have taken a full day.
- The user need not be very familiar with the details of the topic discussed in the text being analyzed: a basic understanding of English is what counts most. In fact, we found that a general understanding of the text’s topic helped, but much less than first anticipated. The user also plays a role in making sure that the accumulated knowledge is reliable: a difficult problem for fully automated KA systems.
- Even if the parse tree on input is not perfect or if HAIKU’s initial analysis is erroneous, particularly in determining the input’s CMP, the user can always enter or select the correct patterns (CMP and CP) and ensure that HAIKU at least accumulates accurate elements of knowledge in its dictionaries. However, this strategy would not be applicable for subsequent steps in semantic analysis, in the MNB, where the

³⁰ In relation with an earlier footnote of section 4.3 about HAIKU’s equivalent to Sergot’s query-the-user facility, this observation is in perfect harmony with [Sergot 83: p.36]: “In operation the query-the-user facility has a very attractive property. At first, it knows little and spends a lot of time asking the user for information. As it assimilates this information however it needs to ask less and less.”.

input's elements must be perfectly analyzed for its integration into the Conceptual Network.

4.7.3 Results for Chapters 1 and 2 of the QUIZ Manual

We list below an excerpt from the contents of HAIKU's KB after processing chapters 1 and 2, starting with empty dictionaries. After chapter 1, HAIKU's KB contained 86 entries in its MDict, 38 entries in its CMPDict, and 38 entries in its CPDict. At the end of chapter 2, the number of entries were, respectively, 119 (+38%), 48 (+26%), and 50 (+32%). A brief discussion will follow in section 4.7.4.

See Appendix H ("KB Accumulated by HAIKU in Chapters 1 and 2") for the full listing of HAIKU's dictionaries at the end of the processing of chapters 1 and 2. The excerpt below presents only the first ten entries for each of the three dictionaries.

MDict

```
mDict(access,[pobj,'pobj-adv','psubj-pobj','psubj-pobj-adv','psubj-
to'],[[adv,[manr]], [pobj,[obj]], [psubj,[agt]], [to,[obj]]]).
mDict(add,[pobj,'pobj-to'],[[pobj,[obj]], [to,[lto]]]).
mDict(affect,['psubj-pobj'],[[pobj,[obj]], [psubj,[agt]]]).
mDict(agree,['on-within'],[[on,[cont]], [within,[lat]]]).
mDict(answer,['psubj-pobj'],[[pobj,[obj]], [psubj,[agt]]]).
mDict(apply,['pobj-adv'],[[adv,[manr]], [pobj,[obj]]]).
mDict(ask,[pobj,'pobj-piobj','psubj-for-like','psubj-
pobj'],[[for,[obj]], [like,[manr]], [piobj,[obj]], [pobj,[obj,recp]],
[psubj,[agt]]]).
mDict(assume,[pobj],[[pobj,[obj]]]).
mDict(avoid,[pobj],[[pobj,[obj]]]).
mDict(base,['pobj-on-for'],[[for,[purp]], [on,[cont]], [pobj,[obj]]]).
etc.
```

CMPDict (example sentences have been replaced with "... " here)

```
cmpDict('down-adv',[['obj-tat', ... ]]).
cmpDict('of-from',[['obj-lfrm', ... ]]).
cmpDict('on-within',[['cont-lat', ... ]]).
cmpDict('pobj-adv',[['obj-caus', ... ], ['obj-manr', ... ]]).
```

```

cmpDict('pobj-as',[['obj-manr', ... ]]).
cmpDict('pobj-as-for-in-adv',[['obj-manr-purp-lto-freq', ... ]]).
cmpDict('pobj-between',[['obj-lat', ... ]]).
cmpDict('pobj-for-adv',[['obj-lto-manr', ... ]]).
cmpDict('pobj-from',[['obj-lfrm', ... ]]).
cmpDict('pobj-in',[['obj-lat', ... ],['obj-lto', ... ]]).
etc.

```

CPDict

```

cpDict('acmp-cont',[check]).
cpDict('agt-caus-obj',[want]).
cpDict('agt-cont',[do]).
cpDict('agt-cont-acmp',[lie]).
cpDict('agt-inst',[read]).
cpDict('agt-lat',[look]).
cpDict('agt-lto',[print]).
cpDict('agt-ltru',[fit]).
cpDict('agt-manr',[respond,sort,use]).
cpDict('agt-manr-freq',[respond]).
etc.

```

4.7.4 Discussion

Some elements of the results deserve further comments. First, on the MDict. The verbs with the largest number of different CMPs are: 'access' (5), 'ask' (4), 'enter' (7), 'link' (6), 'report' (7), 'tell' (5), 'use' (6), 'want' (7), and 'work' (6). Interestingly, these seem to belong to two different but complementary groups. The group of central domain-specific verbs (activities): 'access', 'link', and 'report'; plus, the group of general purpose explanatory verbs: 'ask', 'enter', 'tell', 'use', 'want', and 'work'. As for verbs that occurred with 3 different CMPs or less, 66 (55%) had only one, 27 had two (23%), and 17 (14%) had three CMPs, in this text.

Second, on the CMPDict. The vast majority (94%) of CMPs are of length 3 or smaller: 9 (19%) of length 1, 22 (46%) of length 2, and 14 (29%) of length 3. Two are of length 4 and one of length 5. Also of interest is the number of CPs associated with a given CMP. For 36 of the 48 entries (75%), a single CP is associated with the CMP; 10 entries

(21%) have 1, zero entries have 3 or 4, one entry has 5 and another has 7 different CPs for a unique CMP. For these two (5 and 7 CPs), the CMPs involved are, respectively, `psubj-pobj-piobj` and `psubj-pobj-adv`. This last result seems to confirm our common-sense expectation that these syntactic patterns (i.e. a verb with a subject, a direct object, and an indirect object or an adverbial complement or modifier) cover such a large spectrum of forms that they can hardly be very discriminating as to what semantic pattern is associated with them.

non-discriminating syntactic patterns

Third, on the CPDict. The CPs that were associated with the largest number of verbs are (not surprisingly), `agt-obj` and `agt-obj-manr`, as well as `obj` and `obj-manr`. The latter two illustrate how the imperative form is important in the type of text exemplified by the QUIZ manual. 31 entries (62%), i.e. CPs, are associated with a single verb; 7 (14%) with two verbs, and 5 (10%) with three verbs.

In a nutshell, these results show some support for our claim that Case-based semantic analysis is a valuable tool for knowledge extraction from texts. It seems discriminating enough for most verb (activities) except maybe for input strings involving the ubiquitous CMP and CP patterns `psubj-pobj(-adv)` and `agt-obj(-manr)`. These would greatly benefit from the addition of semantic markers such as WordNet categories (or “hypernyms”) for the nominal groups: as a result, user interaction would be significantly enhanced. See section 5.2 for a few examples illustrating how such semantic markers could be used within HAIKU; see also the end of 5.2 for further details on WordNet’s hypernyms.

We conclude by noting that these results raise a number of interesting questions³¹ that open the door to future research avenues or potential new applications. For instance, if we consider HAIKU’s KB after the processing of a specific text (and assuming this KB was empty when we started out), could we not say that this KB represents a characterization (or “signature”) of the input text? Would texts of the same author tend to have similar Case structures? Would technical texts tend to show similarities in terms of their Case structures, regardless of who is the author? Could this information be useful in the much needed task of seeking a linguistic characterization of technical texts?

³¹ These questions were suggested by Dr. Stan Matwin, a member of the author’s thesis committee.

CHAPTER 5

5. FUTURE WORK ON DIPETT AND HAIKU	182
5.1 A Parse Tree Editor with Learning Capabilities	183
5.1.1 The Editor.....	183
5.1.2 The Editor as a System that Learns.....	185
5.2 Refining HAIKU's CA: a Concept Ontology for Noun Semantics.....	187
5.3 Extending DIPETT's Coverage of English.....	189
5.4 Transforming DIPETT into a Flexible Expectation-Driven Parser	193
5.5 Text Summarization as Parametrizable Text Highlighting.....	196
5.5.1. Introduction.....	196
5.5.2. A Proposal Based on the Notion of Conceptual Distance.....	196

5. FUTURE WORK ON DIPETT AND HAIKU

In this chapter, we briefly describe planned extensions to the work presented in this thesis. The main objective is to develop even further the methodology we proposed in the previous chapters, as well as its computational structure and implementation, so that the resulting system can behave even more “intelligently”. The order of the following subsections is based on the importance of the extension it contains, from more to less urgent.

5.1 A Parse Tree Editor with Learning Capabilities

5.1.1 The Editor

As was mentioned in chapter 3, DIPETT's "best first" parse trees are not always correct. Two typical types of misparses are misattachments (e.g. a prepositional phrase attached to a verb instead of a noun) and erroneous analysis caused by lexical ambiguity (e.g. a word parsed as a verb instead of as a noun — for an example, see 5.3). However, since such misparses can seriously affect semantic analysis, it is part of our short term plans to add to DIPETT-HAIKU another module with an interactive parse tree 'editor'¹. This new module will be responsible for ensuring that before continuing with semantic analysis in HAIKU, the parse tree produced by DIPETT is approved by the user. If not, the user will be allowed to rearrange the tree².

For example, the parse tree #2 in Appendix D contains a misattached prepositional phrase (PP). In "*This would produce the names of the two Boston employees who joined the company since the start of 1984*", the PP with 'since' should have been attached to the verb 'joined' and not as a post-modifier of "the names". The parse tree editor will first show DIPETT's output, but under a much simpler form (see figure 5.1.1 below), and then ask the user if he wishes to make any changes to it. If not, this means that, according to the user's judgment, the parse is correct. Otherwise, the user may make one or more adjustments: this is exactly what would happen with the example sentence above.

The main difficulty in designing such a parse tree editor is the user interface which must be very "friendly". More precisely, since we do not expect the user to be a linguist, we cannot simply assume that the user will be able to easily read and manipulate DIPETT's parse trees. Thus we are looking for an equivalent representation, but one which is much simpler and more intuitive. It must not be too fine-grained, just enough to allow the user to "repair" typical errors. Figure 5.1.1 presents a hypothetical interaction in which the editor shows our example syntactic analysis #2 in an easy-to-read format; we

¹ By 'editing' we mean using a set of simple operations that modify an existing structure. Such a set would definitely not include typing in a complete new structure.

² An alternative strategy would be to allow the parser to backtrack and suggest different analyses (i.e. parses) to the user. However, we see two major difficulties with this strategy: first, the time to produce several different analyses might become unacceptably long for the user, and, second, the user would still be left with the burden of decoding and distinguishing several non-trivial parse trees in order to select the best one.

means "X is modified by Y".)

INPUT: "This would produce the names of the two Boston employees who joined the company since the start of 1984."

SIMPLIFIED PARSE TREE PRODUCED BY DIPETT:

- a) This would produce the names
- b) ↳ of the two Boston employees
- c) ↳ who joined the company
- d) ↳ since the start
- e) ↳ of 1984

IS THE ABOVE PARSE TREE CORRECT [n/Y]? n

PLEASE ENTER THE MODIFICATION COMMANDS, ONE PER LINE ('ok' to exit):

- 1) attach 'd' to 'joined' in 'c'
- 2) ok

HERE IS THE RESULTING UPDATED PARSE TREE:

- [illegible]

IS THE ABOVE PARSE TREE CORRECT [n/Y]? y

Figure 5.1.1. A Hypothetical Interaction with the User During Parse Tree Revision

This is based on the hypothesis that the analyses which must be repaired are, on average, not too distant from the correct analysis. Or, in other words, that the number of errors is small and that these errors are easy to identify and fix. If this hypothesis does hold, which seems to be the case at least for misattachments, then we claim that such an approach to the production of correct parse trees is more productive than other approaches such as [Brown & Nirenburg 90] in which the parser produces multiple parse trees (for ambiguous input strings) that must be dealt with subsequently in semantic

analysis — see also [Blanchon 92] where the proposed system performs disambiguation by asking the user to select the appropriate interpretation amongst a list of possibilities.

There are however a few kinds of parsing errors that might justify a “second-chance” parse. A misparse caused by lexical ambiguity illustrates such a situation. If a word has been parsed as a verb instead of as a noun (again, see 5.3 for an example), it might be the case that the parse tree is so much different from what it ought to be that repairing it with the editor simply does not make sense. A better solution for this situation would be to update the lexicon with the appropriate entries for this particular input and then reparse it entirely. In order to do this, the editor would have a short list of parsing errors for which such an alternate strategy should be proposed to the user. This option might even be offered for any situation since the user may judge the repairs too difficult and thus prefer to reparse after having updated the lexicon or submitted a slightly modified (but semantically equivalent) input string.

The editor should also be prepared to handle fragments. When DIPETT does not manage to find a full analysis under the prescribed CPU time limit, our experience has revealed that, often, the resulting fragmentary analysis is an almost correct parse tree segmented into a few substructures. For such fragments, it appears that the above scheme could be used as a recuperation process which would allow us to transform fragmentary analyses into full analyses by simply connecting the main components *via* the parse tree editor. This requires further work.

Finally, for convenience of use, the editor should be usable during on-line parsing as well as during a preliminary phase of semantic analysis of parse trees already stored in a file, similarly to option 9 in DIPETT’s main menu (see Appendix A).

5.1.2 The Editor as a System that Learns

In the previous subsection, an example showed that the parse tree editor was informed by the user that a certain prepositional phrase (PP) had been misattached. Such information could be accumulated by the editor in a separate KB so that it could be used both by the editor and eventually by the parser in subsequent processing. Similarly to what happens in HAIKU, this would allow the editor module to improve its performance as it accumulates past-processing experience.

First, the editor. As it collects more and more information on parse tree repairs made by the user, it becomes more and more autonomous in recognizing the most frequent (potential) errors when it is given a new input. Based on its KB, the editor could then suggest a small number of potential remedies to the user, thus minimizing the editing task. For instance, suppose that PPs with 'since' are often attached to a preceding noun instead of a preceding verb. The editor could warn the user about this fact after having checked its KB (for an input containing such a PP with 'since').

Second, the parser. If the editor can acquire knowledge on parse tree repairs, it will be possible to generalize its KB in order to propose candidate modifications to the parser's grammar — see [Brill 93] and [Kiyono & Tsujii 93] for related work. (Obviously, this aspect of the editor would be intended for the more specialized user or the person responsible for maintaining the parser's grammar.) In particular, these modifications could take the form of attachment heuristics that would tell DIPETT what are the most likely attachments during the parsing of certain structurally ambiguous input strings. To continue with our PP with 'since' example, generalizing over a certain number of misattachments involving a PP with 'since', the editor would propose a 'since' heuristic rule of the form "when a PP with 'since' follows a complement noun phrase, then attach it to the (preceding) main verb". This facility could thus assist the task of improving the parser's reliability.

5.2 Refining HAIKU's CA: a Concept Ontology for Noun Semantics

As we discussed at the end of chapter 4, probably the most apparent limitation of HAIKU's current representation of syntactic and semantic patterns (i.e. CMPs and CPs) is its lack of semantic categories (or conceptual categories) attached to nouns. Such categories would make HAIKU's stored patterns and suggestions even more precise: HAIKU would simply discard candidate patterns that are associated with nominal fillers which have incompatible semantic categories.

For example, suppose we know that a verb's subject ('psubj') must be of the CONSCIOUS-BEING category and its object ('pobj') must be of the NONDECOMPOSABLE-OBJECT category³. Then, HAIKU's CMP entry for that verb would contain psubj/CONSCIOUS-BEING and pobj/NONDECOMPOSABLE-OBJECT instead of simply psubj and pobj. Assuming that all lexical entries for nouns would be augmented with their appropriate semantic category (or categories), semantic analysis could first ensure that noun fillers of the appropriate semantic category are associated with each Case marker. This extra information would also be added to HAIKU's CPs: we could then have AGT/CONSCIOUS-BEING and OBJ/NONDECOMPOSABLE-OBJECT instead of simply AGT and OBJ. In that way, whenever HAIKU would attempt to suggest candidate CMPs or CPs for a given input, it would systematically filter its potential suggestions through the test of semantic categories.

Let us have another example using semantic categories of the computer science world (CSW)⁴. With the current version of HAIKU, the following two sentences are both associated with the same CMP, i.e. psubj-pobj-adv, and the same CP, i.e. AGT-OBJ-TAT: "*My brother bought a new book yesterday*" and "*My program compiled a new database yesterday*". However, if we augment the current representation with the CSW semantic categories, the first sentence would be associated with the CMP psubj/PERSON-pobj/DOCUMENT-adv and the second with psubj/PROGRAM-pobj/DATA-adv. Similarly for CPs, we would have AGT/PERSON-OBJ/DOCUMENT-TAT and AGT/PROGRAM-OBJ/DATA-TAT. We clearly see that the two sentences which have identical CMPs and CPs within the current framework are no longer

³ We borrowed these categories from [Bateman *et al.* 90].

⁴ These categories were borrowed from the *is-a* hierarchy of [Nirenburg & Raskin 87].

considered similar when CSW semantic categories are added: HAIKU would not propose one as an example of the other during the Case analysis process.

At this point, the reader might be tempted to ask why semantic categories were not added to the design of HAIKU if they seem to have such a positive impact on its processing. There are two good reasons for this. First, it is difficult to find an “off-the-shelf” conceptual ontology that is both relatively standard and not too domain-specific⁵: at present, this type of knowledge is anything but standard⁶. Second, as a consequence of the first point, we believe that using such ontological knowledge would go partly against the fundamental claim of this thesis which argues for TP for KA without *a priori* domain knowledge. Further development and standardization of current ontologies will certainly help to clarify this situation in the future. One possibility that deserves more investigation is the use of WordNet’s hypernyms⁷, which could be considered as semantic categories. For example, [Feng *et al.* 94] discusses how WordNet can be used for disambiguation and for semantic clustering, starting from elements of parse trees produced by DIPETT.

Finally, we note that this type of knowledge, if ever made available to HAIKU, might also influence DIPETT as an expectation-driven processor: see section 5.4 below.

⁵ Ontologies that are fined-grained tend to be too domain-specific; others that are too general, tend to be useless.

⁶ A recent (November 1992) compilation of “ontological concept-systems”, circulated over an electronic news group by Fritz Lehmann, listed 84 different systems!

⁷ A WordNet hypernym is the generic term used to designate a whole class of specific instances: Y is a hypernym of X if X is a (kind of) Y. For example, the sense #1 of the noun ‘file’ has the four following hypernyms, from more specific to more general: line, formation, array or arrangement, group or grouping.

5.3 Extending DIPETT's Coverage of English

It is of the highest importance that DIPETT's coverage of English grammar be as broad as possible (while remaining tractable), at least for the spectrum of syntactic phenomena we encounter in the type of technical text that is of interest to us in this research project. As we briefly discussed at the end of chapter 3, extended testing of the parser has revealed a number of weaknesses, some of which are more consequential than others. We now identify the main areas in which DIPETT's grammar would gain from the addition of new rules, or the extension or revision of the existing rules.

For the following unordered list of phenomena, the parser currently provides either a fragmentary analysis, because it could not find a full analysis, or an erroneous (full) analysis⁸. In other words, input strings containing one or more of these phenomena are not analyzed correctly, if at all, by the current version of DIPETT's grammar. This represents a much more serious problem than simple misattachments. At the time of writing, these drawbacks have not yet been corrected: extensive testing of the parser's coverage was completed only recently, and we also wanted to know what phenomena were most important to add to the current coverage of technical English.

- **reflexive pronouns:** for example, "*A summary operation specifies that a summary for the item is to be reported rather than the value of the item itself*". The parser attempts to analyze 'itself' as a separate and independent pronoun, i.e. which does not refer to 'item'. This might be a difficult problem for a surface parser.
- **genitives with implicit conjunction:** for example, "... , *such as an employee's number and name*". The parser does not associate 'name' with 'employee': 'name' is analyzed as a separate and independent noun. Obviously, both analyses are valid, grammatically speaking. The problem is that the first alternative (i.e. implicit conjunction) is not even considered.
- **punctuation, especially commas:** for example, "*In the next few pages you can learn how to ask the right questions even if you've never used a computer before*". The parser correctly analyzes this string only if a comma is inserted after the "fronted"

⁸ We note that these phenomena are not necessarily totally unknown to the grammar: they are not correctly analyzed. This may be caused either by competing rules that happen to always "win" over the other rules or by erroneous rules.

prepositional phrase, i.e. between ‘pages’ and ‘you’. Similarly with “*QUIZ reads the entire file, and selects the employees you want*”, where the comma in front of the ‘and’ prevents the parser from correctly analyzing the sentence. Certain types of parenthetical input strings also cause problems to the parser. For example, a parenthetical relative clause: “*And the FLOOR function (which rounds down) yields a value of 3*”.

- **enumerations:** for example, “*Functions simplify procedures such as 30-60-90 day aging routines, packing names for addressing, finding the occurrence of one string in another string, pattern matching, rounding numbers up or down, or manipulating characters for titles*”. Although the parser has all the necessary rules to correctly analyze such an input, it appears to be exceeded by some combinatorial aspect of the input string. Because the parser lacks an “enumeration” rule, simpler types of enumerations are also not (fully) analyzed by the parser: for example, “*The user can print any of these things: a letter, a program, or a list of items*”.
- **lexical ambiguity, especially for the noun/verb case:** for example, “*Report 1.2: Report sorted on more than one sort-key*”. The parser analyzes this string as an imperative statement, because of ‘report’ which can function either as a noun or a verb. If we introduce the determiner ‘a’ in front of “Report sorted”, the input is parsed correctly.
- **certain types of structures with inserted, optional modifiers:** for example, “*QUIZ can be tailored, with a little ingenuity, to suit almost any reporting requirement*” is not correctly analyzed, whereas this equivalent form is: “*With a little ingenuity, QUIZ can be tailored to suit almost any reporting requirement*”.
- **questions:** the current coverage of questions is minimal. Although questions are rare in the two texts we studied (i.e. the QUIZ manual and the Tax Guide), this subset of the parser’s grammar would have to be extended to cover other types of technical texts, such as textbooks, in which questions are much more frequent. On the syntax of questions, see [Quirk *et al.* 85: chapter 11].
- **phrasal verbs:** for example, “*He looked up the word in the dictionary*”. Even if the parser (and the lexicon) has the required knowledge to analyze phrasal verbs correctly, it often misparses them for two reasons. First, many particles can also

function as prepositions. Second, the parser gives priority to the analysis of prepositional phrases, because of their importance for Case-based semantic analysis. Thus, for the above example string, the parser mistakenly finds two prepositional phrases: “up the word” and “in the dictionary”. It may be the case that particles represent a more challenging problem than expected: not only for parsing, but also for semantic analysis⁹.

- **conjoined verbs:** for example, “*The bomb, which exploded and caused property damage ...*”. If conjoined verbs can both be transitive according to the lexicon, then the parser assumes that they have the same complement when the second verb is followed by one. In the example here, “property damage” is erroneously assumed to be the implicit complement of ‘exploded’. (Noun semantics would also help here.)
- **past participle constructions:** for example, “*The first amount is included in the total*” is parsed as a passive construction; the past participle construction is not considered. (Also, a few subtle forms of passive are not analyzed properly.)
- **the postposed pronoun ‘both’:** example, “*They told us both to wait*”. The parser only knows about ‘both’ as a predeterminer, not as a postposed pronoun — see [Quirk *et al.* 85: p.382].
- **post-verbal adverbs:** example, “*They are probably at home*”. Adverbs that appear between the verb and the complement, such as disjuncts (see [Quirk *et al.* 85: p.440]), are often misparsed; those that appear just before the verb are correctly analyzed (e.g. “*He rapidly became the leader of the group*”).
- **sentential relative clause:** for example, “*They say he plays truant, which he doesn’t*”. Relative clauses functioning as noun phrase post-modifiers are correctly analyzed but not those at the sentential level — see [Quirk *et al.* 85: p.1118].

We also include in this list the following phenomena which are rather uncommon in well-written technical texts.

⁹ A similar observation appears in [Dick 92: p.237]: “Particles, of all the lexical elements, have caused the most trouble. Although some work has been done on them, their behaviour is not yet fully understood. The dramatic affect they have on the meaning of the verbs to which they attach themselves is enough motivation to provoke curiosity and investigation.”

- **various ‘emphasis’ or stylistic constructions (e.g. topicalization, extraposition):** for example, “*Here is your sister, an envelope in each hand*”.
- **conjoined sentences with elided object:** for example, “*She designed and he implemented the new operating system*”. At present, such a construction is misparsed as a conjunction of two independent clauses.
- **mandative subjunctive:** for example, “*I demanded that the committee reconsider its decision*” — see [Quirk *et al.* 85: p.156].
- **noun phrases with embedded relativizers:** for example, “The report, the cover of which he designed”.

5.4 Transforming DIPETT into a Flexible Expectation-Driven Parser

Another aspect of DIPETT that deserves more research is the transformation of its current mode of parsing, which could be characterized as “best-first” uninformed, into a “best-first” expectation-driven analysis process. Typically, such a parser relies on a repository of information from which it fetches (syntactico-semantic) clues that may help it to process its input strings more reliably or more rapidly. The clues are the elements of information on which the parser can base its “predictive” behaviour. Based on these clues, it expects to find specific, more or less complex constructions in the (remainder of) the input. A gain is obtained when the parser finds that the input at hand matches the clues: in this situation, the clues constitute a “shortcut” to a successful analysis.

syntactico-semantic clues

For the sake of the discussion, let us suppose that the input string to be analyzed is *“The software engineer transferred the old MODULA compiler from partition A to partition B”*. Let us also suppose that the (hypothetical) parser — not only the semantic analyzer — has access to HAIKU’s KB and that the latter contains some information on the verb ‘transfer’. So the parser begins its analysis of the input and gets to the verb ‘transferred’, after having recognized the initial noun phrase (i.e. “The software engineer”) as a candidate subject group. Then it halts and consults the MDict (i.e. the “meaning dictionary”: see 4.3.1) in which it finds that ‘transfer’ has already been associated with the CMP *psubj-pobj-from-to* before.

At this point, the parser has only analyzed the initial noun phrase and the following verb and “knows” this: in the remainder of the input string, it can expect to find a direct object, a PP (prepositional phrase) with ‘from’, and a PP with ‘to’. More, the parser knows that the two PPs should be attached to the main verb, i.e. to ‘transfer’. Using appropriately this set of expectations, the parser can rapidly produce the correct parse, since the example input matches the expectations. By the same token, we know right away what is the CMP that must be associated with the current input, i.e. before getting to Case analysis where it is normally computed at the outset — all this, assuming at least one pattern existed for ‘transfer’ prior to the processing of the current input sentence.

However, if the clues do not match the expectations or if the parser cannot find any expectations in HAIKU’s KB, then the parser must be able to automatically return to its

original mode of parsing. It is in this sense that the parser's expectation mechanism should be flexible: if expectations do not lead quickly to a successful parse, then there is no need to perform any further computations or to interact with the user, and so the parser should simply get back to the default mode of parsing¹⁰.

There is another potential consequence of the addition of this expectation mechanism to the current design: if the clues do indeed match and lead to a successful analysis, not only do we know at once what the input's CMP is (as mentioned above) but, thanks to the CMPDict, we even have at least one candidate CP to propose to the user — say AGT-OBJ-LFRM-LTO for the above example. In that situation, we see that we would manage to find the input's CMP and CP without actually invoking the Case Analyzer (and thus minimizing even further the user's intervention). Of course, some elements of the Case Analyzer might still be needed to compute other results or structures, but the point is that the processing of input strings from which we cannot learn much (since we already know their CMP and CP) would be further automated¹¹.

In brief, as HAIKU would accumulate more and more knowledge in its incremental KB, especially in its MDict and CMPDict dictionaries, the parser's expectation mechanism would tend to become more and more productive and would thus tend to make the user's life more and more easy and less repetitive. This proposal means that the current architecture in which DIPETT and HAIKU are completely independent of one another would be replaced by one in which both modules would indirectly interact through the parser's expectation mechanism.

Finally, three additional remarks. First, we note that if semantic categories are added to HAIKU's patterns, as discussed in section 5.2, the above discussion is still valid¹². The only difference is that the semantic categories would be part of the expectation information (i.e. the clues). For instance, if the parser is looking for a *from*/DEVICE-PP (instead of simply a PP with 'from'), this means that it would have to find a PP with 'from' in which the noun filler would have to be of the semantic category DEVICE. Second, compared to the current approach, expectation-driven parsing is probably easier

¹⁰ We are not saying that the hypothetical mode of parsing described here would be easy to implement within the bounds of the current parser's implementation. In fact, we do not know for sure yet.

¹¹ In a sense, processing would "jump" from parsing to the conclusion of semantic analysis. From the user's point of view, the system would appear more autonomous (i.e. more "intelligent") and, also, more dependable.

¹² In fact, our proposal would be compatible with the notion of "Case parsing": see [Somers 87: chapter 11].

to defend from a cognitive viewpoint: as processors of language, we know that we ourselves use “clues” all the time. Third, the above proposal is moderate when compared to others, such as [Trost *et al.* 88], in which semantic processing always has the lead and is guided by syntactic analysis. However, such approaches do require significant *a priori* (semantic) knowledge.

5.5 Text Summarization as Parametrizable Text Highlighting

5.5.1. Introduction

Text summarization (TS) is a very complex activity. A good summary concentrates on central elements and selectively ignores peripheral details [Lehnert 82: p.376]. Things that are implicit or that can easily be induced or deduced based on the reader's knowledge need not be mentioned. But how does one evaluate importance and relevance in order to perform TS in an automatic TP system?

Early work on TS focused mainly on stories ([DeJong 82], [Lehnert 82]). DeJong's FRUMP system provides summaries based on a top-down script-based extraction of relevant information in specific domains. As is true of most approaches based on story grammars, FRUMP is expectation-driven and thus requires significant *a priori* knowledge stored in its scripts. We cannot adopt such a strategy in TANKA for two reasons. First, we do not want to assume much *a priori* semantic knowledge because it is our goal to incrementally acquire such knowledge through TP. Second, story processing techniques are not necessarily appropriate for expository texts in general and technical texts in particular. Expository texts are organized hierarchically while stories tend to be organized sequentially. The notion of story grammar has no widely accepted equivalent in expository text (but see [Berry-Rogghe & Kaplan 86] for a proposal).

5.5.2. A Proposal Based on the Notion of Conceptual Distance

Although interest in TS seems to have decreased lately, researchers into expository TP have proposed several ideas that could be further developed profitably. In [Nishida *et al.* 86], the authors are interested in global text analysis based on intersentential relations. They suggest the hierarchical structure of expository texts be exploited by considering chapter headings as topic indicators and section headings as topic attributes¹³. In [Fum *et al.* 85], several criteria for evaluating importance are presented. Among these, we consider the following two to be particularly significant: i) the relation between a given sentence and the topic, and ii) references to semantically relevant concepts in the domain.

¹³ See [Hahn 92] for a computational framework that deals with global text structures at the "text coherence" level. The proposed systems recognizes several coherence phenomena such as 'constant theme', 'continuous thematization of rhemes', and 'derived theme'.

The default mode of operation for DIPETT is to process one sentence at a time. We think a desirable feature would be to parse longer input strings such as a paragraph¹⁴ or a free segment of a text (i.e. a certain number of consecutive sentences). However this implies much more extensive processing than is involved in parsing a single sentence — not much would be gained if processing simply provided an ordered sequence of parse trees, one per input sentence (see [Jensen 89] for a similar observation). We envisage more intelligent parsing which would summarize the contents of these longer input strings, i.e. *we are trying to see TS as a special parsing mode*.

Because TANKA is concerned with language analysis rather than language generation, we plan to produce summaries using the actual clauses of the input text. To decide whether a given clause should appear in the summary, we employ the following test: if the clause's conceptual distance $\mathbb{CD}$ is smaller than the granularity parameter Γ , then it becomes a candidate clause for the resulting summary. We assume that the integration of new knowledge in TANKA's knowledge base (Conceptual Network or CN) always take place before TS.

The conceptual distance of a clause is tentatively expressed in terms of five distances:

- 1) the distance between the main concept of the point of focus (the clause) and the current cluster (topic¹⁵) in the CN — the current cluster is associated with the most recent topic shift (e.g. the most recent beginning of a new paragraph);
- 2) the distance between the current cluster and its hierarchical parent in the CN;
- 3) the distance between the point of focus' main concept and the local heading's main concept in the CN;
- 4) the distance between the local heading's main concept and the top level (e.g. chapter) heading's main concept in the CN;
- 5) the "hierarchical distance", i.e. the number of intervening levels, between the local heading and the first level heading in the text.

¹⁴ Interestingly, major topic movements occur predominantly at paragraph boundaries ([Hahn 92: p.28]). Also, the computational theory of the paragraph proposed by [Zadrozny & Jensen 91] should be considered.

¹⁵ On the problem of automatically identifying topic and focus, see [Hajicova & Sgall 86] and [Hajicova & Sgall 93].

The above definition of conceptual distance has a clear computational interpretation: composite distances #1-#4 can be evaluated in terms of edges and vertices in the CN, while composite distance #5 can be evaluated in terms of the decimal notation commonly used to number section headings in technical texts. The granularity parameter Γ is an indication of the level of detail desired in the summary. It could be set to a specific value by the user. A high value of Γ would indicate that we are looking for a summary in which clauses may be located at relatively large conceptual distances from one another: this would produce a detailed summary. A low value of Γ would indicate that we are looking for a more concise and coherent summary.

CHAPTER 6

6. CONCLUSION.....	200
6.1 Significance of the Results.....	202
6.2 Current and Prospective Applications.....	206
6.2.1 DIPETT and HAIKU as MaLTe's Linguistic Module.....	206
6.2.2 A Tool for Assisting Conceptual Data Modelling	207
6.2.3 The Processing of Non-Technical Text.....	208

6. CONCLUSION

This thesis is concerned with the language processing aspects of the task of acquiring knowledge from technical text with little *a priori* domain knowledge. We proposed a semi-automatic approach for which we described and tested a Prolog implementation organized around two central modules, the DIPETT parser and the HAIKU Case-based semantic analyzer. These modules, especially if they are part of an encompassing system such as TANKA, are expected to assist the knowledge engineer's task in situations where technical texts are available. The two modules constitute a semi-automatic linguistic (sub)system, devised for knowledge acquisition purposes, which facilitates the modelling of a domain of interest.

The parser has a broad coverage of technical English and includes mechanisms that help to make it an effective text parsing tool — a tool whose effectiveness should increase even more in the future, as discussed in chapter 5. The semantic analyzer interacts with the user to get his approval of the analysis it proposes and negotiates elements of this analysis whenever necessary. As HAIKU progresses through the input text, it incrementally accumulates knowledge elements that can be used profitably so that it behaves more and more autonomously and thus alleviates the user's participation. Because of their domain independence, both DIPETT and HAIKU contribute to the generality and portability of the approach we put forward.

We illustrate the operation of the system proposed in this thesis by the following practical situation. A knowledge engineer is confronted with the task of modelling the (domain) knowledge contained in a technical manual. In order to assist himself in this undertaking, he wishes to use a system like TANKA, based on a linguistic processor composed of DIPETT plus HAIKU. In an initial phase, he uses DIPETT's simple tagger (see 3.3.5.3) in order to define the manual's vocabulary and build a customized lexicon. Then he calls upon DIPETT to parse, in batch, the technical manual, using the lexicon he prepared earlier. All the parse trees, either full or fragmentary, are collected in a file. In a third phase, he employs HAIKU to semi-automatically perform a Case-based semantic analysis of the parses. At this point, the knowledge engineer has accumulated a repository of knowledge elements, i.e. a KB, with the help of a linguistic processor.

If DIPETT and HAIKU are used within a system like TANKA, then the resulting KB would represent a computerized model of the manual's domain. On the other hand, if DIPETT and HAIKU were used independently of such an encompassing system (i.e. as an independent conceptual analyzer), then HAIKU would nevertheless have accumulated a rich set of knowledge elements within its three main dictionaries of syntactic and semantic patterns. These patterns can be said to represent a particular (partial) model of the text with an emphasis on the knowledge conveyed through the verbs and their complements. Such knowledge is especially useful in showing the main relationships between a domain's main activities and its main objects: the knowledge symbolizes a simple conceptual model. In a sense, this simple model could be said to represent a "conceptual X-ray" of the input text.

6.1 Significance of the Results

We believe the work presented in this thesis has contributed to the fields of NLP and KA by presenting a systematic methodology, supported by a comprehensive implementation, to the difficult problem of incrementally acquiring knowledge from text. In particular, the fact that we did not assume an *a priori* domain-specific KB contributed to the originality of the solution we proposed. As far as we know, no approach close to ours has ever been described in the literature.

We also note that the approach proposed in this thesis differs significantly from the information-retrieval and the statistical paradigms, as mentioned in 2.2. The information-retrieval approach (e.g. [Ciravegna *et al.* 92], [Kim & Moldovan 93], and [Appelt *et al.* 93]) is often based on a frame instantiation strategy, which requires the system to manipulate pre-determined frame structures containing *a priori* knowledge. Such systems tend to be domain-dependent and usually rely on partial parsing or keyword matching. The statistical approach (e.g. [Smadja 91] and [Brent 91a]), usually based on concepts such as word collocations, n-grams, etc., is typically non-incremental and semantically weak. It requires large (often syntactically pre-tagged) corpora and sometimes depends on partial parsing as well. Most of all, results of the statistical approach are rather limited in terms of the goals of this research. For instance, only high-frequency patterns (e.g. collocations) can be statistically validated. In TANKA, our view is that any (semantic) pattern approved by the user is important and may contribute to the knowledge acquisition process.

We now list the central claims made earlier which we think underline the contributing elements of this research, and for which the body of the thesis provides support:

- we have provided a computational setting for the efficient implementation of semi-automatic assistants for KA from technical text. Much of the thesis concentrated on the engineering side of the proposal, providing all the technical details instead of being concerned solely with generalities¹.

¹ This type of description helps present not only what could or can be done about a problem, but what was precisely accomplished and how we actually did (i.e. implemented) it.

- we have demonstrated the practicality of an NLP approach to KA in which the linguistic module does not rely on extensive or complex *a priori* domain knowledge, provided a competent user (see 1.3.2.2) can supervise or guide the system whenever appropriate, especially during the initial “learning” phase since the system will behave more and more autonomously as it accumulates knowledge. In fact, DIPETT and HAIKU are key elements to the transformation of knowledge-based text understanding into KA: if the system cannot “understand” an input, its KB lacks elements of knowledge that are worth acquiring.
- we have demonstrated the appropriateness of a compositional approach using detailed syntactic analysis combined with Case-based semantic analysis: in this context, the surface form of the input text is quite naturally amenable to KA *via* NLP.
- we have illustrated in a specific context how the individual problems of NLP and KA seem to share fundamental processing properties. More precisely, NLP involves KA at all levels: i.e. lexical, syntactic, and semantic; and KA without NLP appears impossible: knowledge is a “cognitive artifact” that is intrinsically human and could not exist without NL.

Scientific research contributes to the advancement of our knowledge not only through complete solutions but also through the identification of limitations, open questions or unresolved issues, and further development. Since we have already covered the topic of future work in chapter 5, and since section 6.2 discusses applications of our approach, the rest of this subsection will consider the other elements we have just listed.

As far as limitations are concerned (excluding those that were discussed in chapter 5), we make three remarks:

- as on-line (*versus* batch) parsing may be a preferable mode in certain practical situations, faster parsing for many types of complicated input strings is a necessary enhancement.

- the current user interface is simplistic and presents a lot of information to the user: a new design, probably based on some graphical tool, is another much needed enhancement.
- on a less technical level, the “quality” of the text submitted on input has a serious impact on the ease of processing: on the one hand, a poorly written text is not likely to represent the domain of interest very well; on the other hand, a text written with too many complicated sentences² is likely to cause serious difficulties to the syntactic and semantic analyzers.

The research work described in this thesis also opened the door to several questions or problems that remain unresolved:

- what is the precise impact of “partial” KA on the resulting KB? That is to say, when semantic analysis is performed on fragmentary parses, what do we lose exactly in the process? One could argue that any NLP approach to KA from text is bound to be incomplete. But how incomplete is it?
- this thesis deals exclusively with the analysis of NL. Are there any elements of our approach that could be useful for the converse problem of language generation? For instance, suppose we analyze a text for which we would like to reformulate (i.e. rewrite) some sections that are poorly written, could we use the KB obtained during analysis in order to guide the generation process?
- are there any “purely linguistic” applications for HAIKU? For instance, could a linguist be interested in using it to study the properties of verbs and their complements in order to determine, say, certain classes of verbs based on some syntactic or semantic patterns? Or to compute frequency statistics of some patterns, possibly occurring with some verbs? Or to study the nature of Case fillers in relation with the different verbs with which they occur? Or to study the relationships between certain syntactic and semantic patterns? (See also the questions raised in the very last paragraph of chapter 4.)

² The Tax Guide is guilty of this writing practice. Sentences containing some 60 or 70 words, including 5 or 6 conjunctions, are not uncommon. They not only give a hard time to DIPETT but also to human parsers!

- can the current CPU time limit strategy, used for setting a maximum parsing time (see 3.4.2), be transformed into an “anytime module”³ strategy? The latter would allow the user to set a time period under which the system would always have to yield a result, maybe incomplete but never useless. In other words, we could launch parsing and whenever it is interrupted (by the user, say), the parser would return the best result it had managed to compute so far. The longer the time period, the better the results. For instance, this time period could function as a parameter influencing the granularity of the parsing process. The shorter the time period, the less the parser’s analysis would be fine-grained.
- in this thesis, we discussed the idea of semi-automatically building *domain-specific* knowledge bases from technical text. What if we applied the same methodology to general textual material, such as the definitions of The Collins COBUILD dictionary, in order to derive a *very general* knowledge base⁴? For example, let us suppose that we would process all definitions of verbs (including the accompanying example sentences). Could we derive a general catalog of CMPs and CPs for verbs that would actually be similar to a list of legal selectional patterns for these verbs? The information collected during such a process would then be available afterwards for easier processing of domain-specific texts, assuming such a general KB would be used as the initial HAIKU KB. Such a general KB would provide additional support for the transformation of DIPETT into an expectation-driven parser (see 5.4) by making available to the parser the kind of information that is useful in expectation-driven parsing.

³ This concept was presented at COLING-92 (on July 23rd) by a German guest speaker whose name is unfortunately a mystery to the author of this thesis!

⁴ This hypothesis was suggested by Dr. Harvey Abramson, a member of the author’s thesis committee.

6.2 Current and Prospective Applications

6.2.1 DIPETT and HAIKU as MaLTe's Linguistic Module

One important claim of our approach is its domain independence and portability. MaLTe, an ongoing project at the Department of Computer Science of the University of Ottawa, provides some support to this claim. We here succinctly describe MaLTe, based on [Matwin & Szpakowicz 92], from which we borrowed a few excerpts with the kind permission of the authors — for more details, see that reference as well as [Delisle *et al.* 94]. This project is a natural but substantial extension of an earlier proof of concept presented in [Delisle *et al.* 91]. One of its main goals was to show the potential of an approach to TP in which NLP and ML techniques are synergistically combined.

The MaLTe system is built upon the linguistic module developed for the TANKA project, i.e. DIPETT and HAIKU. As in TANKA, Case analysis is the centrepiece of linguistic processing of texts in MaLTe. What is specific to MaLTe though is the use of ML techniques that can help extract knowledge from the text by doing appropriate generalization. EBL offers knowledge compilation and generalization methods that seem adequate for the analysis of narratives (describing a domain in generic terms), and examples (showing how this knowledge applies to individuals in the domain). To put it simply, domain theory for EBL is derived from the narrative, while examples serve as data.

In other words, MaLTe attempts to explain the examples using the knowledge contained in the narrative. An explanation is then operationalized (made useful in terms of the system knowledge) and generalized so that it is useful for a whole class of similar examples. The top-level algorithm used in MaLTe is summarized below: the input is a sentence and the associated output is a number of Horn clauses representing this sentence. It is to these Horn clauses that ML techniques will be applied.

- syntactic processing: parsing with DIPETT;
- semantic processing: Case-based semantic analysis with HAIKU;
- translation into logic: conversion, clause by clause, of the syntactic form into Horn clauses;

- prescriptive structure determination: an optional step which is applied to prescriptive structures (in the sense of [Moulin & Rousseau 90]), i.e. which are expressed as if-then rules (e.g. “If you married in 1990, (then) your spouse becomes a supporting person for all of 1990”);
- selection of the theme predicate: determines what single predicate is the most important for the meaning of the sentence;
- unflattening: converts a clause’s predicates into arguments of other predicates.

6.2.2 A Tool for Assisting Conceptual Data Modelling

A data model is a collection of concepts that describe a set of data and operations to manipulate the data. Such conceptual data models, often expressed as a logical (or relational) schema, are used during the design phase of a database (DB). Thus, a schema is a representation of a specific portion of real-life situation. More precisely, a schema is a collection of NL or graphic representations that describe the structure of the data of interest, such as that within one organization⁵.

Typically, DB designers use entity-relationship (ER) diagrams to model the domain. Although the basics of ER diagrams are fairly simple and well-known nowadays, the process of actually building such diagrams for a given complex situation is a difficult task. Based on his comprehension of the enterprise, the DB designer has to identify domain entities, relationships, and attributes of these two. Normally, entities and attributes are expressed as nouns while relationships are expressed as verbs: nouns and verbs serve as descriptors in NL paraphrases.

The application we wish to propose here is the use of DIPETT and HAIKU as the central linguistic module of a computerized tool for assisting the DB designer in the elaboration of a conceptual data model. The main objective of this tool would be to make useful suggestions to the designer as to what could be considered candidate entities, relationships, and attributes, in a given domain. (We are not saying that the output of such a system would be actual ER diagrams. Rather, the output would constitute some kind of a rudimentary conceptual data model that could be further processed for DB design purposes.)

⁵ This introductory paragraph was inspired by the first part of section 2.3 (“Data Models”) in [Batini *et al.* 92].

For this proposal to be feasible, we must have access to texts that describe the domain of interest well. Such texts do exist in medium-size and large organizations: texts that describe how an organization (or its part) is structured and how it functions, its regulations, norms, internal policies, contracts, collective agreements, etc. We hypothesize that if such texts are indeed available and if they are representative of the domain to be modelled, then they could be processed profitably in order to assist the DB designer's task by identifying candidate ER schema elements. In terms proper to DIPETT and HAIKU, these ER elements would be expressed as the most relevant nouns and verbs, and their interrelations (*via* Case structures: see next paragraph).

Since the current version of HAIKU's KB (i.e. its three dictionaries) contains more information on verbs than on nouns (or nominal fillers⁶), a revision of these structures would be necessary to provide a more balanced account of meaningful ER elements. There are two potential additions. First, the addition of ontological markers for nouns, as discussed in 5.2, would seem most desirable. Second, another possible avenue for extending HAIKU's KB would be to store the output Case structures: this would allow us to accumulate detailed information on Case fillers (e.g. their semantic categories). In that way, we could have a bipartite KB with entries indexed either on verbs (as in the current design), or on fillers (e.g. AGT/PERSON).

6.2.3 The Processing of Non-Technical Text

Although many aspects of the work presented in this thesis remain valid for other types of text than the one we considered here, a challenging (experimental) application of DIPETT and HAIKU would be to tackle the processing of other kinds of text. The latter could be technical text containing exchanges between two or more intelligent agents (e.g. a conversation or dialogue about some technical domain), or non-technical text (e.g. a story, personal letter, play, or newspaper article) which normally involves several agents⁷. In what follows, we will be referring to such texts simply as "non-technical" texts.

Obviously, as they stand right now, the current versions of DIPETT and HAIKU are not prepared to properly handle non-technical text. In particular, they lack the required apparatus to deal with most discourse processing aspects of such texts (see 2.1.3.4).

⁶ Strictly speaking, fillers are not simply nouns. Some are nominal clauses or adverbial clauses; clauses may themselves have their own internal fillers, as we saw in chapter 4.

⁷ This involvement may be more or less direct and may take different forms, e.g. quotation or reported speech.

Similarly to what happens within TANKA and MaLT_e, our hypothetical application would consist of using (updated versions of) DIPETT and HAIKU as the basic linguistic processor of an encompassing system dedicated to the processing of non-technical texts. The main goal of this system would be, say, to translate a non-technical input text into a computer-tractable representation.

Let us assume that the encompassing system uses a speech act approach to the semantic representation of those elements of the text which involve exchanges between agents. This approach could be a derivative of a theory such as that of [Cohen & Perreault 79], [Allen & Perreault 80], or [Vanderveken 90, 91]. For instance, in [Moulin *et al.* 92: p.150], the authors describe their method as follows: "Our approach is based on a language/action perspective (Winograd 1987) where speech acts are not considered as unrelated events, but as parts of larger conversational structures. Conversations in which agents cooperate to achieve various discursive goals such as 'to obtain information', 'to come to an agreement' etc., can be found in dialogues and be reported in written texts ...". The use of an approach of that sort seems a requirement for the processing of non-technical text for *we are not only interested in what the text says but also in who says what, and even why*.

Thus, in that context, the objective of the experimental application would be twofold. First, evaluate how feasible and profitable it would be to "export" DIPETT and HAIKU to a TP environment such as the one outlined above. Second, study how the basic hypothesis of this thesis (i.e. TP without *a priori* domain knowledge for KA) would be amenable to this TP environment and, more precisely, how it could best be exploited to realize a productive TP system for the analysis of non-technical (as well as technical) texts.

To illustrate more concretely a potential application of the type discussed above, the author spent several hours with a researcher⁸ who is mainly interested in the analysis of non-technical texts of a rather unusual type, i.e. Spanish plays of the 17th century. He was given a demonstration of DIPETT and HAIKU — using English technical text, of course — and was asked to describe a hypothetical TP environment that would be built on top of DIPETT and HAIKU and which would be adapted to his research needs.

⁸ Professor Ricardo Serrano (Département des Langues Modernes de l'Université du Québec à Trois-Rivières).

Although the researcher was quite pleased with what he saw during the demonstration, he nevertheless identified an impressive list of additional computations or facilities that he considers essential for the final TP system he envisions. We will not discuss whether these should eventually be part of DIPETT, HAIKU, or the hypothetical encompassing system. Instead, the next paragraph lists some of the most important elements which show quite well why discourse processing is still one of the most difficult and challenging domains of modern NLP.

The intended (encompassing) system would analyze a text, e.g. a play, and represent its contents in such a way that it would be able to answer these questions: what is the main topic of the text; for each character⁹, what are the main conceptual elements of what each says; how is the discourse organized and segmented (by conversational exchange and by character); what are the main themes; what is the time-space situation for the characters and the things each says¹⁰; what are the main logical or strategic relationships amongst constituting elements of the segments¹¹; what is the propositional content and what are the illocutionary markers of the phrases, etc. As the reader can judge for himself, no doubt the (even partial) automation of such an elaborate TP system will keep computational linguists busy for years ahead ...

⁹ Or speaker, or actor, or agent.

¹⁰ There are a few subtleties here. In particular, a character may be located in a given time-space continuum while what he utters refers to a different one. For example, I may be talking right now about things that took place a few hundred years ago.

¹¹ For instance, two characters may be in opposition in a text and their utterances may thus represent logical contradictions.

7. REFERENCES

Abbreviations:

AAAI: American Association for Artificial Intelligence;
ACL: Association for Computational Linguistics;
ACM: Association for Computing Machinery;
CACM: Communications of the ACM;
COLING: International Conference on Computational Linguistics;
Conf.: Conference;
IJCAI: International Joint Conference on Artificial Intelligence;
Proc.: Proceedings;
SIGART: Special Interest Group on Artificial Intelligence of the ACM;
TR: Technical Report.

Cross-Reference Information:

The following references are cross-referenced with the associated list of chapter numbers in which they are cited. This list appears at the very end of each individual entry and is delimited with braces. For example, "{S&T,2,3}" means that a given reference is cited in the "Symbols and Terminology" section (at the very beginning of the document) and chapters 2 and 3.

- Abramson, H. & Dahl, V.** (1989), *Logic Grammars*, Springer-Verlag, 1989. {3}
- Agarwal, R. & Boggess, L.** (1992), "A Simple but Useful Approach to Conjunct Identification", *Proc. of the 30th Annual Meeting of the ACL*, 15-21. {2}
- Agarwal, R.** (1990), "Disambiguation of Prepositional Phrase Attachments in English Sentences Using Case Grammar Analysis", M.Sc. Thesis, Department of Computer Science, Mississippi State University. {2}
- Aho, A.V. & Ullman, J.D.** (1972), *The Theory of Parsing, Translation, and Compiling (Volume 1: Parsing)*, Prentice-Hall. {3}
- Akmajian, A., Demers, R.A. & Harnish, R.M.** (1984), *Linguistics (An Introduction to Language and Communication)*, MIT Press. {1}
- Alexander, J.H., Freiling, M.J., Shulman, S.J., Rehfuess, S. & Messick, S.L.** (1988), "Ontological Analysis: an Ongoing Experiment", in B.R. Gaines & J.H. Boose (eds.), *Knowledge Acquisition for Knowledge-Based Systems (Knowledge-Based Systems — Volume 2)*, Academic Press, 25-37. {1}
- Allen, J. & Perreault, C.R.** (1980), "Analyzing Intention in Utterances", *Artificial Intelligence*, 15, 143-178. {6}
- Allen, J.** (1987), *Natural Language Understanding*, Benjamin/Cummings. {S&T,2}
- Allgayer, J., Jansen-Winkel, R., Reddig, C. & Reithinger, N.** (1989), "Bidirectional Use of Knowledge in the Multi-Modal NL Access System XTRA", *Proc. of IJCAI-89*, 1492-1497. {2}

- Alshaw, H.** (1987a), *Memory and Context for Language Interpretation*. Cambridge University Press. {2}
- Alshaw, H.** (1987b), "Processing Dictionary Definitions with Phrasal Hierarchies", *Computational Linguistics*, 13(3-4), 195-202. {2}
- Alshaw, H.** (1990), "Resolving Quasi Logical Forms", *Computational Linguistics*, 16(3), 133-144. {2}
- Altmann, G.** (1988), "Ambiguity, Parsing Strategies, and Computational Models", *Language and Cognitive Processes*, 3(2), 73-97. {2}
- Appelt, D.E., Hobbs, J.R., Bear, J., Israel, D. & Tyson, M.** (1993), "FASTUS: A Finite-state Processor for Information Extraction from Real-world Text", *Proc. of IJCAI-93*, 1172-1178. {6}
- Arinze, B.** (1989), "A Natural Language Front-End for Knowledge Acquisition", *SIGART*, 108, 106-114. {2}
- Asker, L., Gambäck, B. & Samuelsson, C.** (1992), "EBL²: An Approach to Automatic Lexical Acquisition", *Proc. of COLING-92*, 1172-1176. {3}
- Attal, J.P.** (1987), *Grammaire et Usage de l'Anglais*, Duculot. {3}
- Ayuso, D.M.** (1989), "Discourse Entities in Janus", *Proc. of the 27th Annual Meeting of the ACL*, 243-250. {2}
- Ayuso, D.M., Shaked, V. & Weischedel, R.M.** (1987), "An Environment for Acquiring Semantic Information", *Proc. of the 25th Annual Meeting of the ACL*, 32-40. {1,2}
- Baker, C.L.** (1989), *English Syntax*, MIT Press. {3}
- Ballard, B.W. & Stumberger, D.E.** (1986), "Semantic Acquisition in TELI: A Transportable, User-Customized Natural Language Processor", *Proc. of the 24th Annual Meeting of the ACL*, 20-29. {1,2}
- Ballard, B.W.** (1986), "User Specification of Syntactic Case Frames in TELI, a Transportable, User-Customized Natural Language Processor", *Proc. of COLING-86*, 454-460. {1,2}
- Ballard, B.W.** (1988), "A Lexical, Syntactic, and Semantic Framework for TELI: a User Customized Natural Language Processor", in M.W. Evens (ed.), *Relational Models of the Lexicon (Representing Knowledge in Semantic Networks)*, Cambridge University Press, 211-236. {1}
- Barker, K., Copeck, T., Delisle, S., & Szpakowicz, S.** (1993), "An Empirically Grounded Case System", TR-93-08, Computer Science Department, University of Ottawa. {4, Appendix F}
- Barnett, J., Knight, K., Mani, I. & Rich, E.** (1990), "Knowledge and Natural Language Processing", *CACM*, 33(8), 50-71. {1,2}

- Barton, G.E., Berwick, R.C. & Ristad, E.** (1987), *Computational Complexity and Natural Language*, MIT Press. {2,3}
- Basili, R., Pazienza, M.T. & Velardi, P.** (1992), "Computational Lexicons: the Neat Examples and the Odd Exemplars", *Proc. of 3rd Conf. on Applied Natural Language Processing*, 96-103. {2}
- Bateman, J.A., Kasper, R.T., Moore, J.D. & Whitney, R.A.** (1990), "A General Organization of Knowledge for Natural Language Processing: the Penman Upper Model", TR of USC/ISI, Marina del Rey, CA. {5}
- Bates, M.** (1978), "The Theory and Practice of Augmented Transition Network Grammars", in L. Bolc (ed.), *Natural Language Communication with Computers*, Springer-Verlag, 191-260. {2}
- Batini, C., Ceri, S. & Navathe, S.B.** (1992), *Conceptual Database Design (An Entity-Relationship Approach)*, Benjamin/Cummings. {6}
- Belaïd, A. & Belaïd, Y.** (1992), *Reconnaissance des Formes (méthodes et applications)*, InterÉditions. {4}
- Belkin, N.J., Brooks, H.M. & Daniels, P.J.** (1988), "Knowledge Elicitation Using Discourse Analysis", in B.R. Gaines & J.H. Boose (eds.), *Knowledge Acquisition for Knowledge-Based Systems (Knowledge-Based Systems — Volume 1)*, Academic Press, 107-124. {1}
- Berry-Rogghe, G. & Kaplan, R. M.** (1986), "On the Structure of Expository Texts: Preliminaries to Building an Expert System from Manuals", *Proc. of the 6th International Workshop on Expert Systems and Their Applications* (Avignon), 1125-1132. {5}
- Berwick, R.C. & Weinberg, A.S.** (1984), *The Grammatical Basis of Linguistic Performance (Language Use and Acquisition)*, MIT Press. {2,3}
- Berwick, R.C.** (1983), "A Deterministic Parser with Broad Coverage", *Proc. of IJCAI-83*, 710-712. {2}
- Berwick, R.C.** (1985), *The Acquisition of Syntactic Knowledge*, MIT Press. {2}
- Biber, D.** (1989), "A Typology of English Texts", *Linguistics*, 27, 3-43. {1}
- Biber, D.** (1993), "Using Register-Diversified Corpora for General Language Studies", *Computational Linguistics*, 19(2), 219-241. {1}
- Black, E., Lafferty, J. & Roukos, S.** (1992), "Development and Evaluation of a Broad-Coverage Probabilistic Grammar of English-Language Computer Manuals", *Proc. of the 30th Annual Meeting of the ACL*, 185-192. {1,3}
- Blanchon, H.** (1992), "A Solution to the Problem of Interactive Disambiguation", *Proc. of COLING-92*, 1233-1238. {5}
- Bogges, L., Agarwal, R. & Davis, R.** (1991), "Disambiguation of Prepositional Phrases in Automatically Labelled Technical Text", *Proc. of AAAI-91*, 155-159. {2}

- Boguraev, B. & Briscoe, T.** (1987), "Large Lexicons for Natural Language Processing: Utilising the Grammar Coding System of LDOCE", *Computational Linguistics*, 13(3-4), 203-218. {2}
- Brady, M. & Berwick, R.C.** (1983), *Computational Models of Discourse*, MIT Press. {2}
- Brent, M.R.** (1991a), "Automatic Acquisition of Subcategorization Frames from Untagged Text", *Proc. of the 29th Annual Meeting of the ACL*, 209-214. {2,6}
- Brent, M.R.** (1991b), "Automatic Semantic Classification of Verbs from Their Syntactic Contexts: An Implemented Classifier for Stativity", *Proc. of the 5th Conference of the European Chapter of the ACL*, 222-226. {2}
- Brill, E.** (1993), "Automatic Grammar Induction and Parsing Free Text: A Transformation-Based Approach", *Proc. of the 31st Annual Meeting of the ACL*, 259-265. {5}
- Briscoe T., Grover, C., Boguraev, B. & Carroll, J.** (1987), "A Formalism and Environment for the Development of a Large Grammar of English", *Proc. of IJCAI-87*, 703-708. {2}
- Brown, R.D. & Nirenburg, S.** (1990), "Human-Computer Interaction for Semantic Disambiguation", *Proc. of COLING-90*, 42-47. {4,5}
- Bruce, B.** (1975), "Case Systems for Natural Language", *Artificial Intelligence* 6(4), 293-326. {4}
- Buchanan, B.G. & Shortliffe, E.H.** (1984), *Rule-Based Expert Systems (The MYCIN Experiments of the Stanford Heuristics Programming Project)*, Addison-Wesley. {1}
- Bulko, W.C.** (1989), "Understanding Coreference in a System for Solving Physics Word Problems", TR AI89-102 (Ph.D. Thesis), Artificial Intelligence Laboratory, The University of Texas at Austin. {2}
- Byrd, R.J., Calzolari, N., Chodorow, M.S., Klavans, J.L., Neff, M.S. & Rizk, O.A.** (1987), "Tools and Methods for Computational Lexicology", *Computational Linguistics*, 13(3-4), 219-240. {2}
- Carbonell, J.G. & Hayes, P.J.** (1983), "Recovery Strategies for Parsing Extragrammatical Language", *Computational Linguistics*, 9(3-4), 123-146. {3}
- Cardie, C.** (1992a), "Learning to Disambiguate Relative Pronouns", *Proc. of AAAI-92*, 39-43. {3}
- Cardie, C.** (1992b), "Corpus-based Acquisition of Relative Pronoun Disambiguation Heuristics", *Proc. of the 30th Annual Meeting of the ACL*, 216-223. {3}
- Carrol, J., Briscoe, T. & Grover, C.** (1991), "A Development Environment for Large Natural Language Grammars", TR, University of Cambridge Computer Laboratory. {2}

- Carter, D.** (1987), *Interpreting Anaphors in Natural Language Texts*, Ellis Horwood. {2}
- Celce-Murcia, M.** (1975), "Verb Paradigms for Sentence Recognition", *American Journal for Computational Linguistics*, microfiche 38. {4}
- Cercone, N. & McCalla, G.** (1985), "Accessing Knowledge Through Natural Language", TR-85-15, Laboratory for Computer and Communications Research, Simon Fraser University. {2}
- Chomsky, N.** (1986), *Knowledge of Language (Its Nature, Origin, and Use)*, Praeger. {3}
- Chomsky, N.** (1987), *La Nouvelle Syntaxe*, Éditions du Seuil. {3}
- Ciravegna, F., Campia, P. & Colognese, A.** (1992), "Knowledge Extraction from Texts by SINTESI", *Proc. of COLING-92*, 1244-1248. {2,6}
- Cohen, P.R. & Perreault, C.R.** (1979), "Elements of a Plan-Based Theory of Speech Acts", *Cognitive Science*, 3(3), 177-212. {6}
- Colmerauer, A.** (1978), "Metamorphosis Grammars", *Lectures Notes in Computer Science*, Springer-Verlag, 63, 133-189. {3}
- Cook, W.A.** (1979), *Case Grammar: Development of the Matrix Model (1970-1978)*, Georgetown University Press. {4}
- Copeck, T., Barker, K., Szpakowicz, S. & Delisle, S.** (1994), "An Inventory of Properties Characterizing Technical Texts", forthcoming. {1}
- Copeck, T., Delisle, S., & Szpakowicz, S.** (1990), "Intelligent Case Analysis in the KATE System", TR-90-24, Computer Science Department, University of Ottawa. {1,4}
- Copeck, T., Delisle, S., & Szpakowicz, S.** (1992), "Parsing and Case Analysis in TANKA", *Proc. of the 15th International Conference on Computational Linguistics — COLING-92* (Nantes, France), 1008-1012. {1,3,4}
- Corriveau, J.-P.** (1991), "Time-Constrained Memory for Reader-Based Text Comprehension", TR CSRI-246 (Ph.D. Thesis), Computer Systems Research Institute, University of Toronto. {3}
- Covington, M.A.** (1993), "Ten Criteria for Classifying Natural-Language Parsers (April 28, 1993)", unpublished draft distributed by electronic mail, University of Georgia. {3}
- Cowie, J.R.** (1983), "Automatic Analysis of Descriptive Texts", *Proc. of 1st Conf. on Applied Natural Language Processing*, 117-123. {2}
- Cunningham, P. & Veale, T.** (1991), "Organizational Issues Arising from the Integration of the Lexicon and Concept Network in a Text Understanding System", *Proc. of IJCAI-91*, 986-991. {2}

- Dahl, D.A.** (1986), "Focusing and Reference Resolution in PUNDIT", *Proc. of AAAI-86*, 1083-1088. {2}
- Dahl, V. & McCord, M.C.** (1983), "Treating Coordination in Logic Grammar", *Computational Linguistics*, 9(2), 69-91. {2,3}
- Dahl, V.** (1989), "Discontinuous Grammars", *Computational Intelligence*, 5(4), 161-179. {2,3}
- Dahlgren, K.** (1988), *Naive Semantics for Natural Language Understanding*, Kluwer Academic Press. {1}
- Dahlgren, K., McDowell, J. & Stabler, E.P., Jr.** (1989), "Knowledge Representation for Commonsense Reasoning with Text", *Computational Linguistics*, 15(3), 149-170. {2}
- DeJong, G.** (1982a), "An Overview of the FRUMP System", in W.G. Lehnert & M.H. Ringle (eds.), *Strategies for Natural Language Processing*, LEA, 149-176. {5}
- DeJong, G.** (1982b), "Automatic Schema Acquisition in a Natural Language Environment", *Proc. of AAAI-82*, 410-413. {2}
- Delisle, S. & Szpakowicz, S.** (1991), "A Broad-Coverage Parser for Knowledge Acquisition from Technical Texts", *Proc. of the 5th International Conference on Symbolic and Logical Computing — ICEBOL5* (Madison, S.D., USA), April 18-19, 1991, 169-183. {1,3}
- Delisle, S.** (1987), "A Natural Language Interface for an Expert Advisor System", M.C.S. Thesis (also published as TR-86-23), Computer Science Department, University of Ottawa. {2,3}
- Delisle, S.** (1990a), "A Parser for Processing Technical Texts with a Large Coverage of English", TR-90-25, Computer Science Department, University of Ottawa. {1,3}
- Delisle, S.** (1990b), "Arguments Non-Réalisés Syntaxiquement: Une Analyse pour la Compréhension Automatisée du Langage", *Les Cahiers Linguistiques d'Ottawa*, 19, 1-27. {4}
- Delisle, S.** (1991), "Text Processing without A-Priori Domain Knowledge: Semi-Automatic Linguistic Analysis for Incremental Knowledge Acquisition and Conceptual Domain Modelling", Ph.D. Candidacy Paper, Department of Electrical Engineering, University of Ottawa, February 1991. {1,3,4}
- Delisle, S., Barker, K., Copeck, T., & Szpakowicz, S.** (1992), "Interactive Semantic Analysis in TANKA: Designing HAIKU", TR-92-32, Computer Science Department, University of Ottawa. {1,4}
- Delisle, S., Barker, K., Delannoy, J.-F., Matwin, S. & Szpakowicz, S.** (1994), "From Text to Horn Clauses: Combining Linguistic Analysis and Machine Learning", submitted to the *10th Canadian Conference on Artificial Intelligence* (May 1994, Banff, Canada). {6}

- Delisle, S., Copeck, T., Szpakowicz, S. & Barker, K.** (1993), "Pattern Matching for Case Analysis: A Computational Definition of Closeness", in O. Abou-Rabia, C.K. Chang & W.W. Koczkodaj (eds.), *Proc. of the 5th International Conference on Computing and Information — ICCI-93* (Sudbury, Ontario), May 27-28, 1993, 310-315, IEEE Press. {4}
- Delisle, S., Matwin, S., Wang, J. & Zupan, L.** (1991), "Explanation-Based Learning Helps Acquire Knowledge from Natural Language Texts", *Proc. of the 6th International Symposium on Methodologies for Intelligent Systems — ISMIS-91* (Charlotte, N.C., USA), October 16-19, 1991; *Lecture Notes in Artificial Intelligence* #542, Springer-Verlag, 326-337. {3,6}
- DeRose, S.J.** (1988), "Grammatical Category Disambiguation by Statistical Optimization", *Computational Linguistics*, 14(1), 31-39. {2,3}
- Di Eugenio, B.** (1992), "Understanding Natural Language Instructions: The Case of Purpose Clauses", *Proc. of the 30th Annual Meeting of the ACL*, 120-127. {4}
- Diaper, D.** (1989), (ed.), *Knowledge Elicitation: Principles, Techniques and Applications*, Ellis Horwood. {1}
- Dick, J.P.** (1992), "A Conceptual, Case-Relation Representation of Text for Intelligent Retrieval", TR CSRI-265 (Ph.D. Thesis), Computer Systems Research Institute, University of Toronto. {2,4,5}
- Diederich, J.** (1987), "Knowledge-Based Knowledge Elicitation", *Proc. of IJCAI-87*, 201-204. {1}
- Diederich, J., Ruhmann, I. & May, M.** (1988), "KRITON: a Knowledge-Acquisition Tool for Expert Systems", in B.R. Gaines & J.H. Boose (eds.), *Knowledge Acquisition for Knowledge-Based Systems (Knowledge-Based Systems — Volume 2)*, Academic Press, 83-94. {1}
- Dietterich, T.G.** (1989), "Machine Learning", *Annual Review of Computer Science*, 4, 1989-1990, 255-306. {2}
- Dolan, W., Vanderwende, L., & Richardson, S.** (1993), "Automatically Deriving Structured Knowledge Bases from On-line Dictionaries", *Proc. of the 1st Conference of the Pacific Association for Computational Linguistics*, 5-14. {2}
- Dyer, M.G. & Zernik, U.** (1986), "Encoding and Acquiring Meanings for Figurative Phrases", *Proc. of the 24th Annual Meeting of the ACL*, 106-111. {2}
- Ellman, T.** (1989), "Explanation-Based Learning: A Survey of Programs and Perspectives", *ACM Computing Surveys*, 21, 163-221. {S&T}
- Erbach, G.** (1990), "Syntactic Processing of Unknown Words", IWBS Report 131, Wissenschaftliches Zentrum, Institut für Wissensbasierte Systeme. {2}
- Erbach, G.** (1991), "An Environment for Experimentation with Parsing Strategies", *Proc. of IJCAI-91*, 931-936. {3}

- Farreny, H.** (1985), *Les Systèmes Experts (Principes et Exemples)*, Cepadues-Editions. {1}
- Fass, D.** (1989a), "Some Connections Between Knowledge Representation and Natural Language Description", CSS/LCCR TR-89-05, Simon Fraser University. {4}
- Fass, D.** (1989b), "Lexical Semantic Constraints", CSS/LCCR TR-89-11, Simon Fraser University. {2}
- Feng, C., Copeck, T., Szpakowicz, S. & Matwin, S.** (1994), "Semantic Clustering: Acquisition of Partial Ontologies from Public Domain Lexical Sources", to appear in *Proceedings of AAAI Knowledge Acquisition for Knowledge-Based Systems Workshop*, Banff. {3,5}
- Fillmore, C.** (1966), "A Proposal Concerning English Prepositions", F.P. Dincen (ed.), *Report on the 17th Annual Round Table Meeting on Linguistics Language Studies (Monograph Series on Languages and Linguistics 19)*, Georgetown University Press, 19-33. {4}
- Fillmore, C.** (1968), "The Case for Case" In E. Bach and R.T. Harms (eds.), *Universals in Linguistic Theory*, Holt, Rinehart and Winston. {4}
- Fong S. & Berwick, R.C.** (1985), "New Approaches to Parsing Conjunctions Using Prolog", *Proc. of the 23rd Annual Meeting of the ACL*, 118-126. {2}
- Frey, W., Reyle, U. & Rohrer, C.** (1983), "Automatic Construction of a Knowledge Base by Analysing Texts in Natural Language", *Proc. of IJCAI-83*, 727-729. {2}
- Fum, D.** (1985), "Natural Language Processing and the Automatic Acquisition of Knowledge: A Simulative Approach", *Proc. of the 2nd Conference of the European Chapter of the ACL*, 79-88. {2}
- Fum, D., Giovanni, G. & Tasso, C.** (1985), "Evaluating Importance: A Step Towards Text Summarisation", *Proc. of IJCAI-85*, 840-844. {5}
- Gao, Y. & Salveter, S.** (1991), "The Automated Knowledge Engineer: Natural Language Knowledge Acquisition for Expert Systems", *Proc. of the 6th AAAI-Sponsored Knowledge Acquisition for Knowledge-Based Systems Workshop*, 8.1-8.16. {2}
- Gazdar, G. & Mellish, C.** (1989), *Natural Language Processing in PROLOG (An Introduction to Computational Linguistics)*, Addison-Wesley. {S&T, 2}
- Gazdar, G., Franz, A., Osborne, K. & Evans, R.** (1987), *Natural Language Processing in the 1980s*, CSLI Lecture Notes #13. {2}
- Gomez, F.** (1989), "Knowledge Acquisition from Natural Language for Expert Systems Based on Classification Problem-Solving Methods", *Proc. of the 4th AAAI-Sponsored Knowledge Acquisition for Knowledge-Based Systems Workshop*, 15.1-15.18. {2}

- Gomez, F. (1991), "The Acquisition of Word Syntactic Knowledge from Sample Sentences", *Proc. of the 6th AAAI-Sponsored Knowledge Acquisition for Knowledge-Based Systems Workshop*, 9.1-9.17. {2}
- Grefenstette, G. (1992a), "SEXTANT: Exploring Unexplored Contexts for Semantic Extraction from Syntactic Analysis", *Proc. of the 30th Annual Meeting of the ACL*, 324-326. {2}
- Grefenstette, G. (1992b), "Use of Syntactic Context to Produce Term Association Lists for Text Retrieval", *Proc. of SIGIR '92*, ACM, Copenhagen, Denmark, June 21-24, 1992. {2}
- Grimes, J. (1975), *The Thread of Discourse*, Mouton. {4}
- Grishman, R. & Kittredge, R. (1986), (eds.), *Analyzing Language in Restricted Domains: Sublanguage Description and Processing*, LEA. {1}
- Grishman, R. & Sterling, J. (1990), "Information Extraction and Semantic Constraints", *Proc. of COLING-90*, 355-357. {4}
- Grishman, R. & Sterling, J. (1992), "Acquisition of Selectional Patterns", *Proc. of COLING-92*, 658-664. {2}
- Grishman, R. (1986), *Computational Linguistics: An Introduction*, Cambridge University Press. {S&T,2}
- Grishman, R., Hirschman, L. & Friedman, C. (1982), "Natural Language Interfaces Using Limited Semantic Information", *Proc. of COLING-82*, 89-94. {2}
- Grishman, R., Macleod, C. & Sterling, J. (1992), "Evaluating Parsing Strategies Using Standardized Parse Files", *Proc. of 3rd Conf. on Applied Natural Language Processing*, 156-161. {2}
- Grosz, B.J. (1978), "Discourse Knowledge", in D.E. Walker (ed.), *Understanding Spoken Language*, North-Holland, 229-344. {2}
- Grosz, B.J., Appelt, D.E., Martin, P.A. & Pereira, F.C.N. (1987), "TEAM: An Experiment in the Design of Transportable Natural-Language Interfaces", *Artificial Intelligence*, 32, 173-243. {2}
- Grosz, B.J., Sparck Jones, K. & Webber, B.L. (1986), (eds.), *Readings in Natural Language Processing*, Morgan Kaufmann Publishers. {S&T,2}
- Grover, C., Carrol, J., Briscoe, T. (1993), "The Alvey Natural Language Tools Grammar (4th Release)", TR, University of Cambridge Computer Laboratory. {2}
- Gruber, T.R. (1989), *The Acquisition of Strategic Knowledge*, Academic Press. {S&T,1}
- Guha, R.V. & Lenat, D.B. (1990), "Cyc: A Mid-Term Report", *AI Magazine*, 11(3), 32-59. {1}

- Haas, N. & Hendrix, G.G.** (1983), "Learning by Being Told: Acquiring Knowledge for Information Management", in R.S. Michalski, J.G. Carbonell & T.M. Mitchell (eds.), *Machine Learning (An Artificial Intelligence Approach)*, Tioga, 405-427. {2}
- Haddock, N.J.** (1989), "Computational Models of Incremental Semantic Interpretation", *Language and Cognitive Processes*, 4(3/4), SI 337-368. {4}
- Hahn, U. & Reimer, U.** (1986), "TOPIC Essentials", *Proc. of COLING-86*, 497-503. {2}
- Hahn, U.** (1992), "On Text Coherence Parsing", *Proc. of COLING-92*, 25-31. {5}
- Hajicova, E. & Sgall, P.** (1986), "Towards an Automatic Identification of Topic and Focus", *Proc. of the 2nd Conf. of the European Chapter of the ACL*, 263-267. {2,5}
- Hajicova, E. & Sgall, P.** (1993), "Identifying Topic and Focus by an Automatic Procedure", *Proc. of the 6th Conf. of the European Chapter of the ACL*, 178-182. {2,5}
- Hardt, D.** (1992a), "An Algorithm for VP Ellipsis", *Proc. of the 30th Annual Meeting of the ACL*, 9-14. {3}
- Hardt, D.** (1992b), "Some Problematic Cases for VP Ellipsis", *Proc. of the 30th Annual Meeting of the ACL*, 276-278. {3}
- Harris, M.D.** (1985), *Introduction to Natural Language Processing*, Reston. {S&T}
- Hauptmann, A.G.** (1993), "Meaning from Structure in Natural Language Interfaces", Ph.D. Thesis, Computer Science Department, Carnegie-Mellon University. {2}
- Hayes, P.J.** (1985), "The Second Naive Physics Manifesto", in J.R. Hobbs & R.C. Moore (eds.), *Formal Theories of the Commonsense World*, Ablex, 1-36. {1}
- Heidorn, G.E.** (1972), "Natural Language Inputs to a Simulation Programming System", Ph.D. Thesis, Computer Science Department, Yale University. {2}
- Henisz-Dostert, B., Macdonald, R.R. & Zarechnak, M.** (1979), *Machine Translation* (Trends in Linguistics - Studies and Monographs 11), Mouton. {2}
- Herzog, O. & Rollinger, C.-R.** (1991), (eds.), *Text Understanding in LILOG*, Lecture Notes in Artificial Intelligence #546, Springer-Verlag. {2,3}
- Hickman, F.R., Killin, J.L., Land, L., Mulhall, T., Porter, D. & Taylor, R.M.** (1989), *Analysis for Knowledge-Based Systems (A Practical Guide to the KADS Methodology)*, Ellis Horwood. {1}
- Hindle, D. & Rooth, M.** (1991), "Structural Ambiguity and Lexical Relations", *Proc. of the 29th Annual Meeting of the ACL*, 229-236. {2}
- Hirschman, L. & Puder, K.** (1982), "Restriction Grammar in Prolog", *Proc. of the First International Conference on Logic Programming*, 85-90. {3}

- Hirschman, L.** (1986). "Conjunction in Meta-Restriction Grammar", *Journal of Logic Programming*, 4, 299-328. {2}
- Hirst, G.** (1981). Anaphora in Natural Language Understanding: a Survey, Lecture Notes in Computer Science #119, Springer-Verlag. {2,3}
- Hirst, G.** (1987). *Semantic Interpretation and the Resolution of Ambiguity*, Cambridge University Press. {1}
- Hobbs, J.R. & Bear, J.** (1990). "Two Principles of Parse Preference", *Proc. of COLING-90*, 162-167. {2}
- Hobbs, J.R.** (1978). "Resolving Pronoun References", *Lingua* 44, 311-338. {2,3}
- Hobbs, J.R., Appelt, D.E., Bear, J. & Tyson, M.** (1992). "Robust Processing of Real-World Natural-Language Texts", *Proc. of 3rd Conf. on Applied Natural Language Processing*, 186-192. {2}
- Huyck, C.R. & Lytinen, S.L.** (1993). "Efficient Heuristic Natural Language Parsing", *Proc. of AAAI-93*, 386-391. {3}
- Isabelle, P. & Bourbeau, L.** (1985). "TAUM-AVIATION: Its Technical Features and Some Experimental Results", *Computational Linguistics*, 11(1), 18-27. {2}
- Jackendoff, R.** (1983). *Semantics and Cognition*, MIT Press. {1}
- Jacobs, P.S. & Rau, L.F.** (1990). "SCISOR: Extracting Information from On-line News", *CACM*, 33(11), 88-97. {1,2}
- Jacobs, P.S. & Rau, L.F.** (1993). "Innovations in Text Interpretation", *AI Journal*, 63(1-2), Special Issue on Natural Language Processing, 143-191. {3}
- Jacobs, P.S. & Zernik, U.** (1988). "Acquiring Lexical Knowledge from Text: A Case Study", *Proc. of AAAI-88*, 739-744. {2}
- Jacobs, P.S.** (1990). "To Parse or Not to Parse: Relation-Driven Text Skimming", *Proc. of COLING-90*, 194-198. {2}
- Jacobs, P.S.** (1992a). "Joining Statistics with NLP for Text Categorization", *Proc. of 3rd Conf. on Applied Natural Language Processing*, 178-185. {2}
- Jacobs, P.S.** (1992b). "Parsing Run Amok: Relation-Driven Control for Text Analysis", *Proc. of AAAI-92*, 315-321. {2,3}
- Jansen-Winkel, R.M.** (1988). "WASTL: An Approach to Knowledge Acquisition in the Natural Language Domain", *Proc. of the European Knowledge Acquisition Workshop*, 22.1-22.15. {2}
- Jensen, K. & Binot, J.-L.** (1987). "Disambiguating Prepositional Phrase Attachments by Using On-Line Dictionary Definitions", *Computational Linguistics*, 13(3-4), 251-260. {2}

- Jensen, K.** (1989), "A Broad-coverage Natural Language Analysis System", *Proc. of the 1st International Workshop on Parsing Technologies*, 425-441. {2,5}
- Jensen, K., Heidorn, G.E., Miller, L.A. & Ravin, Y.** (1983), "Parse Fitting and Prose Fixing: Getting a Hold on Ill-formedness", *Computational Linguistics*, 9(3-4), 147-160. {3}
- Johnson, M. & Klein, E.** (1986), "Discourse, Anaphora and Parsing", *Proc. of COLING-86*, 669-675. {2}
- Jurafsky, D.** (1992), "An On-Line Computational Model of Human Sentence Interpretation", *Proc. of AAAI-92*, 302-308. {2,3}
- Kamp, H.** (1988), "Discourse Representation Theory: What it is and Where it Ought to Go", in A. Blaser (ed.), *Natural Language at the Computer*, Springer-Verlag, 84-111. {2}
- Kaplan, R.M.** (1989), "Acquiring Knowledge from Text Using Multiple Methodologies to Accomplish Text Understanding", Ph.D. Thesis, Department of Computer Science, Temple University. {2}
- Karp, D., Schabes, Y., Zaidel, M. & Egedi, D.** (1992), "A Freely Available Wide Coverage Morphological Analyzer for English", *Proc. of the 15th International Conference on Computational Linguistics — COLING-92* (Nantes, France), 950-955. {1,3}
- Kay, M.** (1985), "Parsing in Functional Unification Grammar", in D.R. Downty, L. Karttunen, & A.M. Zwicky (eds.), *Natural Language Parsing*, Cambridge University Press, 251-278. {3}
- Kelly, I.D.K.** (1989), *Progress in Machine Translation: Natural Language and Personal Computers*, Sigma Press. {2}
- Kempen, G.** (1991), "Conjunction reduction and gapping in clause-level coordination: an inheritance-based approach", *Computational Intelligence* 7(4), 357-360. {2}
- Kidd, A.L.** (1987), (ed.), *Knowledge Acquisition for Expert Systems (A Practical Handbook)*, Plenum Press. {1}
- Kieras, D.E.** (1985), "Thematic Processes in the Comprehension of Technical Prose", in Britton & Black (eds.), *Understanding Expository Text (A Theoretical and Practical Handbook for Analyzing Explanatory Text)*, LEA, 89-105. {1,2}
- Kim, J.-T. & Moldovan, D.I.** (1993), "Acquisition of Semantic Patterns for Information Extraction from Corpora", *Proc. of the 9th IEEE Conference on AI Applications*, 171-176. {2,6}
- Kimball, J.** (1973), "Seven Principles of Surface Structure Parsing in Natural Language", *Cognition* 2(1), 15-47. {2}
- Kintsh, W. & van Dijk, T.A.** (1978), "Toward a Model of Text Comprehension and Production", *Psychological Review*, 85(5), 363-394. {1}

- Kiyono, M. & Tsujii, J.-i.** (1993). "Linguistic Knowledge Acquisition from Parsing Failures". *Proc. of the 6th Conference of the European Chapter of the ACL*, 222-231. {5}
- Kosy, D.W.** (1986). "Parsing Conjunctions Deterministically". *Proc. of the 24th Annual Meeting of the ACL*, 78-84. {2}
- Kwasny, S.C.** (1980). "Treatment of Ungrammatical and Extra-Grammatical Phenomena in Natural Language Understanding Systems". Ph.D. Thesis, Department of Computer Science, Indiana University. {3}
- Landau, M.-C.** (1990). "Solving Ambiguities in the Semantic Representation of Texts". *Proc. of COLING-90*, 239-244. {2,3}
- Lang, F.-M. & Hirschman, L.** (1988). "Improved Portability and Parsing Through Interactive Acquisition of Semantic Information". *Proc. of the 2nd Conference on Applied Natural Language Processing (ACL)*, 49-57. {1,2}
- Langley, P. & Carbonell, J.G.** (1986). "Language Acquisition and Machine Learning", TR 86-12, University of California, Irvine. {2}
- Lappin, S. & McCord M.** (1990). "Anaphora Resolution in Slot Grammar". *Computational Linguistics*, 16(4), 197-212. {2}
- Larson, M.** (1984), *Meaning-Based Translation: A Guide to Cross-language Equivalence*, University Press of America. {4}
- Lavelli, A. & Stock, O.** (1990). "When Something Is Missing: Ellipsis, Coordination and the Chart", *Proc. of COLING-90*, 184-189. {2}
- LeBlanc, D., Davis, H. & Rosenberg, R.** (1993), "Parsing with Principles", *Proc. of the 1st Conference of the Pacific Association for Computational Linguistics*, 212-221. {3}
- Lehnert, W.G.** (1982), "Plot Units: A Narrative Summarization Strategy", in W.G. Lehnert & M.H. Ringle (eds.), *Strategies for Natural Language Processing*, LEA, 375-412. {5}
- Lenat, D. & Guha, R.** (1990), *Building Large Knowledge-Based Systems (Representation and Inference in the Cyc Project)*, Addison-Wesley. {1}
- Lenat, D., Prakash, M. & Shepherd, M.** (1986), "CYC: Using Common Sense Knowledge to Overcome Brittleness and Knowledge Acquisition Bottlenecks", *AI Magazine*, 6(4), 65-85. {1}
- Lesmo, L. & Torasso, P.** (1985), "Analysis of Conjunctions in a Rule-Based Parser", *Proc. of the 23rd Annual Meeting of the ACL*, 180-187. {2}
- Li, W.-C., Pei, T., Lee, B.-H. & Chiou, C.-F.** (1990), "Parsing Long English Sentences with Pattern Rules", *Proc. of COLING-90*, 410-412. {2}

- Linebarger, M.C., Dahl, D.A., Hirschman, L. & Passonneau, R.J.** (1988). "Sentence Fragments Regular Structures", *Proc. of the 26th Annual Meeting of the ACL*, 7-16. {2}
- Liu, R.-L. & Soo, V.-W.** (1993), "An Empirical Study on Thematic Knowledge Acquisition Based on Syntactic Clues and Heuristics", *Proc. of the 31st Annual Meeting of the ACL*, 243-250. {2}
- Loewen, V.** (1988), "ORDINOTRAD: A Machine Translation System based on Case Grammar", M.A. Thesis, School of Translators and Interpreters, University of Ottawa. {4}
- Longacre, R.E.** (1983), *The Grammar of Discourse*, Plenum Press. {4}
- Longman** (1987), *Longman Dictionary of Contemporary English*, 2nd edition, Longman. {2}
- Magerman, D.M. & Marcus, M.P.** (1991), "Pearl: A Probabilistic Chart Parser", *Proc. of the 5th Conference of the European Chapter of the ACL*, 15-20. {1}
- Manning, C.D.** (1993), "Automatic Acquisition of a Large Subcategorization Dictionary from Corpora", *Proc. of the 31st Annual Meeting of the ACL*, 235-242. {2}
- Marcus, M.** (1980), *A Theory of Syntactic Recognition for Natural Language*, MIT Press. {2}
- Matwin, S. & Szpakowicz, S.** (1992), Machine Learning Techniques in Knowledge Acquisition from Text, *THINK*, 1(2), 37-50. {3,6}
- Mauldin, M.L.** (1991), "Retrieval Performance in FERRET (A Conceptual Information Retrieval System)", *Proc. of the 14th International Conference on Research and Development in Information Retrieval*. {2}
- McCarthy, J. & Hayes, P.J.** (1969), "Some Philosophical Problems from the Standpoint of Artificial Intelligence", in B. Meltzer & D. Michie (eds.), *Machine Intelligence 4*, Edinburgh University Press, 463-502. {1}
- McCawley, J.D.** (1988), *The Syntactic Phenomena of English*, Volumes 1 and 2, The University of Chicago Press. {3}
- McCord, M.C.** (1982), "Using Slots and Modifiers in Logic Grammars for Natural Language", *Artificial Intelligence*, 18, 327-367. {3}
- McCord, M.C.** (1985), "Modular Logic Grammars", *Proc. of the 23rd Annual Meeting of the ACL*, 104-117. {3}
- McDonald, D.D.** (1992), "An Efficient Chart-based Algorithm for Partial-Parsing of Unrestricted Texts", *Proc. of 3rd Conf. on Applied Natural Language Processing*, 193-200. {2}
- McRoy, S.W.** (1992), "Using Multiple Knowledge Sources for Word Sense Discrimination", *Computational Linguistics*, 18(1), 1-30. {3}

- Mellish, C.S.** (1989). "Some Chart-Based Techniques for Parsing Ill-Formed Input", *Proc. of the 27th Annual Meeting of the ACL*, 102-109. {3}
- Merlo, P.** (1993). "For an Incremental Computation of Intrasentential Coreference", *Proc. of IJCAI-93*, 1216-1221. {3}
- Michalski, R.S., Carbonell, J.G. & Mitchell, T.M.** (1983), *Machine Learning (An Artificial Intelligence Approach)*, Tioga. {S&T}
- Miller, G.A.** (1990). (eds.). "WordNet: An On-Line Lexical Database", *International Journal of Lexicography*, 3(4). {1,3}
- Minsky, M. & Paper, S.** (1988). *Perceptrons (An Introduction to Computational Geometry)*, expanded edition, MIT Press. {1}
- Moller, J.-U.** (1988). "Knowledge Acquisition from Texts", *Proc. of the European Knowledge Acquisition Workshop*, 25.1-26.16. {2}
- Montemagni, S. & Vanderwende, L.** (1992). "Structural Patterns vs. String Patterns for Extracting Semantic Information from Dictionaries", *Proc. of COLING-92*, 546-552. {2}
- Mooney, R.J.** (1990), *A General Explanation-Based Learning Mechanism and its Application to Narrative Understanding*, Morgan Kaufmann Publishers. {2}
- Moulin, B. & Rousseau, D.** (1990). "A Knowledge Acquisition System for Analysing Prescriptive Texts", *Proc. of the 5th AAAI-Sponsored Knowledge Acquisition for Knowledge-Based Systems Workshop*, 22.1-22.20. {2,6}
- Moulin, B. & Rousseau, D.** (1991). "Extracting Logical Knowledge from Prescriptive Texts in Order to Build Deontic Knowledge Bases", *Proc. of the 6th AAAI-Sponsored Knowledge Acquisition for Knowledge-Based Systems Workshop*, 17.1-17.20. {2}
- Moulin, B., Rousseau, D. & Vanderveken, D.** (1992). "Speech Acts in a Connected Discourse: A Computational Representation Based on Conceptual Graph Theory", *Journal of Experimental and Theoretical Artificial Intelligence*, 4(1992), 149-165. {6}
- Munch, K.H.** (1986). "Domain Dependent Natural Language Understanding", *Proc. of COLING-86*, 260-262. {2}
- Nirenburg, S. & Raskin, V.** (1987). "The Subworld Concept Lexicon and the Lexicon Management System", *Computational Linguistics*, 13(3-4), 276-289. {1,5}
- Nirenburg, S., Monarch, I., Kaufmann, T., Nirenburg, I. & Carbonell, J.** (1988). "Acquisition of Very Large Knowledge Bases: Methodology, Tools and Applications", *Proc. of the 3rd AAAI-Sponsored Knowledge Acquisition for Knowledge-Based Systems Workshop*, 24.1-24.26. {1,2}
- Nishida, F., Takamatsu, S., Tani, T. & Kusaka, H.** (1986). "Text Analysis and Knowledge Extraction", *Proc. of COLING-86*, 241-243. {2,5}

- Niv, M. (1992), "Right Association Revisited", *Proc. of the 30th Annual Meeting of the ACL*, 285-287. {2}
- Novak, G.S.Jr. & Bulko, W.C. (1990), "Understanding Natural Language with Diagrams", *Proc. of AAAI-90*, 465-470. {2}
- O'Shaughnessy, D.D. (1989), "Parsing with a Small Dictionary for Applications such as Text to Speech", *Computational Linguistics*, 15(2), 97-108. {2}
- Paquet, C. (1993), "Analyse Syntaxique avec le logiciel DIPETT", Rapport final de Projet de Synthèse — 28 juillet 1993, Département de Mathématiques et d'Informatique, Université du Québec à Trois-Rivières, 270p. {3, Appendix C}
- Paquin, L.-C. & Dupuy, L. (1990), "Analyse de texte et acquisition des connaissances: aspects méthodologiques", *ICO Québec*, 95-113. {2}
- Passonneau, R.J. (1989), "Getting at Discourse Referents", *Proc. of the 27th Annual Meeting of the ACL*, 51-59. {2}
- Pereira, F.C.N. & Shieber, S.M. (1987), *Prolog and Natural-Language Analysis*, CSLI Lecture Notes #10. {3}
- Pereira, F.C.N. & Warren, D.H.D. (1980), "Definite Clause Grammars for Language Analysis — A Survey of the Formalism and a Comparison with Augmented Transition Networks", *Artificial Intelligence*, 13, 231-278. {3}
- Pereira, F.C.N. (1981), "Extraposition Grammars", *American Journal for Computational Linguistics*, 7(4), 243-256. {2,3}
- Pollard, C. & Sag, I.A. (1987), *Information-Based Syntax and Semantics (Volume 1 — Fundamentals)*, CSLI Lecture Notes #13. {3}
- Pustejovsky, J. & Bergler, S. (1987), "The Acquisition of Conceptual Structures for the Lexicon", *Proc. of AAAI-87*, 566-570. {2}
- Pustejovsky, J. (1991), "The Generative Lexicon", *Computational Linguistics*, 17(4), 409-441. {2}
- QP-3.1 (1991), *Language and Library*, Quintus Prolog Manual (release 3.1), Quintus Corporation. {4}
- Quénelle, G. & Hourquin, D. (1987), *6000 Verbes Anglais et leurs Composés, Formes et Emplois*, Collection Bescherelle, Editions Hurtubise. {3}
- Quirk, R., Greenbaum, S., Leech, G. & Svartvik, J. (1985). *A Comprehensive Grammar of the English Language*, Longman. {3,4,5}
- Radford, A. (1981), *Transformational Grammar (A First Course)*, Cambridge University Press. {3}

- Regoczei, S. & Hirst, G.** (1989a), "On 'Extracting Knowledge from Text': Modelling the Architecture of Language Users", TR CSRI-225, Computer Systems Research Institute, University of Toronto. {1,2}
- Regoczei, S. & Hirst, G.** (1989b), "Sortal Analysis with SORTAL, a Software Assistant for Knowledge Acquisition", *Proc. of the 4thAAAI-Sponsored Knowledge Acquisition for Knowledge-Based Systems Workshop*, 25.1-25.18. {1,2}
- Regoczei, S. & Plantinga, E.P.O.** (1988), "Creating the Domain of Discourse: Ontology and Inventory", in B.R. Gaines & J.H. Boose (eds.), *Knowledge Acquisition for Knowledge-Based Systems (Knowledge-Based Systems — Volume 2)*, Academic Press, 293-308. {1}
- Reimer, U.** (1990), "Automatic Knowledge Acquisition from Texts: Learning Terminological Knowledge via Text Understanding and Inductive Generalization", *Proc. of the 5thAAAI-Sponsored Knowledge Acquisition for Knowledge-Based Systems Workshop*, 27.1-27.16. {2}
- Riesbeck, C.K.** (1982), "Realistic Language Comprehension", in W.G. Lehnert & M.H. Ringle (eds.), *Strategies for Natural Language Processing*, LEA, 37-54. {4}
- Riesbeck, C.K.** (1983), "Some problems for Conceptual Analysers", in K. Sparck Jones & Y. Wilks (eds.), *Automatic Natural Language Parsing*, Ellis Horwood, 178-181.
- Rieter, E. & Dale, R.** (1992), "A Fast Algorithm for the Generation of Referring Expressions", *Proc. of COLING-92*, 232-238. {2}
- Rinaldo, F.J.** (1989), "Deriving Rules for Medical Expert Systems Using Natural Language Parsing and Discourse Analysis", Ph.D. Thesis, Department of Computer Science, Illinois Institute of Technology. {2}
- Robinson, J.** (1982), "DIAGRAM: A Grammar for Dialogues", *CACM*, 25(1), 27-47. {3}
- Rolbert, M.** (1990), "Des Heuristiques Pour la Recherche du Thème d'un Discours et de l'Antécédent d'un Pronom", *Proc. of COLING-90*, 28-33. {2}
- Rousselot, F., Migault, B. & Igot, P.** (1992), "Elaboration de Techniques d'Analyse Adaptées à la Construction d'une Base de Connaissances", *Proc. of COLING-92*, 483-489. {2}
- Sager, N.** (1981), *Natural Language Information Processing: A Computer Grammar of English and its Applications*, Addison-Wesley. {2}
- Saint-Dizier, P.** (1988), "Contextual Discontinuous Grammars", in V. Dahl & P. Saint-Dizier (eds.), *Natural Language Understanding and Logic Programming II*, North-Holland. {3}
- Salton, G.** (1989), *Automatic Text Processing (The Transformation, Analysis, and Retrieval of Information by Computer)*, Addison-Wesley. {2}

- Schank, R.C. & Riesbeck, C.K. (1981), *Inside Computer Understanding*, LEA. {4}
- Schank, R.C. (1975), *Conceptual Information Processing*, North Holland. {4}
- Schmalhofer, F., Kuhn, O. & Schmidt, G. (1990), "Integrated Knowledge Acquisition from Text, Previously Solved Cases, and Expert Memories", *Proc. of the 5th AAAI-Sponsored Knowledge Acquisition for Knowledge-Based Systems Workshop*, 29.1-29.20. {1}
- Schubert, L.K. (1984), "On Parsing Preferences", *Proc. of COLING-84*, 247-250. {2}
- Sells, P. (1985), *Lectures on Contemporary Syntactic Theories (An Introduction to Government-Binding Theory, Generalized Phrase Structure Grammar, and Lexical-Functional Grammar)*, CSLI Lecture Notes #3. {3}
- Sergot, M. (1983), "A Query-the-User Facility for Logic Programming", in P. Degano & E. Sandewall (eds.), *Integrated Interactive Computing Systems (Proceedings of the European Conference on Integrated Interactive Computing Systems)*, North-Holland Publishing Company. {4}
- Sgall, P., Hajicova, E. & Panevova, J. (1986), *The Meaning of the Sentence in its Semantic and Pragmatic Aspects*, Reidel. {2}
- Shapiro, S.C. & Neal, J.G. (1982), "A Knowledge Engineering Approach to Natural Language Understanding", *Proc. of the 20th Annual Meeting of the ACL*, 136-144. {2}
- Shieber, S.M. (1986), *An Introduction to Unification-Based Approaches to Grammar*, CSLI Lecture Notes #4. {3}
- Silvestro, K. (1988), "Using Explanations for Knowledge-Based Acquisition", *International Journal of Man-Machine Studies*, 29, 159-169. {2}
- Simmons, R.F. & Y.-H. Yu (1992), "The Acquisition and Use of Context-Dependent Grammars for English", *Computational Linguistics*, 18 (4), 391-418. {2}
- Slator, B.M. (1989), "Extracting Lexical Knowledge from Dictionary Text", *SIGART*, 108, 173-174. {2}
- Smadja, F. & McKeown, K. (1991), "Using Collocations for Language Generation", *Computational Intelligence* 7(4), 229-239. {2}
- Smadja, F.A. (1991), "From N-Grams to Collocations: An Evaluation of XTRACT", *Proc. of the 29th Annual Meeting of the ACL*, 279-284. {2,6}
- Somers, H.L. (1987), *Valency and Case in Computational Linguistics*, Edinburgh University Press. {4,5}
- Sowa, J. (1984), *Conceptual Structures: Information Processing in Mind and Machine*, Addison-Wesley, Reading, MA. {1,2}
- Sparck Jones, K. & Boguraev, B.K. (1987), "A Note on the Study of Cases", *Computational Linguistics* 13(1-2), 65-68. {4}

- Stabler, E.P., Jr.** (1986), "Restricting Logic Grammars", *Proc. of AAAI-86*, 1048-1052. {3}
- Stabler, E.P., Jr.** (1987), "Restricting Logic Grammars with Government-Binding Theory", *Computational Linguistics*, 13(1-2), 1-10. {3}
- Steiner, E.H., Schmidt, P. & Zelinsky-Wibbelt, C.** (1988), *From Syntax to Semantics (Insights from Machine Translation)*, Pinter. {2}
- Stock, O.** (1987), "Getting Idioms Into a Lexicon Based Parser's Head", *Proc. of the 25th Annual Meeting of the ACL*, 52-58. {2}
- Stock, O.** (1989), "Parsing with Flexibility, Dynamic Strategies, and Idioms in Mind", *Computational Linguistics*, 1-18. {2}
- Stockwell, R.P., Schachter, P. & Partee, B.H.** (1973), *The Major Syntactic Structures of English*, Holt, Rinehart and Winston. {3}
- Strzalkowski, T.** (1992), "TTP: A Fast and Robust Parser for Natural Language", *Proc. of COLING-92*, 198-204. {3}
- Szpakowicz, S. & Koperczak, Z.** (1990), "Mixed-Strategy Matching in Conceptual Networks", *Methodologies for Intelligent Systems 5 (Proc. of the 5th International Symposium on Methodologies for Intelligent Systems)*, 321-328. {1}
- Szpakowicz, S.** (1990), "Semi-Automatic Acquisition of Conceptual Structure from Technical Texts", *International Journal of Man-Machine Studies*, 33, 385-397. {1}
- Tanenhaus, M.K., Carlson, G. & Trueswell, J.C.** (1989), "The Role of Thematic Structures in Interpretation and Parsing", *Language and Cognitive Processes*, 4(3-4), 211-234. {1}
- Tesnière, L.** (1959), *Eléments de Syntaxe Structurale*, Librairie C. Klincksieck (Paris). {4}
- Tomita, M.** (1984), "Disambiguating Grammatically Ambiguous Sentences by Asking", *Proc. of COLING-84*, 476-480. {2}
- Tomita, M.** (1986), *Efficient Parsing for Natural Language (A Fast Algorithm for Practical Systems)*, Kluwer Academic Publishers. {3}
- Trost, H., Buchberger, E. & Heinz, W.** (1988), "On the Interaction of Syntax and Semantics in a Syntactically Guided CaseFrame Parser", *Proc. of COLING-88*, 677-682. {5}
- Tucker, A.B., Nirenburg, S. & Raskin, V.** (1986), "Discourse and Cohesion in Expository Text", *Proc. of COLING-86*, 181-183. {2}
- van der Linden, E.-J.** (1992), "Idioms, Non-literal Language and Knowledge Representation", *Computational Intelligence*, 8(3), 433-455. {2}

- Vanderveken, D.** (1990), *Meaning and Speech Acts — Volume I: Principles of Language Use*, Cambridge University Press. {6}
- Vanderveken, D.** (1991), *Meaning and Speech Acts — Volume II: Formal Semantics of Success and Satisfaction*, Cambridge University Press. {6}
- Velardi, P., Pazienza, M.T. & Fasolo, M.** (1991), "How to Encode Semantic Knowledge: A Method for Meaning Representation and Computer-Aided Acquisition", *Computational Linguistics*, 17(2), 153-170. {2}
- Velardi, P., Pazienza, M.T. & Magrini, S.** (1989), "Acquisition of Semantic Patterns From a Natural Corpus of Texts", *SIGART*, 108, 115-123. {2}
- Voss, J.F.** (1984), "On Learning and Learning from Text", in H. Mandl, N.L. Stein, & T. Trabasso (eds.), *Learning and Comprehension of Text*, LEA, 193-212. {1}
- Walker, M.A.** (1989), "Evaluating Discourse Processing Algorithms", *Proc. of the 27th Annual Meeting of the ACL*, 251-261. {2}
- Wallace, M.** (1984), *Communicating with Databases in Natural Language*, Ellis Horwood. {2}
- Wegnez, L.F.** (1987), *À la Recherche de l'Intelligence (du Cerveau Humain au Cerveau Electronique vers l'Intelligence Artificielle)*, Office International de Librairie. {1}
- Wehrli, E.** (1988), "Parsing with a GB-Grammar", in U. Reyle & C. Rohrer (eds.), *Natural Language Parsing and Linguistic Theories*, Reidel, 177-201. {3}
- Weischedel, R., Schwartz, R., Palmucci, J., Meteer, M. & Ramshaw, L.** (1993), "Coping with Ambiguity and Unknown Words through Probabilistic Models", *Computational Linguistics*, 19(2), 359-382. {2}
- Weizenbaum, J.** (1967), "Contextual Understanding by Computers", *CACM*, 10 (8), 474-480. {2}
- Weizenbaum, J.** (1976), *Computer Power and Human Reason*, Freeman. {2}
- Whittemore, G., Ferrara, K. & Brunner, H.** (1990), "Empirical Study of Predictive Powers of Simple Attachment Schemes for Post-modifier Prepositional Phrases", *Proc. of the 28th Annual Meeting of the ACL*, 23-30. {2}
- Whittemore, G., Macpherson, M. & Carlson, G.** (1991), "Event-building through Role-filling and Anaphora Resolution", *Proc. of the 29th Annual Meeting of the ACL*, 17-24. {2}
- Wilks, Y.** (1973), "An Artificial Intelligence Approach to Machine Translation", in R.C. Schank & K.M. Colby (eds.), *Computer Models of Thought and Language*, Freeman, 114-151. {2}
- Wilks, Y., Huang, X. & Fass, D.** (1985), "Syntax, Preference, and Right Attachment", *Proc. of IJCAI-85*, 779-784. {2}
- Winograd, T.** (1972), *Understanding Natural Language*, Academic Press. {2,3}

- Winograd, T.** (1983). *Language as a Cognitive Process (Syntax)*. Addison-Wesley. {S&T,1,2,3,4}
- Winograd, T.** (1987). A Language/Action Perspective on the Design of Cooperative Work, Report CSLI-87-98, Center for the Study of Language and Information, Stanford University. {6}
- Woods, W.A.** (1970). "Transition Network Grammars for Natural Language Analysis", *Communications of the ACM*, 13, 591-606. {3}
- Woods, W.A.** (1973). "An Experimental Parsing System for Transition Network Grammars", in R. Rustin (ed.), *Natural Language Processing*, Algorithmic Press. {2}
- Wu, D.** (1993). "An Image-Schematic System of Thematic Roles", *Proc. of the 1st Conference of the Pacific Association for Computational Linguistics*, 323-332. {4}
- Yang, L. & Szpakowicz, S.** (1990a). "A Knowledge Representation Formalism: Conceptual Network", TR-90-26, Department of Computer Science, University of Ottawa. {1}
- Yang, L. & Szpakowicz, S.** (1990b). "Reasoning in Conceptual Networks", TR-90-27, Department of Computer Science, University of Ottawa. {1}
- Yang, L. & Szpakowicz, S.** (1991a). "Inheritance in Conceptual Networks", *Proc. of the Sixth International Symposium on Methodologies for Intelligent Systems*, Charlotte, NC, 191-202. {1}
- Yang, L. & Szpakowicz, S.** (1991b). "Planning in Conceptual Networks", in F. Dehne, F. Fiala and W.W. Koczkodaj (eds.), *Advances in Computing and Information - ICCI-91. Lecture Notes in Computer Science*, vol. 497, Springer-Verlag, 669-671. {1}
- Yang, L.** (1993). "Further Discussion of CN", unpublished manuscript (July 1993), Department of Computer Science, University of Ottawa, {1, Appendix 1}.
- Zadrozny, W. & Jensen, K.** (1991). "Semantics of Paragraphs", *Computational Linguistics*, 17(2), 171-209. {5}
- Zadrozny, W.** (1992). "On Compositional Semantics", *Proc. of COLING-92*, 260-266. {1}
- Zernik, U. & Jacobs, P.S.** (1990). "Tagging for Learning: Collecting Thematic Relations from Corpus", *Proc. of COLING-90*, 34-39. {2}
- Zernik, U.** (1987). "Learning Idioms — With and Without Explanations", *Proc. of IJCAI-87*, 133-136. {2}
- Zernik, U.** (1989). "Lexicon Acquisition: Learning from Corpus by Capitalizing on Lexical Categories", *Proc. of IJCAI-89*, 1556-1562. {2}
- Zernik, U.** (1992). "Shipping Department vs. Shipping Pacemakers: Using Thematic Analysis to Improve Tagging Accuracy", *Proc. of AAAI-92*, 335-342. {2}

Zukerman, I. (1991), "Using Meta-comments to Generate Fluent Text in a Technical Domain", *Computational Intelligence* 7(4), 276-295. {4}

APPENDICES

Appendix A: DIPETT's Driver and User Interface

This appendix shows an abridged, actual log (set in the Courier typeface) of DIPETT's driver in action — details pertaining to the semantic analyzer, HAIKU, are presented in chapter 4. The following log illustrates the user interface dialogue and gives a good idea of the highest level algorithms involved. Explanations were added in the Times typeface, in bold, and enclosed in braces. As well, the original log was slightly reorganized to improve the presentation.

```
Driver ..... version: 30-Jul-93
DIPETT PARSE ..... version: 30-Jul-93
HAIKU SEMANTIC ANALYSER ... version: 20-Jul-93
Copyright(c) 1989-93 S.Delisle, UofO Knowledge Acquisition Lab & UQTR
```

NOTE: default options in the following menus appear in CAPITALS
and can be conveniently selected with CR (carriage return).

```
USE DIPETT AS BATCH PARSE OR BATCH TAGGER (b), OR
  PROCEED WITH DETAILED MENU (d), OR
    JUMP TO MAIN MENU RIGHT AWAY (j) [b/d/J]?
{above: this first menu offers a batch mode ('b' option), either for parsing and/or
tagging, a detailed submenu for parsing ('d' option), which is especially
appropriate for the implementer, and a "jump start" option ('j' option) which uses
default parameters to facilitate the end user's task.}
```

The rest of the log will be organized into three parts, one for each of the three options.

A.1: The 'j' ("jump start") Option

A-1) Log filename (n for none)
[n/"name"/zzz.DIPETT-HAIKU.script]?

{above: the user can ask DIPETT to keep a log of the session in an ASCII file in which everything appearing on the screen will be saved.}

A-2) Parse trees filename (n for none)
[n/"name"/zzz.DIPETT.parse_trees.pl]?

{above: the user can ask DIPETT to collect every parse tree, under its Prolog form, in a (Prolog-readable) file. This file can subsequently be processed directly by the semantic analyzer — see option '9' in the main menu, later. This is particularly useful when using the batch parsing mode.}

B) Dictionary additions filename
["name"/zzz.DIPETT-dict_add.pl]?

!! This file already exists ... OVERWRITE IT [y/N]?

{above: all new lexical entries, i.e. words not found in DIPETT's lexicon, are recorded in this file so that the lexicon can easily be updated afterwards.}

{below: in this 'j' ("jump start") option, the current HAIKU KB file is automatically loaded for consultation and updating. If there is no such file, the initial KB is empty. More on this in chapter 4.}

```
% compiling file
  /@/soleil/pl/dmatensr/delisle/DIPETT/HAIKU_ikbDICT.PL
% HAIKU_ikbDICT.PL compiled in module user, 0.233 sec 11,132 bytes
```

{below: the system always keeps track of the date and time at which a session took place. This information also appears in the output files whose names we saw above.}

```
**=> BEGINNING NEW PARSING SESSION: [date&time: 9/6/93 9:10:36] <=**
```

{below: the system automatically keeps a backup (copy) of the initial HAIKU KB file, in case errors or unfortunate updates should occur during this new session.}

```
> making a backup copy of HAIKU_ikbDICT.PL ...
done! backup file is HAIKU_ikbDICT.BACKUP
```

{below: this is what will be referred to as the "main menu". It presents the user with every operation handled by the driver.}

You may:

0. Change session parameters
1. Input one sentence
2. Read sentences from an existing file
3. Same as #2, but with possible reruns
4. Consult Collins, WordNet, or DIPETT's lexicon
5. Consult or update HAIKU's dictionaries (KB)
6. Reset parsing session counters and stats
7. Display current parsing statistics
8. Set the "Unit #" counter to a specific value
9. Apply HAIKU on parse trees from an existing file
- e. EXIT (end of session)

{above:

- (0) to reconsider all the details of the session parameters, as in option 'd' above — see below.
- (1) to interactively enter one sentence to be analyzed on-line.
- (2) to interactively process sentences from an existing text file to be analyzed on-line.
- (3) same as previous option but with the possibility of reformulating and resubmitting the input sentence if it is rejected.
- (4) simple interface to allow on-line consultation of the Collins dictionary, or the WordNet lexical database, or DIPETT's own lexicon. The version of the latter which is currently in memory may also be updated *via* this option.
- (5) simple interface to dynamically consult or update the current HAIKU KB.
- (6) to reset all counters from which simple statistics are computed for the current session: see below for the list of these statistics.
- (7) to display the current statistics.
- (8) to set the counter which numbers the units (sentences) being processed to a specific value: this is especially handy when sentences in a large corpus are numbered and we want to be able to refer to them individually and uniquely.
- (9) to interactively apply the semantic analyzer to a collection of parse trees accumulated in a file.
- (e) to terminate the current session with the system.}

What do you wish to do [0 @ 9, or e] ...

{above: etc. Here an actual session begins.}

{below: upon selecting the 'e' (exit) option in the main menu, the following statistics are produced, expressed in terms of absolute numbers and percentages. The items are fairly self-explanatory except for (0) which is the default cpu time limit (timing-out mechanism) for the session just completed — see section 3.4.2. We will explain in 3.3.4 and 3.4.4 what fragmentary parsing is.}

```
END OF PARSING SESSION STATISTICS: [date&time: 9/6/93 9:10:41]
0- cpu time limit: 2.0 minute(s).
1- Total nb. of units processed: ...
2- Total nb. of units parsed as fragments: ...
3- Total nb. of units not parsed: ...
4- Total nb. of units skipped: ...
5- Total nb. of units fully parsed: ...
6- Percent of successful parses, full or frag. (excluding #4): ...
7- Total nb. of dictionary additions: ...
8- Total cpu time (parsing & other tasks): ...
9- Average cpu time per input processed: ...
```

```
Dictionary additions (if any) have been copied in file:
zzz.DIPETT-dict_add.pl
```

```
> Save current mDict, cmpDict, & cpDict in a file [n/Y]? n
```

{above: the user can save the current HAIKU KB in a file. This KB represents the initial KB updated with the processing of the text, during the session just completed.}

```
> Updating the journal of KB modifications ...
Please enter your name (signature): Sylvain
absolute path is: /@/soleil/pl/dmatensr/delisle/DIPETT
running 'diff' (new KB vs backup KB) ...
```

{above: the system keeps a journal of KB updates so that we know which elements of the KB were modified and who is responsible of those changes. This would allow us to reconstruct the KB from a given valid state if it were discovered that the most recent updates lead to an incorrect KB, for whatever reasons.}

A.2: The 'd' ("detailed menu") Option

```
USE DIPETT AS BATCH PARSER OR BATCH TAGGER (b), OR
PROCEED WITH DETAILED MENU (d), OR
JUMP TO MAIN MENU RIGHT AWAY (j) [b/d/J]? d
```

{below: 'A-1', 'A-2', and 'B' are as above in the 'j' option.}

```
A-1) Log filename (n for none)
     [n/"name"/zzz.DIPETT-HAIKU.script]?
```

```
A-2) Parse trees filename (n for none)
     [n/"name"/zzz.DIPETT.parse_trees.pl]?
```

```
B) Dictionary additions filename
   ["name"/zzz.DIPETT-dict_add.pl]?
```

```
C-1) Do you wish to DEactivate the scanner's QUIZ bias [y/N]?
{above: our main sample text, for testing purposes, was the QUIZ manual already
mentioned in chapter 1. Since it contains numerous specific lexical conventions,
such as QUIZ reserved words, the user can deactivate the scanner's special rules
for QUIZ in order to keep it domain independent.}
```

```
C-2) Do you wish to (exclusively) activate the tagger [y/N]?
{above: the tagger — see section 3.3.5 — can be activated here for on-line use. See
below for batch activation.}
```

{below: optionally, the user can select a number of options for parsing and/or testing purposes.}

```
D) Do you wish to set one or more parsing options [y/N]? y
   print the scanner's output [y/N]?
{above: optionally, the scanner's output can be displayed.}
   print the lexicallyser's output [y/N]?
{above: optionally, the lexical analyzer's output can be displayed.}
   DEactivate the parser [y/N]?
{above: optionally, the parser can be deactivated. In particular, it is useful for
testing the scanner and the lexical analyzer, or building a lexicon for future
parsing.}
   set a cpu time limit [y/N]?
{above: the user can set an upper limit to the amount of cpu time the parser should
spend on any given input. See 3.4.2.}
   parse in "silent mode" [y/N]?
{above: normally, the parser emits two "beeps" when it completes the analysis,
successful or not, of an input. Optionally, the user can turn off this beeping.}
   Full, Prolog, or Abridged parse trees [f/p/A]?
{above: parse trees produced by DIPETT can be displayed in three formats. 'Full'
means that the pretty-printed tree will show all possible slots, even those that are
empty. 'Prolog' means that the tree shown to the user will be the full tree, as in the
'Full' format, but under its native Prolog form, i.e. un-pretty printed. 'Abridged'
means that the pretty printer will show only the non-empty slots of the parse tree,
thus minimizing the size of the tree.}
   include phrase tokens in the parse trees [n/Y]?
```

{above: optionally, the user can ask DIPETT to associate with every fundamental syntactic unit — essentially, noun phrases, prepositional phrases, verb phrases — the corresponding list of input tokens that were found to belong to this unit.}

set parsing flags [y/N]? y

{above: optionally, the user can activate grammar rule spies that will show when certain rules are activated during the parsing of an input. See 3.3.5.}

{below: this is the list of available grammar rule spies that may be activated.}

list available "spies" [y/N]? y

[sentence, sentence_core, rest_sentence, verbphrase, verbclause,
adj_phrase, complement, subordinate_clause, np_equiv,
nounphrase, nom_cl, np_postmodifiers, prepphrases]

list currently active "spies" [y/N]?

deactivate currently active "spies": Yes, All, or No [y/a/N]?

activate currently inactive "spies" [y/N]?

E) Do you wish to activate the HELPER facility [y/N]?

{above: optionally, the user can activate the *helper* mechanism. See 3.3.5.}

F) Do you wish to activate the MEMORIZER facility [n/Y]?

{above: optionally, the user can activate the memorizer mechanism. See 3.3.5.}

G) Do you wish to activate the partial parsing facility ...

No, eXclusively (full parsing is "off"),

or as "last resort" [n,x,Y]?

{above: the fragmentary parsing facility — see 3.3.4 and 3.4.4 — can be turned off, activated exclusively, or used only as a last resort strategy after having unsuccessfully attempted full parsing.}

H) Do you wish to activate the HAIKU semantic analyzer [n/Y]?

Load HAIKU's KB (i.e. HAIKU_ikbDICT.PL) [n/Y]?

{above: the HAIKU semantic analyzer can be activated (or not), and if so, the user may load the current HAIKU KB file.}

A.3: The 'b' ("batch") Option

```
USE DIPETT AS BATCH PARSER OR BATCH TAGGER (b), OR
PROCEED WITH DETAILED MENU (d), OR
JUMP TO MAIN MENU RIGHT AWAY (j) [b/d/J]? b
```

```
A-1) Log filename (n for none)
    [n/"name"/zzz.DIPETT-HAIKU.script]?
```

```
B) Dictionary additions filename
    ["name"/zzz.DIPETT-dict_add.pl]?
```

{below: the batch mode can be used for parsing and/or tagging. We now present the 'b(oth)' option which covers all aspects of interest to us.}

```
* BATCH MODE: activate the parser, the tagger, or both [P/t/b]? b
```

```
*** Using DIPETT as a batch PARSER and TAGGER ...
*** 1) set the cpu time limit (per sentence),
*** 2) enter the input filename,
***    (which must end with the special string "####.")
*** 3) and initiate the processing -- and see you later!
```

{below: 'A-2' as above in option 'j'.}

```
A-2) Parse trees filename (n for none)
    [n/"name"/zzz.DIPETT.parse_trees.pl]?
```

{below: the user can ask DIPETT to copy all tagging information (see 3.3.5) in a separate Prolog-readable file, which will be available for other processing afterwards.}

```
tagger output filename
    ["name"/zzz.DIPETT-tagger.pl]?
```

{below: as above in option 'd'}.

```
set a cpu time limit [y/N]? y
enter your limit in seconds: 30
```

NOTES:

- a) fragmentary parsing will be attempted if normal parsing fails;
- b) inputs containing unknown words will not be parsed;
- c) unknown words will be skipped if not found in Collins:
the associated info. will appear in the tagger output file;
- d) inputs containing potential "idioms" will remain unchanged;
- e) DIPETT will exit QP at the end of the input file.

{above: On (b) and (c): if a word cannot be found in DIPETT's lexicon, it is automatically looked up the Collins dictionary and the retrieved information is used by DIPETT in order to proceed with parsing. This information is also kept in the output to facilitate subsequent processing (more in 3.3.2).}

```
Input filename? zzz999.test
[File opened successfully]
At what unit number do you want to start the parsing? 1
```

{above: etc. Batch processing begins here.}

Appendix B: DIPETT's English Grammar

This appendix represents an important complement to section 3.3.4: it contains the listings of the two major Prolog files that constitute DIPETT's grammar of English, i.e. `DIPETT_6_PARSER_S_V.PL` and `DIPETT_8_PARSER_N.PL`, plus a simplified Prolog description of these two files. Two other files that are part of DIPETT's grammar but which do not appear below are its tense table (which contains information on all the legal combinations of verbal features) and its small grammar for questions.

B.1: DIPETT_6_PARSER_S_V.PL ("sentence & verb phrase rules")

```

/*****
/* file: dipett_6_parser_s_v.pl
/* version: September 3, 1993
/* author: Sylvain Delisle
*****/

:- assert(dipett_version(3,9,93)).

/*****
/* This file contains the top-level of DIPETT's grammar as well as the set of
/* rules that deal mainly with sentences and verb phrases (thus the '_s_v'). */
/*
/* As far as English grammar is concerned, the main reference used here is:
/*
/*   QUIRK = R. Quirk, S. Greenbaum, G. Leech, & J. Svartvik,
/*   "A Comprehensive Grammar of the English Language",
/*   Longman, 1985.
*****/

/*****
/* PARSE_ENGLISH_INPUT (used by the driver)
*/

parse_english_input(Parse_Tree,I,O) :-
    set_chrono,
    on_exception(Msg,
        compl_sentence(Parse_Tree,I,O),
        display_exception_message_and_fail__6(Msg)).

/* Exception: CPU time limit exceeded.
display_exception_message_and_fail__6(exception(Limit,Problem,Location,I)) :-
    nlp, writep('PARSING STOPPED for the following reason:'),
    nlp, writep(Problem), writep(Location), writep(' :'),
    nlp, writep(' > the current CPU time limit is: '),
    superv_cpu_time_2('7a',Limit),
    writep('At that moment, the input string was:'), nlp,
    get_tokens(I,I2), writep(I2), nlp, !, fail.

/* Exception: otherwise ... (usually an unknown predicate)
display_exception_message_and_fail__6(Exception_Message) :-
    nlp, writep('PARSING STOPPED for the following reason:'),
    nlp, writep(Exception_Message), nlp, nlp, !, fail.

/*****
/* COMPL_SENTENCE (used by parse_english_input)
*/

/* Empty or invalid input!
compl_sentence(_, [dct(punctuation: _/_, []) | _], []) :-
    erase_previous_mem_assertions, !, fail.

/* The input is a "parenthetical unit", a valid input enclosed in "( ... )".
compl_sentence(parenthetical(PU))
    parenthesis_o, {},
    (erase_previous_mem_assertions, compl_sentence(PU), parenthesis_c.

/* The input is an interrogative ("question").
compl_sentence(parse_tree__inter(Question),I,O) :-
    (last(dct(punctuation: '?' /end_of_sentence, []),I) ;
    rev(I, [dct(special_char: '?' /end_of_sentence, []) | _],
    dct(punctuation: '?' /end_of_sentence, []) | _)),
    erase_previous_mem_assertions, !,

```

```

question(Question,I,O2), question_mark(O2,O).

/* Usual case: the input is a declarative or an imperative ("sentence"). */
compl_sentence(parse_tree__decla_or_imper(Ctxt_M,Sentence,Example,Tag)) -->
    (erase_previous_mem_assertions), contextual_marker(Ctxt_M),
    top_level_clause_seeker(1,Sentence,_), opt_quiz_example(Example),
    terminal(Tag).

/* The input is a title or a caption with 2 parts separated by ':' or '--'. */
/* Ex: "Report 1.1: Accessing a single file." */
compl_sentence(parse_tree__title_or_caption(identifier(Id),title(T),Tag)) -->
    (erase_previous_mem_assertions),
    (np_nucleus(_,Id,_); noungp_head(_,_,Id)), (colon; dashes),
    sentence_frag(T), terminal(Tag).

/* Otherwise, the input may be a stand-alone fragment ... */
compl_sentence(parse_tree__frag(FRAG,Tag)) -->
    (erase_previous_mem_assertions), sentence_frag(FRAG), terminal(Tag).

/*****
/* CONTEXTUAL_MARKER (used by compl_sentence) */
/*
/* A "contextual marker" is a NP which identifies the semantic category (in
/* a broad sense) of the text following it. It helps the reader to properly
/* decode the purpose of a sentence or a paragraph (used mostly in T4043). */
contextual_marker(contextual_marker(Ctxt_M)) --> ctxtxt_marker(Ctxt_M).

ctxtxt_marker(answer) --> [dct(word:answer/_,_), colon, {}].
ctxtxt_marker(example) --> [dct(word:example/_,_), colon, {}].
ctxtxt_marker(example) -->
    [dct(word:for/_,_), dct(word:example/_,_), opt_comma, {}].
ctxtxt_marker(examples) --> [dct(word:examples/_,_), colon, {}].
ctxtxt_marker(instance) -->
    [dct(word:for/_,_), dct(word:instance/_,_), opt_comma, {}].
ctxtxt_marker(no) --> [dct(word:no/_,_), colon, {}].
ctxtxt_marker(note) --> [dct(word:note/_,_), colon, {}].
ctxtxt_marker(question) --> [dct(word:question/_,_), colon, {}].
ctxtxt_marker(tax_tip) --> [dct(word:tax/_,_), dct(word:tip/_,_), colon, {}].
ctxtxt_marker(yes) --> [dct(word:yes/_,_), colon, {}].
ctxtxt_marker(Ctxt_M) --> colon_look_ahead, np_equiv(_,1,Ctxt_M,_), colon, {}.
ctxtxt_marker(:nil) --> [].

/* ..... */
colon_look_ahead(String,String) :- colon_in_string(String).

colon_in_string([]) :- !, fail.
colon_in_string([dct(punctuation:':'/pause,_)|_]) :- !.
colon_in_string([_|Rest]) :- colon_in_string(Rest).

/*****
/* TOP_LEVEL_CLAUSE_SEEKER (used by compl_sentence & rest_sentence) */
/*
/* This procedure seeks coordinated clauses at the highest possible level.
/* The Caller parameter indicates whether this predicate is called at the
/* highest level (i.e. by compl_sentence) or not (i.e. by rest_sentence):
/* multiple coordination should only be recognized when it is set to 1. */
top_level_clause_seeker(_,_,_) -->
    end_of_sentence_marker(_), {!, fail}.

top_level_clause_seeker(Caller,Out_Struct,Mood) $ Tokens -->

```

```

evaluate_nb_potential_verbs(NPV,NPA,V_INT), {},
top_level_clause_seeker_h(Caller,NPV,NPA,V_INT,Sentence,Mood),
(get_desired_output_structure(Sentence,Tokens,Out_Struct)).

/* ..... */
/* The inclusion of the original phrase tokens in the parse tree is optional. */
/* This predicate is also used by several other major phrase components. The */
/* flag include_phrase_tokens_in_pt is set by the driver at the beginning. */

get_desired_output_structure(Struct,_,Struct) :-
    include_phrase_tokens_in_pt(off), !.
get_desired_output_structure(Struct_In,Tokens,Struct_Out) :-
    include_phrase_tokens_in_pt(on),
    functor(Struct_In,F,_), F \== '.', !,
    Struct_In =.. [H|T0],
    append(T0,[tokens(Tokens)],T1), Struct_Out =.. [H|T1].
get_desired_output_structure(Struct_In,Tokens,Struct_Out) :-
    include_phrase_tokens_in_pt(on),
    append(Struct_In,[tokens(Tokens)],Struct_Out).

/*****
/* EVALUATE_NB_POTENTIAL_VERBS (used by top_level_clause_seeker & _seeker_r) */
/*
/* We evaluate the number of potential verbs in the current input substring */
/* in order to guide the choice of the most appropriate high-level rule that */
/* the parser should tentatively apply next. For instance, if there is only */
/* one potential verb, then there is no need to apply the 'compound' rule. */
/*
/* In addition, we collect "inter-verbal" intervals (substrings) so that we */
/* can evaluate the density of verbs in the current input string. An inter- */
/* val ('int') is a substring located between two verbs. It is represented as */
/* 'int(POS_A,POS_B,Substring,Flag)' where POS_A and POS_B mark the relative */
/* position (begin/end) of the substring in the current input string. The */
/* second parameter, Substring, contains all the tokens of the substring. */
/* Flag is a boolean which takes the value 'true' if all tokens in the sub- */
/* string (interval) are amongst the following tokens: ['and','or','but',',', */
/* 'not'] -- in that situation, verbs are said to have a high density. */
/* Otherwise, Flag is set to 'false'. */
/* Two special cases are the 'pre_int' and the 'post_int': the first goes */
/* from the very beginning of the input string to the first verb, and the se- */
/* cond goes from the last verb to the end of the input string. */
/*
/* In general: "{pre_int} Verb1 {int#1} Verb2 {int#2} ... VerbN {post_int}." */
/*
/* Note: we also count the number of auxiliaries, but what really matters is */
/* "first-class" verbs, i.e. those that cannot function as auxiliaries. */

evaluate_nb_potential_verbs(0,0,nil,I,I) :-
    end_of_sentence_marker(_,I,_), !.
evaluate_nb_potential_verbs(0,0,nil,[],[]) :- !.
evaluate_nb_potential_verbs(NPV,NPA,V_INT,I,I) :-
    count_potential_verbs(0,NPV,0,NPA,1,[pre_int(1,1,[],nil)],INT_V,I),
    rev(INT_V,V_INT_00),
    subst(1,pre_int(1,1,[],nil),pre_int(0,0,[],nil),V_INT_00,V_INT_0),
    delete(V_INT_0.int(_,_,[],true),V_INT).

count_potential_verbs(0,0,M,M,_,
    [pre_int(A,B,C,nil)],
    [int(A,B2,C2,nil)],[]) :-
    !, rev(C,[dct(_:/end_of_sentence,[])|C2_rev]), rev(C2_rev,C2),
    ((C2 == [], B2 = A) ; (C2 \== [], B2 is B - 1)).
count_potential_verbs(N,N,M,M,_,
    [int(A,B,C,_)|T],

```

```

[post_int(A,B2,C2,nil)|T],[]) :-
!, rev(C,[dct(_:/end_of_sentence,[])|C2_rev]), rev(C2_rev,C2),
((C2 == [], B2 = A) ; (C2 \== [], B2 is B - 1)).
count_potential_verbs(NPV1,NPV2,NPA1,NPA2,Pos,INT_V_in,INT_V_out,[TOKEN|T]) :-
spot_potential_verbs(NPV1,NPV3,NPA1,NPA3,TOKEN,Pos,INT_V_in,INT_V_temp),
Pos_2 is Pos + 1,
count_potential_verbs(NPV3,NPV2,NPA3,NPA2,Pos_2,INT_V_temp,INT_V_out,T).

spot_potential_verbs(N,N1,M,M,dct(_,PofS),Pos,L_in,L_out) :-
\+ memberchk(d(aux/_/_/_),PofS), \+ memberchk(d(modal_aux/_/_),PofS),
memberchk(d(v/_/_/_),PofS), !,
N1 is N + 1,
Pos_A is Pos + 1, append([int(Pos_A,Pos_A,[],true)],L_in,L_out).
spot_potential_verbs(N,N,M,M1,dct(_,PofS),_,L_in,L_in) :-
(memberchk(d(aux/_/_/_),PofS) ; memberchk(d(modal_aux/_/_),PofS)), !,
M1 is M + 1.
spot_potential_verbs(N,N,M,M,dct(Y:X/Z,PofS),Pos_B,
[int(Pos_A,_,SST_in,F_in)|T],
[int(Pos_A,Pos_B,SST_out,F_out)|T]) :-
N \== 0, !, append(SST_in,[dct(Y:X/Z,PofS)],SST_out),
((F_in == true, \+ memberchk(X,[and,or,but,',',not]), F_out = false) ;
F_out = F_in), !.
spot_potential_verbs(0,0,M,M,X,Pos,
[pre_int(1,_,SST_in,nil)],
[pre_int(1,Pos,SST_out,nil)]) :-
!, append(SST_in,[X],SST_out).

/*****
/* TOP_LEVEL_CLAUSE_SEEKER_H (used by top_level_clause_seeker) */
/*
/* The 'H' stands for heuristic: here, we select the most appropriate order */
/* in which high-level rules should be applied. This choice is based on the */
/* number of potential verbs (see evaluate_nb_potential_verbs above) and the */
/* presence or absence of potential sent. coordinators in the current input. */

/* Not even a single verb: fail! */
top_level_clause_seeker_h(_0,0,_,_,_,_) :- !, fail.

/* No sent. coordinators OR only one potential verb: non-compound. */
top_level_clause_seeker_h(Caller,NPV,NPA,_,Sentence,Mood,I,O) :-
(zero_sent_coordinators ; NPV == 1 ; (NPV == 0, NPA == 1)), !,
top_level_clause_seeker_r(non_compound,Caller,Sentence,Mood,I,O).

/* More than 2 potential verbs AND potential correlative sent. coordinators. */
top_level_clause_seeker_h(Caller,NPV,NPA,V_INT,Sentence,Mood,I,O) :-
(NPV > 2 ; (NPV == 0, NPA > 2)),
all_verbal_intervals_dense(V_INT,false),
potential_corr_sent_coor(Coor1,Coor2), Coor1 \== nil, Coor2 \== nil,
must_be_located_in_interverbal_loc(Coor2,V_INT),
(Type = compound ; Type = non_compound),
top_level_clause_seeker_r(Type,Caller,Sentence,Mood,I,O).

/* More than 2 potential verbs AND potential sent. coordinator. */
top_level_clause_seeker_h(Caller,NPV,NPA,V_INT,Sentence,Mood,I,O) :-
(NPV > 2 ; (NPV == 0, NPA > 2)),
all_verbal_intervals_dense(V_INT,F),
potential_sent_coor(Coor), Coor \== nil,
must_be_located_in_interverbal_loc(Coor,V_INT),
((F == true,
(Type = non_compound ; Type = compound)) ;
(F == false,
(\+ all_intervals_have_a_potential_subj(V_INT,Coor),
(Type = compound ; Type = non_compound))) ;

```

```

    (all_intervals_have_a_potential_subj(V_INT,Coor).
      (Type = mult_coor ; Type = compound ; Type = non_compound))),
    top_level_clause_seeker_r(Type,Caller,Sentence,Mood,I,O).

/* Otherwise ...
top_level_clause_seeker_h(Caller,_,_,V_INT,Sentence,Mood,I,O) :-
  all_verbal_intervals_dense(V_INT,F),
  ((F == true,
    (Type = non_compound ;
      (potential_sent_coor(Coor),Coor \== nil,
        must_be_located_in_interverbal_loc(Coor,V_INT),
        Type = compound))) ;
    (F == false,
      (potential_sent_coor(Coor), Coor \== nil,
        must_be_located_in_interverbal_loc(Coor,V_INT),
        Type = compound) ;
      (Type = non_compound))),
  top_level_clause_seeker_r(Type,Caller,Sentence,Mood,I,O).

/* ***** */
/* See explanations about "verbal intervals" in evaluate_nb_potential_verbs. */

/* Testing whether all "verbal intervals" are dense. If so, F is set to true.*/
all_verbal_intervals_dense([],F) :-
  !, ((var(F), F = false) ; nonvar(F)).
all_verbal_intervals_dense([pre_int(_,_,_)|T],F) :-
  all_verbal_intervals_dense(T,F).
all_verbal_intervals_dense([int(_,_,nil)|T],F) :-
  all_verbal_intervals_dense(T,F).
all_verbal_intervals_dense([int(_,_,true)|T],F) :-
  all_verbal_intervals_dense(T,F).
all_verbal_intervals_dense([int(_,_,false)|_],false) :- !.
all_verbal_intervals_dense([post_int(_,_,_)|_],true) :- !.

/* Testing whether Token is located in one of the "verbal intervals". */
must_be_located_in_interverbal_loc(_,[]) :- !, fail.
must_be_located_in_interverbal_loc(Token,[pre_int(_,_,_)|T]) :-
  must_be_located_in_interverbal_loc(Token,T).
must_be_located_in_interverbal_loc(Token,[int(A,B,SUBSTRING,_)|_]) :-
  memberchk(dct(_:Token/_,_),SUBSTRING),
  ((\+ memberchk(Token,[and,or,but]), (B - A + 1) > 2) ;
   memberchk(Token,[and,or,but])).
must_be_located_in_interverbal_loc(Token,[int(_,_,_)|T]) :-
  must_be_located_in_interverbal_loc(Token,T).
must_be_located_in_interverbal_loc(_Token,[post_int(_,_,_)|_]) :- !, fail.

/* Testing whether all "verbal intervals" have at least a potential subject. */
all_intervals_have_a_potential_subj([pre_int(_,_,_)|T],Coor) :-
  all_intervals_have_a_potential_subj(T,Coor).
all_intervals_have_a_potential_subj([int(_,_,SUBSTRING,_)|T],Coor) :-
  rev(SUBSTRING,[dct(_:X/_,_)]),
  (\+ memberchk(X,[and,or,but,',',not]), X \== Coor,
   all_intervals_have_a_potential_subj(T,Coor)).
all_intervals_have_a_potential_subj([int(_,_,_)|_],_) :- !, fail.
all_intervals_have_a_potential_subj([post_int(_,_,_)|_],_) :- !.

/* ***** */
/* TOP_LEVEL_CLAUSE_SEEKER_R (used by top_level_clause_seeker_h) */
/*
/* According to the order specified in top_level_clause_seeker_h above, we
/* now call upon the associated rule: non-compound, compound, or mult-coord. */

```



```

top_level_clause_seeker_r(non_compound,_,Sentence,Mood,I,O) :-
    check_chrono(top_level_clause_seeker_r__non_compound,I),
    find_coordinated_main_clauses(Coor1,[],Mood,_,I,I2),
    sentence(Sent1,Mood,I2,I3),
    rest_sentence(Coor1,[],Sent1,Mood,Sentence,I3,O).

top_level_clause_seeker_r(compound,_,Sentence,Mood,I,O) :-
    check_chrono(top_level_clause_seeker_r__compound,I),
    find_coordinated_main_clauses(Coor1,Coor2,Mood,String1,I,I2), Coor2 \== [],
    call_tentative_sentence(Sent1,Mood,String1),
    rest_sentence(Coor1,Coor2,Sent1,Mood,Sentence,I2,O).

top_level_clause_seeker_r(mult_coor,1,
                           mult_coor_clauses(coord(Coor),
                                                Mood,
                                                [Sent1|S_R]),
                           Mood,I,O)
    :-
    check_chrono(top_level_clause_seeker_r__multiple_coord,I),
    potential_sent_coor(Coor), memberchk(Coor,[and,or,but]),
    find_ahead_next_sent(Coor,I,[],I_H,I_R), memberchk(dct(word:Coor/_,_),I_R),
    evaluate_nb_potential_verbs(N,M,_,I_R,_), (N > 1 ; (N == 0, M > 1)),
    call_tentative_sentence(Sent1,Mood,I_H),
    multiple_coord_clauses(Coor,Mood,S_R,I_R,O), !, length(S_R,L), L > 1.

/* ..... */

find_ahead_next_sent(_,I_R,I_H,I_H,I_R) :-
    end_of_sentence_marker(_,I_R,_), !.
find_ahead_next_sent(Coor,[dct(T:X/Y,Z)|R],I_H,I_H,[dct(T:X/Y,Z)|R]) :-
    (T == word, X == Coor) ; (T == punctuation, X == ',').
find_ahead_next_sent(Coor,[H|R],I_H0,I_H,I_R) :-
    append(I_H0,[H],I_H01), find_ahead_next_sent(Coor,R,I_H01,I_H,I_R).

call_tentative_sentence(Sent,Mood,I) :- sentence(Sent,Mood,I,[]), !.

/*****
/* MULTIPLE_COORD_CLAUSES (used by top_level_clause_seeker_r) */
/*
/* These rule cover the multiple coordination of clauses (see QUIRK, p.925). */
/* In this structure, 'and' and 'or' can link more than two clauses and all */
/* but the final instance of these conjunctions can be omitted. For example: */
/* "The battery may be disconnected, the connection may be loose, or the bulb */
/* may be faulty", where the 2nd clause means ", OR the connection may ..., ". */
/*
/* Notation: 'syndetic': with an explicit coordinator; 'asyndetic': implicit */
/* coordinator which is temporarily "postponed" (thus the 'p_'). */
/*
/* Case (1) : SENT ('.') and/or SENT ... */
multiple_coord_clauses(Coor,Mood,[syndetic_mult_coor_clauses(S2)|S_R],I,O)
    :-
    opt_comma(I,I2), conj(Coor,I2,I3),
    find_ahead_next_sent(Coor,I3,[],I4,I5),
    call_tentative_sentence(S2,Mood,I4),
    multiple_coord_clauses(Coor,Mood,S_R,I5,O).

/* Case (2) : SENT ', ' SENT ... */
multiple_coord_clauses(Coor,Mood,[p_asyndetic_mult_coor_clauses(S2)|S_R],I,O)
    :-
    comma(I,I2),
    find_ahead_next_sent(Coor,I2,[],I3,I4), memberchk(dct(word:Coor/_,_),I4),
    call_tentative_sentence(S2,Mood,I3),
    multiple_coord_clauses(Coor,Mood,S_R,I4,O), S_R \== [].

```

```

multiple_coord_clauses(.,_,[],I,I)
:-
    end_of_sentence_marker(.,I,_, check_chrono(multiple_coord_clauses,I,I)).

/*****
/* OPT_QUIZ_EXAMPLE (used by compl_sentence)
/*
/* Actual examples in the QUIZ manual have been replaced by one of these
/* special markers: 'quiz-example-command', 'quiz-example-commands',
/* 'quiz-example-program', 'quiz-example-prompt', & 'quiz-example-report'.
/* They are classified in the dictionary as special_function_word(s).
*****/

opt_quiz_example(quiz_example(marker(M),QE)) -->
    opt_ex_marker(M), colon, special_function_word(QE,Type), (Type == frag, !),
opt_quiz_example(quiz_example(nil)) --> [].

/* An example marker is "as in", "for example", or "like this".
opt_ex_marker(as_in) -->
    opt_comma, conj(as), prep(in), (!).
opt_ex_marker(for_example) -->
    opt_comma, prep(for), countnoun(example,sg), (!).
opt_ex_marker(like_this) -->
    opt_comma, conj(like), deter(this,_,_), (!).
opt_ex_marker(nil) --> [].

/*****
/* SENTENCE_FRAG (used by compl_sentence & rest_there_sent)
*****/

sentence_frag(_) -->
    end_of_sentence_marker(., (!, fail)).

sentence_frag(quiz_example(marker(for_example),QE)) -->
    check_chrono(sentence_frag),
    prep(for), countnoun(example,sg), colon,
    special_function_word(QE,Type), (Type == frag).

sentence_frag(sentence_fragment(Fragment)) -->
    np_equiv(.,1,Fragment,_).

sentence_frag(sentence_fragment(Fragment)) -->
    required_prepphrases(Fragment,_).

sentence_frag(sentence_fragment(Fragment)) -->
    required_adj_phrase(Fragment).

sentence_frag(sentence_fragment(Fragment)) -->
    required_adverbs(Fragment).

/*****
/* FIND_COORDINATED_MAIN_CLAUSES (used by top_level_clause_seeker_r)
/*
/* This procedure checks whether the input is composed of coordinated main
/* clauses, i.e. basic sentences joined by one or two (for correlatives)
/* coordinating conjunctions. It splits up the input into two main clauses
/* (called Sent1 and Sent2 in what follows) which are joined by a coordinator
/* or a conjunction having the same functionality (i.e. conjunct, subordina-
/* tor, or pair of correlatives). See QUIRK, chap.13 ("Coordination").
/*
/* If the input contains more than two coordinated main clauses, they will be
/* recognized recursively with rest_sentence (via top_level_clause_seeker).
/*
/* We use check_for_potential_verb in order to make sure that what has been

```

```

/* identified as Sent1 (and Sent2) is really a sentence. That is, a sentence */
/* must contain at least one potential verb; otherwise, the parser will be */
/* wasting precious time. So before successfully exiting this procedure, */
/* we require that Sent1 and Sent2 each contain at least one potential verb. */
/* We also require that such coordinated sentences have the same mood. */
/* */
/* Note: the structure illustrated by the following example is NOT covered: */
/* "He designed and she implemented the program" where "the program" is the */
/* elided object of the first sentence in which 'design' is transitive. */

/* COMPOUND: the input contains at least two coordinated main clauses. */
find_coordinated_main_clauses(Coor1,Coor2,Mood,Sent1,I,O) :-
    Coor2 \== [],
    find_coor(Coor1,Sent1,Coor2,I,O2), % Sent2 is a subset of O2
    get_full_O(Sent1,I,O),
    check_for_potential_verb(Sent1,Mood), check_for_potential_verb(O2,Mood).

/* SINGLE MAIN CLAUSE: the input may begin with a conjunction. Such initial */
/* conjunctions function either as intersentential coordinators or as intra- */
/* sentential coordinators, conjuncts, or subordinators. The latter two are */
/* recognized in subordinate_clause (see at the end of this file). */
find_coordinated_main_clauses(ICoor,[],Mood,_,I,O) :-
    initial_coor__opt_comma(ICoor,I,O), check_for_potential_verb(O,Mood).

/* ++++++>+++++ */

get_full_O([],O,O) :- !.
get_full_O([H|T1],[H|T2],O) :- !, get_full_O(T1,T2,O).
get_full_O(Sent1,[_|T],O) :- get_full_O(Sent1,T,O).

initial_coor__opt_comma(IC) --> conj(IC), {coordinator(IC)}, opt_comma, (!).
initial_coor__opt_comma(nil) --> [].

/*****
/* FIND_COOR (used by find_coordinated_main_clauses) */
/* */
/* A coordinator is a "real" coordinator, a semi-coordinator, or an */
/* adverbial conjunct (thus the test "Type == conj" for adverbs). */
/* Potential sentential coordinators were detected during lexical analysis */
/* and asserted there: see programming details in dipett_4_lexicalyser.pl. */
/* */
/* See QUIRK, pp.920+ (coordinators), pp.935+ (correlatives), and */
/* pp.928, 982-3 (semi- and quasi-coordinators). */
/* */
/* It should be noted that the comma does NOT function as a coordinator when */
/* there are only two coordinated main clauses: see QUIRK pp.920+ and p.1615.*/

find_coor(,_,_) --> {zero_sent_coordinators, !, fail}.

find_coor(Coor1,Sent1,Coor2) -->
    sent_coor(Coor1,Coor2), look_ahead_coor([],Sent1,Coor2), {Sent1 \== []}.

zero_sent_coordinators :-
    bagof(X,potential_sent_coor(X),[nil]),
    bagof(Y,potential_corr_sent_coor(Y,_,)[nil]),
    bagof(Z,potential_idiom_sent_coor(Z),[nil]).

/* ..... */

sent_coor(Coor1,Coor2) -->
    {potential_corr_sent_coor(Coor1,Coor2), Coor2 \== nil},
    {[det(word:Coor1/_,_) ; idiomatic_cat(Coor1,_,_)}.

```

```

sent_coor(nil,Coor2) -->
    {potential_idiom_sent_coor(Coor2), Coor2 \== nil}.

sent_coor(nil,Coor2) -->
    {potential_sent_coor(Coor2), Coor2 \== nil}.

/* ..... */

/* Let's try to find the conjunction and thus the end of the first sentence. */
look_ahead_coor(.,.,.) --> end_of_sentence_marker(., {!, fail}).
look_ahead_coor(Sent1,Sent1,Coor2) --> pause, conj(Coor2).
look_ahead_coor(Sent1,Sent1,Coor2) --> conj(Coor2).
look_ahead_coor(Sent1,Sent1,Coor2) --> pause, adv(Coor2.Type), (Type == conj).
look_ahead_coor(Sent1,Sent1,Coor2) --> adv(Coor2.Type), (Type == conj).
look_ahead_coor(Sent0,Sent1,Coor2) -->
    [T], {append(Sent0,[T],Sent01)}, look_ahead_coor(Sent01,Sent1,Coor2).

/*****
/* CHECK_FOR_POTENTIAL_VERB (used by find_coordinated_main_clauses) */
/*
/* This check is performed as a look-ahead to avoid parsing a segment which
/* is bound to be rejected anyway. In other words, a sentence is a sentence
/* if and only if it contains a verb. Obviously, this does not account for
/* exclamatives, irregular sentences (e.g. block language), and nonsentences.
/* See QUIRK, chap.11 ("Sentence types and discourse functions").
/*
/* In a normal declarative sentence, the verb (or auxiliary or modal) is not
/* the very first word, unless we have an imperative or a let-imperative.
*/

check_for_potential_verb(I,_) :-
    (end_of_sentence_marker(.,I,_) ; I == [], !, fail).

check_for_potential_verb([H|T],declarative) :-
    (\+ verb(.,.,.,[H],_) ; ing_word([H])),
    \+ aux(.,.,.,[H],_), \+ modal_aux(.,.,[H],_),
    check_for_potential_verb_rest(T).

check_for_potential_verb(X,imperative) :-
    \+ check_for_let(X,_, !, check_for_potential_verb_rest(X)).

check_for_potential_verb(X,imperative_let) :-
    check_for_let(X,X_rest), check_for_potential_verb_rest(X_rest).

/* ..... */

/* A -ing word might be the very first word: it is not necessarily a verb!
ing_word([dct(word:X/_,_)|_]) :- name(X,X_ascii), append(., "ing", X_ascii).

check_for_potential_verb_rest([]) :- !, fail.
check_for_potential_verb_rest([H|_]) :- verb(.,.,.,[H],_), !.
check_for_potential_verb_rest([_|T]) :- check_for_potential_verb_rest(T).

check_for_let([],[]) :- !, fail.
check_for_let([dct(word:let/_,_)|Rest],Rest) :- !.
check_for_let([_|T],Rest) :- check_for_let(T,Rest).

/*****
/* SENTENCE (used by call_tentative_sentence & top_level_clause_seeker_r) */
/*
/* Two types of sentence are recognized: DECLARATIVE and IMPERATIVE.
/* EXCLAMATIVES are treated as declaratives. On QUESTIONS, see dipett_9.
*/

```

```

sentence(.,_) --> end_of_sentence_marker(,), (!, fail).
sentence(Struct,Sentence_Type) -->
    check_spy_on_this_rule(sentence,nil), check_chrono(sentence),
    subordinate_clause(initial,Initial_Modif),
    sentence_core(Struct,Sentence_Type,Initial_Modif).

/*****
/* SENTENCE_CORE (used by sentence) */
/*
/* These rules define the main types of sentences that DIPETT recognizes. */

sentence_core(.,_,_) -->
    end_of_sentence_marker(,), (!, fail).

/* The special HERE+BE: "HERE IS the solution.", "HERE ARE the tools.", etc. */
/* See QUIRK, pp.521-3 and 1379-81. */
sentence_core(Struct,declarative,Initial_Modif,I,O) :-
    check_spy_on_this_rule(sentence_core,sentence_core_declar_here_be,I,I),
    adv(here,_,I,I2), look_ahead_BE_pres(I2),
    verbphrase(1,_,Tense,here_be__compl,Initial_Modif,Struct,nil,I2,O),
    (Tense == [pres] ; Tense == [pres3s]).

/* The special existential-THERE sentence: "THERE ARE people in the room." */
/* Note: other verbs than BE are also allowed; see QUIRK, pp.1402-14. */
sentence_core(Struct,declarative,Initial_Modif) -->
    check_spy_on_this_rule(sentence_core,sentence_core_declar_there_be),
    special_function_word(there,_),
    verbphrase(1,_,_,there_be__compl,Initial_Modif,Struct0,nil),
    rest_there_sent(Struct0,Struct).

/* The general case: a NP followed by a VP. */
sentence_core(Struct,declarative,Initial_Modif) -->
    check_spy_on_this_rule(sentence_core,sentence_core_declar),
    np_equiv(pre_v,1,Np,Num),
    verbphrase(1,Num,_,Np,Initial_Modif,Struct,nil).

/* Imperative with "let". On directives, see QUIRK, pp.827+. */
sentence_core(Struct,imperative_let,Initial_Modif) -->
    check_spy_on_this_rule(sentence_core,sentence_core_imp_let),
    opt_courtesy_subjunct,
    preposed_verb__let, (let_pron(Subj) ; np_equiv(pre_v,2,Subj,_)),
    opt_negation(Neg),
    verbphrase(1,_,[imp],Subj,Initial_Modif,Struct2,nil),
    (get_imp_let_struct(Struct2,Neg,Struct)).

/* Plain imperative sentence: it must begin with a verb. */
sentence_core(Struct,imperative,Initial_Modif) -->
    check_spy_on_this_rule(sentence_core,sentence_core_imp),
    opt_courtesy_subjunct,
    verbphrase(1,_,[imp],nil,Initial_Modif,Struct,nil).

/* ***** */

look_ahead_BE_pres([dct(word:are/_,_)|_]).
look_ahead_BE_pres([dct(word:is/_,_)|_]).

/* ..... */

/* Recognizes the rest of the "enumerative" existential-THERE sentence. */
/* Ex: "There are two types of keys: UNIQUE AND REPEATING." */
rest_there_sent(Struct0,enum(Struct0,ENUM)) -->
    colon, sentence_frag(ENUM), (!).
rest_there_sent(Struct0,Struct0) --> [].

```

```

/* ..... */

opt_courtesy_subjunct --> adv(please,_), opt_comma, {}.
opt_courtesy_subjunct --> {}.

preposed_verb_let([dct(word:let/_,_)|0],0).

let_pron(her)      --> pers_pron(her,_).
let_pron(him)     --> pers_pron(him,_).
let_pron(it)      --> pers_pron(it,_).
let_pron(me)      --> pers_pron(me,_).
let_pron(them)    --> pers_pron(them,_).
let_pron(us)      --> pers_pron(us,_).
let_pron(yourself) --> reflex_pron(yourself,_).
let_pron(yourselves) --> reflex_pron(yourselves,_).

/*****
/* GET_IMP_LET_STRUCT (used by sentence_core) */
/*
/* In an imperative with "let", the negation (if any) must be found BEFORE
/* the VP, thus neg(yes) in the 2nd clause of test_Neg1_and_replace_by_Neg2. */

get_imp_let_struct(
    statement(Modifier_s,Subj,predicate(Verb1,Voice,Complement,Modifier_p)),
    Neg2,
    statement(Modifier_s,Subj,predicate(Verb2,Voice,Complement,Modifier_p))
    :-
    !, Verb1 =.. X, test_Neg1_and_replace_by_Neg2(X,Y,Neg2), Verb2 =.. Y.

get_imp_let_struct(
    statement(Modifier_s,Subj,predicate(Verb1,Voice,Complement,Modifier_p),T),
    Neg2,
    statement(Modifier_s,Subj,predicate(Verb2,Voice,Complement,Modifier_p),T))
    :-
    Verb1 =.. X, test_Neg1_and_replace_by_Neg2(X,Y,Neg2), Verb2 =.. Y.

/* Conjoined main verbs. */
test_Neg1_and_replace_by_Neg2(
    [Reg_or_Cop,conj_verbs(Conj,Conj_Verbs_in)],
    [Reg_or_Cop,conj_verbs(Conj,Conj_Verbs_out)],Neg2)
    :-
    !, replace_Neg1_in_conj_verbs(Conj_Verbs_in,Conj_Verbs_out,Neg2).

/* Single main verb. */
test_Neg1_and_replace_by_Neg2(
    [Reg_or_Cop,verb(MV,Tense,Partic,neg(yes),Adv,Trans)],
    [Reg_or_Cop,verb(MV,Tense,Partic,neg(Neg2),Adv,Trans)],Neg2).

/* ..... */

replace_Neg1_in_conj_verbs([],[],_) :- !.
replace_Neg1_in_conj_verbs([V_H|V_T],[V_H_ok|V_T_ok],Neg2) :-
    test_Neg1_and_replace_by_Neg2([_,V_H],[_,V_H_ok],Neg2),
    replace_Neg1_in_conj_verbs(V_T,V_T_ok,Neg2).

/*****
/* REST_SENTENCE (used by top_level_clause_seeker_r) */
/*
/* Rules for compounds (#3-4) allow an unbound number of coordinated clauses
/* via top_level_clause_seeker which uses sentence and rest_sentence. */
/* "single-main-clause" input: SIMPLE SENTENCE. */

```

```

rest_sentence(Coor1,[],Sent,Mood,
    simple_sentence(
        structure(
            single_main_clause(coord(Coor1),Str)))
    -->
    check_spy_on_this_rule(rest_sentence,rest_sentence_simple),
    {\+ functor_memberchk(subordinate,Sent), Str =.. [Mood,Sent]}).

/* "single-main-clause" input: COMPLEX SENTENCE. */
rest_sentence(Coor1,[],Sent,Mood,
    complex_sentence(
        structure(
            single_main_clause(coord(Coor1),Str)))
    -->
    check_spy_on_this_rule(rest_sentence,rest_sentence_complex),
    {Str =.. [Mood,Sent]}).

rest_sentence(_,_,_,_,_,_,_)
    -->
    end_of_sentence_marker(_), {!, fail}).

/* Coordinated main clauses (compound): FIRST COORDINATOR IS "EMPTY". */
rest_sentence(nil,Coor2,Sent1,Mood,
    compound_sentence(
        structure(
            head_main_clause(coord(nil),Str1),
            next_main_clause(coord(Coor2,Opt_conj),Str2)))
    -->
    check_spy_on_this_rule(rest_sentence,rest_sentence_compound_nilstcoor),
    {!}, opt_pause, [dct(_:Coor2/_,_)], opt_coor_conj(Opt_conj),
    top_level_clause_seeker(2,Sent2,Mood),
    {Str1 =.. [Mood,Sent1], Str2 =.. [Mood,Sent2]}).

/* Coordinated main clauses (compound): CORRELATIVES. */
rest_sentence(Coor1,Coor2,Sent1,Mood,
    correlative_compound_sentence(
        structure(
            head_main_clause(coord(Coor1),Str1),
            next_main_clause(coord(Coor2),Str2)))
    -->
    {Coor1 \== nil, Coor2 \== [], Coor2 \== nil},
    check_spy_on_this_rule(rest_sentence,rest_sentence_correlatives),
    opt_pause, [dct(_:Coor2/_,_)],
    top_level_clause_seeker(2,Sent2,Mood),
    {Str1 =.. [Mood,Sent1], Str2 =.. [Mood,Sent2]}).

/* ..... */

/* These five coordinators do not allow another conjunction to precede them: */
/* see QUIRK, pp.922-3. */
opt_coor_conj(C) -->
    conj(C), {\+ memberchk(C,[and,or,but,for,so_that]), !}.
opt_coor_conj(nil) --> [].

/.....*/
/* FUNCTOR_MEMBERCHK (used by rest_sentence) */
/* */
/* Checks whether a certain functor F is part of an arbitrary structure S. */

functor_memberchk(_,S) :- (atomic(S) : var(S)), !, fail.
functor_memberchk(F,S) :- functor(S,F,_), !.
functor_memberchk(_,[]) :- !, fail.
functor_memberchk(F,[S_H|S_T]) :-
    functor_memberchk(F,S_H) ; functor_memberchk(F,S_T).

```

```

functor_memberchk(F,S)      :-
    functor(S,G,_), !, G \== '.', S =.. X, functor_memberchk(F,X).

/* *****
/* VERBPHRASE (used by question, rel_clause, sentence_core,
/*   sentence_core_for_questions, sentence_core_s,
/*   sentence_for_wh_i_d_clause, & wh_i_d_clause)
/*
/* Allows a conjunction of verb phrases. Here "tense" includes modality:
/*   Time: temps chronologique; Tense: temps des verbes;
/*   Aspect: facon dont l'action s'accomplit (debut, duree, etc.);
/*   Modality: nuances dans l'accomplissement de l'action.
/*

verbphrase(_,_,_,_,_,_,_)_      -->
    end_of_sentence_marker(_), (!, fail).

verbphrase(_ ,Num,Tense,Subj,Initial_Modif,Out_Struct,M_el) $ Tokens -->
    already_parsed(verbphrase,[Num,Tense,Subj,Initial_Modif,VP,M_el]),
    (get_desired_output_structure(VP,Tokens,Out_Struct)).

verbphrase(Type,Num,Tense,Subj,Initial_Modif,Out_Struct,M_el,I,O) :-
    check_spy_on_this_rule(verbphrase,nil,I,I),
    check_chrono(verbphrase,I),
    verbphrase_Reg_or_Stat(Type,Num,Tense,Subj,VP1,M_el,I,O2),
    ajust_Num_for_conj_vb(Num,Tense,Num_ok),
    rest_verbphrase(Type,Num_ok,Subj,Initial_Modif,VP1,M_el,VP,O2,O),
    simili_dollar_device(I,O,Tokens),
    keep_parse_in_mem(verbphrase,[Num,Tense,Subj,Initial_Modif,VP,M_el],I,O),
    get_desired_output_structure(VP,Tokens,Out_Struct).

/* ..... */

/* Adjusting 'Num' for conjoined main verbs: the sg3 special case.
/*
ajust_Num_for_conj_vb(sg3,[H|_],sg) :-
    memberchk(H,[be_to,can,could,dare,had_better,ought_to,may,might,
    must,shall,should,used_to,will,would,would_rather]), !.
ajust_Num_for_conj_vb(Num,_ ,Num).

/* *****
/* VERBPHRASE_REG_OR_STAT (used by conj_vp & verbphrase)
/*
/*
/* Verb phrases are divided into two major types: REGULAR (action verbs),
/* and STATIVE ("to be", "to become", "to appear", etc.). Within regular
/* verb phrases, the passive and active voices are distinguished.
/*
/* REGULAR ACTIVE.
verbphrase_Reg_or_Stat(Type,Num,Tense,Subj,statement(subj(Subj),Struct),M_el)
-->
    verb_clause(Type,Num,active,Tense,_ ,_,Struct,M_el).

/* REGULAR PASSIVE.
verbphrase_Reg_or_Stat(Type,Num,Tense,Compl,statement(subj(Subj),Struct),M_el)
-->
    verb_clause(Type,Num,passive,Tense,Subj,Compl,Struct,M_el).

/* STATIVE.
verbphrase_Reg_or_Stat(Type,Num,Tense,Subj,statement(subj(Subj),Struct),M_el)
-->
    verb_clause(Type,Num,stative,Tense,_ ,_,Struct,M_el).

/* *****
/* REST_VERBPHRASE (used by verbphrase)
/*

```



```

/*
/* The verbphrase and rest_verbphrase rules recognize these structures:
/* 1) single-VP, 2) conjoined-VPs, 3) single-VP '(' single-VP ')', and
/* 4) single-VP '(' conjoined-VPs ')'.
/* However, the same rules do not recognize structures like #3 & 4 in which
/* the first element, i.e. single-VP, would be replaced by conjoined-VPs.
/*
/* It should be noted that this remark also applies to these structures:
/* i) rest_np_nucleus (with NPs instead of VPs),
/* ii) rest_nom_cl (with nominal clauses instead of VPs), and
/* iii) rest_prepphrases (with PPs instead of VPs).
*/

rest_verbphrase(Type,Num,Subj,Initial_Modif,statement(Agent,VP1),M_el,
               statement(subord_clauses(initial(Initial_Modif)),
                           Agent,
                           conj_vps(CONJS,[VP1_ok,parenthetical(VPS_ok)])))

    -->
    parenthesis_o,
    conj_vp(1,Type,Num,Subj,Agent,M_el,CONJS,VPS),
    (nonvar(CONJS), CONJS \== []),
    parenthesis_c,
    (check_elided_modal_or_aux(VP1,VP1_ok,VPS,VPS_ok)).

rest_verbphrase(Type,Num,Subj,Initial_Modif,statement(Agent,VP1),M_el,
               statement(subord_clauses(initial(Initial_Modif)),
                           Agent,
                           conj_vps(CONJS,[VP1_ok|VPS_ok])))

    -->
    conj_vp(1,Type,Num,Subj,Agent,M_el,CONJS,VPS),
    (nonvar(CONJS), CONJS \== []),
    (check_elided_modal_or_aux(VP1,VP1_ok,VPS,VPS_ok)).

rest_verbphrase(.,.,Initial_Modif,statement(Agent,VP1),.,
               statement(subord_clauses(initial(Initial_Modif)),
                           Agent,
                           VP1_ok))

    -->
    (vp_tense_table_info(VP1,VP1_ok)).

/* ..... */

/* conj_vp requires that Conj be overt but not necessarily unique. It thus
/* accepts structures like "A and B then C". These are relatively rare and
/* may be rather ambiguous (e.g. with 'and' and 'or'). See QUIRK pp.920+.
/* Note: Conj must be overt AND unique for conj_np, conj_nom_cl, and conj_pp.
/*
/* The N parameter indicates whether we are currently trying to parse a VP
/* that follows the very first VP. If so, we don't allow "VP ',' CONJ VP".
/* Unfortunately, this discards cases like "VP ',' CONJ VP ',' CONJ VP".
/* However, this seems to be the price to pay to reduce ambiguity in the par-
/* sing of conj. items: only a costly look-ahead mechanism might be helpful.
/* This also applies to conj_np, conj_nom_cl, and conj_pp (in dipett_8).
*/

/* Case (1) : VP ('') CONJ VP ...
conj_vp(N,Type,Num,Subj,Agent,M_el,[Conj|C_R],[VP1|VPS])
    ({N == 1} ; ({N \== 1}, opt_comma)), conj(Conj),
    verbphrase_Reg_or_Stat(Type,Num,.,Subj,statement(Agent,VP1),M_el),
    conj_vp(.,Type,Num,Subj,Agent,M_el,C_R,VPS).

/* Case (2) : VP ',' VP ...
conj_vp(.,Type,Num,Subj,Agent,M_el,CONJS,[p_asyncdetic_conj_vp(VP1)|VPS]) -->
    comma, verbphrase_Reg_or_Stat(Type,Num,.,Subj,statement(Agent,VP1),M_el),
    conj_vp(.,Type,Num,Subj,Agent,M_el,CONJS,VPS), (VPS \== []).

```

```

conj_vp(.,_,.,_.,[],[]):-!,[]:-!.
check_chrono(conj_vp).

/* ***** */
/* VP_TENSE_TABLE_INFO (used by rest_verbphrase)                                */
/*                                                                           */
/* Replacing the original tense sequence by a standard tense name.          */
/* ***** */

vp_tense_table_info(X_in,X_out):-
    X_in=.. [predicate|[V1|REST]],
    access_tense_table(V1,V1_ok,._), X_out=.. [predicate|[V1_ok|REST]].

/* ***** */
/* ACCESS_TENSE_TABLE (used by harmonize_tenses_1_and_2,                      */
/*   process_conjoined_verbs, tense2_is_subset_of_tensel,                  */
/*   & vp_tense_table_info)                                                */
/* ***** */

/* Conjoined main verbs.                                                    */
access_tense_table(Vs,Vs_ok,.-):-
    Vs=.. [Reg_or_Cop,conj_verbs(Conj,Conj_Verbs_in)], !,
    process_conjoined_verbs(Conj_Verbs_in,Conj_Verbs_out),
    Vs_ok=.. [Reg_or_Cop,conj_verbs(Conj,Conj_Verbs_out)].

/* Single main verb.                                                        */
access_tense_table(V1,V1_ok,Tensela,Tenselb):-
    V1=.. [Reg_or_Cop,verb(MV1,tense(Tensela),Partic,Neg,Advs,Trans)],
    tense_table(Tensela,Tenselb),
    V1_ok=.. [Reg_or_Cop,verb(MV1,tense(Tenselb),Partic,Neg,Advs,Trans)].

/* ..... */

process_conjoined_verbs([],[]) :- !.
process_conjoined_verbs([V_H|V_T],[V_H_ok|V_T_ok]) :-
    V=.. [dummy,V_H], access_tense_table(V,V_ok,.-),
    V_ok=.. [dummy,V_H_ok], process_conjoined_verbs(V_T,V_T_ok).

/* ***** */
/* CHECK_ELIDED_MODAL_OR_AUX (used by rest_verbphrase)                    */
/*                                                                           */
/* We must check whether the VPs following the very first VP, in a conjunc- */
/* tion of VPs, contain an elided modal or auxiliary. Here's an example:   */
/* "I will print my messages and read them tonight" should be interpreted as */
/* "I will print my messages and WILL read them tonight".                   */
/* ***** */

check_elided_modal_or_aux(_,_,[],[]) :- !.

check_elided_modal_or_aux(VP1,VP1_ok,[VP2|VP_R],[VP2_ok|VP_R_ok]):-
    elided_modal_or_aux(VP1,VP1_ok,VP2,VP2_ok),
    check_elided_modal_or_aux(VP1,_,VP_R,VP_R_ok).

/* ..... */

/* Conjoined VPs: the asyndetic case.                                       */
elided_modal_or_aux(X_in,X_out,Y_in,Y_out):-
    Y_in=.. [p_asyndetic_conj_vp|[Y_1]], !, Y_1=.. [predicate|[V2|REST2]],
    X_in=.. [predicate|[V1|REST1]],
    harmonize_tenses_1_and_2(V1,V1_ok,V2,V2_ok),
    X_out=.. [predicate|[V1_ok|REST1]],
    Y_2=.. [predicate|[V2_ok|REST2]], Y_out=.. [p_asyndetic_conj_vp|[Y_2]].

/* Conjoined VPs: otherwise ...                                             */
elided_modal_or_aux(X_in,X_out,Y_in,Y_out):-
    Y_in=.. [predicate|[V2|REST2]],
    X_in=.. [predicate|[V1|REST1]],

```

```

    harmonize_tenses_1_and_2(V1,V1_ok,V2,V2_ok),
    X_out =.. {predicate|[V1_ok|REST1]}],
    Y_out =.. {predicate|[V2_ok|REST2]}].

/* ..... */

/* Here, we convert the sequence of "tenses" into a more standard name. */
harmonize_tenses_1_and_2(V1,V1_ok,V2,V2_ok) :-
    access_tense_table(V1,V1_ok,Tensela,Tenselb),
    tense2_is_subset_of_tensel(Tensela,Tenselb,V2,V2_ok).

/* When Tensel1 is a "subset" (equal or implicitly equal) of Tense2. */
tense2_is_subset_of_tensel(Tensela,Tenselb,V2,V2_ok) :-
    V2 =.. [Reg_or_Cop,conj_verbs(Conj,Conj_Verbs_in)], !,
    tense2_vs_tensel_in_conj_verbs(Conj_Verbs_in,Conj_Verbs_out,Tensela,Tenselb),
    V2_ok =.. [Reg_or_Cop,conj_verbs(Conj,Conj_Verbs_out)].

tense2_is_subset_of_tensel(Tensela,Tenselb,V2,V2_ok) :-
    V2 =.. [Reg_or_Cop,verb(MV2,tense(Tense2),Partic,Neg,Adv,Trans)],
    is_subset(Tense2,Tensela),
    V2_ok =.. [Reg_or_Cop,verb(MV2,tense(Tenselb),Partic,Neg,Adv,Trans)].

/* When Tensel1 is different from Tense2. */
tense2_is_subset_of_tensel(_,_,V2,V2_ok) :-
    access_tense_table(V2,V2_ok,_,_).

/* ..... */

/* Conjoined main verbs. */
tense2_vs_tensel_in_conj_verbs([],[],_,_) :- !.
tense2_vs_tensel_in_conj_verbs([V_H|V_T],[V_H_ok|V_T_ok],T1a,T1b) :-
    V =.. [dummy,V_H], tense2_is_subset_of_tensel(T1a,T1b,V,V_ok),
    V_ok =.. [dummy,V_H_ok], tense2_vs_tensel_in_conj_verbs(V_T,V_T_ok,T1a,T1b).

/* This subset definition is definitely simplistic (but works fine)! */
is_subset(Tense2,Tense2) :- !.
is_subset(Tense2,Tensela) :- length(Tense2,N2), length(Tensela,N1a), N2 < N1a.

/* ..... */
/* VERB_CLAUSE (used by ing_clause, noun, object2_or_action_form, */
/* past_part_phrase, rel_clause, simple_to_infinitive_clause, & */
/* verbphrase_Reg_or_Stat) */
/* ..... */
/* These rules extract the verb sequence with its FINAL modifiers (if any). */

verb_clause(_____,_____,_____,_____) -->
    end_of_sentence_marker(_), (!, fail).

/* General verb clause (TYPE 1). */
verb_clause(1,Num,Voice,Tense,Subj,CPL,Struct,M_el) -->
    already_parsed(verb_clause1,[Num,Voice,Tense,Subj,CPL,Struct,M_el]).

verb_clause(1,Num,Voice,Tense,Subj,CPL,Struct,M_el,I,O) :-
    check_spy_on_this_rule(verb_clause,verb_clause1,I,I),
    check_chrono(verb_clause1,I), there_must_be_a_verb_here(I),
    verbphrase_RegAct_or_RegPass(Num,Voice,Tense,Pre_V,V,
                                Medial_Modif,Compl,M_el,I,O2),
    ((O = O2, Final_Modif = []) ;
     (opt_pause(O2,O3), required_subordinate_clause(final,Final_Modif,O3,O))),
    set_struct(Voice,Tense,Pre_V,V,
               Subj,Compl,Medial_Modif,CPL,Final_Modif,Struct),
    keep_parse_in_mem(verb_clause1,[Num,Voice,Tense,Subj,CPL,Struct,M_el],I,O).

```

```

/* Simplified verb clause (TYPE 2): no final modifiers allowed.
verb_clause(2,Num,Voice,Tense,Subj,CPL,Struct,M_el)
    already_parsed(verb_clause2, [Num,Voice,Tense,Subj,CPL,Struct,M_el]).

verb_clause(2,Num,Voice,Tense,Subj,CPL,Struct,M_el,I,O) :-
    check_spy_on_this_rule(verb_clause,verb_clause2,I,I),
    check_chrono(verb_clause2,I), there_must_be_a_verb_here(1),
    verbphrase_RegAct_or_RegPass(Num,Voice,Tense,Pre_V,V,
                                Medial_Modif,Compl,M_el,I,O),
    set_struct(Voice,Tense,Pre_V,V,
               Subj,Compl,Medial_Modif,CPL,[],Struct),
    keep_parse_in_mem(verb_clause2, [Num,Voice,Tense,Subj,CPL,Struct,M_el],I,O).

/*****
/* THERE_MUST_BE_A_VERB_HERE (used by verb_clause &
/*     verbseq_BeGoingTo_or_else)
/*
/*
/* Trying to avoid unnecessary parsing. There must be a verb, a modal, or an
/* auxiliary at this point otherwise the parsing is bound to be rejected.
/*
/* This procedure is also useful for several NP components via verb_clause.
*****/

there_must_be_a_verb_here([_:(special_char:['_/,_'])|T]) :-
    there_must_be_a_verb_here(T).

there_must_be_a_verb_here([H|T]) :-
    adv(_,[H],_), there_must_be_a_verb_here(T).

there_must_be_a_verb_here([H|_]) :- verb(_,[H],_), !.
there_must_be_a_verb_here([H|_]) :- aux(_,[H],_), !.
there_must_be_a_verb_here([H|_]) :- modal_aux(_,[H],_), !.
there_must_be_a_verb_here(_ ) :- !, fail.

/*****
/* SET_STRUCT (used by verb_clause)
*/

/* REGULAR ACTIVE.
set_struct(active,Tense,Pre_V,V,_,CPL,Medial_Modif,CPL,
           Final_Modif,
           predicate(regular(VERB),voice(active),complement(CPL),
                     subord_clauses(medial(Medial_Modif),
                                     final(Final_Modif))))
:-
get_verb_trans(V,Trans), Trans \== stat,
conciliate__pre_v__v(Tense,Pre_V,V,VERB).

/* REGULAR PASSIVE.
set_struct(passive,Tense,Pre_V,V,Subj,Compl_List,Medial_Modif,Compl,
           Final_Modif,
           predicate(regular(VERB),voice(passive),complement(COMPL),
                     subord_clauses(medial(Medial_Modif),
                                     final(Final_Modif))))
:-
get_verb_trans(V,Trans), Trans \== stat,
conciliate__pre_v__v(Tense,Pre_V,V,VERB),
Compl_List =.. [Compl_Type,[Subj|CPL]], append(CPL,[Compl],C),
COMPL =.. [Compl_Type,C].

/* STATIVE.
set_struct(stative,Tense,Pre_V,V,_,CPL,Medial_Modif,CPL,
           Final_Modif,
           predicate(copula(VERB),s_qualifier(CPL),
                     subord_clauses(medial(Medial_Modif),

```

```

                                final(Final_Modif))) :-
get_verb_trans(V,stat), conciliate__pre_v__v(Tense,Pre_V,V,VERB).

/*****
/* CONCILIATE __PRE_V__V (used by set_struct) */
/*
/* This procedure "conciliates" (or "unifies") negation markers and adverbs
/* that were found before (thus the pre_v name for "pre-verbal") the main
/* verb(s) (e.g. before the aux. group) with the ones found right beside it. */

/* The trivial case: a single verb. */
conciliate__pre_v__v(Tense,pre_v(Neg,Advs),
                    verb(V,_,P,neg(Neg1),advs(Advs1),Tr),
                    verb(V,tense(Tense),P,neg(Neg2),advs(Advs2),Tr))
                    :- !, resolve_neg(Neg,Neg1,Neg2), append(Advs,Advs1,Advs2).

/* Otherwise, conjoined verbs (see verbseq_core): we assume that the scope
/* of pre-verbal negation 'Neg' and adverbials 'Advs' is the whole sequence
/* of conjoined verbs unless they are joined by the coordinator 'but'. In
/* that case, Neg and Advs only affect the very first verb. */

conciliate__pre_v__v(Tense,pre_v(Neg,Advs),
                    conj_verbs(but,
                                [verb(V,_,P,neg(Neg1),advs(Advs1),Tr)|R1]),
                    conj_verbs(but,
                                [verb(V,tense(Tense),P,neg(Neg2),advs(Advs2),Tr)|R2]))
                    :-
                    !, resolve_neg(Neg,Neg1,Neg2), append(Advs,Advs1,Advs2),
                    neg_and_advs__conj_v_scope(Tense,_,nil,R1,R2).

conciliate__pre_v__v(Tense,pre_v(Neg,Advs),conj_verbs(C,L1),conj_verbs(C,L2))
                    :-
                    neg_and_advs__conj_v_scope(Tense,Neg,Advs,L1,L2).

/* ..... */

neg_and_advs__conj_v_scope(Tense,Neg,Advs,
                            [verb(V,_,P,neg(Neg1),advs(Advs1),Tr)|R1],
                            [verb(V,tense(Tense),P,neg(Neg2),advs(Advs2),Tr)|R2])
                    :-
                    resolve_neg(Neg,Neg1,Neg2),
                    ((Advs \== nil, append(Advs,Advs1,Advs2)) ; (Advs == nil, Advs2 = Advs1)),
                    ((Advs == [], ADVS = Advs1) ; (Advs \== [], ADVS = Advs)),
                    neg_and_advs__conj_v_scope(Tense,Neg,ADVS,R1,R2).

neg_and_advs__conj_v_scope(.,.,.,[],[]).

/*****
/* RESOLVE_NEG (used by be_going_to_g, conciliate__pre_v__v,
/* neg_and_advs__conj_v_scope, verbseq, verbseq_BeGoingTo_or_else, &
/* verbseq_Have_or_Do_aux) */
/*
/* Negation unification: yields to 'yes' iff both subcomponents are 'yes'. */

resolve_neg(yes,yes,yes) :- !.
resolve_neg(.,.,no).

/*****
/* VERBPHRASE_REGACT_OR_REGPASS (used by verb_clause) */
/*
/* Handles the regular verb phrases, active or passive. When M_el is present
/* as last argument, it signals the presence of an empty NP (see rel_clause).*/

verbphrase_RegAct_or_RegPass(Num,Voice,Tense,Pre_V,V,Medial_Modif,CPL,M_el)

```

```

-->
verbseq(Num,Voice,Tense,Pre_V,V),
medial_vp_subordinate_clause(Medial_Modif),
complement(Voice,V,M_el,CPL).

medial_vp_subordinate_clause(nil)          --> [].

medial_vp_subordinate_clause(subordinate(MSC)) -->
    med_correlative_marks_o(N),
    (adverbial_subordinate(medial, MSC) ; nominal_subordinate(medial, MSC)),
    med_correlative_marks_c(N).

/* ..... */

med_correlative_marks_o(1) --> comma.
med_correlative_marks_o(2) --> dashes.
med_correlative_marks_o(3) --> parenthesis_o.

med_correlative_marks_c(1) --> comma.
med_correlative_marks_c(2) --> dashes.
med_correlative_marks_c(3) --> parenthesis_c.

/* ***** */
/* VERBSEQ (used by verbphrase_RegAct_or_RegPass) */
/*
/* The basic form of a verb sequence is based on Winograd's description:
/* "modal_aux | be_going_to | have/do | be_1 | be_2 | verb"
/* be_1: information about the continuity of the action.
/* be_2: passive or active form.
/*
/* Ou encore: i) modal + be + verbe en -ing
/*             ii) modal + have + participe passe
/*             iii) modal + have been + participe present.
/*
/* All these different elements, except the verb of course, are optional:
/* see QUIRK, chap.3; on marginal modals & modal idioms, see pp.138+.
/*

/* The verb sequence contains a "HAD BETTER" modal idiom.
verbseq(.,Voice,[had_better,pres|Tense],pre_v(Neg,Adv),V) -->
    check_chrono(verbseq),
    (modal_aux(had,.) ; modal_aux(had_would,.)),
    adj(good,comp), opt_negation(Neg1),
    verbseq__BeGoingTo_or_else(2,.,Voice,Tense,pre_v(Neg2,Adv),V),
    (resolve_neg(Neg1,Neg2,Neg)).

/* The verb sequence contains a "WOULD RATHER" modal idiom.
verbseq(.,Voice,[would_rather,pres|Tense],pre_v(Neg,Adv),V) -->
    (modal_aux(would,.) ; modal_aux(had_would,.)),
    adv(rather,.), opt_negation(Neg1),
    verbseq__BeGoingTo_or_else(2,.,Voice,Tense,pre_v(Neg2,Adv),V),
    (resolve_neg(Neg1,Neg2,Neg)).

/* The verb sequence contains a modal (different from the marginals below).
verbseq(.,Voice,[Modal,Modality|Tense],pre_v(Neg,Adv),V) -->
    modal_aux(Modal,Modality),
    (Modal \== dare, Modal \== need, Modal \== ought_to, Modal \== used_to),
    opt_adv_negation(Adv1,Neg1),
    verbseq__BeGoingTo_or_else(2,.,Voice,Tense,pre_v(Neg2,Adv2),V),
    (append(Adv1,Adv2,Adv), resolve_neg(Neg1,Neg2,Neg)).

/* The verb sequence contains a "DARE" marginal modal.
verbseq(.,Voice,[dare,Modality|Tense],pre_v(Neg,Adv),V) -->
    modal_aux(dare,Modality), opt_negation(Neg1),

```

```

verbseq__BeGoingTo_or_else(2,_,Voice,Tense,pre_v(Neg2,Adv),V),
(resolve_neg(Neg1,Neg2,Neg)).

/* The verb sequence contains a "NEED" marginal modal. */
verbseq(,Voice,[need,Modality|Tense],pre_v(Neg,Adv),V) -->
modal_aux(need,Modality), opt_negation(Neg1),
verbseq__BeGoingTo_or_else(2,_,Voice,Tense,pre_v(Neg2,Adv),V),
(resolve_neg(Neg1,Neg2,Neg)).

/* The verb sequence contains a "OUGHT TO" marginal modal. */
verbseq(,Voice,[ought_to,Modality|Tense],pre_v(Neg,Adv),V) -->
modal_aux(ought_to,Modality), opt_negation(Neg1), infinitive_marker,
verbseq__BeGoingTo_or_else(2,_,Voice,Tense,pre_v(Neg2,Adv),V),
(resolve_neg(Neg1,Neg2,Neg)).

/* The verb sequence contains a "USED TO" marginal modal. */
verbseq(,Voice,[used_to,Modality|Tense],pre_v(Neg,Adv),V) -->
modal_aux(used_to,Modality), opt_negation(Neg1), infinitive_marker,
verbseq__BeGoingTo_or_else(2,_,Voice,Tense,pre_v(Neg2,Adv),V),
(resolve_neg(Neg1,Neg2,Neg)).

/* The verb sequence does not contain a modal. */
verbseq(Num,Voice,Tense,Pre_V,V) -->
verbseq__BeGoingTo_or_else(1,Num,Voice,Tense,Pre_V,V).

/*****
/* INFINITIVE_MARKER (used by be_going_to, have_to, rel_clause, */
/* simple_to_infinitive_clause, verbseq, verbseq__Be__aux, & */
/* verbseq__Have_or_Do__aux) */
infinitive_marker --> special_function_word(to,_).

/*****
/* VERBSEQ__BEGOINGTO_OR_ELSE (used by verbseq) */
/* The "BE_GOING_TO" and "HAVE_TO" cases. */
verbseq__BeGoingTo_or_else(,Num,Voice,Tense,Pre_V,V,I,O) :-
  (be_going_to(Num,Tense1,Neg1,Adv1,I,O2) ;
   have_to(Num,Tense1,Neg1,Adv1,I,O2)),
  there_must_be_a_verb_here(O2),
  ((verbseq__HaveBe__aux(,Voice,[inf,contin],pre_v(Neg2,Adv2),V,O2,O),
    Tense2 = [contin], append(Adv1,Adv2,Adv),
    resolve_neg(Neg1,Neg2,Neg), Pre_V = pre_v(Neg,Adv)) ;
   (verbseq_core(,Voice,[inf],V,O2,O),
    Tense2 = [simple], Pre_V = pre_v(Neg1,Adv1))),
  append(Tense1,Tense2,Tense).

/* The GENERAL case ... */
verbseq__BeGoingTo_or_else(Modal_flag,Num,Voice,Tense,Pre_V,V) -->
verbseq__Have_or_Do__aux(Modal_flag,Num,Voice,Tense,Pre_V,V).

/*****
/* BE_GOING_TO & HAVE_TO (both used by verbseq__BeGoingTo_or_else) */
/* The "BE_GOING_TO" (near future) & */
/* "HAVE_TO" (obligation or logical necessity) SEMI-AUXILIARIES: */
/* see QUIRK, pp.143+ (semi-auxiliaries). */

be_going_to(Num,Tense,Neg,Adv) -->
  adverbs(Adv1), be_going_to_g(Neg,Num,Adv2,Tense),
  be_going_to_mainvb, infinitive_marker, {append(Adv1,Adv2,Adv)}.

be_going_to_g(Neg,Num,Adv,[Tense,perf,near_fut]) -->

```

```

    aux(have,Num,Tense), opt_advs_negation(Advs,Neg1),
    aux(be,_,perf), opt_negation(Neg2), {resolve_neg(Neg1,Neg2,Neg)}.

be_going_to_g(Neg,Num,Advs,[Tense,near_fut]) -->
    aux(be,Num,Tense), opt_advs_negation(Advs,Neg).

be_going_to_mainvb([dct(word:going/_._)|O,O].

/* ..... */

have_to(Num,[Tense],Neg,Advs) -->
    adverbs(Advs1), aux(have,Num,Tense), opt_advs_negation(Advs2,Neg),
    infinitive_marker, {append(Advs1,Advs2,Advs)}.

/* ***** */
/* VERBSEQ_HAVE_OR_DO_AUX (used by verbseq_BeGoingTo_or_else), */
/* VERBSEQ_HAVEBE_AUX (used by verbseq_BeGoingTo_or_else & */
/* verbseq_Have_or_Do_aux), */
/* & VERBSEQ_BE_AUX (used by verbseq_Have_or_Do_aux & */
/* verbseq_HaveBe_aux) */
/* ----- HAVE as an auxiliary ----- */
/* Modal_flag = 1 : for NON-MODALS. */
verbseq_Have_or_Do_aux(1,Num,Voice,[Tense1,perf|Tense2],
    pre_v(Neg,Advs),V) -->
    adverbs(Advs1), aux(have,Num,Tense1), {Tense1 \== inf},
    opt_advs_negation(Advs2,Neg1),
    verbseq_HaveBe_aux(Num,Voice,[perf|Tense2],pre_v(Neg2,Advs3),V),
    {append(Advs1,Advs2,Advs4), append(Advs4,Advs3,Advs),
    resolve_neg(Neg1,Neg2,Neg)}.

/* Modal_flag = 2 : for MODALS. */
verbseq_Have_or_Do_aux(2,Num,Voice,[perf|Tense],pre_v(Neg,Advs),V) -->
    adverbs(Advs1), aux(have,Num,inf), opt_advs_negation(Advs2,Neg1),
    verbseq_HaveBe_aux(Num,Voice,[perf|Tense],pre_v(Neg2,Advs3),V),
    {append(Advs1,Advs2,Advs4), append(Advs4,Advs3,Advs),
    resolve_neg(Neg1,Neg2,Neg)}.

/* ----- DO as an auxiliary ----- */
verbseq_Have_or_Do_aux(_,Num,Voice,[Tense],pre_v(Neg,Advs),V) -->
    adverbs(Advs1), aux(do,Num,Tense), {Tense \== perf},
    opt_advs_negation(Advs2,Neg),
    verbseq_core(_,Voice,[inf],V), {append(Advs1,Advs2,Advs)}.

/* No HAVE or DO auxiliaries found in the verb sequence ... */
verbseq_Have_or_Do_aux(Modal_flag,Num,Voice,Tense,Pre_V,V) -->
    verbseq_Be_aux(Modal_flag,Num,Voice,Tense,Pre_V,V).

/* ----- BE as an auxiliary ----- */
/* first two clauses: when following HAVE ... */
verbseq_HaveBe_aux(Num,Voice,[Tense,contin],pre_v(Neg,Advs),V) -->
    aux(be,Num,Tense), opt_advs_negation(Advs,Neg),
    verbseq_core(Num,Voice,[progr],V).

verbseq_HaveBe_aux(Num,Voice,Tense,Pre_V,V) -->
    verbseq_Be_aux(_,Num,Voice,Tense,Pre_V,V).

/* ..... */
/* Modal_flag = 1 : for NON-MODALS; the "BE TO" modal idiom. */

```



```

verbseq__Be__aux(1,Num,Voice,[be_to,Tense,pres],pre_v(Neg,Adv),V) -->
    adverbs(Adv),
    aux(be,Num,Tense), {(Tense == past ; Tense == pres ; Tense == pres3s)},
    opt_adv_negation([],Neg), infinitive_marker,
    verbseq_core(_,Voice,[inf],V).

/* Modal_flag = 1 : for NON-MODALS, the general case. */
verbseq__Be__aux(1,Num,Voice,[Tense,contin],pre_v(Neg,Adv),V) -->
    adverbs(Adv1), aux(be,Num,Tense), {Tense \== inf},
    opt_adv_negation(Adv2,Neg),
    verbseq_core(Num,Voice,[progr],V), {append(Adv1,Adv2,Adv)}.

/* Modal_flag = 2 : for MODALS. */
verbseq__Be__aux(2,Num,Voice,[pres,contin],pre_v(Neg,Adv),V) -->
    adverbs(Adv1), aux(be,Num,inf), opt_adv_negation(Adv2,Neg),
    verbseq_core(Num,Voice,[progr],V), {append(Adv1,Adv2,Adv)}.

/* ----- No auxiliaries found ! ----- */
verbseq__Be__aux(_,Num,Voice,Tense,pre_v(yes,[]),V) -->
    verbseq_core(Num,Voice,Tense,V).

/* ***** */
/* VERBSEQ_CORE (used by verbseq__Be__aux, verbseq__BeGoingTo_or_else, */
/* verbseq__Have_or_Do__aux, & verbseq__HaveBe__aux) */
/* */
/* These rules recognize the simple (usual) case in which a VP contains a */
/* single main verb. They also recognize the more complex case in which a */
/* sequence of conjoined main verbs is used instead. Although the latter can */
/* also be analysed in terms of conjoined verb phrases with several elided */
/* constituents, we follow the analysis of QUIRK (see pp.943+ & pp.967+) and */
/* consider it as a case of conjoined main verbs with, often, an implied */
/* chronological sequence. Here are two typical examples taken from QUIRK: */
/* Ex.1: "Paul WASHED AND DRIED the dishes." */
/* Ex.2: "He WAS BORN, LIVED, AND DIED in the city of Vienna." */

verbseq_core(Num,Voice,[Tense],V) -->
    check_chrono(verbseq_core),
    verbal(Num,Voice,Tense,Vb1), rest_verbseq_core(Num,Voice,Coor,Vb_R),
    {pack_verbseq_core(Vb1,Coor,Vb_R,V)}.

/* ..... */

rest_verbseq_core(Num,Voice,Coor,[Vb2|Vb_R]) -->
    opt_comma, real_coordinator_3(Coor), verbal(Num,Voice,_,Vb2),
    rest_verbseq_core(Num,Voice,Coor,Vb_R), {!}.
rest_verbseq_core(Num,Voice,Coor,[Vb2|Vb_R]) -->
    comma, verbal(Num,Voice,_,Vb2),
    rest_verbseq_core(Num,Voice,Coor,Vb_R), {Vb_R \== [], !}.
rest_verbseq_core(_,_,_,[]) --> [].

pack_verbseq_core(Vb1,_,[],Vb1) :- !.
pack_verbseq_core(Vb1,Coor,Vb_R,conj_verbs(Coor,[Vb1|Vb_R])) :-
    nonvar(Coor), last(Last_V,Vb_R), get_verb_trans(Last_V,Last_V_trans),
    append(Vbs_but_last,[Last_V],[Vb1|Vb_R]),
    check_trans_compatibility(Vbs_but_last,Last_V_trans).

/* ..... */

/* Making sure that we are not dealing with incompatible verbs, as far as */
/* their transitivity is concerned. Otherwise, they should be considered as */
/* conjoined VPs, not conjoined verbs. See also get_verb_trans below. */

```

```

check_trans_compatibility(_,Trans)    :- Trans \== intr, Trans \== tr, !.
check_trans_compatibility(VbsL,intr)  :- not_incompatible_intr_vs_tr(tr,VbsL).
check_trans_compatibility(VbsL,tr)    :- not_incompatible_intr_vs_tr(intr,VbsL).

not_incompatible_intr_vs_tr(_,[])      :- !.
not_incompatible_intr_vs_tr(T,[V_H|_]) :-
    get_verb_trans(V_H,Trans), Trans == T, !, fail.
not_incompatible_intr_vs_tr(T,[V_H|V_R]) :-
    get_verb_trans(V_H,Trans), Trans \== T, not_incompatible_intr_vs_tr(T,V_R).

/* ***** */

/*                                     ACTIVE VERB                                */
verbal(Num,active,Tense,
       verb(MainVb,tense(Tense),partic(Par),neg(Neg),advs(Advs),trans(Trans)))
-->
    adverbs(Advs),
    verb__in_double_quotes_or_not(Tense,Trans,Num,MainVb),
    opt_particle(MainVb,Par), opt_negation(Neg),
    (check_Num_Tense_agreement(Num,Tense)).

/*                                     "PASSIVE" VERB                             */
verbal(Num,passive,Tense,
       verb(MainVb,tense(Tense),partic(Par),neg(Neg),advs(Advs),trans(Trans)))
-->
    adverbs(Advs), aux(be,Num,Tense), opt_negation(Neg),
    verb__in_double_quotes_or_not(perf,Trans,Num,MainVb),
    opt_particle(MainVb,Par),
    (check_Num_Tense_agreement(Num,Tense)).

/*                                     STATIVE VERB                               */
verbal(Num,stative,Tense,
       verb(MainVb,tense(Tense),partic(nil),neg(Neg),advs(Advs),trans(stat)))
-->
    adverbs(Advs),
    ((verb__in_double_quotes_or_not(Tense,stat,Num,MainVb),
      opt_negation(Neg)) ;
     (opt_negation(Neg),
      verb__in_double_quotes_or_not(Tense,stat,Num,MainVb))).
    (check_Num_Tense_agreement(Num,Tense)).

/* ..... */

check_Num_Tense_agreement(Num,_)      :- var(Num), !.
check_Num_Tense_agreement(sg3,pres3s) :- !.
check_Num_Tense_agreement(sg3,Tense)  :- Tense \== pres, Tense \== inf, !.
check_Num_Tense_agreement(Num,Tense)  :- Num \== sg3, Tense \== pres3s.

/* ***** */
/* VERB__IN_DOUBLE_QUOTES_OR_NOT (used by past_part, pres_part, & verbal) */
/*
/* We recognize verbs enclosed in double quotes (e.g. "tells", "enter").
/* A word W in double quotes is represented as double_quotes(W),
/* but a sequence of words SW in double quotes is represented as string(SW). */

verb__in_double_quotes_or_not(Tense,Trans,Num,double_quotes(V)) -->
    (Tense == imp), double_quotes, verb(imp,Trans,Num,V), double_quotes, (!).

verb__in_double_quotes_or_not(Tense,Trans,Num,double_quotes(V)) -->
    double_quotes, verb(Tense,Trans,Num,V), (Tense \== imp), double_quotes.

verb__in_double_quotes_or_not(Tense,Trans,Num,V) -->

```

```

{Tense == imp}, verb(imp,Trans,Num,V), {}).

verb__in_double_quotes_or_not(Tense,Trans,Num,V)          -->
    verb(Tense,Trans,Num,V), {Tense \== imp}.

/*****
/* COMPLEMENT (used by verbphrase_RegAct_or_RegPass)      */
/*                                                         */
/* Defines the different types of complement accepted by a verb sequence: */
/*     intr = intransitive : verbs that do not occur with objects.          */
/*     stat = stative      : copulas (see verb_clause & set_struct).         */
/*     tr  = transitive    : verbs that occur with one or more object.       */
/*     tr_intr : verbs that can be used either as intr or as tr.            */
*****/

complement(_,_ ,M_el,nil)      --> {M_el == ing_noun, !}).

complement(Voice,V,M_el,CPL) -->
    check_chrono(complement(tr)),
    {get_verb_trans(V,Trans), (Trans == tr_intr ; Trans == tr)},
    check_spy_on_this_rule(complement,complement_tr),
    complement_tr(Voice,M_el,CPL).

complement(Voice,V,M_el,CPL) -->
    check_chrono(complement(intr)),
    {get_verb_trans(V,Trans), (Trans == tr_intr ; Trans == intr), !},
    check_spy_on_this_rule(complement,complement_intr),
    complement_intr(Voice,M_el,CPL).

complement(Voice,V,M_el,CPL) -->
    check_chrono(complement(stat)),
    {get_verb_trans(V,stat)},
    check_spy_on_this_rule(complement,complement_stat),
    complement_stat(Voice,M_el,CPL).

/*****
/* GET_VERB_TRANS (used by complement, not_incompat. le_intr_vs_tr,      */
/*     pack_verbseq_core, & set_struct)                                     */
*****/

/* The trivial case: a single verb.                                         */
get_verb_trans(verb(_,_ ,_ ,_ ,_ ,trans(Trans)),Trans).

/* Otherwise, conjoined verbs: we assume that the last verb determines Trans.*/
get_verb_trans(conj_verbs(_ ,Vlist),Trans) :-
    last(verb(_,_ ,_ ,_ ,_ ,trans(Trans)),Vlist).

/* *****
/* TR (i.e. transitives, ditransitives, & complex transitives)          */
/*                                                         */
/* Some rules apply to ditransitives as well as to complex transitives;   */
/* see QUIRK, pp.1170+ (16.20: "Types of verb complementation").          */
/*                                                         */
/* Notation: S(ubject), V(erb), O(bject), C(omplement), A(dverbial).       */
*****/

/* ----- ACTIVE STRUCTURES ----- */

complement_tr(active,referring_np(M_el),svoo_svoc_svoa([M_el,X])) -->
    np_equiv(post_v,2,X,_), {}).

complement_tr(active,referring_np(M_el),svoo_svoc_svoa([M_el,X])) -->
    required_prepphrases(X,_), {}).

complement_tr(active,referring_np(M_el),svo(M_el)) -->
    [].

```

B.1: DIPETT_6_PARSER_S_V.PL ("sentence & verb phrase rules")

```

complement_tr(active,def_prep(M_el),svoo_svoc_svoa([Obj,pp(P,M_el)])) -->
  np_equiv(post_v,2,Obj,_), prep(P), not_followed_by_a_np.

complement_tr(active,def_prep(M_el),svo(pp(P,M_el))) -->
  prep(P), not_followed_by_a_np.

complement_tr(active,nil,svoc(Compar)) -->
  comparative_clause(Compar).

complement_tr(active,nil,svo(X)) -->
  np_equiv(post_v,0,X,_); required_prepphrases(X,_).

complement_tr(active,nil,Compl_Struct) -->
  np_equiv(post_v,2,Obj,_),
  rest_complement_tr_act_np(Obj,Compl_Struct).

complement_tr(active,nil,svoo_svoc_svoa([Obj,PP_obj])) -->
  required_prepphrases(PP_obj,_), np_equiv(post_v,2,Obj,_).

/* ----- PASSIVE STRUCTURES ----- */

complement_tr(passive,referring_np(M_el),svoo([Agent,M_el])) -->
  by_phrase(Agent); {Agent = pass_subj(nil)}.

complement_tr(passive,nil,Compl_Struct) -->
  by_phrase(Agent), rest_complement_tr_pass_by(Agent,Compl_Struct).

complement_tr(passive,nil,Compl_Struct) -->
  required_prepphrases(Iobj,by),
  rest_complement_tr_pass_pp(Iobj,Compl_Struct).

complement_tr(passive,nil,Compl_Struct) -->
  np_equiv(post_v,0,Obj,_),
  rest_complement_tr_pass_np(Obj,Compl_Struct).

complement_tr(passive,nil,svoo_svoc_svoa([Agent,attributes(A)])) -->
  required_adj_phrase(A), by_phrase(Agent).

/* ..... */

rest_complement_tr_act_np(Obj,svoc([Obj,Compar])) -->
  comparative_clause(Compar).

rest_complement_tr_act_np(Iobj,svoo_svoc_svoa([X,iobj(Iobj)])) -->
  np_equiv(post_v,0,X,_).

rest_complement_tr_act_np(Obj,svoo_svoc_svoa([Obj,X])) -->
  required_prepphrases(X,_).

rest_complement_tr_act_np(Obj,svoc_svoa([Obj,X])) -->
  required_adj_phrase(X).

/* ..... */

rest_complement_tr_pass_by(Agent,svoo_svoc_svoa([Agent,X])) -->
  np_equiv(post_v,0,X,_); required_prepphrases(X,_), (!).
rest_complement_tr_pass_by(Agent,svo([Agent,nil])) --> (!).

rest_complement_tr_pass_pp(Iobj,svoo_svoc_svoa([Agent,iobj(Iobj)])) -->
  by_phrase(Agent), (!).
rest_complement_tr_pass_pp(Iobj,svoo([pass_subj(nil),iobj(Iobj)])) --> (!).

rest_complement_tr_pass_np(Obj,svoo_svoc_svoa([Agent,Obj])) -->
  by_phrase(Agent), (!).

```

```

rest_complement_tr_pass_np(Obj,svoo([pass_subj(nil),Obj])) --> [].

/* *****
/* Note: in fact this might be too restrictive. What we really mean is "not
/* followed by something that can ONLY function as a NP". Yes, tricky indeed.*/

not_followed_by_a_np(I,I) :-
    (noungp_head(_,_,_I,_)) ; np_nucleus(_,_,_I,_), required_adverbs(_I,_), !.
not_followed_by_a_np(I,I) :- noungp_head(_,_,_I,_), !, fail.
not_followed_by_a_np(I,I) :- np_nucleus(_,_,_I,_), !, fail.
not_followed_by_a_np(I,I) :- !.

/* Note: this rule recognizes "by Tom and Bob" but not "by Tom and by Bob". */
by_phrase(pass_subj(Agent)) --> prep(by), np_equiv(post_v,2,Agent,_).

/* *****
/* INTR (i.e. intransitives) */

complement_intr(active,nil,sv(nil)) --> [].

/* *****
/* STAT (i.e. statives or copulars) */

complement_stat(stative,referring_np(M_el),svc_sva(M_el)) --> [].

complement_stat(stative,nil,svc_sva(A)) -->
    compl_adj_phrase(A).

complement_stat(stative,nil,svc_sva(Obj)) -->
    np_equiv(pre_v,0,Obj,_).

/* *****
/* REQUIRED_SUBORDINATE_CLAUSE (used by verb_clause) */
/* & SUBORDINATE_CLAUSE (used by required_subordinate_clause & sentence) */
/*
/* Sentence modifiers may occur at the beginning or at the end of a sentence */
/* or a verbphrase. They generally affect the meaning of the sentence by */
/* specifying particular circumstances, time or manner constraints on the */
/* main clause. Thus, relative to the latter, they are subordinate clauses. */
/*
/* Note: "modifiers" and "subordinate clauses" are used as synonyms here. */

required_subordinate_clause(POS,RCM) -->
    subordinate_clause(POS,RCM), {RCM \== []}.

subordinate_clause(initial,_) -->
    end_of_sentence_marker(_), {!, fail}.
subordinate_clause(final,[I,I,I]) :-
    end_of_sentence_marker(_I,_), !.
subordinate_clause(_,[]) --> [].
subordinate_clause(POS,Out_Struct) $ Tokens -->
    modif(POS,M1), rest_sub_cl(POS,Coor,M_R,M1), {pack_sub_cl(M1,Coor,M_R,M_L)},
    {get_desired_output_structure(M_L,Tokens,Out_Struct)}.

/* ..... */

/* The following rules require that an initial "modifier" must be followed by */
/* a comma if it's not a simple adverbial clause. Otherwise, it's optional. */

rest_sub_cl(POS,Coor,[M2|M_R],_)
    -->
    opt_comma, opt_real_coordinator(Coor), modif(POS,M2),

```

```

    rest_sub_cl(POS,Coor,M_R,M2), []).
rest_sub_cl(initial,_,[],ph(subordinate(adverbial(simple_adv_clause(_)),_))
    -->
    [].
rest_sub_cl(initial,_,[],_)
    -->
    comma.
rest_sub_cl(final,_,[],_)
    -->
    opt_comma.

pack_sub_cl(M1,_,[],[M1]) :- !.
pack_sub_cl(M1,Coor0,M_R,conj_subord_cl(Coor,[M1|M_R])) :-
    (var(Coor0), Coor = implicit_and) ; (nonvar(Coor0), Coor = Coor0).

/.....*/
/* MODIF (used by rest_sub_cl & subordinate_clause) */
modif(POS,subordinate(M)) -->
    already_parsed(subordinate_clause,[POS,M]).

modif(POS,subordinate(M),I,O) :-
    check_spy_on_this_rule(subordinate_clause,nil,I,I), check_chrono(modif,I),
    modif_phrase(POS,M,I,O),
    keep_parse_in_mem(subordinate_clause,[POS,M],I,O).

/* Major categories of subordinate clauses: adverbial and nominal clauses. */
/* Others are sentential relative & comparative (see QUIRK, pp.1047+). */
/* Note: sentential relative is syntactically similar to nominal relative */
/* clause (though semantically different), thus, not included here. */
/* For more details, see QUIRK p.1118. On subordinators, see QUIRK, pp.998+. */

modif_phrase(POS,Out_Struct) $ Tokens -->
    comparative_subordinate(POS,M),
    (get_desired_output_structure(M,Tokens,Out_Struct)).
modif_phrase(POS,Out_Struct) $ Tokens -->
    adverbial_subordinate(POS,M),
    (get_desired_output_structure(M,Tokens,Out_Struct)).
modif_phrase(POS,Out_Struct) $ Tokens -->
    nominal_subordinate(POS,M),
    (get_desired_output_structure(M,Tokens,Out_Struct)).

comparative_subordinate(initial,comparative(M)) -->
    comparative_clause(M).

adverbial_subordinate(_,adverbial(M)) -->
    adverbial_clause(M).
adverbial_subordinate(_,adverbial(M)) -->
    adv_group(M).
adverbial_subordinate(initial,adverbial(subordinator(Sub_Conj),M)) -->
    opt_subord_conj(Sub_Conj), required_prepphrases(M,_).

nominal_subordinate(initial,nominal(subordinator(Sub_Conj),M)) -->
    opt_subord_conj(Sub_Conj), nominal_clause(_,M,_).

/* ..... */

opt_subord_conj(Sub_Conj) -->
    conj(Sub_Conj), {\+ real_coordinator(Sub_Conj), !}.

```

```

opt_subord_conj(Sub_Conj) -->
    adv(Sub_Conj,Type), (Type == conj, !).
opt_subord_conj(nil)      --> [].

/*.....
/* ADVERBIAL_CLAUSE & ADV_GROUP (both used by adverbial_subordinate) */

adverbial_clause(adv_clause(subordinator(Adv_or_Conj),Clause)) -->
    (required_adverbs(Adv_or_Conj) ;
    (opt_subord_conj(Adv_or_Conj), (Adv_or_Conj \== nil))),
    rest_adverbial_clause(Clause).

rest_adverbial_clause(Clause) -->
    nominal_clause(ing_clause,Clause,_,_).
rest_adverbial_clause(Clause) -->
    sentence_core_s(Clause).
rest_adverbial_clause(past_part_phrase(Agent,PPP)) -->
    past_participle_phrase(.,main_clause_subj,Agent,PPP).

/* ..... */

adv_group(simple_adv_clause(Adv)) --> required_adverbs(Adv).
adv_group(comp_adv_clause(Comp,Object)) -->
    compare_p_adv(Comp),
    (np_equiv(post_v,2,Object,_) ; sentence_core_s(Object)).

compare_p_adv(comparator__than(Deg_Mark,Adv)) -->
    compar_degree_marker(Deg_Mark), adv_s(Adv), conj(than).
compare_p_adv(comparator__than(Deg_Mark,Adj)) -->
    compar_degree_marker(Deg_Mark), adj(Adj,comp), conj(than).

/*.....
/* SENTENCE_CORE_S (used by adv_group, comparative_clause, question, */
/* rest_adverbial_clause, that_clause, wh_i_d_clause & whether_if_clause) */
/* ..... */
/* Similar to 'sentence_core' except that we do not allow subordinate_clause */
/* or np_equiv of type 1 in order to avoid potential infinite left recursion.*/

sentence_core_s(Struct) -->
    check_chrono(sentence_core_s),
    np_equiv(pre_v,2,Np,Num), verbphrase(2,Num,_,Np,[],Struct,nil).

/*.....
/* COMPARATIVE_CLAUSE (used by comparative_subordinate, compl_adj_phrase, */
/* complement_tr, & rest_complement_tr_act_np) */
/* ..... */
/* See QUIRK, pp.1127+. */

comparative_clause(_)
    -->
    end_of_sentence_marker(_), (!, fail).

/* 'THAN' comparison: " ... a much better student than your own brother." */
comparative_clause(comparative_clause__than(
    prep(Prep),degree(Deg_Mark),compar__than(Adj_C),
    compar_obj1(Obj1), compar_obj2(Obj2)))
    -->
    (prep(Prep) ; {Prep = nil}),
    compar_degree_marker(Deg_Mark), compar_adj(Deg_Mark,Adj_C),
    (np_equiv(post_v,2,Obj1,_) ; {Obj1 = nil}),
    conj(than), (sentence_core_s(Obj2) ; np_equiv(post_v,2,Obj2,_)).

```

```

/* 'AS-AS' comparison: " ... as intelligent as your own brother." */
comparative_clause(comparative_clause__as_as(
    degree(Deg_Mark),compar__as_as(Adj_C),
    compar_obj(Obj)))
-->
adv(as,_),
compar_degree_marker(Deg_Mark),
(compar_adj(Deg_Mark,Adj_C) ;
    np_equiv(post_v,2,Adj_C,_)),
adv(as,_),
(sentence_core_s(Obj) ;
    np_equiv(post_v,2,Obj,_)) ;
required_adj_phrase(Obj)).

compar_degree_marker(least) --> quant(least,_,_), {!}.
compar_degree_marker(less) --> quant(less,_,_), {!}.
compar_degree_marker(many) --> quant(many,_,_), {!}.
compar_degree_marker(more) --> quant(more,_,_), {!}.
compar_degree_marker(most) --> quant(most,_,_), {!}.
compar_degree_marker(much) --> quant(much,_,_), {!}.
compar_degree_marker(so) --> adv(so,_), {!}.
compar_degree_marker(nil) --> [].

compar_adj(_,Adj_C) --> adj_s(Adj_C,_).
compar_adj(_,Adj_C) --> adv_s(Adj_C,_).
compar_adj(Deg_Mark,nil) --> {Deg_Mark \== nil}.

/*****
/* LAST_RESORT_PARSING_OPTION (used by parse in dipett_1) */
/*
/* When an input is unrecognizable by the grammar or when it cannot be par-
/* sed inside the allotted CPU time limit, the user can (option G in parsing
/* options) ask DIPETT to reparse the input as a series of fragments. This
/* option is obviously less desirable than a full (correct) parse but much
/* more preferable to a completely unsuccessful parse. This approach to
/* "partial parsing" allows us to ensure, up to a certain point, that DIPETT
/* will at least attempt to extract partial "knowledge" elements out of any
/* well-formed input. This is, in a sense, a last resort parsing strategy. */

/* set_chrono: relative to the original CPU time limit; not a new one. */
last_resort_parsing_option(Frags,I,O) :-
    ((fragmentary_parsing(on),
        writep('** Reparse as a series of fragments [n/Y]? '),
        ((user:batch_parsing(on), Answer = y, nlp) ;
            (\+ user:batch_parsing(on), read_answer(Answer))),
        Answer \== n) ;
        (frag_parsing_only(on), nlp)),
    set_chrono,
    on_exception(Msg,
        with_timing('fragmentary parsing ',
            last_resort_parsing(Frags,I,O)),
        display_exception_message_and_fail__6(Msg)),
    display_fp_percents(Frags).

/* As is the case in 'compl_sentence' above, an input may be parenthetical. */
last_resort_parsing(parenthetical(parse_tree__frag_series(FRAGS,Tag))) -->
    parenthesis_o, {!},
    (erase_previous_mem_assertions),
    frag_series(FRAGS),
    terminal(Tag),
    parenthesis_c.

```



```

last_resort_parsing(parse_tree__frag_series(FRAGS,Tag))      --\
  {erase_previous_mem_assertions},
  frag_series(FRAGS),
  terminal(Tag).

frag_series([Frag_H|Frag_R]) -->
  frag_element(Frag_H), frag_series(Frag_R).
frag_series([])             --> {}.

frag_element(sentence_fragment(FE)) --> sentence_core_s(FE).
frag_element(sentence_fragment(FE)) --> verbphrase(2,_,_,nil,[],FE,nil).
frag_element(FE)                   --> sentence_frag(FE).
frag_element(_)                     --> terminal(_), {!, fail}.
frag_element(skipped_token(ST_ok)) -->
  [dct(_:ST/_,_)].
  {name(ST,ST_a), append([39|ST_a],[39],ST_b), name(ST_ok,ST_b)}.

/*****
/* DISPLAY_FP_PERCENTS (used by last_resort_parsing_option) */

display_fp_percents([]) :- !.
display_fp_percents(F) :-
  display_number_of_frags(F), display_percent_skips(F).

display_number_of_frags(F) :-
  fas(sentence_fragment,F,L), length(L,Number),
  writep('** Number of fragments      : '), writep(Number), nlp,
  ((Number > 0, ctr_inc(nb_fragments_parsed,_)) ;
   ctr_inc(bad_units,_)), !.

display_percent_skips(F) :-
  fas(skipped_token,F,L), length(L,Number),
  writep('** Number of tokens not parsed : '), writep(Number), nlp,
  writep('** % of input string recognized : '),
  sentence_tokens(ST), length(ST,LST),
  Percent is ((LST - Number)/LST) * 100,
  writep(Percent), writep('%'), nlp, nlp.

*****/

```

273

```

    (get_desired_output_structure(NP,Tokens,Out_Struct))).

/* EITHER/OR Nps.
either_or__np(POS,conj_nps(either_or,NUM,[Np_e,Np_o]),NUM) -->
    conj(either), np_nucleus_and_mod(POS,Np_e,Num_e),
    conj(or), np_nucleus_and_mod(POS,Np_o,Num_o),
    (plur_rule_n(Num_e,Num_o,or,NUM))).

/* NEITHER/NOR Nps.
either_or__np(POS,conj_nps(neither_nor,NUM,[Np_e,Np_o]),NUM) -->
    conj(neither), np_nucleus_and_mod(POS,Np_e,Num_e),
    conj(nor), np_nucleus_and_mod(POS,Np_o,Num_o),
    (plur_rule_n(Num_e,Num_o,or,NUM))).

rest_np_nucleus(POS,Np0,Num0,conj_nps(Coor,Num,[Np0,parenthetical(Np_R)]),Num)
-->
    parenthesis_o, conj_np(1,POS,Num0,Coor,Np_R,Num), parenthesis_c, (!).

rest_np_nucleus(POS,Np0,Num0,conj_nps(Coor,Num,[Np0|Np_R]),Num)
-->
    conj_np(1,POS,Num0,Coor,Np_R,Num), (!).

rest_np_nucleus(_,Np0,Num0,Np0,Num0)
-->
    [].

/* ..... */
/* On the N parameter, see remarks in dipett_6 (conj_vp predicate). */

/* Case (1) : NP ('') COOR NP ...
conj_np(N,POS,Num_H,Coor,[Np_H2|Np_R],Num) -->
    ({N == 1} ; ({N \== 1}, opt_comma)), real_coordinator_3(Coor),
    np_nucleus_and_mod(POS,Np_H2,Num_H2), (plur_rule_n(Num_H,Num_H2,Coor,Num)),
    conj_np(_,POS,Num_H,Coor,Np_R,Num).

/* Case (2) : NP '' NP ...
conj_np(_,POS,Num_H,Coor,[p_asyndetic_conj_np(Np_H2)|Np_R],Num) -->
    comma,
    np_nucleus_and_mod(POS,Np_H2,_),
    conj_np(_,POS,Num_H,Coor,Np_R,Num), {Np_R \== []}.

conj_np(_,_,_,Coor,[],_) -->
    check_chrono(conj_np), {nonvar(Coor)}.

/* ..... */
/* PLUR_RULE_N (used by conj_np & either_or__np) */

plur_rule_n(X,Y,or,sg3) :- (X == sg ; X == sg3), (Y == sg ; Y == sg3).
plur_rule_n(X,Y,or,pl) :- X == pl ; Y == pl.
plur_rule_n(_,_,and,pl).
plur_rule_n(_,_,C,Num) :- C \== and, C \== or, (Num = sg3 ; Num = pl).

/* ..... */
/* NOMINAL_CLAUSE (used by adj_complement, apposition_group,
/* nominal_subordinate, np_equiv_form, np_for_pp & rest_adverbial_clause) */
/*
/* On the first parameter: 'CL' is used to constrain the kind of nominal
/* clause when necessary (e.g. it must be a to_infinite_clause). */

nominal_clause(_,_,_,_) -->
    end_of_sentence_marker(_), (!, fail).
nominal_clause(CL,Out_Struct,NUM,COMPL) $ Tokens -->

```

```

    either_or__nom_cl(CL,NC,COMPL), {NUM = sg3 ; NUM = pl ; NUM = sg},
    {get_desired_output_structure(NC,Tokens,Out_Struct)}).
nominal_clause(CL,Out_Struct,NUM,COMPL) $ Tokens -->
    nom_cl(CL,Nc0,COMPL), rest_nom_cl(CL,Nc0,COMPL,NC),
    {NUM = sg3 ; NUM = pl ; NUM = sg},
    {get_desired_output_structure(NC,Tokens,Out_Struct)}).

/* EITHER/OR nominal clauses. */
either_or__nom_cl(CL,conj_nom_cl(either_or,[Nc_e,Nc_o]),COMPL) -->
    conj(either), nom_cl(CL,Nc_e,COMPL), conj(or), nom_cl(CL,Nc_o,COMPL).

/* NEITHER/NOR nominal clauses. */
either_or__nom_cl(CL,conj_nom_cl(neither_nor,[Nc_e,Nc_o]),COMPL) -->
    conj(neither), nom_cl(CL,Nc_e,COMPL), conj(nor), nom_cl(CL,Nc_o,COMPL).

rest_nom_cl(CL,Nc0,COMPL,conj_nom_cl(Coor,[Nc0,parenthetical(Nc_R)])) -->
    parenthesis_o, conj_nom_cl(1,CL,COMPL,Coor,Nc_R), parenthesis_c, (!).

rest_nom_cl(CL,Nc0,COMPL,conj_nom_cl(Coor,[Nc0|Nc_R])) -->
    conj_nom_cl(1,CL,COMPL,Coor,Nc_R), (!).

rest_nom_cl(_,Nc0,_,Nc0) --> [].

/* ..... */
/* On the N parameter, see remarks in dipett_6 (conj_vp predicate). */

/* Case (1) : NP_subst ('') COOR NP_subst ... */
conj_nom_cl(N,CL,COMPL,Coor,[Nc_H2|Nc_R]) -->
    ({N == 1} ; ({N \== 1}, opt_comma)), real_coordinator_3(Coor),
    nom_cl(CL,Nc_H2,COMPL), conj_nom_cl(_,CL,COMPL,Coor,Nc_R).

/* Case (2) : NP_subst '', NP_subst ... */
conj_nom_cl(_,CL,COMPL,Coor,[p_asyndetic_conj_nom_cl(Nc_H2)|Nc_R]) -->
    comma,
    nom_cl(CL,Nc_H2,COMPL), conj_nom_cl(_,CL,COMPL,Coor,Nc_R), {Nc_R \== []}.

conj_nom_cl(_,_,_,Coor,[]) -->
    check_chrono(conj_nom_cl), {nonvar(Coor)}.

/.....*/
/* NOM_CL (used by conj_nom_cl, either_or__nom_cl, & nominal_clause) */
/*
/* Following QUIRK, these are the four most common types of nominal clauses. */

nom_cl(_,_,_) -->
    end_of_sentence_marker(_), (!, fail).

nom_cl(to_infinitive_clause,Out_Struct,_) $ Tokens -->
    check_spy_on_this_rule(nom_cl,nom_to_infinitive_clause),
    to_infinitive_clause(2,NC),
    {get_desired_output_structure(nominal_clause(NC),Tokens,Out_Struct)}.

nom_cl(ing_clause,Out_Struct,_) $ Tokens -->
    check_spy_on_this_rule(nom_cl,nom_ing_clause),
    ing_clause(2,NC),
    {get_desired_output_structure(nominal_clause(NC),Tokens,Out_Struct)}.

nom_cl(wh_interro_declarative_clause,Out_Struct,_) $ Tokens -->
    check_spy_on_this_rule(nom_cl,nom_wh_interdeclar_clause),
    wh_interro_declarative_clause(NC),
    {get_desired_output_structure(nominal_clause(NC),Tokens,Out_Struct)}.

```

```

nom_cl(that_clause,Out_Struct,COMPL) $ Tokens          ---
  check_spy_on_this_rule(nom_cl,nom_that_clause),
  that_clause(NC,COMPL),
  (get_desired_output_structure(nominal_clause(NC),Tokens,Out_Struct)).

/*****
/* TO_INFINITIVE_CLAUSE (used by nom_cl)                */
/*
/* Ex: "TO COMPLETE THIS WORK IN TWO DAYS is definitely impossible."
/* Ex: "Tom would prefer FOR THE CHILDREN TO FINISH THEIR SOUP."
*/

to_infinite_clause(Type,to_infinite_clause(X))          -->
  simple_to_infinite_clause(Type,X).

to_infinite_clause(Type,to_infinite_clause(wh_word(Q),X)) -->
  interro_pron(Q,_), simple_to_infinite_clause(Type,X).

to_infinite_clause(Type,to_infinite_clause(for(Np),X))   -->
  prep(for), np_equiv(pre_v,2,Np,_), simple_to_infinite_clause(Type,X).

/*****
/* SIMPLE_TO_INFINITIVE_CLAUSE (used by adj_complement,  */
/*   real_passive_rel_clause_test, to_infinite_clause, wh_i_d_clause, */
/*   & whether_if_clause)                                */

simple_to_infinite_clause(Type,X) -->
  check_chrono(simple_to_infinite_clause),
  infinite_marker, verb_clause(Type,_,_,[inf],nil,_,X,nil).

/*****
/* ING_CLAUSE (used by nom_cl)                            */
/*
/* Gerundive: when a verb in present participle form is used as a noun.
/* Ex.1: "LAUGHING WITH FRIENDS makes you live longer.",
/* Ex.2: "MY FORGETTING her name was very unfortunate."
/* See QUIRK, pp.1063+ (including the genitive case -- see Ex.2 above).
*/

ing_clause(Type,genitive_ing_clause(DP,X)) -->
  ic_deter_pron(DP), verb_clause(Type,_,_,[progr],nil,_,X,nil).

ing_clause(Type,ing_clause(X)) -->
  verb_clause(Type,_,_,[progr],nil,_,X,nil).

ic_deter_pron(deter_pron(DP)) -->
  deter(DP,_,_), {memberchk(DP,[her,his,its,my,our,their,your])}.

/*****
/* WH_INTERRO_DECLARATIVE_CLAUSE (used by nom_cl)        */
/*
/* Note: covers the nominal relative clause (see QUIRK, pp.1244+).
*/

wh_interro_declarative_clause(
  wh_interro_declarative_clause(wh_subj(QW),Clause)) -->
  check_chrono(wh_interro_declarative_clause),
  there_must_be_a_wh_word(QW), wh_i_d_clause(QW,Clause).

wh_interro_declarative_clause(
  wh_interro_declarative_clause(Clause)) -->
  check_chrono(wh_interro_declarative_clause),
  whether_if_clause(Clause).

there_must_be_a_wh_word(QW,I,I) :- interro_pron(QW,_,I,_).

```

```

/*****
/* WH_I_D_CLAUSE (used by wh_interro_declarative_clause) */
/* Ex: "I want to know WHAT TO EXPECT ... " */
wh_i_d_clause(QW,Clause) -->
    interro_pron(QW,_), simple_to_infinitive_clause(2,Clause).

/* Ex: "I want to know WHAT WAS DONE ... " */
wh_i_d_clause(QW,Clause) -->
    interro_pron(QW,_), verbphrase(2,_,_,wh_subj,[],Clause,nil).

/* Ex: "I want to know WHICH PEOPLE HAVE PRINTED THESE FILES." */
wh_i_d_clause(QW,Clause) -->
    interro_pron(QW,_), sentence_for_wh_i_d_clause(Clause,wh_subj).

/* Ex: " I want to know WHICH FILE YOU HAVE PRINTED." */
wh_i_d_clause(QW,Clause) -->
    opt_syn_elem(wh_word_of(QW)), np_equiv(pre_v,2,Obj,_),
    sentence_for_wh_i_d_clause(Clause,Obj).

/* Ex: "I want to know WHAT YOU HAVE DONE." */
wh_i_d_clause(QW,Clause) -->
    opt_syn_elem(wh_word_of(QW)), sentence_core_s(Clause).

/* ..... */

sentence_for_wh_i_d_clause(Sent,M_el) -->
    np_equiv(pre_v,2,Np,Num),
    verbphrase(2,Num,_,Np,[],Sent,referring_np(M_el)).

wh_word_of(Q) --> interro_pron(Q,_), opt_syn_elem(pre(of)).

/*****
/* WHETHER_IF_CLAUSE (used by appositive, apposition_group &
/*   wh_interro_declarative_clause) */
/*
/* This clause is also called the "yes-no" (interrogative) clause: for more
/* details, see QUIRK, pp. 1053+. */
/* Ex: "I don't know WHETHER TO SEE MY DOCTOR TODAY." */
whether_if_clause(whether_if_clause(OrNot,WI_Clause)) -->
    conj(whether), opt_or_not(_,OrNot_2),
    simple_to_infinitive_clause(2,WI_Clause), opt_or_not(OrNot_2,OrNot).

/* Ex: "It's not clear to me IF SHE LIKES THE PRESENT." */
whether_if_clause(whether_if_clause(OrNot,WI_Clause)) -->
    conj(if), sentence_core_s(WI_Clause), opt_or_not(_,OrNot).

/* Ex: "It's not clear to me WHETHER SHE LIKES THE PRESENT." */
whether_if_clause(whether_if_clause(OrNot,WI_Clause)) -->
    conj(whether), opt_or_not(_,OrNot_2),
    sentence_core_s(WI_Clause), opt_or_not(OrNot_2,OrNot).

opt_or_not(OrNot,or_not) -->
    {OrNot \== or_not}, conj(or), special_function_word(not,_), {!}.
opt_or_not(OrNot,nil) --> {OrNot \== or_not, !}.
opt_or_not(OrNot,or_not) --> {OrNot == or_not}.

/*****
/* THAT_CLAUSE (used by nom_cl) */
/*

```

```

/* The 2nd rule handles the case in which "that" is optional, that is when
/* this clause functions as a complement: "I hope (THAT) HE ARRIVES SOON.";
/* see QUIRK, pp.1179+. Unfortunately, the 2nd rule tends to overgeneralize.

that_clause(that_clause(CL),COMPL) -->
    {var(COMPL)}, conj(that), sentence_core_s(CL).

that_clause(that_clause(CL),COMPL) -->
    {COMPL == complement}, opt_syn_elem(conj(that)), sentence_core_s(CL).

/*****
/* NP_NUCLEUS_AND_MOD (used by conj_np, either_or__np, & nounphrase) */
/*
/* The nounphrase is a common noun, a proper noun, or one of its equivalent
/* forms (see nucleus_equiv below), potentially followed by a modifier. Noti-
/* ce that several of these equiv. can also behave as noun equivalents, see
/* details in noungp_head.
/* See QUIRK, chap.5 ("Nouns and determiners") & 17 ("The noun phrase").
/*
/* Nounphrase postmodifiers are allowed only if NPPM == 1 (not if NPPM == 0).
/* For the 'proper noun' special case, NPPM == 2 (see details in np_nucleus).*/

np_nucleus_and_mod(.,.,.) -->
    end_of_sentence_marker(.), {!, fail}.
np_nucleus_and_mod(POS,Out_Struct,NUM) $ Tokens -->
    check_chrono(np_nucleus_and_mod),
    np_nucleus(NPPM,Npl,NUM),
    get_heuristic_indicator(POS,NPPM,POS_ok,Heuristic_Flag,Clue),
    determine_most_probable_np(POS_ok,Npl,NUM,Heuristic_Flag,Clue,NP),
    {get_desired_output_structure(NP,Tokens,Out_Struct)}.

/* ..... */

get_heuristic_indicator(POS,0,POS,nil,.,I,I).
get_heuristic_indicator(POS,1,POS,Heuristic_Flag,Clue,I,I) :-
    heuristic_look_ahead(POS,Heuristic_Flag,Clue,I).
get_heuristic_indicator(POS,2,POS,true,pp,I,I) :-
    heuristic_look_ahead(POS,true,pp,I).
get_heuristic_indicator(.,2,pre_v,Heuristic_Flag,Clue,I,I) :-
    heuristic_look_ahead(pre_v,Heuristic_Flag,Clue,I), Clue \== pp.

determine_most_probable_np(.,NP,.,nil,.,NP) --> {!}.
determine_most_probable_np(POS,Npl,NUM,true,Clue,NP) -->
    check_chrono(determine_most_probable_np),
    np_postmodifiers(POS,Clue,Npl,PM,NUM), {integrate_np_pm(Npl,PM,NP)}.
determine_most_probable_np(.,NP,.,.,.,NP) --> {!}.
determine_most_probable_np(POS,Npl,NUM,false,.,NP) -->
    check_chrono(determine_most_probable_np), {(Clue = rel ; Clue = app)},
    np_postmodifiers(POS,Clue,Npl,PM,NUM), {integrate_np_pm(Npl,PM,NP)}.

/* ..... */

integrate_np_pm(Entity_In,NP_Post_Modif,Entity_Out) :-
    Entity_In =.. List_In, append(List_In,[NP_Post_Modif],List_Out),
    Entity_Out =.. List_Out.

/*****
/* HEURISTIC_LOOK_AHEAD (used by get_heuristic_indicator) */
/*
/* After having parsed a single NP (nucleus), we must decide whether what
/* follows may be parsed as a modifier of that NP: we use heuristic rules.
/*
/* looks like a PREPOSITIONAL PHRASE (pp) */

```

```

/* ***note***: (O=I ; adv_s(_,I,O)), prep(X,O,_), (\+ conj(X,O,_); X==like). */
heuristic_look_ahead(pre_v,true,pp,I) :-
    (O = I ; adv_s(_,I,O)), prep(_,O,_).
heuristic_look_ahead(post_v,true,pp,I) :-
    (O2 = I ; conj(either,I,O2) ; conj(neither,I,O2)), (O = O2 ; adv_s(_,O2,O)),
    prep(X,O,_), memberchk(X,[about,below,including,of,regarding]).

/* looks like a RELATIVE CLAUSE (rel) */
heuristic_look_ahead(post_v,true,rel,I) :-
    (O = I ; conj(either,I,O) ; conj(neither,I,O)), past_parc(_,O,_).
heuristic_look_ahead(_,true,rel,I) :-
    (O = I ; conj(either,I,O) ; conj(neither,I,O)),
    (rel_pron(_,O,_); ing_word(O)).
heuristic_look_ahead(_,true,rel,I) :- prep(_,I,O), rel_pron(_,O,_).

/* looks like an APPOSITIVE (app) */
heuristic_look_ahead(_,true,app,I) :-
    comma(I,O), (conj(that,O,_); conj(whether,O,_); conj(if,O,_)).
heuristic_look_ahead(_,true,app,I) :-
    (comma(I,O); parenthesis_o(I,O); dashes(I,O)),
    explicit_app_ind(X,O,_), \+ real_coordinator(X), X \== of.
heuristic_look_ahead(_,true,app,I) :-
    (parenthesis_o(I,O); dashes(I,O)), \+ explicit_app_ind(_,O,_).
heuristic_look_ahead(_,true,app,I) :- conj(whether,I,_); conj(if,I,_).

/* looks like a RESTRICTED COMPARISON (rc) */
heuristic_look_ahead(_,true,rc,I) :- conj(than,I,_).

/* Otherwise, we could not find any clues ... */
heuristic_look_ahead(_,false,_).

/* ***** */
/* NP_NUCLEUS (used by not_followed_by_a_np & np_nucleus_and_mod) */
/* */
/* The first parameter indicates whether the nounphrase (nucleus) can be fol- */
/* lowed by a modifier (e.g. a relative clause): 0 means 'no', and 1 'yes': */
/* 2 means "only those of the pre-verbal position (pre_v)". */
/* */

/* COMMON NOUN */
np_nucleus(1,entity(intensifier(Adv),Dets,ATTRS,HEAD,number(NUM)),NUM) -->
    ((Adv = nil) ; adv_s(Adv)),
    determinatives(Dets,Num1), {adv_det_filter(Adv,Dets), predeter_filter(Dets)},
    rest_common_noun_np(Dets,Num1,ATTRS,HEAD,NUM).

/* GENITIVE (with implicit ellipsis) */
np_nucleus(0,entity(attributes([]),genitive(X),number(Num)),Num) -->
    [dct(genitive(_):X/_,_)], {(Num = sg ; Num = pl)}.

/* EQUIVALENT FORM */
np_nucleus(0,entity(intensifier(Adv),X,number(Num)),Num) -->
    ((Adv = nil) ; adv_s(Adv)),
    nucleus_equiv(X,Num), {(var(Num), (Num = sg ; Num = pl)) ; nonvar(Num)}.

/* PROPER NOUN */
np_nucleus(2,entity(attributes(ATTRS),proper_noun(PN),number(Num)),Num) -->
    adj_phrase(ATTRS), propernoun(PN,Num).

/* ADJECTIVE SUBSTANTIVE CLAUSE */
np_nucleus(1,Adj_subst_cl,Num) -->
    adj_substantive_clause(Adj_subst_cl,Num).

/* ***** */
/* EQUIV: PERSONAL PRONOUN */

```



```

nucleus_equiv(ref(pers_pron(P,X)),Num)    --\
    partiv_pron(P,Num), pers_pron(X,_).
nucleus_equiv(ref(pers_pron(X)),Num)      --\ pers_pron(X,Num).

/* EQUIV: POSSESSIVE PRONOUN
nucleus_equiv(ref(poss_pron(P,X)),Num)    - >
    partiv_pron(P,Num), poss_pron(X,_).
nucleus_equiv(ref(poss_pron(X)),Num)      --> poss_pron(X,Num).

/* EQUIV: INDEFINITE PRONOUN
nucleus_equiv(ref(pron(X)),Num)           -->
    pron(X,Num,_), not_followed_by_or_more(X), not_followed_by_a_np.

/* EQUIV: QUANTIFIER
nucleus_equiv(ref(quant(X)),Num)          -->
    quant(X,Num,_), not_followed_by_a_np.

/* EQUIV: NUMBER (GENERAL)
nucleus_equiv(ref(X),Num)                 --> number_gen(X,Num).

/* EQUIV: REFLEXIVE PRONOUN
nucleus_equiv(ref(reflex_pron(X)),Num)     --> reflex_pron(X,Num).

/* EQUIV: CHARACTER
nucleus_equiv(ref(X),Num)                 --> noun(character,Num,X).

/* EQUIV: STRING
nucleus_equiv(ref(X),Num)                 --> noun(string,Num,X).

/* EQUIV: QUIZ TOKEN
nucleus_equiv(ref(quiz_token(X)),_)       -->
    [dct(Type:X/_,_)],
    {(Type == [quiz_token,_] ; Type == ref_or_var_or_quiz_item)}.

/* EQUIV: REFERENCE OR VARIABLE
nucleus_equiv(ref(ref_or_var(X)),_)       --> [dct(ref_or_var:X/_,_)].

/* EQUIV: DATE
nucleus_equiv(ref(X),_)                   --> date_form(X).

/* EQUIV: SPECIAL NUMBER
nucleus_equiv(ref(number_special(X)),_)   --> [dct(number_special:X/_,_)].

/* EQUIV: DOLLARS
nucleus_equiv(ref(dollars(X,or_more)),_)  -->
    [dct(dollars:X/_,_)], conj(or), quant(more,_,_,O,_).
nucleus_equiv(ref(dollars(X,or_less)),_)  -->
    [dct(dollars:X/_,_)], conj(or), quant(less,_,_,O,_).
nucleus_equiv(ref(dollars(X)),_)          --> [dct(dollars:X/_,_)].

/* EQUIV: PERCENTAGE
nucleus_equiv(ref(percentage(X)),_)       --> [dct(percentage:X/_,_)].

/* .....

/* Disambiguating 'one': indefinite pronoun vs cardinal.
not_followed_by_or_more(X,I,I) :- X \== one, !.
not_followed_by_or_more(one,I,I) :- conj(or,I,O), quant(more,_,_,O,_), !, fail.
not_followed_by_or_more(one,I,I) :- conj(or,I,O), quant(less,_,_,O,_), !, fail.
not_followed_by_or_more(one,I,I).

/* *****
/* ADV_DET_FILTER & PREDETER_FILTER (both used by np_nucleus)

```

```

/* If an adverb precedes the determinatives, the latter must be non-empty. */
adv_det_filter(Adv,determinatives(predeter(nil),deter(nil),postdeter(nil)))
:- Adv \== nil, !, fail.
adv_det_filter(_,_).

/* Ruling out empty comparatives in determinative constructions. */
predeter_filter(determinatives(
    predeter(np_intensifier__than(_)),deter(nil),postdeter(nil)))
:- !, fail.
predeter_filter(_).

/*****
/* REST_COMMON_NOUN_NP (used by np_nucleus) */
rest_common_noun_np(Dets,Num1,attributes(ATTRS),head_noun(HEAD),Num)
-->
    adj_phrase(ATTRS),
    (noun_subst(Dets,Nt,Num2,HEAD) ; noungp_head(Nt,Num2,HEAD)),
    (check_nums(Num1,Num2,Num), check_type_agree(Dets,Num,Nt)).

/* ADJ & NOUN are enclosed in double quotes (e.g. "supporting person"); */
/* see note on double_quotes vs 'string' in verb_in_double_quotes_or_not. */
/* Notation (string): _b stands for begin(ning) and _e stands for end(ing). */
rest_common_noun_np(Dets,Num1,string_b(attributes(ATTRS)),
    string_e(head_noun(HEAD)),Num)
-->
    double_quotes, adj_phrase(ATTRS), {ATTRS \== []},
    (noun_subst(Dets,Nt,Num2,HEAD) ; noungp_head(Nt,Num2,HEAD)),
    double_quotes, {check_nums(Num1,Num2,Num), check_type_agree(Dets,Num,Nt)}.

/* ..... */

noun_subst(determinatives(_ ,deter(X),_),countnoun,sg,noun_subst(one)) -->
    {memberchk(X,[that,the,this])}, pron(one,sg3,_).
noun_subst(determinatives(_ ,deter(X),_),countnoun,pl,noun_subst(ones)) -->
    {memberchk(X,[the,these,those])}, pron(ones,pl,_).

check_nums(Num1,sg,sg3)      :- var(Num1).
check_nums(Num1,sg,sg)       :- var(Num1), !.
check_nums(Num1,pl,pl)       :- var(Num1), !.
check_nums(sg3,sg,sg3)      :- !.
check_nums(sg,sg,sg)        :- !.
check_nums(pl,pl,pl)        :- !.
check_nums(Num1,_,sg3)      :- Num1 == sp_sg3, !.
check_nums(Num1,Num2,Num2)  :- Num1 == sp_N.

check_type_agree(determinatives(_ ,deter(X),_),Num,massnoun) :-
    (Num == sg ; Num == sg3),
    \+ memberchk(X,[a,an,each,either,every,neither]).
check_type_agree(_Dets,Num,countnoun) :-
%   Dets \== determinatives(predeter(nil),deter(nil),postdeter(nil)),
    (Num == sg ; Num == sg3 ; Num == pl).
check_type_agree(Dets,pl,countnoun) :-
    Dets == determinatives(predeter(nil),deter(nil),postdeter(nil)).
check_type_agree(_ ,_,NounType) :-
    NounType \== massnoun, NounType \== countnoun.

/*****
/* ADJ_SUBSTANTIVE_CLAUSE (used by np_nucleus) */
/*
/* Covers the special case where an adjective is used as a noun. */

```

```

/* Ex: "THE EXTREMELY POOR has great difficulty ..."

adj_substantive_clause(entity(Dets,intensifier(Adv),Adj,number(Num)),Num) -->
    determinatives(Dets,Num),
    (Dets \== determinatives(predeter(nil),deter(nil),postdeter(nil))),
    ((Adv = nil) ; adv_s(Adv)), adj_subst(Adj), not_followed_by_a_np.

adj_subst(Adj) --> adj(Adj,_).
adj_subst(double_quotes(Adj)) --> double_quotes, adj(Adj,_), double_quotes.

/*****
/* PARTIV_PRON (used by nucleus_equiv) */

partiv_pron(P,Num) --> partitive_predeter_partiv_pron(P,Num). % e.g. ALL OF us
partiv_pron(P,Num) --> partitive_deter_partiv_pron(P,Num). % e.g. EACH OF us
partiv_pron(P,Num) --> closed_postdeter_partiv_pron(P,Num). % e.g. ONE OF us

partitive_predeter_partiv_pron(partitive_predeter(PRE,one,of),Num) -->
    predeter(PRE,Num,_), pron(one,_,_),
    (valid_predeter_partiv_pron(PRE)), prep(of).

partitive_predeter_partiv_pron(partitive_predeter(PRE,of),Num) -->
    predeter(PRE,Num,_), (valid_predeter_partiv_pron(PRE)), prep(of).

valid_predeter_partiv_pron(PRE) :-
    \+ memberchk(PRE,[double,once,such,twice,what,whole]).

/* ..... */

partitive_deter_partiv_pron(partitive_deter(D,one,of),Num) -->
    deter(D,Num,_), pron(one,_,_), (valid_deter_partiv_pron(D)), prep(of).

partitive_deter_partiv_pron(partitive_deter(D,of),Num) -->
    deter(D,Num,_), (valid_deter_partiv_pron(D)), prep(of).

valid_deter_partiv_pron(D) :-
    memberchk(D,[any,each,either,enough,neither,some,whatever,which,whichever]).

/* ..... */

/* See below (in opt_postdeter) for definitions of postdeter_gp & postdeter. */
closed_postdeter_partiv_pron(partitive_postdeter(POST,Of),Num) -->
    postdeter_gp(POST,Num), part_of_partiv_pron(Of).

closed_postdeter_partiv_pron(partitive_postdeter(POST,Of),Num) -->
    postdeter(POST,Num), part_of_partiv_pron(Of).

part_of_partiv_pron([or,Q,of]) -->
    conj(or), quant(Q,_,_), {(Q == more ; Q == less)}, prep(of).
part_of_partiv_pron(of) --> prep(of).

/*****
/* NUMBER_GEN (used by card_num_c, date_form, noun_post_modifier &
/* nucleus_equiv) */

/* Ex: "seven", "seven or more"
number_gen(card_num(X,or_more),Num) -->
    card_num(X,Num,_), conj(or), quant(more,_,_).
number_gen(card_num(X,or_less),Num) -->

```

```

        card_num(X,Num,_), conj(or), quant(less,_,_).
number_gen(card_num(X),Num) -->
        card_num(X,Num,_), not_the_one_pronoun(X).

/* Ex: "5", "5 or more" */
number_gen(num(X,or_more),Num) -->
        [dct(number:X/_,_), conj(or), quant(more,_,_),
        {((X >= 2, Num = pl) ; Num = sg)}].
number_gen(num(X,or_less),Num) -->
        [dct(number:X/_,_), conj(or), quant(less,_,_),
        {((X >= 2, Num = pl) ; Num = sg)}].
number_gen(num(X),Num) -->
        [dct(number:X/_,_), {((X >= 2, Num = pl) ; Num = sg)}].

/* Ex: "(2)", "(2 or more)" */
number_gen(parenthetical_num(X,or_more),Num) -->
        parenthesis_o, [dct(number:X/_,_), conj(or), quant(more,_,_),
        parenthesis_c, {((X >= 2, Num = pl) ; Num = sg)}].
number_gen(parenthetical_num(X,or_less),Num) -->
        parenthesis_o, [dct(number:X/_,_), conj(or), quant(less,_,_),
        parenthesis_c, {((X >= 2, Num = pl) ; Num = sg)}].
number_gen(parenthetical_num(X),Num) -->
        parenthesis_o, [dct(number:X/_,_), parenthesis_c,
        {((X >= 2, Num = pl) ; Num = sg)}].

/* Ex: "1(c)(iii)" */
number_gen(composite_num(X,PSN),Num) -->
        [dct(number:X/_,_),
        parenthetical_sub_num(PSN), {((X >= 2, Num = pl) ; Num = sg)}].

/* ..... */

/* Disambiguating 'one': indefinite pronoun vs cardinal. */
not_the_one_pronoun(X,I,I) :- X \== one, !.
not_the_one_pronoun(one,I,I) :-
        \+ end_of_sentence_marker(_,I,_), \+ aux(_,_,_,I,_), \+ modal_aux(_,_,I,_),
        (\+ verb(_,_,_,I,_)) ; ing_word(I).

/*****
/* PARENTHETICAL_SUB_NUM (used by number_gen) */
/*
/* Deals with "sub-numeral" (letter) numbering: Ex: '(a)' in "37(a)". */
parenthetical_sub_num([L|R]) -->
        parenthesis_o,
        [dct(word:L/low,_), {name(L,A_L), length(A_L,Le), Le =< 3},
        parenthesis_c, rest_parenthetical_sub_num(R)].

rest_parenthetical_sub_num([L|R]) -->
        parenthesis_o,
        [dct(word:L/low,_), {name(L,A_L), length(A_L,Le), Le =< 3},
        parenthesis_c, rest_parenthetical_sub_num(R), (!)].
rest_parenthetical_sub_num([]) --> [].

/*****
/* DATE_FORM (used by noun_post_modifier, noun_pre_modif, & nucleus_equiv) */
/* Standard abbreviated format as recognized by the scanner */
date_form(date(X)) --> [dct(date:X/_,_)].

/* Ex: "April 30, 1991" */
date_form(date(month(Month),day(Day),year(Year))) -->
        propernoun(Month,_), {valid_month(Month)},

```

```

number_gen(num(Day),_), {Day > 0, Day < 32}, opt_comma,
number_gen(num(Year),_), {Year > 0}.

/* Ex: "April 30"
date_form(date(month(Month),day(Day))) -->
propernoun(Month,_), {valid_month(Month)},
number_gen(num(Day),_), {Day > 0, Day < 32}.

/* Ex: "April 1991"
date_form(date(month(Month),year(Year))) -->
propernoun(Month,_), {valid_month(Month)},
number_gen(num(Year),_), {Year > 0}.

valid_month(M) :- memberchk(M,[january,february,march,april,may,june,july,
                             august,september,october,november,december]).

/*****
/* NP_POSTMODIFIERS (used by determine_most_probable_np) */
/*
/* Noun phrase postmodifiers are clauses that are attached to a noun phrase. */
/* Warning: some clauses (especially PPs) may in fact belong to a VP;
/* only semantic K or user interaction will allow complete disambiguation.
/* See QUIRK, chap.17 ("The noun phrase") and, in particular, 17.9-50.
*/

np_postmodifiers(_,_,_,_)_ -->
end_of_sentence_marker(_), {!, fail}.
np_postmodifiers(POS,Clue,NpRel,np_postmodifiers(NP_PM),Num) -->
check_spy_on_this_rule(np_postmodifiers,nil),
np_postmodif(POS,Clue,NpRel,NP_PM,Num).

/* Note: the definition of rel_clause covers the 3 cases where a ing_clause, */
/* a to_infinitive_clause, or a past part. phrase acts as a relative clause. */
np_postmodif(_ ,rel,NpRel,X,Num) --> {nonvar(NpRel)}, rel_clause(NpRel,X,Num).
np_postmodif(POS,app,_ ,X,Num) --> appositive(POS,X,Num).
np_postmodif(_ ,rc,_ ,X,_ ) --> restricted_comparison(X).
np_postmodif(_ ,pp,_ ,X,_ ) --> required_prepphrases(X,_ ).

/* Recognizes a restricted case of comparison: "... a bigger car THAN THAT." */
restricted_comparison(restricted_comparative_clause__than(Compar_Obj)) -->
conj(than), np_equiv(post_v,2,Compar_Obj,_ ).

/*****
/* REL_CLAUSE (used by np_postmodif) */
/*
/* Missing subject (last parameter): M_el = missing element (referring NP). */
/* Typical case (1).
/* Ex: "He always reads the letters THAT APPEAR IN THE LOCAL JOURNAL."
rel_clause(Np,rel_clause(rel_marker(RM),VP),Num)
-->
rel_marker(1,RM), verbphrase(2,Num,_ ,Np,[],VP,nil).

/* The case where the preposition is deferred ("dangling" preposition).
/* Ex: "The man WHOM I GAVE IT TO."
rel_clause(Np,rel_clause(rel_marker(RM),Sent),_)
-->
rel_marker(1,RM),
np_equiv(pre_v,2,Np1,Num1), verbphrase(2,Num1,_ ,Np1,[],Sent,def_prep(Np)).

/* The case where a preposition precedes the relative marker.
/* Ex: "The man TO WHOM I GAVE IT."

```

```

rel_clause(Np,rel_clause(rel_marker(RM),Sent),_)
-->
  prep(P), rel_marker(1,RM),
  (np_equiv(pre_v,2,Subj,Num) ; {(Num = sg3 ; Num = pl ; Num = sg)}),
  verbphrase(2,Num,_,Subj,[],Sent,referring_np(pp(P,Np))).

/* A -ing clause as a relative clause: "An architect HAVING LUNCH." */
rel_clause(_,rel_clause(rel_marker(nil),VC),_)
-->
  verb_clause(2,_,_,[progr],nil,_,VC,nil).

/* A past participle phrase as a relative clause: "A report WRITTEN BY TOM." */
rel_clause(Np,rel_clause(rel_marker(nil),subj(Agent),EC),Num)
-->
  past_participle_phrase(Num,Np,Agent,EC).

/* Nonfinite relative clause (1): "A cause FOR WHICH TO DIE." */
rel_clause(Np,rel_clause(rel_marker(RM),to_infinitive_clause(VC)),_)
-->
  prep(P), rel_marker(1,RM),
  infinitive_marker,
  verb_clause(2,_,_,[inf],nil,_,VC,referring_np(pp(P,Np))).

/* Typical case (2): the relative clause is introduced by a relative marker. */
/* Ex: "He always reads the letters (WHICH) SHE WRITES TO HIM." */
rel_clause(Np,rel_clause(rel_marker(prepp(P),RM),Sent),_)
-->
  ((prep(P), rel_marker(1,RM)) ; {P = nil}, rel_marker(0,RM)),
  np_equiv(pre_v,2,Subj,Num),
  verbphrase(2,Num,_,Subj,[],Sent,referring_np(Np)).

/* Nonfinite relative clause (2): "A cause FOR YOU TO SUPPORT." */
rel_clause(Np,rel_clause(
  p_rel_marker(for),subj(Subj),to_infinitive_clause(Sent)),_)
-->
  prep(for), np_equiv(pre_v,2,Subj,_),
  infinitive_marker,
  verb_clause(2,_,_,[inf],nil,_,Sent,referring_np(Np)).

/* ..... */

rel_marker(_,RM) --> rel_pron(RM), (!). % rel_marker(1,X) => MANDATORY
rel_marker(0,nil) --> []. % rel_marker(0,X) => OPTIONAL

/*****
/* PAST_PARTICIPLE_PHRASE (used by rel_clause & rest_adverbial_clause) */

past_participle_phrase(Num,Compl,Subj,EC,I,O) :-
  check_chrono(past_participle_phrase,I),
  real_pass_rel_clause_test(I),
  ((Num == pl, W = were) ; (Num \== pl, W = was)),
  verb_clause(2,_,passive,[past],Subj,Compl,EC,nil,
    [dct(word:W/low,[d(aux/be,Num,past)])|I],O).

real_pass_rel_clause_test([]) :- !, fail.
real_pass_rel_clause_test(I) :- verb(_,_,_,I,O), preppphrase(_,_,O,_), !.
real_pass_rel_clause_test(I) :- simple_to_infinitive_clause(2,_,I,_), !.
real_pass_rel_clause_test([_|T]) :- real_pass_rel_clause_test(T).

/*****
/* APPOSITIVE (used by np_postmodif) */
/*
/* This rule recognizes appositives: linguistic units, mostly noun phrases, */

```

```

/* which are identical in reference; see QUIRK, chap.17 (17.65-93). */
/* General case: it requires correlative markers.
appositive(POS,appositive(APPN),Num) -->
    app_correlative_marks_o(N), apposition_core(APPN,Num),
    app_correlative_marks_c(POS,N).

/* Two cases that do not require correlative markers.
appositive(_,appositive(APPN),Num) --> propernoun(APPN,Num).
appositive(_,appositive(APPN),_) --> whether_if_clause(APPN).

/* Special case (1): special characters (e.g. "the ampersand symbol (&)").
appositive(_,appositive(special_symbol(APPN)),_) -->
    parenthesis_o, special_char_seq(APPN), parenthesis_c.

/* Special case (2): QUIZ command (e.g. "the item subtotal (SUBTOTAL)").
appositive(_,appositive(APPN),_) -->
    parenthesis_o, [dct(Type:APPN/_,_)], {Type =.. [quiz_token,_]},
    parenthesis_c.

app_correlative_marks_o(1) --> comma.
app_correlative_marks_o(2) --> dashes.
app_correlative_marks_o(3) --> parenthesis_o.

app_correlative_marks_c(_,1) --> comma.
app_correlative_marks_c(post_v,1,I,I) :- end_of_sentence_marker(_,I,_).
app_correlative_marks_c(_,2) --> dashes.
app_correlative_marks_c(post_v,2,I,I) :- end_of_sentence_marker(_,I,_).
app_correlative_marks_c(_,3) --> parenthesis_c.

/* ..... */

apposition_core([EAI,AG],Num) -->
    explicit_app_ind(EAI), opt_comma, apposition_group(AG,Num).

apposition_core([EAI,AG],Num) -->
    apposition_group(AG,Num), opt_comma, explicit_app_ind(EAI).

apposition_core([Adv,AG],Num) -->
    required_adverbs(Adv), apposition_group(AG,Num).

apposition_core(AG,Num) -->
    apposition_group(AG,Num).

apposition_core(EAI,_) -->
    explicit_app_ind(EAI),
    ((EAI == for_example ; EAI == for_instance ;
      EAI == in_other_words ; EAI == in_particular)).

/* Ex: "Jones, THE DISTINGUISHED ART CRITIC, died in his sleep last night." */
apposition_group(AG,Num) --> np_equiv(pre_v,2,AG,Num).

/* Ex: "My friend ANNA was here last night." */
apposition_group(AG,Num) --> propernoun(AG,Num).

/* Ex: "His explanation, THAT HE COULDN'T SEE THE CAR, is unsatisfactory." */
apposition_group(AG,_) --> nominal_clause(that_clause,AG,_,_).

/* Ex: "The question WHETHER TO CONFESS OR NOT troubled the girl." */
apposition_group(AG,_) --> whether_if_clause(AG).

/* ***** */

```

```

explicit_app_ind(EAI)          --> adv(EAI,Type), (Type == app).
explicit_app_ind(EAI)          --> [det(_:EAI/_), {appos_indicator(EAI)}].
explicit_app_ind(for_example)  --> idiomatic_cat(for_example,_,_,_).
explicit_app_ind(for_instance) --> idiomatic_cat(for_instance,_,_,_).
explicit_app_ind(in_other_words) --> idiomatic_cat(in_other_words,_,_,_).
explicit_app_ind(in_particular) --> idiomatic_cat(in_particular,_,_,_).
explicit_app_ind(such_as)      --> idiomatic_cat(such_as,_,_,_).
explicit_app_ind(that_is_to_say) --> idiomatic_cat(that_is_to_say,_,_,_).
explicit_app_ind(that_is)      --> idiomatic_cat(that_is,_,_,_).

appos_indicator('e.g.').
appos_indicator('i.e.').
appos_indicator(and).
appos_indicator(chiefly).
appos_indicator(eg).
appos_indicator(especially).
appos_indicator(for_example).
appos_indicator(for_instance).
appos_indicator(ie).
appos_indicator(in_other_words).
appos_indicator(in_particular).
appos_indicator(included).
appos_indicator(including).
appos_indicator(mainly).
appos_indicator(mostly).
appos_indicator(namely).
appos_indicator(notably).
appos_indicator(of).
appos_indicator(or).
appos_indicator(particularly).
appos_indicator(say).
appos_indicator(simply).
appos_indicator(such_as).
appos_indicator(technically).
appos_indicator(that_is).
appos_indicator(that_is_to_say).
appos_indicator(viz).

/*****
/* REQUIRED_PREPPHRASES (used by adj_complement, adverbial_subordinate,
/* complement_tr, np_postmodifier, rest_complement_tr_act_np,
/* rest_complement_tr_pass_by, & sentence_frag)
/* & PREPPHRASES (used by required_prepphrases)
*/

required_prepphrases(RPPS,Restriction) -->
    prepphrases(RPPS,Restriction), {RPPS \== []}.

prepphrases([],_,I,I) :-
    end_of_sentence_marker(_,I,_), !.
prepphrases(Out_Struct,Restriction) $ Tokens -->
    check_spy_on_this_rule(prepphrases,nil),
    (either_or_pp(PPS,Restriction) ;
     (prepphrase(PP,Restriction), rest_prepphrases(PP,PPC,Restriction))),
    (get_desired_output_structure(PPS,Tokens,Out_Struct), !).
prepphrases([],_) --> [].

/* EITHER/OR PPs.
either_or_pp([conj_pps(either_or,[PP_e,PP_o])],Restriction) -->
    conj(either), prepphrase(PP_e,Restriction),
    conj(or), prepphrase(PP_o,Restriction).

```



```

/* NEITHER/NOR PPs.
either_or_pp([conj_pps(neither_nor,[PP_e,PP_o])],Restriction) -->
    conj(neither), prepphrase(PP_e,Restriction),
    conj(nor), prepphrase(PP_o,Restriction).

rest_prepphrases(PP,[PP,parenthetical(conj_pps(Coor,PPs))],Restriction) -->
    parenthesis_o, conj_pp(1,Coor0,PPs,Restriction), {PPs \== []},
    {((var(Coor0), Coor = implicit_and) ; (nonvar(Coor0), Coor = Coor0))},
    parenthesis_c, (!).

rest_prepphrases(PP,[conj_pps(Coor,[PP|PPs])],Restriction) -->
    conj_pp(1,Coor0,PPs,Restriction), {PPs \== []},
    {((var(Coor0), Coor = implicit_and) ; (nonvar(Coor0), Coor = Coor0))}, (!).

rest_prepphrases(PP,[PP],_) -->
    [].

/* ..... */
/* On the N parameter, see remarks in dipett_6 (conj_vp predicate). */

/* Case (1) : PP (',' ) COOR PP ... */
conj_pp(N,Coor,[PP_H|PP_R],Restriction) -->
    {(N == 1) ; {(N \== 1), opt_comma}}, real_coordinator_3(Coor),
    prepphrase(PP_H,Restriction), conj_pp(_,Coor,PP_R,Restriction).

/* Case (2) : PP (',' ) PP ... */
conj_pp(_,Coor,[p_xsyndetic_conj_pp(PP_H)|PP_R],Restriction) -->
    opt_comma,
    prepphrase(PP_H,Restriction), conj_pp(_,Coor,PP_R,Restriction).

conj_pp(_,_,[],_) -->
    check_chrono(conj_pp).

/***** */
/* PREPPHRASE (used by conj_pp, either_or_pp, prepphrases, & */
/* real_pass_rel_clause_test) */

prepphrase(_,_) --> end_of_sentence_marker(_), (!, fail).
prepphrase(PP,_) --> already_parsed(prepphrase,PP).
prepphrase(PP,Restriction,I,O) :-
    check_chrono(prepphrase,I),
    prepphrase_core(PP,Restriction,I,O), keep_parse_in_mem(prepphrase,PP,I,O).

prepphrase_core(Out_Struct,Restriction) $ Tokens -->
    required_adverbs(Advs), prep_pp(P,Restriction), np_for_pp(Np),
    {get_desired_output_structure(pp([Advs,P],Np),Tokens,Out_Struct)}.
prepphrase_core(Out_Struct,Restriction) $ Tokens -->
    prep_pp(P,Restriction), np_for_pp(Np),
    {get_desired_output_structure(pp(P,Np),Tokens,Out_Struct)}.
prepphrase_core(Out_Struct,Restriction) $ Tokens -->
    prep_pp(P,Restriction), required_adverbs(Advs),
    {get_desired_output_structure(pp_adv(P,Advs),Tokens,Out_Struct)}.

prep_pp(conj_preps(C,[P1,P2]),Restriction) -->
    prep(P1), real_coordinator_3(C), prep(P2), opt_colon,
    {(var(Restriction) ; (P1 \== Restriction, P2 \== Restriction))}.
prep_pp(P,Restriction) -->
    prep(P), opt_colon, {(var(Restriction) ; P \== Restriction)}.

np_for_pp(Np) --> nominal_clause(wh_interro_declarative_clause,Np,_,_).

```

```

np_for_pp(Np) --> nominal_clause(ing_clause,Np,_,_).
np_for_pp(Np) --> np_equiv(post_v,2,Np,_).

/*****
/* DETERMINATIVES (used by adj_substantive_clause & np_nucleus) */
/*
/* Determiners, including pre- & post-determiners. See QUIRK, pp.253+. */

determinatives(determinatives(predeter(PRE),deter(D),postdeter(POST)),Num) -->
    opt_predeter(PRE,Num_pre), opt_deter(D,Num_d), opt_postdeter(POST,Num_post),
    {determinatives_num(PRE,Num_pre,D,Num_d,POST,Num_post,Num)}.

/*****
/* OPT_PREDETER (used by determinatives) */

opt_predeter(PRE,sg3) --> predeter_whole(PRE), {!}.
opt_predeter(PRE,Num) --> predeter(PRE,Num,_), rest_predeter(PRE,PRE), {!}.
opt_predeter(PRE,_) --> compare_p_np(PRE), {!}.
opt_predeter(nil,_) --> [].

predeter_whole(a_whole) --> deter(a,sg3,_), predeter(whole,_,_).
predeter_whole(the_whole) --> deter(the,sg3,_), predeter(whole,_,_).

rest_predeter(PRE,[PRE,of]) --> prep(of), {!}.
rest_predeter(PRE,PRE) --> [].

/* Recognizes "less than a ... ", "more than one", "fewer than 100", etc. */
compare_p_np(np_intensifier__than(Adv)) --> adv(Adv,_), conj(than).
compare_p_np(np_intensifier__than(Adj)) --> adj(Adj,comp), conj(than).

/*****
/* OPT_DETER (used by determinatives) */

opt_deter(D,Num) --> partitive_deter(D,Num), {!}.
opt_deter(D,Num) --> deter(D,Num,_), {!}.
opt_deter(nil,_) --> [].

partitive_deter(partitive_deter(D1,one,of,D2),Num) -->
    deter(D1,Num0,_), pron(one,_,_), prep(of), deter(D2,_,_),
    {valid_deter_partitive(D1,D2), ((Num0 == sg3, Num = sp_sg3) ; Num = sp_N)}.

partitive_deter(partitive_deter(D1,of,D2),Num) -->
    deter(D1,Num0,_), prep(of), deter(D2,_,_),
    {valid_deter_partitive(D1,D2), ((Num0 == sg3, Num = sp_sg3) ; Num = sp_N)}.

valid_deter_partitive(D1,D2) :-
    memberchk(D1,[any,each,either,enough,neither,some,whatever,whichever]),
    memberchk(D2,[each,every,her,his,its,my,our,such,that,the,their,these,
        this,those,your]).

/*****
/* OPT_POSTDETER (used by determinatives) */

opt_postdeter(POST,Num) -->
    poss_ng(P), rest1_postdeter(P,POST,Num), {!}.
opt_postdeter(POST,Num) -->
    closed_postdeter(CP,Num), rest2_postdeter(CP,POST), {!}.
opt_postdeter(POST,pl) -->
    open_class_quant(POST), {!}.

```

290

```

open_class_quant(open_quant(lots_of)) --> quant(lots,_,_), prep(of).
open_class_quant(open_quant(plenty_of)) --> quant(plenty,_,_), prep(of).

/*****
/* DETERMINATIVES_NUM (used by determinatives) */
/*
/* Checking number agreement in determinatives. The general pattern is:
/* "determinatives_num(PRE,Num_pre,D,Num_d,POST,Num_post,Resulting_Num)"
/* where PRE = predeterminer, D = determiner, and POST = postdeterminer. */

/* #1: PRE: no, D: no, POST: no */
determinatives_num(nil,_,nil,_,nil,_,_) :- !.

/* #2: special case for partitive determiners */
determinatives_num(_,_,D,Num_d,_,_,Num_d) :-
    D \== nil, (Num_d == sp_sg3 ; Num_d == sp_N), !.

/* #3: PRE: yes (_N|pl|sg3), D: no, POST: no */
determinatives_num(PRE,Num_pre,nil,_,nil,_,Num_pre) :- PRE \== nil, !.

/* #4: PRE: no, D: yes (_N|pl|sg3), POST: no */
determinatives_num(nil,_,D,Num_d,nil,_,Num_d) :- D \== nil, !.

/* #5: PRE: no, D: no, POST: yes (_N|pl|sg3) */
determinatives_num(nil,_,nil,_,POST,Num_post,Num_post) :- POST \== nil, !.

/* #6: PRE: yes (_N|pl|sg3), D: yes (_N|pl|sg3), POST: no */
determinatives_num(PRE,Num_pre,D,Num_d,nil,_,Num) :-
    PRE \== nil, D \== nil, !, determinatives_num_agree(Num_pre,Num_d,Num).

/* #7: PRE: yes (_N|pl|sg3), D: no, POST: yes (_N|pl|sg3) */
determinatives_num(PRE,Num_pre,nil,_,POST,Num_post,Num) :-
    PRE \== nil, POST \== nil, !,
    determinatives_num_agree(Num_pre,Num_post,Num).

/* #8: PRE: no, D: yes (_N|pl|sg3), POST: yes (_N|pl|sg3) */
determinatives_num(nil,_,D,Num_d,POST,Num_post,Num) :-
    D \== nil, POST \== nil, !, determinatives_num_agree(Num_d,Num_post,Num).

/* #9: PRE: yes (_N|pl|sg3), D: yes (_N|pl|sg3), POST: yes (_N|pl|sg3) */
determinatives_num(PRE,Num_pre,D,Num_d,POST,Num_post,Num) :-
    PRE \== nil, D \== nil, POST \== nil, !,
    determinatives_num_agree(Num_pre,Num_d,Num_post,Num).

/*****
/* DETERMINATIVES_NUM_AGREE/3 & /4 (used by determinatives_num) */

determinatives_num_agree(N1,N1,N1) :- nonvar(N1).
determinatives_num_agree(N1,N2,_) :- var(N1), var(N2).
determinatives_num_agree(N1,N2,N1) :- nonvar(N1), var(N2).
determinatives_num_agree(N1,N2,N2) :- var(N1), nonvar(N2).

determinatives_num_agree(N1,N1,N1,N1) :- nonvar(N1).
determinatives_num_agree(N1,N2,N3,_) :- var(N1), var(N2), var(N3).
determinatives_num_agree(N1,N2,N2,N2) :- var(N1), nonvar(N2).
determinatives_num_agree(N1,N2,N1,N1) :- var(N2), nonvar(N1).
determinatives_num_agree(N1,N1,N3,N1) :- var(N3), nonvar(N1).
determinatives_num_agree(N1,N2,N3,N3) :- var(N1), var(N2).
determinatives_num_agree(N1,N2,N3,N2) :- var(N1), var(N3).
determinatives_num_agree(N1,N2,N3,N1) :- var(N2), var(N3).

/*****
/* REQUIRED_ADJ_PHRASE (used by compl_adj_phrase, complement_stat,

```

```

/*      complement_tr, rest_complement_tr_act_np, & sentence_frag)
/* & ADJ_PHRASE (used by np_nucleus, required_adj_phrase, &
/*      rest_common_noun_np)
/*
/* Note: we also cover the case where and adverb is used as an adjective;
/* see QUIRK, chap.7 ("Adjectives and adverbs") and pp.405+.

required_adj_phrase(Adj_P) --> adj_phrase(Adj_P), {Adj_P \== []}.

adj_phrase([],I,I) :-
    end_of_sentence_marker(.,I,.), !.
adj_phrase(Out_Struct) $ Tokens -->
    check_spy_on_this_rule(adj_phrase,nil),
    adj_equiv(Adj,Deg), rest_adj_phrase(Coor,Adj_P_R),
    (pack_adj_phrase(Adj,Deg,Coor,Adj_P_R,ADJP)),
    (get_desired_output_structure(ADJP,Tokens,Out_Struct), !).
adj_phrase([]) --> [].

/* ..... */

rest_adj_phrase(Coor,[attrs(adj(Adj,Deg))|Adjs]) -->
    opt_comma, opt_real_coordinator(Coor),
    adj_equiv(Adj,Deg), rest_adj_phrase(Coor,Adjs), (!).
rest_adj_phrase(.,[]) --> [].

pack_adj_phrase(Adj,Deg,_,[],[attrs(adj(Adj,Deg))]) :- !.
pack_adj_phrase(Adj,Deg,Coor0,Adj_P_R,
    conj_adj_p(Coor,[attrs(adj(Adj,Deg))|Adj_P_R])) :-
    (var(Coor0), Coor = implicit_and) ; (nonvar(Coor0), Coor = Coor0).

/* ..... */

adj_equiv(Adj,Deg) --> adj_s(Adj,Deg).
adj_equiv(double_quotes(Adj),Deg) -->
    double_quotes, adj_s(Adj,Deg), double_quotes.
adj_equiv(Adj,pos) --> past_part(Adj).
adj_equiv(Adj,pos) --> pres_part(Adj).
adj_equiv(Adj,pos) --> adv_s(Adj).
adj_equiv(double_quotes(Adj),pos) --> double_quotes, adv_s(Adj), double_quotes.

past_part(Past_Part) --> verb__in_double_quotes_or_not(perf,_,_,Past_Part).
pres_part(Pres_Part) --> verb__in_double_quotes_or_not(progr,_,_,Pres_Part).

/* In order to reduce lexical ambiguity, we discard "false" adverbs.
/* ##note##: this may be a bit too restrictive ...
adv_s(A,I,O) :- adv(A,_,I,O), \+ conj(A,I,_, \+ prep(A,I,_.

/* ..... */
/* NOUNGP_HEAD (used by not_followed_by_a_np, object1_form, question, &
/*      rest_common_noun_np)
/*
/* The head of a noun group is a noun that may be preceded by pre-modifiers
/* and potentially followed by a post-modifier; see QUIRK, pp.1238+.
/* On 'string': see verb__in_double_quotes_or_not and np_nucleus.
/*

/* Noun gp. string type (1): "pre-modif noun post-modif"
nongp_head(NounType,Num,
    string(noun(Noun,
        noun_modifiers(pre_modif(Pre),post_modif(Post))))
-->

```

```

double_quotes, noun_pre_modifiers(Pre), {Pre \== []},
noun(NounType,Num,Noun), noun_post_modifier(Post), {Post \== nil},
double_quotes.

/* Noun gp. string type (2): "pre-modif noun" post-modif */
noungp_head(NounType,Num,
    noun(string_e(Noun),
        noun_modifiers(string_b(pre_modif(Pre)),post_modif(Post))))
-->
double_quotes, noun_pre_modifiers(Pre), {Pre \== []},
noun(NounType,Num,Noun), double_quotes, noun_post_modifier(Post).

/* Noun gp. string type (3): pre-modif "noun post-modif" */
noungp_head(NounType,Num,
    noun(string_b(Noun),
        noun_modifiers(pre_modif(Pre),string_e(post_modif(Post)))))
-->
noun_pre_modifiers(Pre), double_quotes, noun(NounType,Num,Noun),
noun_post_modifier(Post), {Post \== nil}, double_quotes.

noungp_head(NounType,Num,
    noun(Noun,noun_modifiers(pre_modif(Pre),post_modif(Post))))
-->
noun_pre_modifiers(Pre), noun(NounType,Num,Noun), noun_post_modifier(Post).

/*****
/* NOUN_PRE_MODIFIERS (used by noungp_head) */
/*
/* All nouns preceding the head noun are called "pre-modifiers".
/* Compound nouns can be constructed from:
/*      i) verb + noun (ex: swimming-pool, shooting star) [verb: -ing|inf]
/*      ii) noun + verb (ex: birth-control) [verb: -ing]
/*      iii) noun + noun (ex: motor-cycle, love story).
/* The pre-modifiers cover cases i and iii when dashes are not needed;
/* otherwise, it's a compound and treated as a single-token word.
/*
/* On NP pre-modification in general, see QUIRK, chap.17 (17.94-120).
*****/

noun_pre_modifiers(NPM) -->
    (noun_pre_modif(CL1) ; propernoun(CL1,_);
    (noun(_,_ ,CL1), not_followed_by_ing_word)),
    rest_noun_pre_modifiers(Coor,CL_R),
    {pack_noun_pre_modifiers(CL1,Coor,CL_R,NPM), !}).
noun_pre_modifiers({}) --> [].

not_followed_by_ing_word(I,I) :- ing_word(I), verb(_,_ ,_,I,_), !, fail.
not_followed_by_ing_word(I,I) :- !.

/* ..... */

rest_noun_pre_modifiers(Coor,[CL2|CL_R]) -->
    opt_real_coordinator(Coor),
    (noun_pre_modif(CL2) ; propernoun(CL2,_);
    (noun(_,_ ,CL2), not_followed_by_ing_word)),
    rest_noun_pre_modifiers(Coor,CL_R), {!}.
rest_noun_pre_modifiers(_,[]) --> [].

pack_noun_pre_modifiers(CL1,_,[],[CL1]) :- !.
pack_noun_pre_modifiers(CL1,Coor0,CL_R,conj_noun_pre_mod(Coor,[CL1|CL_R])) :-
    (var(Coor0), Coor = implicit_and) ; (nonvar(Coor0), Coor = Coor0).

/* ..... */

```

```

noun_pre_modif(quiz_token(X))      -->
  [dct(Type:X/_._)],
  {(Type =.. [quiz_token,_] ; Type == ref_or_var_or_quiz_item)}.

noun_pre_modif(ref_or_var(X))      --> [dct(ref_or_var:X/_._)].

noun_pre_modif(X)                  --> date_form(X).

noun_pre_modif(number_special(X))  --> [dct(number_special:X/_._)].

noun_pre_modif(dollars(X))         --> [dct(dollars:X/_._)].

noun_pre_modif(percentage(X))      --> [dct(percentage:X/_._)].

/* ***** */
/* NOUN (used by noun_pre_modifiers, noungp_head, np_equiv_form, */
/*   nucleus_equiv, & rest_noun_pre_modifiers) */
noun(countnoun,Num,Noun)           --> countnoun(Noun,Num).
noun(countnoun,Num,double_quotes(Noun)) -->
  double_quotes, countnoun(Noun,Num), double_quotes.

noun(massnoun,sg,Noun)             --> massnoun(Noun).
noun(massnoun,sg,double_quotes(Noun)) -->
  double_quotes, massnoun(Noun), double_quotes.

noun(Noun_Type,Num,ing_noun(Ing_Noun)) -->
  {(Noun_Type = countnoun ; Noun_Type = massnoun)},
  verb_clause(2,Num_vc,_,[progr],nil,nil,Ing_Noun,ing_noun),
  {(Num_vc == pl, Num = pl, Noun_Type = countnoun) ;
   (Num_vc \== pl, Num = sg,
    (Noun_Type = countnoun ; Noun_Type = massnoun))}.
noun(Noun_Type,Num,double_quotes(ing_noun(Ing_Noun))) -->
  double_quotes,
  verb_clause(2,Num_vc,_,[progr],nil,nil,Ing_Noun,ing_noun), double_quotes,
  {(Num_vc == pl, Num = pl, Noun_Type = countnoun) ;
   (Num_vc \== pl, Num = sg,
    (Noun_Type = countnoun ; Noun_Type = massnoun))}.

/* ----- NOUN EQUIVALENTS OR SUBSTITUTES ----- */

noun(quiz_token,_,quiz_token(X))  -->
  [dct(Type:X/_._)],
  {(Type =.. [quiz_token,_] ; Type == ref_or_var_or_quiz_item)}.

noun(quiz_example,_,quiz_example(X)) -->
  special_function_word(X,Type), {Type == frag}.

noun(special_symbol,_,special_symbol(X)) -->
  double_quotes, special_char_seq(X), double_quotes.

noun(character,_,character(X))     -->
  atomic_character(X), {X \== a, X \== i}.

noun(string,_,string(X))          -->
  double_quotes, get_string_tokens(X).

/* ***** */

special_char_seq([H|T]) -->
  [dct(special_char:H/_.[])], {H \== ''}, rest_special_char_seq(T).

rest_special_char_seq([H|T]) -->

```

```

        [dct(special_char:H/_,[ ]), (H \== ' '), rest_special_char_seq(T), (!).
rest_special_char_seq([ ]) --> [ ].

/* Remark: a string (e.g. a title used as an "address") is not parsed. */
get_string_tokens([H|T]) --> [dct(_:H/_,[ ]), (H \== ' '), get_string_tokens(T).
get_string_tokens([ ]) --> double_quotes.

atomic_character(X) -->
    double_quotes,
    [dct(word:X/all_caps,[ ]), (name(X,L), length(L,1)), double_quotes.
atomic_character(X) -->
    parenthesis_o,
    [dct(word:X/all_caps,[ ]), (name(X,L), length(L,1)), parenthesis_c.
atomic_character(X) -->
    [dct(word:X/all_caps,[ ]), (name(X,L), length(L,1)).

/*****
/* NOUN_POST_MODIFIER (used by noungp_head) */

noun_post_modifier(X) --> number_gen(X,[ ]), (!).

noun_post_modifier(quiz_token(X)) -->
    [dct(Type:X/_,[ ]),
    ((Type == [quiz_token,[ ] ; Type == ref_or_var_or_quiz_item), (!).

noun_post_modifier(ref_or_var(X)) --> [dct(ref_or_var:X/_,[ ]), (!).

noun_post_modifier(X) --> date_form(X), (!).

noun_post_modifier(number_special(X)) --> [dct(number_special:X/_,[ ]), (!).

noun_post_modifier(dollars(X)) --> [dct(dollars:X/_,[ ]), (!).

noun_post_modifier(percentage(X)) --> [dct(percentage:X/_,[ ]), (!).

noun_post_modifier(nil) --> [ ].

/*****
/* COMPL_ADJ_PHRASE (used by complement_stat) */
/*
/* Un adjectif peut avoir un complement qui est l'un des 6 groupes suivants: */
/* i) un groupe prepositionnel (prepositional phrase) */
/* ii) une proposition substantive introduite par 'that' (that clause) */
/* iii) une substantive avec un verbe a l'infinitif (to-infinitive clause) */
/* iv) une substantive avec un verbe en -ing (nominal -ing clause) */
/* v) une question indirecte (see wh_interro_declarative_clause above) */
/* vi) une proposition substantive introduite par 'than' (than clause). */
/*
/* Voir QUIRK, chap.16 ("Complementation of verbs and adjectives"); pp.1220+ */

compl_adj_phrase(_) --> end_of_sentence_marker(_), (!, fail).

compl_adj_phrase(Comparison) --> comparative_clause(Comparison).

compl_adj_phrase(attrs_compl(ADJP)) -->
    required_adj_phrase(Adj_P), rest_compl_adj_phrase(Adj_P,ADJP).

compl_adj_phrase(attrs_compl(Adj_P)) --> required_adverbs(Adj_P).

compl_adj_phrase(attrs_compl(Adj_C)) --> adj_complement(Adj_C).

/* ..... */

```



```

rest_compl_adj_phrase(Adj_P,[Adj_P,ADJP]) --> adj_complement(ADJP), (!).
rest_compl_adj_phrase(ADJP,ADJP)          --> [].

/* ++++++ */

adj_complement(AC) --> required_prepphrases(AC,_).

adj_complement(AC) --> nominal_clause(that_clause,AC,_,_).

adj_complement(AC) --> nominal_clause(to_infinite_clause,AC,_,_).

adj_complement(AC) --> nominal_clause(ing_clause,AC,_,_).

adj_complement(AC) --> nominal_clause(wh_interro_declarative_clause,AC,_,_).

adj_complement(nonfinite_rel_clause_equiv(for(NpX),to_infinite_clause(X)))
-->
    prep(for), np_equiv(pre_v,2,NpX,_), simple_to_infinite_clause(2,X).

/* ***** */
/* REQUIRED_ADVERBS (used by adv_group, adverbial_clause, apposition_core, */
/* complement_stat, opt_particle, prepphrase_core, & sentence_frag) */
/* & ADVERBS (used by be_going_to, have_to, opt_advs_negation, question, */
/* required_adverbs, verbal, verbseq__Be__aux, & */
/* verbseq__Have_or_Do__aux) */

required_adverbs(Advs) --> adverbs(Advs), {Advs \== []}.

adverbs([],I,I) :-
    end_of_sentence_marker(_,I,_), !.
adverbs(Out_Struct) $ Tokens -->
    (adv_s(Adv1) ; adv_not_s(Adv1)), rest_adverbs(Coor,Advs_R),
    {pack_adverbs(Adv1,Coor,Advs_R,ADVS)},
    {get_desired_output_structure(ADVS,Tokens,Out_Struct), !}.
adverbs([]) --> [].

/* ..... */

rest_adverbs(Coor,[Adv2|Advs_R]) -->
    opt_comma, opt_real_coordinator(Coor), (adv_s(Adv2) ; adv_not_s(Adv2)),
    rest_adverbs(Coor,Advs_R), (!).
rest_adverbs(_,[]) --> [].

pack_adverbs(Adv1,_,[],[Adv1]) :- !.
pack_adverbs(Adv1,Coor0,Advs_R,[conj_advs(Coor,[Adv1|Advs_R])]) :-
    (var(Coor0), Coor = implicit_and) ; (nonvar(Coor0), Coor = Coor0).

adv_not_s(A,I,O) :- adv(A,_,I,O), (conj(A,I,_) ; prep(A,I,_) ).

/* ~~~~~ */
/* Punctuation and a few special characters used throughout the parser. */

opt_pause --> pause, (!).
opt_pause --> [].

pause --> colon ; comma ; dashes ; semi_colon.

opt_colon --> colon, (!).
opt_colon --> [].

```

```

opt_comma    --> comma, {!}.
opt_comma    --> [].

colon        --> [dct(punctuation:':'/pause,[])].
comma        --> [dct(punctuation:','/pause,[])].
dashes       --> [dct(punctuation:'--'/pause,[])].
semi_colon   --> [dct(punctuation:';/pause,[])].

double_quotes --> [dct(special_char:''/_,[[])].

parenthesis_o --> [dct(special_char: '('/_,[[])].
parenthesis_c --> [dct(special_char: ')'/_,[[])].

question_mark --> [dct(punctuation:'?'/_,[[])].

/* ..... */
/* TERMINAL (used by compl_sentence) */
/* ..... */
/* On tags, see QUIRK, pp.983-4. */
/* ..... */

terminal(end_of_input(Tag,Terminal)) -->
    opt_comma, tag_coor(Tag), end_of_sentence_marker(Terminal).
terminal(end_of_input(Terminal))      --> end_of_sentence_marker(Terminal).

/* ..... */

tag_coor(Tcoor) --> [dct(word:Tcoor/_,_), {tag_coordinator(Tcoor)}].
tag_coor(Tcoor) -->
    [dct(word:Tcoora/_,_), dct(word:Tcoorb/_,_), dct(word:Tcoorc/_,_),
     dct(word:Tcoord/_,_), dct(word:Tcoore/_,_), dct(word:Tcoorf/_,_)],
    {tag_coordinator([Tcoora,Tcoorb,Tcoorc,Tcoord,Tcoore,Tcoorf],Tcoor)}.
tag_coor(Tcoor) -->
    [dct(word:Tcoora/_,_), dct(word:Tcoorb/_,_), dct(word:Tcoorc/_,_)],
    {tag_coordinator([Tcoora,Tcoorb,Tcoorc],Tcoor)}.

tag_coordinator(and_so_forth).
tag_coordinator(and_so_on).
tag_coordinator(and_so_on_and_so_forth).
tag_coordinator(and_the_like).
tag_coordinator(etc).

tag_coordinator([and,so,forth],and_so_forth).
tag_coordinator([and,so,on,and,so,forth],and_so_on_and_so_forth).
tag_coordinator([and,so,on],and_so_on).
tag_coordinator([and,the,like],and_the_like).

/* ..... */

end_of_sentence_marker(period)      -->
    [dct(punctuation:'.'/end_of_sentence,[])].
end_of_sentence_marker(ellipsis)    -->
    [dct(punctuation:'...'/end_of_sentence,[])].
end_of_sentence_marker(exclamation_mark) -->
    [dct(punctuation:'!'/end_of_sentence,[])].

/* ..... */

```

```

/* OPT_NEGATION (used by be_going_to_g, opt_advs_negation, sentence_core,
/*      verbal, & verbseq)

opt_negation(no) --> special_function_word(not,_), (!).
opt_negation(yes) --> [].

/*****
/* OPT_ADVS_NEGATION (used by be_going_to_g, have_to, verbseq,
/*      verbseq_Be__aux, verbseq_Have_or_Do__aux, & verbseq_HavePe__aux)

opt_advs_negation(Advs,Neg) --> opt_negation(Neg), adverbs(Advs).
opt_advs_negation(Advs,Neg) -->
    adverbs(Advs), {Advs \== [not]}, opt_negation(Neg).

/*****
/* OPT_PARTICLE (used by verbal)

opt_particle(_,nil) --> [].

opt_particle(V,[Advs,Par]) -->
    {(V =.. [double_quotes,MainVb] ; MainVb = V)},
    required_adverbs(Advs), adv_particle(Par),
    {adv_particle_list(MainVb,List_of_Par), memberchk(Par,List_of_Par)}.

opt_particle(V,Par) -->
    {(V =.. [double_quotes,MainVb] ; MainVb = V)},
    adv_particle(Par),
    {adv_particle_list(MainVb,List_of_Par), memberchk(Par,List_of_Par)}.

/*****
/* OPT_SYN_ELEM (used by that_clause, wh_i_d_clause, & wh_word_of)

opt_syn_elem(Syn_Elem,In,Out) :-
    Syn_Elem =.. [Pred|Args], append(Args,[In,Out],Args2),
    Syn_Elem2 =.. [Pred|Args2], call(Syn_Elem2), !.

opt_syn_elem(_,In,In).

/*****
/* OPT_REAL_COORDINATOR (used by rest_adj_phrase, rest_adverbs,
/*      rest_noun_pre_modifiers, & rest_sub_cl)
/* & REAL_COORDINATOR_3 (used by card_num_c, conj_nom_cl, conj_np, conj_pp,
/*      opt_real_coordinator, prep_pp, & rest_verbseq_core)

opt_real_coordinator(Coor) --> real_coordinator_3(Coor), (!).
opt_real_coordinator(_) --> [].

real_coordinator_3(Coor) --> [dct(word:Coor/_,_), {real_coordinator(Coor)}].

/*****

```



```

ctxt_marker => "for", "example", opt_comma.
ctxt_marker => "examples", colon.
ctxt_marker => "for", "instance", opt_comma.
ctxt_marker => "no", colon.
ctxt_marker => "note", colon.
ctxt_marker => "question", colon.
ctxt_marker => "tax", "tip", colon.
ctxt_marker => "yes", colon.
ctxt_marker => np_equiv, colon.
ctxt_marker => [].

/*****/

top_level_clause_seeker_r(non_compound, ... =>
    sentence, rest_sentence.

top_level_clause_seeker_r(compound, ... =>
    call_tentative_sentence, rest_sentence.

top_level_clause_seeker_r(mult_coor, ... =>
    potential_sent_coor,
    call_tentative_sentence, multiple_coord_clauses.

call_tentative_sentence          => sentence.

/*****/
/* These rule cover the multiple coordination of clauses (see QUIRK, p.925). */
/* In this structure, 'and' and 'or' can link more than two clauses and all */
/* but the final instance of these conjunctions can be omitted. For example: */
/* "The battery may be disconnected, the connection may be loose, or the bulb */
/* may be faulty", where the 2nd clause means ", OR the connection may ...." */
/* */
/* Notation: 'syndetic': with an explicit coordinator; 'asyndetic': implicit */
/* coordinator which is temporarily "postponed" (thus the 'p_'). */
/* */

/* Case (1) : SENT (',' ) and/or SENT ... */
multiple_coord_clauses =>
    opt_comma, conj, call_tentative_sentence, multiple_coord_clauses.

/* Case (2) : SENT ', ' SENT ... */
multiple_coord_clauses =>
    comma, call_tentative_sentence, multiple_coord_clauses.

multiple_coord_clauses =>
    end_of_sentence_marker.

/*****/
/* Actual examples in the QUIZ manual have been replaced by one of these */
/* special markers: 'quiz-example-command', 'quiz-example-commands', */
/* 'quiz-example-program', 'quiz-example-prompt', & 'quiz-example-report'. */
/* They are classified in the dictionary as special_function_word(s). */

opt_quiz_example => opt_ex_marker, colon, special_function_word.
opt_quiz_example => [].

/* An example marker is "as in", "for example", or "like this". */
opt_ex_marker => opt_comma, "as in".
opt_ex_marker => opt_comma, "for example".
opt_ex_marker => opt_comma, "like this".
opt_ex_marker => [].

/*****/

sentence_frag => "for example", colon, special_function_word.

```

```

sentence_frag => np_equiv.
sentence_frag => required_prepphrases.
sentence_frag => required_adj_phrase.
sentence_frag => required_adverbs(Fragment).

/*****
/* Two types of sentence are recognized: DECLARATIVE and IMPERATIVE.      */
/* EXCLAMATIVES are treated as declaratives. On QUESTIONS, see dipett_9.    */
*/

sentence => subordinate_clause, sentence_core.

/* The special HERE+BE: "HERE IS the solution.", "HERE ARE the tools.", etc. */
/* See QUIRK, pp.521-3 and 1379-81.                                          */
sentence_core => "here", verbphrase.

/* The special existential-THERE sentence: "THERE ARE people in the room." */
/* Note: other verbs than BE are also allowed; see QUIRK, pp.1402-14.      */
sentence_core => "there", verbphrase, rest_there_sent.

/* The general case: a NP followed by a VP.                                */
sentence_core => np_equiv, verbphrase.

/* Imperative with "let". On directives, see QUIRK, pp.827+.                */
sentence_core => opt_courtesy_subjunct,
               preposed_verb_let, (let_pron ; np_equiv), opt_negation,
               verbphrase.

/* Plain imperative sentence: it must begin with a verb.                    */
sentence_core => opt_courtesy_subjunct, verbphrase.

/* ..... */

/* Recognizes the rest of the "enumerative" existential-THERE sentence.      */
/* Ex: "There are two types of keys: UNIQUE AND REPEATING."                 */
rest_there_sent => colon, sentence_frag.
rest_there_sent => [].

opt_courtesy_subjunct => "please", opt_comma.
opt_courtesy_subjunct => [].

let_pron =>
    "her" ; "him" ; "it" ; "me" ; "them" ; "us" ; "yourself" ; "yourselves".

/*****
/* Rules for compounds (#3-4) allow an unbound number of coordinated clauses */
/* via top_level_clause_seeker which uses sentence and rest_sentence.        */
*/

/* "single-main-clause" input: SIMPLE SENTENCE.                            */
rest_sentence(simple_sentence( ... => ...

/* "single-main-clause" input: COMPLEX SENTENCE.                            */
rest_sentence(complex_sentence( ... => ...

/* Coordinated main clauses (compound): FIRST COORDINATOR IS "EMPTY".        */
rest_sentence(compound_sentence( ... =>
    opt_pause, coordinator, opt_coor_conj, top_level_clause_seeker_r.

/* Coordinated main clauses (compound): CORRELATIVES.                        */
rest_sentence(correlative_compound_sentence( ..=>
    opt_pause, coordinator, top_level_clause_seeker_r.

/* ..... */

```

```

/* These five coordinators do not allow another conjunction to precede them: */
/* see QUIRK, pp.922-3. */
opt_coor_conj => conj(C), {\+ memberchk(C,[and,or,but,for,so_that])}.
opt_coor_conj => [].

/*****
/* Allows a conjunction of verb phrases. Here "tense" includes modality: */
/*      Time: temps chronologique; Tense: temps des verbes: */
/*      Aspect: facon dont l'action s'accomplit (debut, duree, etc.); */
/*      Modality: nuances dans l'accomplissement de l'action. */

verbphrase => verbphrase_Reg_or_Stat, rest_verbphrase.

/* Verb phrases are divided into two major types: REGULAR (action verbs), */
/* and STATIVE ("to be", "to become", "to appear", etc.). Within regular */
/* verb phrases, the passive and active voices are distinguished. */

verbphrase_Reg_or_Stat => verb_clause(active).
verbphrase_Reg_or_Stat => verb_clause(passive).
verbphrase_Reg_or_Stat => verb_clause(stative).

/*****
/* The verbphrase and rest_verbphrase rules recognize these structures: */
/*      1) single-VP, 2) conjoined-VPs, 3) single-VP '(' single-VP ')', and */
/*      4) single-VP '(' conjoined-VPs ')'. */
/* However, the same rules do not recognize structures like #3 & 4 in which */
/* the first element, i.e. single-VP, would be replaced by conjoined-VPs. */
/* */
/* It should be noted that this remark also applies to these structures: */
/*      i) rest_np_nucleus (with NPs instead of VPs), */
/*      ii) rest_nom_cl (with nominal clauses instead of VPs), and */
/*      iii) rest_prepphrases (with PPs instead of VPs). */

rest_verbphrase => parenthesis_o, conj_vp, parenthesis_c.
rest_verbphrase => conj_vp.

/* ..... */

/* conj_vp requires that Conj be overt but not necessarily unique. It thus */
/* accepts structures like "A and B then C". These are relatively rare and */
/* may be rather ambiguous (e.g. with 'and' and 'or'). See QUIRK pp.920+. */
/* Note: Conj must be overt AND unique for conj_np, conj_nom_cl, and conj_pp. */
/* */
/* The N parameter indicates whether we are currently trying to parse a VP */
/* that follows the very first VP. If so, we don't allow "VP ',' CONJ VP". */
/* Unfortunately, this discards cases like "VP ',' CONJ VP ',' CONJ VP". */
/* However, this seems to be the price to pay to reduce ambiguity in the par- */
/* sing of conj. items: only a costly look-ahead mechanism might be helpful. */
/* This also applies to conj_np, conj_nom_cl, and conj_pp (in dipett_8). */

/* Case (1) : VP ('(',') CONJ VP ... */
conj_vp => opt_comma, conj, verbphrase_Reg_or_Stat, conj_vp.

/* Case (2) : VP ',' VP ... */
conj_vp => comma, verbphrase_Reg_or_Stat, conj_vp.

/*****
/* These rules extract the verb sequence with its FINAL modifiers (if any). */

/* General verb clause (TYPE 1). */
verb_clause(1, ... =>
    verbphrase_RegAct_or_RegPass,
    ((opt_pause, required_subordinate_clause) ; true).

```

```

/* Simplified verb clause (TYPE 2): no final modifiers allowed. */
verb_clause(2, ... =>
    verbphrase_RegAct_or_RegPass.

/*****
/* Handles the regular verb phrases, active or passive. When M_el is present */
/* as last argument, it signals the presence of an empty NP (see rel_clause).*/

verbphrase_RegAct_or_RegPass =>
    verbseq, medial_vp_subordinate_clause, complement.

medial_vp_subordinate_clause => [].
medial_vp_subordinate_clause =>
    med_correlative_marks_o(N),
    (adverbial_subordinate ; nominal_subordinate),
    med_correlative_marks_c(N).

/* ..... */

med_correlative_marks_o(1) => comma.
med_correlative_marks_o(2) => dashes.
med_correlative_marks_o(3) => parenthesis_o.

med_correlative_marks_c(1) => comma.
med_correlative_marks_c(2) => dashes.
med_correlative_marks_c(3) => parenthesis_c.

/*****
/* The basic form of a verb sequence is based on Winograd's description: */
/* "modal_aux | be_going_to | have/do | be_1 | be_2 | verb" */
/* be_1: information about the continuity of the action. */
/* be_2: passive or active form. */
/* */
/* Ou encore: i) modal + be + verbe en -ing */
/* ii) modal + have + participe passe */
/* iii) modal + have been + participe present. */
/* */
/* All these different elements, except the verb of course, are optional: */
/* see QUIRK, chap.3; on marginal modals & modal idioms, see pp.138+. */

/* The verb sequence contains a "HAD BETTER" modal idiom. */
verbseq =>
    (modal_aux(had) ; modal_aux(had_would)), "better",
    opt_negation, verbseq_BeGoingTo_or_else.

/* The verb sequence contains a "WOULD RATHER" modal idiom. */
verbseq =>
    (modal_aux(would) ; modal_aux(had_would)), "rather",
    opt_negation, verbseq_BeGoingTo_or_else.

/* The verb sequence contains a modal (different from the marginals below). */
verbseq =>
    modal_aux(Modal),
    {Modal \== dare, Modal \== need, Modal \== ought_to, Modal \== used_to},
    opt_adv_negation, verbseq_BeGoingTo_or_else.

/* The verb sequence contains a "DARE" marginal modal. */
verbseq =>
    modal_aux(dare), opt_negation, verbseq_BeGoingTo_or_else.

/* The verb sequence contains a "NEED" marginal modal. */
verbseq =>
    modal_aux(need), opt_negation, verbseq_BeGoingTo_or_else.

```



```

/* The verb sequence contains a "OUGHT TO" marginal modal. */
verbseq =>
    modal_aux(ought_to), opt_negation, "to", verbseq__BeGoingTo_or_else.

/* The verb sequence contains a "USED TO" marginal modal. */
verbseq =>
    modal_aux(used_to), opt_negation, "to", verbseq__BeGoingTo_or_else.

/* The verb sequence does not contain a modal. */
verbseq =>
    verbseq__BeGoingTo_or_else.

/*****
/* The "BE_GOING_TO" and "HAVE_TO" cases. */

verbseq__BeGoingTo_or_else =>
    (be_going_to ; have_to), (verbseq__HaveBe__aux ; verbseq_core).

/* The GENERAL case ... */
verbseq__BeGoingTo_or_else =>
    verbseq__Have_or_Do__aux.

/*****
/* The "BE_GOING_TO" (near future) &
/* "HAVE_TO" (obligation or logical necessity) SEMI-AUXILIARIES:
/* see QUIRK, pp.143+ (semi-auxiliaries).

be_going_to => adverbs, be_going_to_g, "going", "to".

be_going_to_g => aux, opt_advs_negation, aux, opt_negation.
be_going_to_g => aux, opt_advs_negation.

have_to => adverbs, aux, opt_advs_negation, "to".

/*****
/* ----- HAVE as an auxiliary ----- */

/* Modal_flag = 1 : for NON-MODALS. */
verbseq__Have_or_Do__aux(1, ..=>
    adverbs, aux, opt_advs_negation, verbseq__HaveBe__aux.

/* Modal_flag = 2 : for MODALS. */
verbseq__Have_or_Do__aux(2, ..=>
    adverbs, aux, opt_advs_negation, verbseq__HaveBe__aux

/* ----- DO as an auxiliary ----- */

verbseq__Have_or_Do__aux>
    adverbs, aux, opt_advs_negation, verbseq_core.

/* No HAVE or DO auxiliaries found in the verb sequence ... */
verbseq__Have_or_Do__aux>
    verbseq__Be__aux.

/* ----- BE as an auxiliary ----- */

/* first two clauses: when following HAVE ... */
verbseq__HaveBe__aux=> aux, opt_advs_negation, verbseq_core.
verbseq__HaveBe__aux=> verbseq__Be__aux.

/* ..... */

/* Modal_flag = 1 : for NON-MODALS; the "BE TO" modal idiom. */
verbseq__Be__aux(1, ...=>

```

```

    adverbs, aux, opt_advs_negation, "to", verbseq_core.

/* Modal_flag = 1 : for NON-MODALS, the general case. */
verbseq__Be__aux(1, ... =>
    adverbs, aux, opt_advs_negation, verbseq_core.

/* Modal_flag = 2 : for MODALS. */
verbseq__Be__aux(2, ... =>
    adverbs, aux, opt_advs_negation, verbseq_core.

/* ----- No auxiliaries found ! ----- */
verbseq__Be__aux => verbseq_core.

/*****
/* These rules recognize the simple (usual) case in which a VP contains a
/* single main verb. They also recognize the more complex case in which a
/* sequence of conjoined main verbs is used instead. Although the latter can
/* also be analysed in terms of conjoined verb phrases with several elided
/* constituents, we follow the analysis of QUIRK (see pp.943+ & pp.967+) and
/* consider it as a case of conjoined main verbs with, often, an implied
/* chronological sequence. Here are two typical examples taken from QUIRK:
/* Ex.1: "Paul WASHED AND DRIED the dishes."
/* Ex.2: "He WAS BORN, LIVED, AND DIED in the city of Vienna."
*****/

verbseq_core => verbal, rest_verbseq_core.

/* ..... */

rest_verbseq_core=>
    opt_comma, real_coordinator_3, verbal, rest_verbseq_core.
rest_verbseq_core=>
    comma, verbal, rest_verbseq_core.
rest_verbseq_core=> [].

/* ***** */

/* ACTIVE VERB */
verbal =>
    adverbs, verb__in_double_quotes_or_not, opt_particle, opt_negation.

/* "PASSIVE" VERB */
verbal =>
    adverbs, aux, opt_negation, verb__in_double_quotes_or_not, opt_particle.

/* STATIVE VERB */
verbal =>
    adverbs,
    ((verb__in_double_quotes_or_not, opt_negation) ;
    (opt_negation, verb__in_double_quotes_or_not)).

/*****
/* We recognize verbs enclosed in double quotes (e.g. "tells", "enter").
/* A word W in double quotes is represented as double_quotes(W),
/* but a sequence of words SW in double quotes is represented as string(SW).
*****/

verb__in_double_quotes_or_not => double_quotes, verb, double_quotes.
verb__in_double_quotes_or_not => verb.

/*****
/* Defines the different types of complement accepted by a verb sequence:
/* intr = intransitive : verbs that do not occur with objects.
/* stat = stative : copulas (see verb_clause & set_struct).
/* tr = transitive : verbs that occur with one or more object.
*****/

```

```

/*      tr_intr : verbs that can be used either as intr or as tr.      */

complement      => complement_tr.
complement      => complement_intr.
complement      => complement_stat..

/* Some rules apply to ditransitives as well as to complex transitives; */
/* see QUIRK, pp.1170+ (16.20: "Types of verb complementation").        */
/*                               */
/* Notation: S(subject), V(erb), O(bject), C(omplement), A(dverbial).    */
/*                               */
/* ----- ACTIVE STRUCTURES ----- */

complement_tr(active,referring_np(M_el),svoo_svoc_svoa( ... =>
    np_equiv.

complement_tr(active,referring_np(M_el),svoo_svoc_svoa( ... =>
    required_prepphrases.

complement_tr(active,referring_np(M_el),svo( ...      =>
    []).

complement_tr(active,def_prep(M_el),svoo_svoc_svoa( ... =>
    np_equiv, prep, not_followed_by_a_np.

complement_tr(active,def_prep(M_el),svo( ...      =>
    prep, not_followed_by_a_np.

complement_tr(active,nil,svoc( ...      =>
    comparative_clause.

complement_tr(active,nil,svo( ...      =>
    np_equiv ; required_prepphrases.

complement_tr(active,nil, ...      =>
    np_equiv, rest_complement_tr_act_np.

complement_tr(active,nil,svoo_svoc_svoa( ...      =>
    required_prepphrases, np_equiv.

/* ----- PASSIVE STRUCTURES ----- */

complement_tr(passive,referring_np(M_el),svoo( ...      =>
    by_phrase.

complement_tr(passive,nil, ...      =>
    by_phrase, rest_complement_tr_pass_by.

complement_tr(passive,nil, ...      =>
    required_prepphrases, rest_complement_tr_pass_pp.

complement_tr(passive,nil, ...      =>
    np_equiv, rest_complement_tr_pass_np.

complement_tr(passive,nil,svoo_svoc_svoa( ...      =>
    required_adj_phrase, by_phrase.

/* ..... */

rest_complement_tr_act_np( ... svoc( ...      =>
    comparative_clause.

rest_complement_tr_act_np( ... svoo_svoc_svoa( ...      =>

```

```

    np_equiv.

rest_complement_tr_act_np( ... svoo_svoc_svoa( ... =>
    required_prepphrases.

rest_complement_tr_act_np( ... svoc_svoa( ... =>
    required_adj_phrase.

/* ..... */

rest_complement_tr_pass_by( ... svoo_svoc_svoa( ... =>
    np_equiv ; required_prepphrases.
rest_complement_tr_pass_by( ... svo( ... => [].

rest_complement_tr_pass_pp( ... svoo_svoc_svoa( ... =>
    by_phrase.
rest_complement_tr_pass_pp( ... svoo( ... => [].

rest_complement_tr_pass_np( ... svoo_svoc_svoa( ... =>
    by_phrase.
rest_complement_tr_pass_np( ... svoo( ... => [].

/* ++++++ */
/* Note: in fact this might be too restrictive. What we really mean is "not */
/* followed by something that can ONLY function as a NP". Yes, tricky indeed.*/

not_followed_by_a_np=> (noungp_head ; np_nucleus), required_adverbs.
not_followed_by_a_np=> noungp_head, fail.
not_followed_by_a_np=> np_nucleus, fail.
not_followed_by_a_np=> [].

/* Note: this rule recognizes "by Tom and Bob" but not "by Tom and by Bob". */
by_phrase => "by", np_equiv.

/* ##### */
/* INTR (i.e. intransitives) */

complement_intr(active,nil,sv( ... => [].

/* ##### */
/* STAT (i.e. statives or copulars) */

complement_stat(stative,referring_np(M_el),svc_sva( .=> [].

complement_stat(stative,nil,svc_sva( ... =>
    compl_adj_phrase.

complement_stat(stative,nil,svc_sva( ... =>
    np_equiv.

/* ..... */
/* Sentence modifiers may occur at the beginning or at the end of a sentence */
/* or a verbphrase. They generally affect the meaning of the sentence by */
/* specifying particular circumstances, time or manner constraints on the */
/* main clause. Thus, relative to the latter, they are subordinate clauses. */
/* */
/* Note: "modifiers" and "subordinate clauses" are used as synonyms here. */

required_subordinate_clause=> subordinate_clause(RCM), {RCM \== []}.

subordinate_clause .=> [].
subordinate_clause => modif, rest_sub_cl.

/* ..... */

```

```

/* The following rules require that an initial "modifier" must be followed by*
/* a comma if it's not a simple adverbial clause. Otherwise, it's optional.  */

rest_sub_cl => opt_comma, opt_real_coordinator, modif, rest_sub_cl.
rest_sub_cl => [].
rest_sub_cl => comma.
rest_sub_cl => opt_comma.

/*****

modif => modif_phrase.

/* Major categories of subordinate clauses: adverbial and nominal clauses.  */
/* Others are sentential relative & comparative (see QUIRK, pp.1047+).        */
/* Note: sentential relative is syntactically similar to nominal relative    */
/* clause (though semantically different), thus, not included here.          */
/* For more details, see QUIRK p.1118. On subordinators, see QUIRK, pp.998+. */

modif_phrase => comparative_subordinate.
modif_phrase => adverbial_subordinate.
modif_phrase => nominal_subordinate.

comparative_subordinate=> comparative_clause.

adverbial_subordinate => adverbial_clause.
adverbial_subordinate => adv_group.
adverbial_subordinate => opt_subord_conj, required_prepphrases.

nominal_subordinate => opt_subord_conj, nominal_clause.

/* ..... */

opt_subord_conj => conj(Sub_Conj), {\+ real_coordinator(Sub_Conj)}.
opt_subord_conj => adv(conj).
opt_subord_conj => [].

/*****

adverbial_clause =>
    (required_adverbs ;
    (opt_subord_conj(Adv_or_Conj), (Adv_or_Conj \== nil))),
    rest_adverbial_clause.

rest_adverbial_clause => nominal_clause.
rest_adverbial_clause => sentence_core_s.
rest_adverbial_clause => past_participle_phrase.

/* ..... */

adv_group => required_adverbs.
adv_group => compare_p_adv, (np_equiv ; sentence_core_s).

compare_p_adv=> compar_degree_marker, adv_s, "than".
compare_p_adv=> compar_degree_marker, adj, "than".

/*****
/* Similar to 'sentence_core' except that we do not allow subordinate_clause */
/* or np_equiv of type 1 in order to avoid potential infinite left recursion.*/

sentence_core_s => np_equiv, verbphrase(2, ... ).

/*****
/* COMPARATIVE_CLAUSE (used by comparative_subordinate, compl_adj_phrase, */

```



```

nounphrase => either_or_np.
nounphrase => np_nucleus_and_mod. rest_np_nucleus.

/* EITHER/OR Nps.
either_or_np( ... conj_nps( ... =>
    "either", np_nucleus_and_mod, "or", np_nucleus_and_mod.
/* NEITHER/NOR Nps.
either_or_np( ... conj_nps( ... =>
    "neither", np_nucleus_and_mod, "nor", np_nucleus_and_mod.

rest_np_nucleus( ... conj_nps( ...=>
    parenthesis_o, conj_np, parenthesis_c.
rest_np_nucleus( ... conj_nps( ...=>
    conj_np.
rest_np_nucleus( ... => [].

/* ..... */

/* Case (1) : NP (',' ) COOR NP ...
conj_np => opt_comma, real_coordinator_3, np_nucleus_and_mod, conj_np.

/* Case (2) : NP ',' NP ...
conj_np => comma, np_nucleus_and_mod, conj_np.

conj_np => [].

/*****

nominal_clause => either_or_nom_cl.
nominal_clause => nom_cl, rest_nom_cl.

/* EITHER/OR nominal clauses.
either_or_nom_cl( ... conj_nom_cl( ... => "either", nom_cl, "or", nom_cl.
/* NEITHER/NOR nominal clauses.
either_or_nom_cl( ... conj_nom_cl( ... => "neither", nom_cl, "nor", nom_cl.

rest_nom_cl( ... conj_nom_cl( ...=>
    parenthesis_o, conj_nom_cl, parenthesis_c.
rest_nom_cl( ... conj_nom_cl( ...=>
    conj_nom_cl.
rest_nom_cl( ... => [].

/* ..... */

/* Case (1) : NP_subst (',' ) COOR NP_subst ...
conj_nom_cl =>
    opt_comma, real_coordinator_3, nom_cl, conj_nom_cl.

/* Case (2) : NP_subst ',' NP_subst ...
conj_nom_cl( ... [p_asyndetic_conj_nom_cl( ... =>
    comma, nom_cl, conj_nom_cl.

conj_nom_cl => [].

/*****
/* Following QUIRK, these are the four most common types of nominal clauses. */

nom_cl => to_infinitive_clause(2, ... ).
nom_cl => ing_clause(2, ... ).
nom_cl => wh_interro_declarative_clause.
nom_cl => that_clause.

/*****
/* Ex: "TO COMPLETE THIS WORK IN TWO DAYS is definitely impossible." */

```

```

/* Ex: "Tom would prefer FOR THE CHILDREN TO FINISH THEIR SOUP." */

to_infinitive_clause( ... to_infinitive_clause( ...=>
    simple_to_infinitive_clause.
to_infinitive_clause( ... to_infinitive_clause( ...=>
    interro_pron, simple_to_infinitive_clause.
to_infinitive_clause( ... to_infinitive_clause( ...=>
    "for", np_equiv, simple_to_infinitive_clause.

simple_to_infinitive_clause=> "to", verb_clause.

/*****
/* Gerundive: when a verb in present participle form is used as a noun. */
/* Ex.1: "LAUGHING WITH FRIENDS makes you live longer.", */
/* Ex.2: "MY FORGETTING her name was very unfortunate." */
/* See QUIRK, pp.1063+ (including the genitive case -- see Ex.2 above). */

ing_clause( ... genitive_ing_clause( ...=> ic_deter_pron, verb_clause.
ing_clause( ... ,ing_clause( ... => verb_clause.

ic_deter_pron=>
    "her" ; "his" ; "its" ; "my" ; "our" ; "their" ; "your".

/*****
/* Note: covers the nominal relative clause (see QUIRK, pp.1244+). */

wh_interro_declarative_clause(
    wh_interro_declarative_clause(wh_subj( ...=>
        there_must_be_a_wh_word, wh_i_d_clause.
wh_interro_declarative_clause(
    wh_interro_declarative_clause( ... =>
        whether_if_clause.

there_must_be_a_wh_word=> interro_pron.

/*****

/* Ex: "I want to know WHAT TO EXPECT ... " */
wh_i_d_clause=> interro_pron, simple_to_infinitive_clause.

/* Ex: "I want to know WHAT WAS DONE ... " */
wh_i_d_clause=> interro_pron, verbphrase.

/* Ex: "I want to know WHICH PEOPLE HAVE PRINTED THESE FILES." */
wh_i_d_clause=> interro_pron, sentence_for_wh_i_d_clause.

/* Ex: " I want to know WHICH FILE YOU HAVE PRINTED." */
wh_i_d_clause=> (wh_word_of ; true), np_equiv, sentence_for_wh_i_d_clause.

/* Ex: "I want to know WHAT YOU HAVE DONE." */
wh_i_d_clause=> (wh_word_of ; true), sentence_core_s.

/* ..... */

sentence_for_wh_i_d_clause =>
    np_equiv, verbphrase(2, ... , referring_np(...)).

wh_word_of => interro_pron, ("of" ; true).

/*****
/* This clause is also called the "yes-no" (interrogative) clause: for more */
/* details, see QUIRK, pp. 1053+. */

/* Ex: "I don't know WHETHER TO SEE MY DOCTOR TODAY." */

```



```

whether_if_clause(whether_if_clause( ...=>
    "whether", opt_or_not, simple_to_infinitive_clause, opt_or_not.

/* Ex: "It's not clear to me IF SHE LIKES THE PRESENT."
whether_if_clause(whether_if_clause( ...=>
    "if", sentence_core_s, opt_or_not.

/* Ex: "It's not clear to me WHETHER SHE LIKES THE PRESENT."
whether_if_clause(whether_if_clause( ...=>
    "whether", opt_or_not, sentence_core_s, opt_or_not.

opt_or_not    => "or not".
opt_or_not    => [].

/*****
/* The 2nd rule handles the case in which "that" is optional, that is when
/* this clause functions as a complement: "I hope (THAT) HE ARRIVES SOON.";
/* see QUIRK, pp.1179+. Unfortunately, the 2nd rule tends to overgeneralize. */

that_clause(that_clause( ... => "that", sentence_core_s.
that_clause(that_clause( ... => ("that" ; true), sentence_core_s.

/*****
/* The nounphrase is a common noun, a proper noun, or one of its equivalent
/* forms (see nucleus_equiv below), potentially followed by a modifier. Noti-
/* ce that several of these equiv. can also behave as noun equivalents, see
/* details in noungp_head.
/* See QUIRK, chap.5 ("Nouns and determiners") & 17 ("The noun phrase").
/*
/* Nounphrase postmodifiers are allowed only if NPPM == 1 (not if NPPM == 0).
/* For the 'proper noun' special case, NPPM == 2 (see details in np_nucleus).*/

np_nucleus_and_mod => np_nucleus.

/*****
/* The first parameter indicates whether the nounphrase (nucleus) can be fol-
/* lowed by a modifier (e.g. a relative clause): 0 means 'no', and 1 'yes';
/* 2 means "only those of the pre-verbal position (pre_v)".
/*
/* COMMON NOUN
np_nucleus(1,entity( ..=>
    (true ; adv_s), determinatives, rest_common_noun_np.

/* GENITIVE (with implicit ellipsis)
np_nucleus(0,entity( ..=>
    genitive.

/* EQUIVALENT FORM
np_nucleus(0,entity( ..=>
    (true ; adv_s), nucleus_equiv.

/* PROPER NOUN
np_nucleus(2,entity( ..=>
    adj_phrase, propernoun.

/* ADJECTIVE SUBSTANTIVE CLAUSE
np_nucleus(1, ... =>
    adj_substantive_clause.

/* *****
/* EQUIV: PERSONAL PRONOUN
nucleus_equiv(ref(pers_pron( ...    => partiv_pron, pers_pron.

```

```

nucleus_equiv(ref(pers_pron( ...      => pers_pron.

/* EQUIV: POSSESSIVE PRONOUN                                     */
nucleus_equiv(ref(poss_pron( ...      => partiv_pron, poss_pron.
nucleus_equiv(ref(poss_pron( ...      => poss_pron.

/* EQUIV: INDEFINITE PRONOUN                                     */
nucleus_equiv(ref(pron( ...            =>
    pron(P), not_followed_by_or_more(P), not_followed_by_a_np.

/* EQUIV: QUANTIFIER                                           */
nucleus_equiv(ref(quant( ...           => quant, not_followed_by_a_np.

/* EQUIV: NUMBER (GENERAL)                                     */
nucleus_equiv(ref( ...                 => number_gen.

/* EQUIV: REFLEXIVE PRONOUN                                     */
nucleus_equiv(ref(reflex_pron( ...     => reflex_pron.

/* EQUIV: CHARACTER                                           */
nucleus_equiv(ref( ...                 => noun(character, ... ).

/* EQUIV: STRING                                              */
nucleus_equiv(ref( ...                 => noun(string, ... ).

/* EQUIV: QUIZ TOKEN                                           */
nucleus_equiv(ref(quiz_token( ...      => quiz_token.

/* EQUIV: REFERENCE OR VARIABLE                               */
nucleus_equiv(ref(ref_or_var( ...      => ref_or_var.

/* EQUIV: DATE                                                 */
nucleus_equiv(ref( ...                 => date_form.

/* EQUIV: SPECIAL NUMBER                                       */
nucleus_equiv(ref(number_special( ...  => number_special.

/* EQUIV: DOLLARS                                              */
nucleus_equiv(ref(dollars( ...         => dollars, "or more".
nucleus_equiv(ref(dollars( ...         => dollars, "or less".
nucleus_equiv(ref(dollars( ...         => dollars.

/* EQUIV: PERCENTAGE                                           */
nucleus_equiv(ref(percentage( ...      => percentage.

/* ..... */

/* Disambiguating 'one': indefinite pronoun vs cardinal.      */
not_followed_by_or_more(P) => P \== one, !.
not_followed_by_or_more(one) => "or more", !, fail.
not_followed_by_or_more(one) => "or less", !, fail.
not_followed_by_or_more(one).

/*****/

rest_common_noun_np          =>
    adj_phrase, (noun_subst ; noungp_head).

/* ADJ & NOUN are enclosed in double quotes (e.g. "supporting person"); */
/* see note on double_quotes vs 'string' in verb_in_double_quotes_or_not. */
/* Notation (string): _b stands for begin(ning) and _e stands for end(ing). */
rest_common_noun_np( ... , string_b(...), string_e( ... ) =>
    double_quotes, adj_phrase, (noun_subst ; noungp_head), double_quotes.

```

```

noun_subst => ("that" ; "the" ; "this"), "one".
noun_subst => ("the" ; "these" ; "those"), "ones".

/*****
/* Covers the special case where an adjective is used as a noun.
/* Ex: "THE EXTREMELY POOR has great difficulty ..."
*/

adj_substantive_clause(entity( ..=>
    determinatives,(true ; adv_s), adj_subst, not_followed_by_a_np.

adj_subst => adj.
adj_subst => double_quotes, adj, double_quotes.

/*****

partiv_pron => partitive_predeter_partiv_pron. % e.g. ALL OF us
partiv_pron => partitive_deter_partiv_pron.    % e.g. EACH OF us
partiv_pron => closed_postdeter_partiv_pron.   % e.g. ONE OF us

partitive_predeter_partiv_pron(partitive_predeter( ... => predeter, "one of".
partitive_predeter_partiv_pron(partitive_predeter( ... => predeter, "of".

/* ..... */

partitive_deter_partiv_pron(partitive_deter( ..=> deter, "one of".
partitive_deter_partiv_pron(partitive_deter( ..=> deter, "of".

/* ..... */

/* See below (in opt_postdeter) for definitions of postdeter_gp & postdeter. */
closed_postdeter_partiv_pron(partitive_postdeter( =>.
    postdeter_gp, part_of_partiv_pron.
closed_postdeter_partiv_pron(partitive_postdeter( =>.
    postdeter, part_of_partiv_pron.

part_of_partiv_pron => "or more of" ; "or less of".
part_of_partiv_pron => "of".

/*****

/* Ex: "seven", "seven or more"
number_gen(card_num( ..=> card_num, ("or more" ; "or less").
number_gen(card_num( ..=> card_num, not_the_one_pronoun.

/* Ex: "5", "5 or more"
number_gen(num( ... => number, (true ; "or more" ; "or less").

/* Ex: "(2)", "(2 or more)"
number_gen(parenthetical_num( ...=>
    parenthesis_o, number, (true ; "or more" ; "or less"), parenthesis_c.

/* Ex: "1(c)(iii)". Note: 'parenthetical_sub_num' deals with "sub-numeral"
/* (letter) numbering: Ex: '(a)' in "37(a)".
number_gen(composite_num( ... => number, parenthetical_sub_num.

/*****
/* Standard abbreviated format as recognized by the scanner
/* Ex: "April 30, 1991"
/* Ex: "April 30"
/* Ex: "April 1991"

date_form(date( ... => date.

```

```

/*****
/* Noun phrase postmodifiers are clauses that are attached to a noun phrase. */
/* Warning: some clauses (especially PPs) may in fact belong to a VP: */
/* only semantic K or user interaction will allow complete disambiguation. */
/* See QUIRK, chap.17 ("The noun phrase") and, in particular, 17.9-50. */

np_postmodifiers( ... np_postmodifiers( ... => np_postmodif.

/* Note: the definition of rel_clause covers the 3 cases where a ing_clause, */
/* a to_infinitive_clause, or a past part. phrase acts as a relative clause. */
np_postmodif => rel_clause.
np_postmodif => appositive.
np_postmodif => restricted_comparison.
np_postmodif => required_prepphrases.

/* Recognizes a restricted case of comparison: "... a bigger car THAN THAT." */
restricted_comparison(restricted_comparative_clause__than( .=>
    "than", np_equiv.

/*****
/* Missing subject (last parameter): M_el = missing element (referring NP). */

/* Typical case (1). */
/* Ex: "He always reads the letters THAT APPEAR IN THE LOCAL JOURNAL." */
rel_clause( ... rel_clause(rel_marker( .=>
    rel_marker, verbphrase(2, ... ).

/* The case where the preposition is deferred ("dangling" preposition). */
/* Ex: "The man WHOM I GAVE IT TO." */
rel_clause( ... rel_clause(rel_marker( ... =>
    rel_marker, np_equiv, verbphrase(2, ... ).

/* The case where a preposition precedes the relative marker. */
/* Ex: "The man TO WHOM I GAVE IT." */
rel_clause( ... rel_clause(rel_marker( ... =>
    prep, rel_marker, (np_equiv ; true),
    verbphrase(2, ... , referring_np( ... )).

/* A -ing clause as a relative clause: "An architect HAVING LUNCH." */
rel_clause( ... rel_clause(rel_marker( ... =>
    verb_clause(2 ... ).

/* A past participle phrase as a relative clause: "A report WRITTEN BY TOM." */
rel_clause( ... rel_clause(rel_marker(nil), ... =>
    past_participle_phrase.

/* Nonfinite relative clause (1): "A cause FOR WHICH TO DIE." */
rel_clause( ... rel_clause(rel_marker( ... =>
    prep, rel_marker, "to", verb_clause(2, ... ).

/* Typical case (2): the relative clause is introduced by a relative marker. */
/* Ex: "He always reads the letters (WHICH) SHE WRITES TO HIM." */
rel_clause( ... rel_clause(rel_marker( ... =>
    ((prep, rel_marker) ; rel_marker), np_equiv, verbphrase(2, ... ).

/* Nonfinite relative clause (2): "A cause FOR YOU TO SUPPORT." */
rel_clause( ... rel_clause(p_rel_marker(for), .=>
    "for", np_equiv, "to", verb_clause(2, ... ).

/*****

past_participle_phrase =>
    real_pass_rel_clause_test, % making sure it really is such a clause.
    verb_clause(2,_,passive, ... ).

```

```

/*****
/* This rule recognizes appositives: linguistic units, mostly noun phrases,
/* which are identical in reference; see QUIRK, chap.17 (17.65-93).

/* General case: it requires correlative markers.
appositive =>
    app_correlative_marks_o(N), apposition_core, app_correlative_marks_c(N).

/* Two cases that do not require correlative markers.
appositive => propernoun.
appositive => whether_if_clause.

/* Special case (1): special characters (e.g. "the ampersand symbol (&)").
appositive =>
    parenthesis_o, special_char_seq, parenthesis_c.

/* Special case (2): QUIZ command (e.g. "the item subtotal (SUBTOTAL)").
appositive =>
    parenthesis_o, quiz_token, parenthesis_c.

app_correlative_marks_o(1) => comma.
app_correlative_marks_o(2) => dashes.
app_correlative_marks_o(3) => parenthesis_o.

app_correlative_marks_c(1) => comma.
app_correlative_marks_c(1) => end_of_sentence_marker.
app_correlative_marks_c(2) => dashes.
app_correlative_marks_c(2) => end_of_sentence_marker.
app_correlative_marks_c(3) => parenthesis_c.

/* ..... */

apposition_core => explicit_app_ind, opt_comma, apposition_group.
apposition_core => apposition_group, opt_comma, explicit_app_ind.
apposition_core => required_adverbs, apposition_group.
apposition_core => apposition_group(AG,Num).
apposition_core =>
    explicit_app_ind(EAI),
    ((EAI == for_example ; EAI == for_instance ;
     EAI == in_other_words ; EAI == in_particular)).

/* Ex: "Jones, THE DISTINGUISHED ART CRITIC, died in his sleep last night." */
apposition_group => np_equiv.

/* Ex: "My friend ANNA was here last night." */
apposition_group => propernoun.

/* Ex: "His explanation, THAT HE COULDN'T SEE THE CAR, is unsatisfactory." */
apposition_group => nominal_clause.

/* Ex: "The question WHETHER TO CONFESS OR NOT troubled the girl." */
apposition_group => whether_if_clause.

/* ***** */

explicit_app_ind(EAI) => adv(EAI,Type), {Type == app}.
explicit_app_ind(EAI) => [dct(_:EAI/_,_)], {appos_indicator(EAI)}.
explicit_app_ind(for_example) => idiomatic_cat(for_example,_,_,_).
explicit_app_ind(for_instance) => idiomatic_cat(for_instance,_,_,_).
explicit_app_ind(in_other_words) => idiomatic_cat(in_other_words,_,_,_).
explicit_app_ind(in_particular) => idiomatic_cat(in_particular,_,_,_).

```

```

explicit_app_ind(such_as)      => idiomatic_cat(such_as,_,_,_).
explicit_app_ind(that_is_to_say) => idiomatic_cat(that_is_to_say,_,_,_).
explicit_app_ind(that_is)      => idiomatic_cat(that_is,_,_,_).

appos_indicator('e.g.').
appos_indicator('i.e.').
appos_indicator(and).
appos_indicator(chiefly).
appos_indicator(eg).
appos_indicator(especially).
appos_indicator(for_example).
appos_indicator(for_instance).
appos_indicator(ie).
appos_indicator(in_other_words).
appos_indicator(in_particular).
appos_indicator(included).
appos_indicator(including).
appos_indicator(mainly).
appos_indicator(mostly).
appos_indicator(namely).
appos_indicator(notably).
appos_indicator(of).
appos_indicator(or).
appos_indicator(particularly).
appos_indicator(say).
appos_indicator(simply).
appos_indicator(such_as).
appos_indicator(technically).
appos_indicator(that_is).
appos_indicator(that_is_to_say).
appos_indicator(viz).

/*****

required_prepphrases=> prepphrases(RPPS, ... ), (RPPS \== []).

prepphrases => either_or_pp ; (prepphrase, rest_prepphrase).
prepphrases => [].

/* EITHER/OR PPs.                                     */
either_or_pp([conj_pps(either_or, ..=>
    "either", prepphrase,"or", prepphrase.

/* NEITHER/NOR PPs.                                   */
either_or_pp([conj_pps(neither_nor, ...=>
    "neither", prepphrase, "nor", prepphrase.

rest_prepphrases( ... [PP,parenthetical(conj_pps( =>.
    parenthesis_o, conj_pp, parenthesis_c.
rest_prepphrases( ... [conj_pps( ...      =>
    conj_pp.
rest_prepphrases                                => [].

/* ..... */
/* On the N parameter, see remarks in dipett_6 (conj_vp predicate). */

/* Case (1) : PP ('.') COOR PP ... */
conj_pp      =>
    opt_comma, real_coordinator_3, prepphrase, conj_pp.

/* Case (2) : PP ('.') PP ... */

```

```

conj_pp( ... [p_asyndetic_conj_pp=>
    opt_comma, prepphrase, conj_pp.

conj_pp          =>
    [].

/*****

prepphrase      => prepphrase_core.

prepphrase_core => required_adverbs, prep_pp, np_for_pp.
prepphrase_core => prep_pp, np_for_pp.
prepphrase_core => prep_pp, required_adverbs.

prep_pp(conj_preps( ...=> prep, real_coordinator_3, prep, opt_colon.
prep_pp          => prep, opt_colon.

np_for_pp => nominal_clause(wh_interro_declarative_clause, ... ).
np_for_pp => nominal_clause(ing_clause, ... ).
np_for_pp => np_equiv.

/*****
/* Determiners, including pre- & post-determiners. See QUIRK, pp.253+.      */
determinatives(determinatives(predeter(PRE),deter(D),postdeter(POST)) ...
=>
    opt_predeter(PRE, ... ), opt_deter(D, ... ), opt_postdeter(POST, ... ).

/*****

opt_predeter => predeter_whole.
opt_predeter => predeter, rest_predeter.
opt_predeter => compare_p_np.
opt_predeter => [].

predeter_whole => "a whole".
predeter_whole => "the whole".

rest_predeter=> "of".
rest_predeter=> [].

/* Recognizes "less than a ... ", "more than one", "fewer than 100", etc.      */
compare_p_np(np_intensifier__than( ..=> adv, "than".
compare_p_np(np_intensifier__than( ..=> adj(comp), "than".

/*****

opt_deter => partitive_deter.
opt_deter => deter.
opt_deter => [].

partitive_deter(partitive_deter( ... => deter, "one of", deter.
partitive_deter(partitive_deter( ... => deter, "of", deter.

/*****

opt_postdeter=> poss_ng, rest1_postdeter.
opt_postdeter=> closed_postdeter, rest2_postdeter.
opt_postdeter=> open_class_quant.
opt_postdeter=> [].

poss_ng      => genitive.

```

```

rest1_postdeter( ... genitive( ..=> closed_postdeter.
rest1_postdeter( ... genitive( ..=> []).

rest2_postdeter( ... genitive( ..=> poss_ng.
rest2_postdeter( ...      => [].

/* ++++++ */

closed_postdeter => (postdeter_gp ; postdeter), rest3_postdeter.

postdeter_gp(ord_and_card( ...=> ord_num, card_num_c.

card_num_c(conj_card_nums( ...=>
    number_gen, real_coordinator_3, number_gen.
card_num_c(
    =>
    number_gen.

postdeter(card( ... => adv_or_adj_post, card_num_c.
postdeter(card( ... => card_num_c.
postdeter(ord( ... => adv_or_adj_post, ord_num.
postdeter(ord( ... => ord_num.
postdeter(quant( ...=> quant.

/* An adverb or an adjective used as an adverb (see QUIRK, pp.405+). */
adv_or_adj_post => adv_s.
adv_or_adj_post => adj_s(Deg), {Deg \== comp, Deg \== sup}. %

adj_s( ... ) => adj(A), \+ prep(A).

/* ..... */

rest3_postdeter( ... closed( ... => part_of.
rest3_postdeter( ... closed( ... => [].

part_of(PO)      =>
    "or", {"more" ; "less"} ; (adj_s(Deg), {Deg \== sup}) ; adv_s),
    "of", rest_part_of.
part_of([of, ... => "of", deter.
part_of      => "of".

rest_part_of => deter.
rest_part_of => [].

open_class_quant(open_quant(lots_of ... => "lots of".
open_class_quant(open_quant(plenty_of ..=> "plenty of".

/* ++++++ */
/* Note: we also cover the case where and adverb is used as an adjective; */
/* see QUIRK, chap.7 ("Adjectives and adverbs") and pp.405+. */

required_adj_phrase => adj_phrase(Adj_P), {Adj_P \== []}.

adj_phrase      => adj_equiv, rest_adj_phrase.
adj_phrase      => [].

rest_adj_phrase( ... , {attrs(adj( ..=>
    opt_comma, opt_real_coordinator, adj_equiv, rest_adj_phrase.
rest_adj_phrase      => [].

/* ..... */

```



```

adj_equiv => adj_s.
adj_equiv => double_quotes, adj_s, double_quotes.
adj_equiv => past_part.
adj_equiv => pres_part.
adj_equiv => adv_s.
adj_equiv => double_quotes, adv_s, double_quotes.

past_part => verb_in_double_quotes_or_not(perf, ... ).
pres_part => verb_in_double_quotes_or_not(progr, ... ).

/* In order to reduce lexical ambiguity, we discard "false" adverbs. */
/* ##note##: this may be a bit too restrictive ... */
adv_s => adv(A), \+ conj(A), \+ prep(A).

/*****
/* The head of a noun group is a noun that may be preceded by pre-modifiers */
/* and potentially followed by a post-modifier; see QUIRK, pp.1238+. */
/* On 'string': see verb_in_double_quotes_or_not and np_nucleus. */

/* Noun gp. string type (1): "pre-modif noun post-modif" */
noungp_head( ... string(noun( ... , noun_modifiers( ... =>
    double_quotes, noun_pre_modifiers, noun, noun_post_modifier, double_quotes.

/* Noun gp. string type (2): "pre-modif noun" post-modif */
noungp_head( ... noun(string_e( ... , noun_modifiers(string_b( =>
    double_quotes, noun_pre_modifiers, noun, double_quotes, noun_post_modifier.

/* Noun gp. string type (3): pre-modif "noun post-modif" */
noungp_head( ... noun(string_b( ... , noun_modifiers( ... =>
    noun_pre_modifiers, double_quotes, noun, noun_post_modifier, double_quotes.

noungp_head( ... noun( ... noun_modifiers( ... =>
    noun_pre_modifiers, noun, noun_post_modifier.

/*****
/* All nouns preceding the head noun are called "pre-modifiers". */
/* Compound nouns can be constructed from: */
/* i) verb + noun (ex: swimming-pool, shooting star) [verb: -ing|inf] */
/* ii) noun + verb (ex: birth-control) [verb: -ing] */
/* iii) noun + noun (ex: motor-cycle, love story). */
/* The pre-modifiers cover cases i and iii when dashes are not needed; */
/* otherwise, it's a compound and treated as a single-token word. */
/* */
/* On NP pre-modification in general, see QUIRK, chap.17 (17.94-120). */

noun_pre_modifiers =>
    (noun_pre_modif ; propernoun ; (noun, not_followed_by_ing_word)),
    rest_noun_pre_modifiers.
noun_pre_modifiers => [].

not_followed_by_ing_word> ing_word(I), verb(.,.,.,I,.), !, fail.
not_followed_by_ing_word> !.

/* ..... */

rest_noun_pre_modifiers=>
    opt_real_coordinator,
    (noun_pre_modif ; propernoun ; (noun, not_followed_by_ing_word)),
    rest_noun_pre_modifiers.
rest_noun_pre_modifiers=> [].

/* ..... */

```

```

noun_pre_modif(quiz_token( ...=> quiz_token.
noun_pre_modif(ref_or_var( ... => ref_or_var.
noun_pre_modif      => date_form.
noun_pre_modif(number_special( ...=> number_special.
noun_pre_modif(dollars( ... => dollars.
noun_pre_modif(percentage( ...=> percentage.

/*****

noun => countnoun.
noun => double_quotes, countnoun, double_quotes.

noun => massnoun.
noun => double_quotes, massnoun, double_quotes.

noun( ... ing_noun( ...      =>
    verb_clause(2, ... [progr], ... ).
noun( ... double_quotes(ing_noun( ...=>
    double_quotes, verb_clause(2, ... [progr], ... ), double_quotes.

/* ----- NOUN EQUIVALENTS OR SUBSTITUTES ----- */

noun => quiz_token.
noun => special_function_word(Type), {Type == frag}.
noun => double_quotes, special_char_seq, double_quotes.
noun => atomic_character.
noun => double_quotes, get_string_tokens.

/*****

noun_post_modifier => number_gen.
noun_post_modifier => quiz_token.
noun_post_modifier => ref_or_var.
noun_post_modifier => date_form.
noun_post_modifier => number_special.
noun_post_modifier => dollars.
noun_post_modifier => percentage.
noun_post_modifier => [].

/*****
/* Un adjectif peut avoir un complement qui est l'un des 6 groupes suivants: */
/* i)  un groupe prepositionnel (prepositional phrase) */
/* ii) une proposition substantive introduite par 'that' (that clause) */
/* iii) une substantive avec un verbe a l'infinitif (to-infinitive clause) */
/* iv) une substantive avec un verbe en -ing (nominal -ing clause) */
/* v)  une question indirecte (see wh_interro_declarative_clause above) */
/* vi) une proposition substantive introduite par 'than' (than clause). */
/* */
/* Voir QUIRK, chap.16 ("Complementation of verbs and adjectives"); pp.1220+ */

compl_adj_phrase      => comparative_clause.

compl_adj_phrase(attrs_compl( ...=>
    required_adj_phrase, rest_compl_adj_phrase.

compl_adj_phrase(attrs_compl( ...=> required_adverbs.

compl_adj_phrase(attrs_compl( ...=> adj_complement.

rest_compl_adj_phrase => adj_complement.
rest_compl_adj_phrase => [].

```

```

/* .....
adj_complement => required_prepphrases.
adj_complement => nominal_clause(that_clause, ... ).
adj_complement => nominal_clause(to_infinite_clause, ... ).
adj_complement => nominal_clause(ing_clause, ... ).
adj_complement => nominal_clause(wh_interro_declarative_clause, ... ).
adj_complement(nonfinite_rel_clause_equiv( ... =>
    "for", np_equiv, simple_to_infinite_clause.

/.....

required_adverbs => adverbs(AdvS), {AdvS \== []}.

adverbs => (adv_s ; adv_not_s), rest_adverbs.
adverbs => [].

/* ..... */

rest_adverbs =>
    opt_comma, opt_real_coordinator, (adv_s ; adv_not_s), rest_adverbs.
rest_adverbs => [].

adv_not_s => adv(A), (conj(A) ; prep(A)).

/* ~~~~~ */
/* Punctuation and a few special characters used throughout the parser. */

opt_pause => pause.
opt_pause => [].

pause => colon ; comma ; dashes ; semi_colon.

opt_colon => colon.
opt_colon => [].

opt_comma => comma.
opt_comma => [].

colon => ":".
comma => ",".
dashes => "--".
semi_colon => ";".

double_quotes => "".

parenthesis_o => "( ".
parenthesis_c => ") ".

question_mark => "?".

/.....
/* On tags, see QUIRK, pp.983-4. */

terminal(end_of_input( ... => opt_comma, tag_coor, end_of_sentence_marker.
terminal(end_of_input( ... => end_of_sentence_marker.

tag_coor => tag_coordinator.

tag_coordinator(and_so_forth).
tag_coordinator(and_so_on).
tag_coordinator(and_so_on_and_so_forth).
tag_coordinator(and_the_like).
tag_coordinator(etc).

```

```

tag_coordinator([and,so,forth],and_so_forth).
tag_coordinator([and,so,on,and,so,forth],and_so_on_and_so_forth).
tag_coordinator([and,so,on],and_so_on).
tag_coordinator([and,the,like],and_the_like).

/* ..... */

end_of_sentence_marker(period)    => ".".
end_of_sentence_marker(ellipsis)  => "...".
end_of_sentence_marker(exclamation_mark) => "!".

/...../

opt_negation => "not".
opt_negation => [].

opt_advs_negation=> opt_negation, adverbs.
opt_advs_negation=> adverbs(Advs), (Advs \== [not]), opt_negation.

/...../

opt_particle => [].
opt_particle => required_adverbs, adv_particle.
opt_particle => adv_particle(Par),

/...../

opt_real_coordinator=> real_coordinator_3.
opt_real_coordinator=> [].

real_coordinator_3 => real_coordinator.

/...../

```

Appendix C: Parsing Statistics for the QUIZ Manual*

Since we wanted to study the influence of parsing time on the parser's performance, the following statistics were computed for five different CPU time limits, i.e. 30, 60, 90, 120, and 300 seconds, on a SPARC-2 workstation. For all inputs, DIPETT first attempted to get a full analysis in the allotted time; if it failed to do so, it would then attempt to find a fragmentary analysis (see 3.4.4.2), still with the same time limit. Results are discussed in section 3.6.4.

CHAPTER 1

Chapter 1: 171 units	30 sec.	60 sec.	90 sec.	2 min.	5 min.
(1) #units fully analyzed	87	89	89	90	91
(2) #units stopped by the CPU time limit, before success or failure	53	50	49	46	43
(3) #units rejected, i.e. extra-grammatical	31	32	33	35	37
(4) percent of units fully analyzed	50.9%	52.0%	52.0%	52.6%	53.2%
(5) #units analyzed by fragments: from (2) + (3)	58	61	66	65	66
(6) #units stopped by the CPU time limit (before success or failure): from (5)	24	20	15	15	14
(7) #units rejected, i.e. extra-grammatical: from (5)	2	1	1	1	1
(8) percent of units analyzed by fragments: from (5)	69.0%	74.4%	80.5%	80.2%	81.5%
(9) #units analyzed, either fully or by fragments	145	150	155	155	157
(10) percent of units analyzed, either fully or by fragments	84.8%	87.7%	90.6%	90.6%	91.8%

* Je remercie Caroline Paquet, une finissante au Baccalauréat en Informatique à l'Université du Québec à Trois-Rivières, qui a effectué d'innombrables tests sur DIPETT et colligé de nombreuses observations sur les résultats. De plus, elle a calculé la plupart des statistiques qui apparaissent dans cette annexe et qui sont tirées du chapitre 2 de [Paquet 93].

CHAPTER 2

Chapter 2: 100 units	30 sec.	60 sec.	90 sec.	2 min.	5 min.
(1) #units fully analyzed	41	42	43	43	44
(2) #units stopped by the CPU time limit, before success or failure	32	31	29	26	23
(3) #units rejected, i.e. extra-grammatical	27	27	28	31	33
(4) percent of units fully analyzed	41.0%	42.0%	43.0%	43.0%	44.0%
(5) #units analyzed by fragments: from (2) + (3)	50	53	52	52	51
(6) #units stopped by the CPU time limit (before success or failure): from (5)	9	5	5	5	5
(7) #units rejected, i.e. extra-grammatical: from (5)	0	0	0	0	0
(8) percent of units analyzed by fragments: from (5)	84.7%	91.4%	91.2%	91.2%	91.1%
(9) #units analyzed, either fully or by fragments	91	95	95	95	95
(10) percent of units analyzed, either fully or by fragments	91.0%	95.0%	95.0%	95.0%	95.0%

CHAPTER 3

Chapter 3: 162 units	30 sec.	60 sec.	90 sec.	2 min.	5 min.
(1) #units fully analyzed	73	76	80	80	82
(2) #units stopped by the CPU time limit, before success or failure	53	46	40	40	30
(3) #units rejected, i.e. extra-grammatical	36	40	42	42	50
(4) percent of units fully analyzed	45.1%	46.9%	49.3%	49.3%	50.6%
(5) #units analyzed by fragments: from (2) + (3)	75	75	72	72	71
(6) #units stopped by the CPU time limit (before success or failure): from (5)	14	11	10	10	9
(7) #units rejected, i.e. extra-grammatical: from (5)	0	0	0	0	0
(8) percent of units analyzed by fragments: from (5)	84.3%	87.2%	87.8%	87.8%	88.8%
(9) #units analyzed, either fully or by fragments	148	151	152	152	153
(10) percent of units analyzed, either fully or by fragments	91.4%	93.2%	93.8%	93.8%	94.4%

CHAPTER 4

Chapter 4: 111 units	30 sec.	60 sec.	90 sec.	2 min.	5 min.
(1) #units fully analyzed	47	49	51	51	52
(2) #units stopped by the CPU time limit, before success or failure	37	25	21	20	18
(3) #units rejected, i.e. extra-grammatical	27	37	39	40	41
(4) percent of units fully analyzed	42.3%	44.1%	45.9%	45.9%	46.8%
(5) #units analyzed by fragments: from (2) + (3)	59	59	59	59	59
(6) #units stopped by the CPU time limit (before success or failure): from (5)	5	3	1	1	0
(7) #units rejected, i.e. extra-grammatical: from (5)	0	0	0	0	0
(8) percent of units analyzed by fragments: from (5)	92.2%	95.2%	98.3%	98.3%	100%
(9) #units analyzed, either fully or by fragments	106	108	110	110	111
(10) percent of units analyzed, either fully or by fragments	95.5%	97.3%	99.1%	99.1%	100%

CHAPTER 5

Chapter 5: 111 units	30 sec.	60 sec.	90 sec.	2 min.	5 min.
(1) #units fully analyzed	50	51	52	52	52
(2) #units stopped by the CPU time limit, before success or failure	32	31	26	25	21
(3) #units rejected, i.e. extra-grammatical	29	29	33	34	38
(4) percent of units fully analyzed	45.0%	45.9%	46.8%	46.8%	46.8%
(5) #units analyzed by fragments: from (2) + (3)	50	50	50	52	54
(6) #units stopped by the CPU time limit (before success or failure): from (5)	11	10	9	7	5
(7) #units rejected, i.e. extra-grammatical: from (5)	0	0	0	0	0
(8) percent of units analyzed by fragments: from (5)	82.0%	83.3%	84.7%	88.1%	91.5%
(9) #units analyzed, either fully or by fragments	100	101	102	104	106
(10) percent of units analyzed, either fully or by fragments	90.1%	91.0%	91.9%	93.7%	95.5%

CHAPTER 6

Chapter 6: 112 units	30 sec.	60 sec.	90 sec.	2 min.	5 min.
(1) #units fully analyzed	47	50	51	53	54
(2) #units stopped by the CPU time limit, before success or failure	41	32	30	27	23
(3) #units rejected, i.e. extra-grammatical	24	30	31	32	35
(4) percent of units fully analyzed	42.0%	44.6%	45.5%	47.3%	48.2%
(5) #units analyzed by fragments: from (2) + (3)	54	53	53	51	50
(6) #units stopped by the CPU time limit (before success or failure): from (5)	11	9	8	8	8
(7) #units rejected, i.e. extra-grammatical: from (5)	0	0	0	0	0
(8) percent of units analyzed by fragments: from (5)	83.1%	85.5%	86.9%	86.4%	86.2%
(9) #units analyzed, either fully or by fragments	101	103	104	104	104
(10) percent of units analyzed, either fully or by fragments	90.2%	92.0%	92.9%	92.9%	92.9%

CHAPTERS 1 TO 6

Chapters 1-6: 767 units	30 sec.	60 sec.	90 sec.	2 min.	5 min.
(1) #units fully analyzed	345	357	366	369	375
(2) #units stopped by the CPU time limit, before success or failure	248	215	195	184	158
(3) #units rejected, i.e. extra-grammatical	174	195	206	214	234
(4) percent of units fully analyzed	45.0%	46.6%	47.7%	48.1%	48.9%
(5) #units analyzed by fragments: from (2) + (3)	346	351	352	351	351
(6) #units stopped by the CPU time limit (before success or failure): from (5)	74	58	48	46	41
(7) #units rejected, i.e. extra-grammatical: from (5)	2	1	1	1	1
(8) percent of units analyzed by fragments: from (5)	82.0%	85.6%	87.8%	88.2%	89.5%
(9) #units analyzed, either fully or by fragments	691	708	718	720	726
(10) percent of units analyzed, either fully or by fragments	90.1%	92.3%	93.6%	93.9%	94.7%

Appendix D: Example Parse Trees

This appendix shows five actual parse trees produced by DIPETT. The first two inputs come from the QUIZ manual and the three others from the T4043 Tax Guide — on these two texts, see 3.6.2 and 3.6.3. Timings are for a SPARC-10. In the following presentation, we added a few comments to identify imperfect (i.e. “non-best” according to our knowledge of the language) parse trees, i.e. for trees #2, (D.2) #3 (D.3), and #5 (D.5). In these comments, the symbol PP stands for prepositional phrase.

D.1: Example Parse Tree #1

Unit #1: For your purposes , though , it would be more convenient to have an alphabetical list .

Process this unit (Y/n), try another Version, or Exit [Y/n/v/e]?

lexical analysis (cpu time) : 0.167 second(s).

First parse for input sentence:

MEMORIZER results: # of units fetched (not re-parsed unnecessarily).

```
noun phrase(s)           : 2
prepositional phrase(s)   : 0
verb phrase(s)            : 0
verbal clause(s) of type 1 : 0
verbal clause(s) of type 2 : 7
subordinate clause(s)     : 0
```

parsing (cpu time) : 0.817 second(s).

```
parse_tree__decla_or_imper
  complex_sentence
    structure
      single_main_clause
        declarative
          statement
            subordinate_clauses
              initial
                conj_subord_cl  implicit_and
                  subordinate
                    adverbial
                      pp  for
                        entity
                          determinatives
                            deter  your
                              head_noun
                                noun  purpose
                                  number  pl
                                    tokens  your  purposes
                                      tokens  for  your  purposes
                                        tokens  for  your  purposes
                                          tokens  for  your  purposes
                                            subordinate
                                              adverbial
                                                simple_adv_clause  though
                                                  tokens  though
                                                    tokens  though
                                                        tokens  for  your  purposes ,  though ,
subj
  entity
    ref
      pron  it
        number  sg3
          tokens  it
            predicate
              copula
                verb  be
                  tense  conditional_present_simple
```

```

      neg yes
      trans stat
s_qualifier
  svc_sva
    attrs_compl
      conj_adj_p implicit_and
        attrs
          adj more pos
        attrs
          adj convenient pos
        tokens more convenient
      nominal_clause
        to_infinitive_clause
          predicate
            regular
              verb have
                tense inf
                  neg yes
                    trans tr
                      voice active
                        complement
                          svo
                            entity
                              determinatives
                                deter an
                              attributes
                                attrs
                                  adj alphabetical pos
                                tokens alphabetical
                              head_noun
                                noun list
                                  number sg3
                                    tokens an alphabetical list
                                  tokens to have an alphabetical list
                                tokens would be more convenient to have an
                                alphabetical list
                                  tokens for your purposes , though , it would be more
                                convenient to have an alphabetical list
                                  end_of_input period

```

D.2: Example Parse Tree #2

Unit #2: This would produce the names of the two Boston employees who joined the company since the start of 1984 .

Process this unit (Y/n), try another Version, or Exit [Y/n/v/e]?

lexical analysis (cpu time) : 0.15 second(s).

First parse for input sentence:

MEMORIZER results: # of units fetched (not re-parsed unnecessarily).

```
noun phrase(s)      : 0
prepositional phrase(s) : 0
verb phrase(s)      : 0
verbal clause(s) of type 1 : 0
verbal clause(s) of type 2 : 0
subordinate clause(s) : 0
```

parsing (cpu time) : 0.484 second(s).

{below: this tree contains a misattached PP. "since the start of 1984" should not be at the same level as the preceding 'of'-PP. Instead, it should have been attached to the preceding verb 'joined'. When presented with a series of PPs, the parser first attempts to analyze them as conjoined PPs, unless it finds a syntactic clue that indicates a potential embedded PP, such as an 'of'-PP which usually acts as noun post-modifier.}

```
parse_tree__decla_or_imper
  simple_sentence
    structure
      single_main_clause
        declarative
          statement
            subj
              entity
                ref
                  pron this
                  number sg3
                  tokens this
                predicate
                  regular
                    verb produce
                    tense conditional_present_simple
                    neg yes
                    trans tr_intr
                    voice active
                    complement
                      svo
                        entity
                          determinatives
                            deter the
                          head_noun
                            noun name
                            number pl
                            np_postmodifiers
                              conj_pps implicit_and
                              pp of
```

```

entity
  determinatives
    deter the
    postdeter
      closed
      card
        card_num two
  head_noun
    noun employee
    noun_modifiers
      pre_modif boston
  number pl
  np_postmodifiers
    rel_clause
      rel_marker who
      statement
        subj
          entity
            determinatives
              deter the
              postdeter
                closed
                card
                  card_num two
            head_noun
              noun employee
              noun_modifiers
                pre_modif boston
            number pl
          predicate
            regular
              verb join
                tense past_simple
                neg yes
                trans tr_intr
            voice active
            complement
              svo
                entity
                  determinatives
                    deter the
                    head_noun
                      noun company
                      number sg3
                      tokens the company
                tokens joined the company
                tokens the two boston employees who
joined the company
tokens of the two boston employees
who joined the company
  p_asyndetic_conj_pp
    pp since
      entity
        determinatives
          deter the
          head_noun
            noun start
            number sg3

```



```

np_postmodifiers
  pp of
    entity
      ref
        num 1984
        number pl
        tokens 1984
        tokens of 1984
        tokens of 1984
        tokens the start of 1984
        tokens since the start of 1984
        tokens of the two boston employees who
joined the company since the start of 1984
        tokens the names of the two boston
employees who joined the company since the start of 1984
        tokens would produce the names of the two boston
employees who joined the company since the start of 1984
        tokens this would produce the names of the two boston
employees who joined the company since the start of 1984
        end_of_input period

```

D.3: Example Parse Tree #3

Unit #3: Refer to lines 304 and 305 in the 1990 General Tax Guide for more information on these non-refundable tax credits .

Process this unit (Y/n), try another Version, or Exit [Y/n/v/e]?

lexical analysis (cpu time) : 0.25 second(s).

First parse for input sentence:

MEMORIZER results: # of units fetched (not re-parsed unnecessarily).

noun phrase(s)	: 0
prepositional phrase(s)	: 0
verb phrase(s)	: 0
verbal clause(s) of type 1	: 0
verbal clause(s) of type 2	: 0
subordinate clause(s)	: 0

parsing (cpu time) : 0.3 second(s).

{below: this parse tree contains two misattached PPs. First, the PP “in the 1990 General Tax Guide” should have been attached to the preceding noun phrase “lines 304 and 305”. Second, the PP “on these non-refundable tax credits” should have been attached to the preceding noun phrase “more information”. The explanation for this analysis is the same as the one for tree #2 above.}

```

parse_tree__decla_or_imper
  simple_sentence
    structure
      single_main_clause
        imperative
          statement
            predicate
              regular
                verb refer
                  tense imperative
                  neg yes
                  trans tr_intr
                voice active
                complement
                  svo
                    conj_pps implicit_and
                      pp to
                        conj_nps and pl
                          entity
                            head_noun
                              noun line
                                noun_modifiers
                                  post_modif
                                    num 304
                                  number pl
                                    tokens lines 304
                                entity
                                  ref
                                    num 305
                                  number pl
                                    tokens 305

```

```

tokens lines 304 and 305
tokens to lines 304 and 305
p_asyndetic_conj_pp
  pp in
    entity
      determinatives
        deter the
        postdeter
          closed
            card
              num 1990
      attributes
        attrs
          adj general pos
          tokens general
        head_noun
          noun guide
          noun_modifiers
            pre_modif tax
          number sg
          tokens the 1990 general tax guide
          tokens in the 1990 general tax guide
p_asyndetic_conj_pp
  pp for
    entity
      determinatives
        postdeter
          closed
            quant more
      head_noun
        noun information
        number sg3
        tokens more information
        tokens for more information
p_asyndetic_conj_pp
  pp on
    entity
      determinatives
        deter these
      attributes
        attrs
          adj non-refundable pos
          tokens non-refundable
        head_noun
          noun credit
          noun_modifiers
            pre_modif tax
          number pl
          tokens these non-refundable tax credits
        tokens on these non-refundable tax
credits
  tokens to lines 304 and 305 in the 1990
  general tax guide for more information on these non-
  refundable tax credits
  tokens refer to lines 304 and 305 in the 1990
  general tax guide for more information on these non-
  refundable tax credits

```

tokens refer to lines 304 and 305 in the 1990 general
tax guide for more information on these non-refundable tax
credits
end_of_input period

D.4: Example Parse Tree #4

Unit #4: Since Rosa is the supporting person with the lower net income , she would claim the child care expenses .
Process this unit (Y/n), try another Version, or Exit [Y/n/v/e]?

lexical analysis (cpu time) : 0.1 second(s).
First parse for input sentence:

MEMORIZER results: # of units fetched (not re-parsed unnecessarily).

noun phrase(s)	: 0
prepositional phrase(s)	: 0
verb phrase(s)	: 0
verbal clause(s) of type 1	: 0
verbal clause(s) of type 2	: 0
subordinate clause(s)	: 0

parsing (cpu time) : 0.267 second(s).

```
parse_tree__decla_or_imper
  complex_sentence
    structure
      single_main_clause
        declarative
          statement
            subordinate_clauses
              initial
                subordinate
                  adverbial
                    adv_clause since
                      tokens since
                        statement
                          subj
                            entity
                              proper_noun rosa
                              number sg3
                              tokens rosa
                            predicate
                              copula
                                verb be
                                  tense present_simple
                                  neg yes
                                  trans stat
                                s_qualifier
                                  svc_sva
                                    entity
                                      determinatives
                                        deter the
                                      attributes
                                        attrs
                                          adj support pos
                                          tokens supporting
                                        head_noun
                                          noun person
                                          number sg3
                                          np_postmodifiers
                                            pp with
```

```

entity
  determinatives
    deter the
  attributes
    conj_adj_p implicit_and
    attrs
      adj low comp
    attrs
      adj net pos
    tokens lower net
  head_noun
    noun income
  number sg3
  tokens the lower net income
  tokens with the lower net
income
  tokens with the lower net
income
  tokens the supporting person with
the lower net income
  tokens is the supporting person with
the lower net income
  tokens since rosa is the supporting person
with the lower net income
  tokens since rosa is the supporting person
with the lower net income ,
  subj
    entity
      ref
        pers_pron she
        number sg3
        tokens she
    predicate
      regular
        verb claim
        tense conditional_present_simple
        neg yes
        trans tr
      voice active
      complement
        svo
          entity
            determinatives
              deter the
            head_noun
              noun expense
            noun_modifiers
              pre_modif
                conj_noun_pre_mod implicit_and child
care
  number pl
  tokens the child care expenses
  tokens would claim the child care expenses
  tokens since rosa is the supporting person with the
lower net income , she would claim the child care expenses
end_of_input period

```

D.5: Example Parse Tree #5

Unit #5: Omar and Elke maintained their residential ties with Canada because a member of their family occupied their residence while they lived outside the country , and their absence is not permanent .
Process this unit (Y/n), try another Version, or Exit [Y/n/v/e]?

lexical analysis (cpu time) : 0.25 second(s).
First parse for input sentence:

MEMORIZER results: # of units fetched (not re-parsed unnecessarily).
noun phrase(s) : 86
prepositional phrase(s) : 57
verb phrase(s) : 9
verbal clause(s) of type 1 : 0
verbal clause(s) of type 2 : 9
subordinate clause(s) : 0

parsing (cpu time) : 23.233 second(s).

{below: for reasons similar to the situation for PPs in trees #2 and #3 above, the "while they lived outside the country" subordinate clause should have been attached to the preceding 'occupied' clause, instead of being analyzed at the same level as that of the 'because' clause.}

```
parse_tree__decla_or_imper
  compound_sentence
    structure
      head_main_clause
        declarative
          statement
            subj
              conj_nps and pl
                entity
                  proper_noun omar
                  number sg3
                  tokens omar
                entity
                  proper_noun elke
                  number sg3
                  tokens elke
              tokens omar and elke
            predicate
              regular
                verb maintain
                tense past_simple
                neg yes
                trans tr
              voice active
              complement
                svoo_svoc_svoa
                entity
                  determinatives
                    deter their
                  attributes
                    attrs
```

```

      adj residential pos
      tokens residential
      head_noun
      noun tie
      number pl
      tokens their residential ties
pp with
  entity
    proper_noun canada
    number sg3
    tokens canada
    tokens with canada
    tokens with canada
subordinate_clauses
final
  conj_subord_cl implicit_and
  subordinate
    adverbial
      adv_clause because
      statement
        subj
          entity
            determinatives
            deter a
            head_noun
            noun member
            number sg3
            np_postmodifiers
            pp of
            entity
              determinatives
              deter their
              head_noun
              noun family
              number sg3
              tokens their family
              tokens of their family
              tokens of their family
              tokens a member of their family
            predicate
              regular
                verb occupy
                tense past_simple
                neg yes
                trans tr
              voice active
              complement
                svo
                  entity
                    determinatives
                    deter their
                    head_noun
                    noun residence
                    number sg
                    tokens their residence
                    tokens occupied their residence
                    tokens because a member of their family
                    occupied their residence

```



```

subordinate
  adverbial
    adv_clause while
      statement
        subj
          entity
            ref
              pers_pron they
              number pl
              tokens they
        predicate
          regular
            verb live
            tense past_simple
            neg yes
            trans tr_intr
          voice active
          complement
            svo
              pp outside
              entity
                determinatives
                  deter the
                  head_noun
                    noun country
                    number sg
                    tokens the country
              tokens outside the country
              tokens outside the country
              tokens lived outside the country
              tokens while they lived outside the
country
  tokens because a member of their family
  occupied their residence while they lived outside the country
  tokens maintained their residential ties with
  canada because a member of their family occupied their
  residence while they lived outside the country
  next_main_clause
    coord and
    declarative
      simple_sentence
        structure
          single_main_clause
            declarative
              statement
                subj
                  entity
                    determinatives
                      deter their
                      head_noun
                        noun absence
                        number sg3
                        tokens their absence
              predicate
                copula
                  verb be
                  tense present_simple
                  neg no

```

```
trans stat
s_qualifier
svc_sva
attrs_compl
attrs
adj permanent pos
tokens permanent
tokens is not permanent
tokens their absence is not permanent
tokens omar and elke maintained their residential ties
with canada because a member of their family occupied their
residence while they lived outside the country , and their
absence is not permanent
end_of_input period
```

Appendix E: HAIKU and Situations 1 to 5

This appendix shows an abridged, actual log (set in the `Courier` typeface) of HAIKU in action in the five different Case analysis situations presented in 4.4. Explanations were added in the Times typeface, in bold, and enclosed in braces. To improve the presentation, very simple example inputs were selected; as well, the original log was slightly reorganized .

E.1: Situation $\mathcal{A}_1$

```

Unit #1: Bob printed a file .
lexical analysis (cpu time) : 0.016 second(s).
First parse for input sentence:

{ ... }

parsing (cpu time) : 0.084 second(s).

parse_tree__decla_or_imper
  simple_sentence
    structure
      single_main_clause
        declarative
{ ... }

HAIKU: initiating Case-based semantic analysis ...

UPDATED PARSE
TREE(parse_tree__decla_or_imper(simple_sentence(structure(single_main
_clause(declarative(*statement1*))),tokens([bob,printed,a,file])),end
_of_input(period)))

LOGICAL CLAUSAL STRUCTURE
  top_structure
    [*statement1*]

SYMBOLIC A-LISTS
  coordinators
  clauses

[*statement1*,statement(subj(entity(proper_noun(bob),number(sg3),toke
ns([bob]))),predicate(regular(verb(print,tense([past_simple]),neg(yes
),trans(tr_intr))),voice(active),complement(svo(entity(determinatives
(deter(a)),head_noun(noun(file)),number(sg3),tokens([a,file]))))),tok
ens([printed,a,file])))]

INPUT STRING([bob,printed,a,file,.])

HAIKU: Case Analysis of current input's clauses ...

CURRENT CLAUSE: *statement1*
CURRENT SUBJ   :
subj(entity(proper_noun(bob),number(sg3),tokens([bob])))
CURRENT VERB   : print
CURRENT COMPL  :
complement(svo(entity(determinatives(deter(a)),head_noun(noun(file)),
number(sg3),tokens([a,file]))))
CURRENT MODIFS : [main_verb_modif(nil),clause_modif(nil)]

CMP found by HAIKU: psubj-pobj
> Do you wish to overrule this CMP
  (enter d to discard this clause, or
   a to discard the entire input) [y/N/d/a]?

```

{below: notice, HAIKU identifies the current clause as of the $\mathcal{A}_1$ type.}

S1: known CMP already associated with known verb: CP is known or new.

candidate CPs with example sentences:

CP: agt-obj Ex: [bob,printed,a,file]

current clause's main verb (infinitive): print

current clause's CMP (or a permutation): psubj-pobj

most probable or closest CPs, which are of compatible length:

agt-obj (1);

if appropriate, please select one of these CPs;

enter a number between 1 and 2 (or 0 if none is appropriate): 1

CMP & CP will be paired as follows: psubj/agt pobj/obj ; Correct
[n/Y]?

replace cmpDict's example with current clause [y/N]?

CASE STRUCTURES

clauses

[*statement1*,

case_structure(

verb(print,tense([past_simple]),neg(yes),trans(tr_intr)),

psubj-pobj,

agt-obj,

entity(proper_noun(bob),number(sg3),tokens([bob])),

entity(determinatives(deter(a),head_noun(noun(file)),

number(sg3),tokens([a,file])),

[main_verb_modif(nil),clause_modif(nil)])]

HAIKU semantic analysis (cpu time) : 1.233 second(s).

E.2: Situation δ_2

Unit #2: Beth erased all her files , suddenly .

lexical analysis (cpu time) : 0.15 second(s).

First parse for input sentence:

```
{ ... }
```

parsing (cpu time) : 0.25 second(s).

```
parse_tree__decla_or_imper
  complex_sentence
    structure
      single_main_clause
        declarative
  { ... }
```

HAIKU: initiating Case-based semantic analysis ...

UPDATED PARSE

```
TREE(parse_tree__decla_or_imper(complex_sentence(structure(single_main_clause(declarative(*statement4*))),tokens([beth,erased,all,her,file
s,,,suddenly])),end_of_input(period)))
```

LOGICAL CLAUSAL STRUCTURE

```
top_structure
  [*statement4*]
```

SYMBOLIC A-LISTS

```
coordinators
clauses
```

```
[*statement4*,statement(subj(entity(proper_noun(beth),number(sg3),tokens([beth])),predicate(regular(verb(erase,tense([past_simple]),neg(yes),trans(tr_intr))),voice(active),complement(svo(entity(determinatives(predeter(all),deter(her)),head_noun(noun(file)),number(pl),tokens([all,her,files])))),subordinate_clauses(final([subordinate(adverbial(simple_adv_clause([suddenly,tokens([suddenly])),tokens([suddenly]))),tokens([suddenly])))),tokens([erased,all,her,files,,,suddenly])))]
```

INPUT STRING([beth,erased,all,her,files,,,suddenly,.])

HAIKU: Case Analysis of current input's clauses ...

CURRENT CLAUSE: *statement4*

CURRENT SUBJ :

```
subj(entity(proper_noun(beth),number(sg3),tokens([beth])))
```

CURRENT VFRB : erase

CURRENT COMPL :

```
complement(svo(entity(determinatives(predeter(all),deter(her)),head_noun(noun(file)),number(pl),tokens([all,her,files]))))
```

CURRENT MODIFS :

```
[main_verb_modif(nil),clause_modif(subordinate_clauses([final([subordinate(adverbial(simple_adv_clause([suddenly,tokens([suddenly]))),tokens([suddenly]))),tokens([suddenly])))))]
```

```
CMP found by HAIKU: psubj-pobj-adv
> Do you wish to overrule this CMP
  (enter d to discard this clause, or
   a to discard the entire input) [y/N/d/a]?
```

{below: notice, HAIKU identifies the current clause as of the δ_2 type.}

S2: known CMP not yet associated with known verb; CP is known or new.

```
candidate CPs with example sentences:
  CP: agt-obj-manr Ex:
[bob,always,prints,and,reads,his,messages]
current clause's main verb (infinitive): erase
current clause's CMP (or a permutation): psubj-pobj-adv
most probable or closest CPs, which are of compatible length:
  agt-obj-manr (1) ;

if appropriate, please select one of these CPs;
  enter a number between 1 and 1 (or 0 if none is appropriate): 1
CMP & CP will be paired as follows: psubj/agt pobj/obj adv/manr ;
Correct [n/Y]?
  replace cmpDict's example with current clause [y/N]?

UPDATING mDict for erase ...
=> erase [psubj-pobj,psubj-pobj-adv,psubj-pobj-with]
[[adv,[manr]], [pobj,[obj]], [psubj,[agt]], [with,[inst]]]
UPDATING cpDict for agt-obj-manr ...
=> agt-obj-manr [erase,need,print]
```

CASE STRUCTURES

```
clauses
[*statement4*,
  case_structure(
    verb(erase,tense([past_simple]),neg(yes),trans(tr_intr)),
    psubj-pobj-adv,
    agt-obj-manr,
    entity(proper_noun(beth),number(sg3),tokens([beth])),
    entity(determinatives(predeter(all),deter(her)),
      head_noun(noun(file)),number(pl),tokens([all,her,files])),
    [main_verb_modif(nil),
      clause_modif(subordinate_clauses([final([subordinate(adverbial
        (simple_adv_clause([suddenly,tokens([suddenly]))],
          tokens([suddenly]))],tokens([suddenly]))])))]?)
```

HAIKU semantic analysis (cpu time) : 1.65 second(s).

E.3: Situation 83

Unit #3: Tom erased a file with a new command .
 lexical analysis (cpu time) : 0.05 second(s).
 First parse for input sentence:

```
{ ... }
```

parsing (cpu time) : 0.633 second(s).

```
parse_tree__decla_or_imper
  simple_sentence
    structure
      single_main_clause
        declarative
  { ... }
```

HAIKU: initiating Case-based semantic analysis ...

UPDATED PARSE

```
TREE(parse_tree__decla_or_imper(simple_sentence(structure(single_main
_clause(declarative(*statement3*))),tokens([tom,erased,a,file,with,a,
new,command])),end_of_input(period)))
```

LOGICAL CLAUSAL STRUCTURE

```
top_structure
  [*statement3*]
```

SYMBOLIC A-LISTS

```
coordinators
clauses
```

```
[*statement3*,statement(subj(entity(proper_noun(tom),number(sg3),toke
ns([tom]))),predicate(regular(verb(erase,tense([past_simple]),neg(yes
),trans(tr_intr))),voice(active),complement(svoo_svoc_svoa([entity(de
terminatives(deter(a)),head_noun(noun(file)),number(sg3),tokens([a,fi
le])),[pp(with,entity(determinatives(deter(a)),attributes([attrs(adj(
new,pos)),tokens([new]))),head_noun(noun(command)),number(sg3),tokens
([a,new,command])),tokens([with,a,new,command])),tokens([with,a,new,c
ommand]))])),tokens([erased,a,file,with,a,new,command]))]
```

```
INPUT STRING([tom,erased,a,file,with,a,new,command,.])
```

HAIKU: Case Analysis of current input's clauses ...

CURRENT CLAUSE: *statement3*

CURRENT SURJ :

```
subj(entity(proper_noun(tom),number(sg3),tokens([tom])))
```

CURRENT VERB : erase

CURRENT COMPL :

```
complement(svoo_svoc_svoa([entity(determinatives(deter(a)),head_noun(
noun(file)),number(sg3),tokens([a,file])),[pp(with,entity(determinati
ves(deter(a)),attributes([attrs(adj(new,pos)),tokens([new]))),head_no
un(noun(command)),number(sg3),tokens([a,new,command])),tokens([with,a
,new,command])),tokens([with,a,new,command]))]))
```

```
CURRENT MODIFS : [main_verb_modif(nil),clause_modif(nil)]
```



```

CMP found by HAIKU: psubj-pobj-with
> Do you wish to overrule this CMP
  (enter d to discard this clause, or
   a to discard the entire input) [y/N/d/a]?

```

{below: notice, HAIKU identifies the current clause as of the δ_3 type. Note: all the intermediate results returned by the metric are only shown here as a trace of execution; what matters to the user is the closest CP proposed by HAIKU.}

S3: entirely new CMP with known verb; CP is known or new.

```

entirely new CMP; no obvious candidate CPs to propose
list of close CMPs proposed by the "closeness metric":
[THE INPUT CMP PROPERLY UNDERMATCHES:
 ,match(3.75, [psubj,pobj,with,from]),
THE INPUT CMP NON-PROPERLY UNDERMATCHES:
 ,match(1.11111, [psubj,pobj,to]),match(1.11111, [psubj,pobj,piobj]),mat
ch(1.11111, [psubj,pobj,on]),match(1.11111, [psubj,pobj,from]),match(1.
1111, [psubj,pobj,adv]),match(0.833333, [psubj,pobj,on,adv]),match(0.1
1111, [psubj,on,in]),
THE INPUT CMP PROPERLY OVERMATCHES:
 ,match(1.55556, [psubj,with]),match(1.11111, [psubj,pobj]),match(0.3333
33, [pobj]),match(0.222222, [psubj]),
THE INPUT CMP NON-PROPERLY OVERMATCHES:
 ,match(0.166667, [pobj,adv]),match(0.111111, [psubj,to]),match(0.111111
 , [psubj,for]),match(0.111111, [psubj,adv])]
```

Candidate CPs with example sentences for the closest CMP (psubj-pobj-with-from):

```

CP: agt-obj-inst-lfrm Ex:
[this, letter, was, prepared, by, tom, with, the, new, computer, from, the, other
, room]
current clause's main verb (infinitive): erase
current clause's CMP (or a permutation): psubj-pobj-with
most probable or closest CPs, which are of compatible length: {}

```

NEW CP: list of 28 legal Cases in HAIKU

```

agt:agent; benf:beneficiary; expr:experiencer; inst:instrument;
obj:object; recp:recipient; caus:cause; manr:manner; val:value;
eff:effect; purp:purpose; freq:frequency; tat:time_at;
tfrm:time_from; tto:time_to; ttru:time_through; dir:direction;
lat:location_at; lfrm:location_from; lto:location_to; ord:order;
ltru:location_through; ornt:orientation; cont:content;
excl:exclusion; acmp:accompaniment; matr:material; meas:measure.
REM: A CASE MAY BE SUFFIXED WITH A '?' TO DENOTE UNCERTAINTY

```

(e.g. lat?).

```

here is what the cmDict knows about this Case marker: with
{format: Case(COMmonness,Example); CO=1:common, CO=2:uncommon}
matr(1,The pie was made WITH APPLES.)
acmp(1,He came WITH THEM.)
tfrm(1,WITH THE GUESTS GONE, things have returned to normal.)
manr(1,He did it WITH PANACHE.)
inst(1,The window was broken WITH A ROCK.)
cont(1,I am concerned WITH THE ECONOMY.)
recp(1,She left her car WITH ME.)
caus(2,He trembled WITH FEAR.)
eff(2,The battle will end WITH DEATH.)

```

please enter the new CP (e.g. agt-obj-tat),
 or enter 'h' to see the current input string and CMP,
 or CR to abort ["new CP"/h/CR]? agt-obj-inst
 CMP & CP will be paired as follows: psubj/agt pobj/obj with/inst ;
 Correct [n/Y]?

```

    UPDATING mDict for erase ...
    => erase [psubj-pobj,psubj-pobj-with]
    [[pobj,[obj]],[psubj,[agt]],with,[inst]]
    UPDATING cmpDict for psubj-pobj-with ...
    => psubj-pobj-with [[agt-obj-
inst,['tom,erased,a,file,with,a,new,command,..']]
    UPDATING cpDict for agt-obj-inst ...
    => agt-obj-inst [erase]

```

CASE STRUCTURES

```

clauses
[*statement3*,
  case_structure(
    verb(erase,tense([past_simple]),neg(yes),trans(tr_intr)),
    psubj-pobj-with,
    agt-obj-inst,
    entity(proper_noun(tom),number(sg3),tokens([tom])),
    [entity(determinatives(deter(a)),head_noun(noun(file)),
      number(sg3),tokens([a,file])),[pp(with,
        { ... },
        [main_verb_modif(nil),clause_modif(nil)])]]

```

HAIKU semantic analysis (cpu time) : 2.234 second(s).

E.4: Situation δ_4

```

Unit #4: Bob erased a file .
lexical analysis (cpu time) : 0.067 second(s).
First parse for input sentence:

{ ... }

parsing (cpu time) : 0.083 second(s).

parse_tree__decla_or_imper
  simple_sentence
    structure
      single_main_clause
        declarative
{ ... }

HAIKU: initiating Case-based semantic analysis ...

UPDATED PARSE
TREE(parse_tree__decla_or_imper(simple_sentence(structure(single_main
_clause(declarative(*statement2*))),tokens([bob,erased,a,file])),end_
of_input(period)))

LOGICAL CLAUSAL STRUCTURE
  top_structure
    [*statement2*]

SYMBOLIC A-LISTS
  coordinators
  clauses

[*statement2*,statement(subj(entity(proper_noun(bob),number(sg3),toke
ns([bob]))),predicate(regular(verb(erase,tense([past_simple]),neg(yes
),trans(tr_intr))),voice(active),complement(svo(entity(determinatives
(deter(a)),head_noun(noun(file)),number(sg3),tokens([a,file]))))),tok
ens([erased,a,file]))]

INPUT STRING([bob,erased,a,file,..])

HAIKU: Case Analysis of current input's clauses ...

CURRENT CLAUSE: *statement2*
CURRENT SUBJ   :
subj(entity(proper_noun(bob),number(sg3),tokens([bob])))
CURRENT VERB   : erase
CURRENT COMPL  :
complement(svo(entity(determinatives(deter(a)),head_noun(noun(file)),
number(sg3),tokens([a,file]))))
CURRENT MODIFS : [main_verb_modif(nil),clause_modif(nil)]

CMP found by HAIKU: psubj-pobj
> Do you wish to overrule this CMP
  (enter d to discard this clause, or
   a to discard the entire input) [y/N/d/a]?

```

{below: notice, HAIKU identifies the current clause as of the δ_4 type.}

S4: new verb with known CMP; new CP (relative to this verb).

candidate CPs with example sentences:

CP: agt-obj Ex: [bob,printed,a,file]
 current clause's main verb (infinitive): erase
 current clause's CMP (or a permutation): psubj-pobj
 most probable or closest CPs, which are of compatible length:
 [agt-obj]

NEW CP: list of 28 legal Cases in HAIKU

agt:agent; benf:beneficiary; expr:experiencer; inst:instrument;
 obj:object; recp:recipient; caus:cause; manr:manner; val:value;
 eff:effect; purp:purpose; freq:frequency; tat:time_at;
 tfrm:time_from; tto:time_to; ttru:time_through; dir:direction;
 lat:location_at; lfrm:location_from; lto:location_to; ord:order;
 ltru:location_through; ornt:orientation; cont:content;
 excl:exclusion; acmp:accompaniment; matr:material; meas:measure.
 REM: A CASE MAY BE SUFFIXED WITH A '?' TO DENOTE UNCERTAINTY
 (e.g. lat?).

please enter the new CP (e.g. agt-obj-tat),

or enter 'h' to see the current input string and CMP,

or CR to abort ["new CP"/h/CR]? agt-obj

CMP & CP will be paired as follows: psubj/agt pobj/obj ; Correct
 [n/Y]?

UPDATING mDict for erase ...

=> erase [psubj-pobj] [[pobj,[obj]],[psubj,[agt]]]

UPDATING cpDict for agt-obj ...

=> agt-obj

[ask,complete,copy,crash,delete,erase,get,give,know,like,look,pay,pre
 fer,prepare,print,read,recreate,report,save,sort,store,think,try,want
 ,work,write]

CASE STRUCTURES

clauses

["statement2",

case_structure(

verb(erase,tense([past_simple]),neg(yes),trans(tr_intr)),

psubj-pobj,

agt-obj,

entity(proper_noun(bob),number(sg3),tokens([bob])),

entity(determinatives(deter(a)),head_noun(noun(file)),

number(sg3),tokens([a,file])),

[main_verb_modif(nil),clause_modif(nil)])]

HAIKU semantic analysis (cpu time) : 1.25 second(s).

E.5: Situation 85

Unit #5: Bill unexpectedly modified his data file with the REPL command .

lexical analysis (cpu time) : 0.1 second(s).

First parse for input sentence:

{ ... }

parsing (cpu time) : 2.05 second(s).

```

parse_tree__decla_or_imper
  simple_sentence
    structure
      single_main_clause
        declarative

```

{ ... }

HAIKU: initiating Case-based semantic analysis ...

UPDATED PARSE

```

TREE(parse_tree__decla_or_imper(simple_sentence(structure(single_main
_clause(declarative(*statement6*))),tokens([bill,unexpectedly,modifie
d,his,data,file,with,the,repl,command])),end_of_input(period)))

```

LOGICAL CLAUSAL STRUCTURE

```

top_structure
  [*statement6*]

```

SYMBOLIC A-LISTS

```

coordinators
clauses

```

```

[*statement6*,statement(subj(entity(proper_noun(bill),number(sg3),tok
ens([bill]))),predicate(regular(verb(modify,tense([past_simple]),neg(
yes),advs([unexpectedly,tokens([unexpectedly]))),trans(tr)),voice(ac
tive),complement(svoo_svoc_svoa([entity(determinatives(deter(his)),he
ad_noun(noun(file,noun_modifiers(pre_modif([datum]))),number(sg3),to
kens([his,data,file])),[pp(with,entity(determinatives(deter(the)),hea
d_noun(noun(command,noun_modifiers(pre_modif([repl]))),number(sg),to
kens([the,repl,command])),tokens([with,the,repl,command])),tokens([wi
th,the,repl,command]))])),tokens([unexpectedly,modified,his,data,fil
e,with,the,repl,command]))])

```

INPUT

```

STRING([bill,unexpectedly,modified,his,data,file,with,the,repl,comman
d,.])

```

HAIKU: Case Analysis of current input's clauses ...

CURRENT CLAUSE: *statement6*

CURRENT SUBJ :

```

subj(entity(proper_noun(bill),number(sg3),tokens([bill]))

```

CURRENT VERB : modify

CURRENT COMPL :

```

complement(svoo_svoc_svoa([entity(determinatives(deter(his)),head_nou

```

```

n(noun(file,noun_modifiers(pre_modif([datum]))),number(sg3),tokens([
his,data,file])).[pp(with,entity(determinatives(deter(the)),head_noun
(noun(command,noun_modifiers(pre_modif([repl]))),number(sg),tokens([
the,repl,command])),tokens([with,the,repl,command])),tokens([with,the
,repl,command]))]]))
CURRENT MODIFS :
[main_verb_modif(advs([unexpectedly,tokens([unexpectedly]))),clause_
modif(nil)]

```

```

CMP found by HAIKU: psubj-pobj-with-adv
> Do you wish to overrule this CMP
  (enter d to discard this clause, or
   a to discard the entire input) [y/N/d/a]?

```

{below: notice, HAIKU identifies the current clause as of the 85 type. Note: all the intermediate results returned by the metric are only shown here as a trace of execution; what matters to the user is the closest CP proposed by HAIKU.}

S5: new verb with (entirely) new CMP; new CP.

```

entirely new CMP; no obvious candidate CPs to propose
list of close CMPs proposed by the "closeness metric":
[THE INPUT CMP PROPERLY UNDERMATCHES: ,
THE INPUT CMP NON-PROPERLY UNDERMATCHES:
,match(2.8125,[psubj,pobj,with,from]),match(1.6875,[psubj,pobj,on,adv
]),
THE INPUT CMP PROPERLY OVERMATCHES:
,match(2.5,[psubj,pobj,with]),match(1.5,[psubj,pobj,adv]),match(1.166
67,[psubj,with]),match(0.833333,[psubj,pobj]),match(0.666667,[pobj,ad
v]),match(0.5,[psubj,adv]),match(0.25,[pobj]),match(0.166667,[psubj])
,match(0.083333,[adv]),
THE INPUT CMP NON-PROPERLY OVERMATCHES:
,match(0.555556,[psubj,pobj,to]),match(0.555556,[psubj,pobj,piobj]),m
atch(0.555556,[psubj,pobj,on]),match(0.555556,[psubj,pobj,from]),matc
h(0.083333,[psubj,to]),match(0.083333,[psubj,for]),match(0.0555556,
[psubj,on,in])

```

Candidate CPs with example sentences for the closest CMP (psubj-pobj-with-from):

```

CP: agt-obj-inst-lfrm Ex:
[this,letter,was,prepared,by,tom,with,the,new,computer,from,the,other
,room]
current clause's main verb (infinitive): modify
current clause's CMP (or a permutation): psubj-pobj-with-adv
most probable or closest CPs, which are of compatible length:
[agt-obj-inst-lfrm]

```

NEW CP: list of 28 legal Cases in HAIKU

```

agt:agent; benf:beneficiary; expr:experiencer; inst:instrument;
obj:object; recp:recipient; caus:cause; manr:manner; val:value;
eff:effect; purp:purpose; freq:frequency; tat:time_at;
tfrm:time_from; tto:time_to; ttru:time_through; dir:direction;
lat:location_at; lfrm:location_from; lto:location_to; ord:order;
ltr:location_through; ornt:orientation; cont:content;
excl:exclusion; acmp:accompaniment; matr:material; meas:measure.
REM: A CASE MAY BE SUFFIXED WITH A '?' TO DENOTE UNCERTAINTY
(e.g. lat?).

```

```

here is what the cmDict knows about this Case marker: with
(format: Case(Commonness,Example); CO=1:common, CO=2:uncommon)
  matr(1,The pie was made WITH APPLES.)
  acmp(1,He came WITH THEM.)
  tfirm(1,WITH THE GUESTS GONE, things have returned to normal.)
  manr(1,He did it WITH PANACHE.)
  inst(1,The window was broken WITH A ROCK.)
  cont(1,I am concerned WITH THE ECONOMY.)
  recp(1,She left her car WITH ME.)
  caus(2,He trembled WITH FEAR.)
  eff(2,The battle will end WITH DEATH.)

```

```

please enter the new CP (e.g. agt-obj-tat),
  or enter 'h' to see the current input string and CMP,
  or CR to abort ["new CP"/h/CR]? agt-obj-inst-manr
CMP & CP will be paired as follows: psubj/agt pobj/obj with/inst
adv/manr ; Correct [n/Y]?

```

```

UPDATING mDict for modify ...
=> modify [psubj-pobj-with-adv]
[[adv,[manr]], [pobj,[obj]], [psubj,[agt]], [with,[inst]]]
UPDATING cmpDict for psubj-pobj-with-adv ...
=> psubj-pobj-with-adv [[agt-obj-inst-
manr, '[bill,unexpectedly,modified,his,data,file,with,the,repl,command
,.]']]
UPDATING cpDict for agt-obj-inst-manr ...
=> agt-obj-inst-manr [modify]

```

CASE STRUCTURES

```

clauses
[*statement6*,
  case_structure(
    verb(modify,tense([past_simple]),neg(yes),
      advs([unexpectedly,tokens([unexpectedly])]),trans(tr)),
    psubj-pobj-with-adv,
    agt-obj-inst-manr,
    entity(proper_noun(bill),number(sg3),tokens([bill])),
    [entity(determinatives(deter(his)),head_noun(noun(file),
      noun_modifiers(pre_modif([datum]))),number(sg3),
      { ... }
    [main_verb_modif(advs([unexpectedly,tokens([unexpectedly])])),
      clause_modif(nil)]]]

```

HAIKU semantic analysis (cpu time) : 2.417 second(s).

Appendix F: HAIKU's CA Processor

This appendix contains the listings of the two major Prolog files that constitute the HAIKU Case-based semantic analyzer: `HAIKU_SA.PL`, the Case-Analyzer driver, and `HAIKU_CA.PL`, the individual-clause Case Analyzer — see section 4.3. Two other files that are part of HAIKU but which do not appear below are its list of (domain-independent) prepositions and the Cases they mark (which was extracted from [Barker *et al.* 93]), plus the closeness metric (which is described in section 4.6).

Note: the end of section F.1 (`HAIKU_SA.PL`) also presents the Prolog code for the pronoun explicitizer (see 3.5). This processing takes place right after parsing and just before Case analysis.

F.1: The CA Driver

```

/*****
/* file: HAIKU_SA.PL
/* version: July 20, 1993
/* author: Sylvain Delisle
*****/

:- dynamic previous_PT/1.

:- volatile previous_PT/1.

:- assert(haiku_version(20,7,93)).

/*****
/* SA : (used by proceed_with_haiku_sa in dipett_1)
*****/

sa(Syn,Sem) :-
    on_exception(Msg,
        haiku_sa(Syn,Sem),
        display_exception_message_and_fail__SA(Msg)).

display_exception_message_and_fail__SA(Exception_Message) :-
    nlp, nlp, writep('HAIKU_SA STOPPED for the following reason:'),
    nlp, writep(Exception_Message), nlp, nlp, !, fail.

/*****
/* HAIKU_SA : (used by sa)
/*
/*
/* In order to perform CA we must collect all clauses. They are located in
/* nominal clauses (to_infinitive_clause, genitive_ing_clause, ing_clause,
/* certain rel_clauses, wh_interro_declarative_clause, and that_clause) or in
/* statements. Amongst the 6 types of nominal clauses, the first four are
/* not defined in terms of 'statement', the 5th may, and the 6th is. This ex-
/* plains why we first search for the 4 first types of nominal clauses and
/* then for statements (see haiku_sa_0 & extract_top_nom_clauses).
*****/

haiku_sa(In_PT,nil) :-
    fas('parse_tree__inter',In_PT,LSAT), LSAT \== [],
    nlp, writep('Semantic analysis ABORTED: the current input is a '),
    writep('question!'),
    nlp, nlp, replace_previous_PT(In_PT).

%haiku_sa(In_PT,nil) :-
%    fas('parse_tree__title_or_caption',In_PT,LSAT), LSAT \== [],
%    nlp, writep('Semantic analysis ABORTED: the current input is a '),
%    writep('title or caption!'),
%    nlp, nlp, replace_previous_PT(In_PT).

haiku_sa(In_PT,Sem_Struct) :-
    fas('parse_tree__frag',In_PT,LSAT1), LSAT1 \== [],
    fas('sentence_fragment',In_PT,LSAT2), LSAT2 \== [],
    nlp, writep('** CA: the current input is a series of fragments ...'), nlp,
    replace_previous_PT(In_PT),
    ext(n, frags, sentence_fragment, In_PT, _Out_PT, A_List),
    A_List =.. [frags|[FRAGMENTS]],
    haiku_sa_fragments(FRAGMENTS,[],Sem_Struct).

/* LOOKING FOR CONJOINED VERB PHRASES (VPs) ...
haiku_sa(Input_PT,Sem_Struct) :-

/* Getting rid of 'nil'/'[]' elements ...

```

```

zap_nils(Input_PT, In_PT_ok), !,

ext(1, conj_VP1, conj_vps, In_PT_ok, _, A_List_VPs),
((A_List_VPs == conj_VP1([]), % No conjoined VPs!
  PTS = [In_PT_ok]) ;
(A_List_VPs \== conj_VP1([]), % Yes. conjoined VPs!
  A_List_VPs = conj_VP1([_, CONJ_VPS])),
ext(n, conj_VP2, predicate, CONJ_VPS, _, A_VP_List),
A_VP_List = conj_VP2(VP_List), length(VP_List, L1),
nlp, writep('WARNING! The input contains '), writep(L1),
writep(' DISTINCT CONJOINED VP(s):'), nlp, writep(CONJ_VPS), nlp, nlp,
writep('Semantic Analysis will proceed according to this '),
writep('inference rule:'),
nlp, writep(' SUBJ [VP1 & VP2 & ... & VPn] =>'), nlp,
writep(' SUBJ VP1, SUBJ VP2, ... , SUBJ VPn.'), nlp,
writep('An input containing N (distinct) '),
writep('conjoined VPs will generate N clauses for HAIKU.'), nlp, nlp,
haiku_sa_1_VPs(1, In_PT_ok, CONJ_VPS, VP_List, [], PTS)),
haiku_sa_1_Verbs(PTS, [], Sem_Struct).

/* ++++++ */

haiku_sa_fragments([], Sem_Struct, Sem_Struct) :- !.
haiku_sa_fragments([_, FRAG_H] | FRAG_T, Sem_Struct_in, Sem_Struct_out) :-
  haiku_sa(FRAG_H, Sem_Struct_H), !,
  ((Sem_Struct_H = [logical_input(_, _, _, 'CASE STRUCTURES'([X]))], var(X),
    writep('> EMPTY CLAUSE: CA cannot be performed on this partial input.'),
    nlp, nlp) ;
  true), !,
  append(Sem_Struct_in, [Sem_Struct_H], Sem_Struct_0),
  haiku_sa_fragments(FRAG_T, Sem_Struct_0, Sem_Struct_out).

/* ++++++ */

haiku_sa_1_VPs(1, In_PT, CVPS, [], PTS, PTS) :- !.
haiku_sa_1_VPs(N, In_PT, CVPS, [_, VP] | T, PTS_in, PTS_out) :-
  subst(1, CVPS, VP, In_PT, Out_PT_temp), append(PTS_in, [Out_PT_temp], PTS_in_2),
  writep('>> CONJOINED VPs: clause #'), writep(N), writep(' ... '), nlp,
  writep(VP), nlp, M is N + 1,
  haiku_sa_1_VPs(M, In_PT, CVPS, T, PTS_in_2, PTS_out).

/* LOOKING FOR CONJOINED VERBS ... */
haiku_sa_1_Verbs([], Sem_Struct, Sem_Struct) :- !.
haiku_sa_1_Verbs([Input_PT | T], Sem_Struct_in, Sem_Struct) :-
  ext(1, conj_V1, conj_verbs, Input_PT, _, A_List_Verbs),
  ((A_List_Verbs == conj_V1([]), % No conjoined verbs!
    haiku_sa_0(Input_PT, A, B, C, D, E),
    Sem_Struct_temp = logical_input(A, B, C, D, E)) ;
  (A_List_Verbs \== conj_V1([]), % Yes. conjoined verbs!
    A_List_Verbs = conj_V1([_, CONJ_VERBS])),
  ext(n, conj_V2, verb, CONJ_VERBS, _, A_Verb_List),
  A_Verb_List = conj_V2(Verb_List), length(Verb_List, L2),
  nlp, writep('WARNING! The input contains '), writep(L2),
  writep(' DISTINCT CONJOINED VERB(S):'), nlp, writep(CONJ_VERBS), nlp, nlp,
  writep('Semantic Analysis will proceed according to this '),
  writep('inference rule:'),
  nlp, writep(' SUBJ [Verb1 & Verb2 & ... & VerbN] COMPL =>'), nlp,
  writep(' SUBJ Verb1 COMPL, SUBJ Verb2 COMPL, ... , '),
  writep('SUBJ VerbN COMPL.'), nlp,
  writep('An input containing N (distinct) '),
  writep('conjoined verbs will generate N clauses for HAIKU.'),
  nlp, nlp, haiku_sa_1(1, Input_PT, CONJ_VERBS, Verb_List, Sem_Struct_temp)),
  append(Sem_Struct_in, [Sem_Struct_temp], Sem_Struct_out),

```

```

haiku_sa_1_Verbs(PT_T, Sem_Struct_out, Sem_Struct).

haiku_sa_1(_, _In_PT, CV, [], []) :- !.
haiku_sa_1(N, In_PT, CV, [[_, VERB] | T], [logical_input(A, B, C, D, E) | R]) :-
    subst(1, CV, VERB, In_PT, Out_PT_temp),
    writep('>> CONJOINED VERBS: clause #'), writep(N), writep(' ... '), nlp,
    writep(VERB), nlp,
    haiku_sa_0(Out_PT_temp, A, B, C, D, E), M is N + 1,
    haiku_sa_1(M, In_PT, CV, T, R).

/* ***** */
haiku_sa_0(In_PT,
    'INPUT STRING'(Input_Tokens),
    'UPDATED PARSE TREE'(Updated_PT),
    'LOGICAL CLAUSAL STRUCTURE'(LOGIC_STRUCT_ok),
    'SYMBOLIC A-LISTS'([A_List_top_coords, A_List_clauses]),
    'CASE STRUCTURES'([Clausal_Case_Structures]))
:-
/* Let's make sure it's not a question, a caption, or a series of fragments. */
fas('parse_tree__inter', In_PT, LSAT1), LSAT1 == [],
% fas('parse_tree__title_or_caption', In_PT, LSAT2), LSAT2 == [],
fas('parse_tree__frag', In_PT, LSAT3), LSAT3 == [],

/* Fetching the current input's list of tokens ... */
sentence_tokens(Input_Tokens), !, % this is asserted in DIPETT_1.

/* "Explicitization" of pronouns ... */
pro_explicitizer(In_PT, In_PT_ok), !.

/* Extracting clauses from the input parse tree ... */
extract_top_nom_clauses(nil, In_PT_ok, PT_Out, [], A_List_CL_Out_r),
rev(A_List_CL_Out_r, A_List_Nom_clauses), !.

/* Extracting the top-level structure ... */
top_st(PT_Out, TOP_STRUCT, Updated_PT, A_List_Nom_clauses,
    CLAUSES_L, A_List_clauses, A_List_top_coords), !.

/* Extracting the 'sub' top-level structure ... */
sub_st(CLAUSES_L, SUB_STRUCT), append([TOP_STRUCT], SUB_STRUCT, LOGIC_STRUCT),
zap_nils(LOGIC_STRUCT, LOGIC_STRUCT_ok), !.

TOK = 'INPUT STRING'(Input_Tokens),
UPPT = 'UPDATED PARSE TREE'(Updated_PT),
LS = 'LOGICAL CLAUSAL STRUCTURE'(LOGIC_STRUCT_ok),
SAL = 'SYMBOLIC A-LISTS'([A_List_top_coords, A_List_clauses]),
nlp, writep(UPPT), nlp,
nlp, writep_list(LS),
nlp, writep_list(SAL),
nlp, writep(TOK), nlp, !.

/* The ultimate step: perform CA for each and every clause. */
clausal_Case_Analysis(A_List_clauses, Clausal_Case_Structures_0),
((Clausal_Case_Structures_0 \== []),
    Clausal_Case_Structures =.. [clauses|[Clausal_Case_Structures_0]],
    CS = 'CASE STRUCTURES'([Clausal_Case_Structures]),
    nlp, writep_list(CS), nlp),
(Clausal_Case_Structures_0 == [], nlp)),
replace_previous_PT(In_PT_ok).

/*****/

```

```

/* WRITEP_LIST (used by haiku_sa_0) */
/*
/* A simple "pretty-printer" for HAIKU's output structures. */

writep_list(L) :-
    L =.. [H|[T]], H \== '.', !, writep(H), nlp, writep_list2(T).
writep_list(L) :-
    L =.. [H|T], H \== '.', !, writep(H), nlp, writep_list2(T).
writep_list(X) :- writep(X), nlp.

writep_list2([]) :- !.
writep_list2([L|M]) :-
    L =.. [H|[T]], H \== '.', !, writep(' '), writep(H), nlp,
    writep_list3(T), writep_list2(M).
writep_list2([L|M]) :-
    L =.. [H|T], H \== '.', !, writep(' '), writep(H), nlp,
    writep_list3(T), writep_list2(M).
writep_list2(X) :- writep(' '), writep(X), nlp.

writep_list3([]) :- !.
writep_list3([H|T]) :- writep(' '), writep(H), nlp, writep_list3(T).
writep_list3(X) :- writep(' '), writep(X), nlp.

/* ***** */
/* For pronoun explicitization: saving the previous parse tree. */

replace_previous_PT(In_PT) :-
    retractall(previous_PT(_)), assert(previous_PT(In_PT)).

/***** */
/* ZAP NILS (used by haiku_sa_0) */
/*
/* This predicate eliminates nils and []'s from the input parse tree. We */
/* need two phases (i.e. passes) in order to eliminate 2nd order nils. */
/* 'zap_nils' also eliminates redundant list of tokens that were introduced */
/* in the parse tree if the "include phrase tokens in the parse tree" sub- */
/* option, in option D of the parsing menu, was selected (i.e. activated). */

zap_nils(In,Out) :- nils_off(In,Out0), nils_off(Out0,Out).

nils_off(X,X) :- var(X), !.
nils_off(X,X) :- atomic(X), !.
nils_off([tokens(L) | [tokens(L) | T]], [tokens(L) | R]) :- nils_off(T,R).
nils_off([H|T],Y) :-
    functor(H,F,1), F \== '.', (H =.. [F,nil] ; H =.. [F,[]]), !, nils_off(T,Y).
nils_off([H|T],[H2|T2]) :-
    !, nils_off(H,H2), nils_off(T,T2).
nils_off(X,Y) :-
    functor(X,F,_), X =.. Z, nils_off(Z,Y0),
    ((Y0 == [F], Y =.. [F,nil]) ; % Introducing an artificial 'nil':
    (Y0 \== [F], Y =.. Y0)). % we eliminate these in the 2nd pass.

/***** */
/* EXTRACT_TOP_NOM_CLAUSES (used by haiku_sa_0) */
/*
/* Starting from the highest level, this predicate extracts nominal clauses */
/* and handles correctly embedded clauses. The first parameter, 'Restric', */
/* is necessary in order to avoid infinite recursion: since 'ext' handles */
/* embedded situations for a given functor, we must avoid attempting to find */
/* such a functor within an occurrence of the same functor here. */

extract_top_nom_clauses(Restric, In_PT, Out_PT, A_List_CL_In, A_List_CL_Out)
:-

```

```

extract_nom_clauses(Restric,rel_clause,
                    In_PT,PT_out1,A_List_CL_In,A_List_CL_out1),

extract_nom_clauses(Restric,to_infinitive_clause,
                    PT_out1,PT_out2,A_List_CL_out1,A_List_CL_out2),

extract_nom_clauses(Restric,genitive_ing_clause,
                    PT_out2,PT_out3,A_List_CL_out2,A_List_CL_out3),

extract_nom_clauses(Restric,ing_clause,
                    PT_out3,PT_out4,A_List_CL_out3,A_List_CL_out4),

extract_nom_clauses(Restric,wh_interro_declarative_clause,
                    PT_out4,Out_PT,A_List_CL_out4,A_List_CL_Out).

extract_nom_clauses(R,R,In_PT,In_PT,A_List_CL_In,A_List_CL_In)      :- !.
extract_nom_clauses(_,NOM_CL,In_PT,Out_PT,A_List_CL_In,A_List_CL_Out) :-
    ext(n,out,NOM_CL,In_PT,Out_PT,A_List), A_List =.. [out|[CLAUSES]],
    CLAUSES \== [], !,
    extract_nom_clauses_2nd_lev(NOM_CL,CLAUSES,CL_Out,A_List_CL_In,A_List_CL_1),
    append(A_List_CL_1,CL_Out,A_List_CL_Out).
extract_nom_clauses(_,_,In_PT,In_PT,A_List_CL_In,A_List_CL_In)      :- !.

extract_nom_clauses_2nd_lev(statement,In_PT,In_PT,A_List_CL_In,A_List_CL_In) :-
    !.
extract_nom_clauses_2nd_lev(Restric,In_PT,Out_PT,A_List_CL_In,A_List_CL_Out) :-
    extract_top_nom_clauses(Restric,In_PT,PT_out1,A_List_CL_In,A_List_CL_out1),
    extract_nom_clauses(nil,
                        statement,PT_out1,Out_PT,A_List_CL_out1,A_List_CL_Out).

/*****
/* TOP_ST: (used by haiku_sa_0)
/*
/* This predicate extracts the top-level structure of DIPETT parse trees.
/* Such a structure contains the main clauses (statements) as well their
/* joining coordinators, if any. A call to top_st(PT,A,B,C,D) will unify A
/* with the top-level structure, B with the updated parse tree (i.e. the ori-
/* ginal input parse tree PT in which symbolic tags have replaced top-level
/* clauses and coordinators), C with the a-list of clauses, and D with the
/* a-list of coordinators.
/* Note: We have 'mult_coor_cl', 'correlative ...', or 'coord' (or nil).
*****/

top_st(In_PT,top_structure(L),PT_2,A_List_Nom_clauses,
        CLAUSES_L,A_List_clauses,A_List_top_coords) :-

    ext(n,clauses,statement,In_PT,PT_1,A_List_statements),

    ((ext(n,mult_coord,mult_coor_clauses,PT_1,_,A_List_top_coords_0),
        A_List_top_coords_0 =.. [mult_coord|[MC]], MC \== [],
        ext(1,mcc_c,coord,MC,_,mcc_c([[SYMB_COORD,COORD]])),
        A_List_top_coords = coordinators([[SYMB_COORD,COORD]]),
        PT_2 = PT_1, MULT = true) ;
        (ext(n,coordinators,coord,PT_1,PT_2,A_List_top_coords),
        MULT = false)), !,

    fas('',PT_2,L0),

    A_List_statements =.. [clauses|[CLAUSES]],
    append(CLAUSES,A_List_Nom_clauses,CLAUSES_L),
    A_List_clauses =.. [clauses|[CLAUSES_L]],

    ((fas('correlative_compound_sentence',PT_2,CCS), CCS \== [], !,

```

```

L0 = [C1,S1,C2,S2].
find_emb_subordinators(S1,CLAUSES_L,New_S1).
find_emb_subordinators(S2,CLAUSES_L,New_S2).
L000 = [[C1,New_S1],[C2,New_S2]] :
ftst(L0,CLAUSES_L,L000)).

((MULT == true, insert_COORD_inMULT(L000,SYMB_COORD,L00)) ;
 (MULT == false, L00 = L000)).

length(L00,Le). ((Le <= 1, L = L00) ; (Le > 1, L = [L00])).

/* ***** */
/* Handles the 'mult_coor_clauses' special case by explicitly re-introducing */
/* the coordinator between the three or more constituting statements. */
/* For example, we must change ['statement1','statement2','statement3'] into */
/* ['statement1','coord1','statement2','coord1','statement3']. */
/* ***** */

insert_COORD_inMULT([],_,[]) :- !.
insert_COORD_inMULT([H|T],SYMB_COORD,[H|New_T]) :-
    fas('statement',H,L), L \== [], !,
    ((T \== [], New_T = [SYMB_COORD|Rest]) ;
     (T == [], New_T = Rest)),
    insert_COORD_inMULT(T,SYMB_COORD,Rest).

/***** */
/* FTST: (used by top_st) */
/* */
/* ftst (find top structure): fine-grained analysis launched in 'top_st'. */
/* Ex. arg. #1: ['statement1','coord1','statement2','coord2','statement3']. */
/***** */

ftst([],_,[]) :- !.
ftst([H|T],CLAUSES,[H|New_T]) :-
    fas('coord',H,L), L \== [], length(T,LT), LT > 1, !, ftst(T,CLAUSES,New_T).
ftst([H|T],CLAUSES,[H|New_T]) :-
    fas('coord',H,L), L \== [], !, ftst(T,CLAUSES,New_T).
ftst([H|T],CLAUSES,[New_H|New_T]) :-
    find_emb_subordinators(H,CLAUSES,New_H), ftst(T,CLAUSES,New_T).

find_emb_subordinators(H,CLAUSES,New_H) :-
    fetch_Hclause(H,CLAUSES,Hclause),
    ext(n,sub_clauses,subordinate,Hclause,_,sub_clauses(SUB_CL_A_List)),
    ((SUB_CL_A_List == [], New_H = H) ;
     (SUB_CL_A_List \== [],
      extract_sub_cl(H,SUB_CL_A_List,CLAUSES,New_H_0),
      (New_H_0 = [New_H] ; New_H = New_H_0))), !.

fetch_Hclause(_,[],_) :- !, fail.
fetch_Hclause(H,[[H,Hclause]|_],Hclause) :- !.
fetch_Hclause(H,[_|T],Hclause) :- fetch_Hclause(H,T,Hclause).

extract_sub_cl(_,[],_) :- !.
extract_sub_cl(H,[_|SUB_CL]|T,CLAUSES,New_HT) :-
    ext(1,subord,subordinator,SUB_CL,_,LL),
    ((LL \== subord([]), LL = subord([[_|SUB]]) ,
     fas('statement',SUB_CL,CL), CL \== [], CL = [CL_H],
     find_emb_subordinators(CL_H,CLAUSES,New_CL),
     New_HT = [[H,SUB,New_CL]|New_T],
     extract_sub_cl(H,T,CLAUSES,New_T)) ;
     ((T == [], New_HT = [H|New_T]) ; (T \== [], New_HT = New_T))),
    extract_sub_cl(H,T,CLAUSES,New_T)).

```

```

/*****
/* SUB_ST: (used by haiku_sa_3) */
/*
/* We here identify the sub-structure (vs 'top') of main statements by lo-
/* cating 'conj_nom_cl' within 'statement'(s). */

sub_st([],[]) :- !.
sub_st([H|T],[structure(S,S_st)|R]) :-
    H = [S,ST], ext(n,conj_cl,conj_nom_cl,ST,_,conj_cl([_,S_st])),
    sub_st(T,R).
sub_st([_|T],R) :- sub_st(T,R).

/*****
/* EXT: extracts all the different ST_ID (functor name) subtrees of an input */
/* parse tree In_PT and outputs the parse tree with all the ST_ID subtrees */
/* replaced by a symbolic tag, as well as the a-list of each tag with their */
/* associated subtree. 'ext' handles correctly parse trees (or in fact, any */
/* legal Prolog structure) in which there are many occurrences or embedded oc- */
/* currences of ST_ID structures. However, it should be noted that 'ext' does */
/* not handle entirely correctly duplicates which happen to be also embedded: */
/* this is so for ext(b_list,b,a(b(99),f(w),b(b(99))),K,L) because of b(99). */
/*
/* Here is an example ("extract subtrees beginning with a 'b' functor ..."): */
/* ext(symb,b,a(12,b(f(g)),h(q),b(33,a,b(b(99)))),X,Y) returns */
/* X = a(12,'*b7*',h(q),'*b8*'), */
/* Y = symb([['*b7*',b(f(g))], */
/*          ['*b8*',b(33,a,'*b9*')],['*b9*',b('*b10*')],['*b10*',b(99)]]). */
/* Note: alien symbols, introduced by 'ext', begin and end with '*'. */
/*
/* Remark: this predicate is also used in several places in 'HAIKU_CA.PL'. */

ext(TYPE,ST_ID_F,ST_ID,In_PT,NewPT,Symb_Assoc__A_List) :-
    (In_PT \== nil, In_PT \== [], In_PT \== error), !,
    ste(ST_ID,In_PT,ST_List),
    subst_in_PT(TYPE,ST_ID,In_PT,ST_List,A_List_temp,NewPT,[],Embl),
    process_emb(ST_ID,A_List_temp,Embl,A_List),
    filter_A_List(NewPT,A_List,A_List,A_List_ok),
    Symb_Assoc__A_List =.. [ST_ID_F|[A_List_ok]].
ext(_,_,_,PT,PT,[]) :- !.

/*****
/* STE: (used by ext, and several others) */
/*
/* "Sub-Tree Extract" is a predicate which finds all the subtrees of an arbi- */
/* trary Prolog tree T having a specific functor name F. If S does not con- */
/* tain at least one occurrence of F, 'ste' returns '[]'. 'ste' handles cor- */
/* rectly multi-occurring Fs, as well as embedded Fs. */
/*
/* Example: ste(b,a(1,b(2),3,c(4,b(5),e(b)),b(b(b(99)))),L) returns */
/* L = [b(2),b(5),b(b(b(99))),emb([b(b(99)),emb([b(99)])])]. */

ste(_,T,[]) :- (atomic(T) ; var(T)), !.
ste(F,T,L) :- (functor(T,F,_); % functor F, OR F0
               (functor(T,F0,_), fas(F,F0,[F0])), % with F as a prefix.
               F \== '.', T =.. TL, ste(F,TL,L1), append_emb([T],L1,L), !.
               !, fail.
ste(F,[T],L) :- ste(F,T,L), !.
ste(F,[TH|TT],L) :- ste(F,TH,T1), ste(F,TT,T2), append(T1,T2,L).
ste(F,T,L) :- functor(T,G,_), !, G \== '.', T =.. TL, ste(F,TL,L).

append_emb(X,[],X) :- !.
append_emb(X,Y,Z) :- append(X,[emb(Y)],Z).

```

```

/*****
/* SUBST_IN_PT (used by ext)
/*
/* This predicate serves two functions. First, it controls the substitution
/* of subtrees with symbolic tags in a parse tree. Second, it collects all
/* the symbolic tags associated with embedded subtrees in order to process
/* these separately afterwards (see 'process_emb' below).

subst_in_PT(_,_,NewPT,[],[],NewPT,Embl,Embl)
    !.
subst_in_PT(T,ST_ID,PT,[emb(ST)|ST_T],A_List,NewPT,Embl0,Embl)
    append(Embl0,[emb(ST)],Embl1), !,
    subst_in_PT(T,ST_ID,PT,ST_T,A_List,NewPT,Embl1,Embl).
subst_in_PT(1,ST_ID,PT,[ST_H|ST_T],[[Gen_ST_ID,ST_H]|ST_T],NewPT,Embl,Embl) :-
    !, gensym_''(ST_ID,Gen_ST_ID), subst(1,ST_H,Gen_ST_ID,PT,NewPT).
subst_in_PT(T,ST_ID,PT,[ST_H|ST_T],[[Gen_ST_ID,ST_H]|R],NewPT,Embl0,Embl) :-
    gensym_''(ST_ID,Gen_ST_ID), subst(T,ST_H,Gen_ST_ID,PT,NewPT_temp),
    subst_in_PT(T,ST_ID,NewPT_temp,ST_T,R,NewPT,Embl0,Embl).

/* We concatenate a '' character to the beginning and the end of the gensym */
/* symbol in order to differentiate symbols introduced automatically with */
/* those found in the input. For example: 'statement1'', 'coord17'', etc. */
/* Note: used by 'subst_in_PT' and 'update_A_List_with_emb'. */
gensym_''(In,Out) :-
    atom(In), gensym(In,Out2), name(Out2,L2),
    append([42],L2,L1), append(L1,[42],L0), name(Out,L0).

/*****
/* SUBST: (used by subst_in_PT, update_A_List_with_emb, and several others) */
/*
/* A classic: see details in Clocksin & Mellish or Sterling & Shapiro!
/* subst(n,Old,New,OldTerm,NewTerm): NewTerm is the result of replacing ALL
/* occurrences of Old in OldTerm by New.
/*
/* The "1st only" version was provided by Feliks Kluzniak (via S.Szapkowicz).
/* subst(1, + A, + B, + X, +- Y):
/* If the term X contains occurrences of term A, replace the FIRST occurrence
/* of A with B and try to unify the result with Y. If X does not contain any
/* occurrences of A, try to unify X with Y.

subst(n,Old,New,Old,New) :- !.
subst(n,_Old,_New,Val,Val) :- atomic(Val), !.
subst(n,Old,New,Val,NewVal) :-
    functor(Val,Fn,N), functor(NewVal,Fn,N), subst_args(N,Old,New,Val,NewVal).

subst(1,A,B,X,Y) :- s(A,B,X,Z), !, Y = Z.
subst(1,_A,_B,X,X).

/* ***** */

/* subst/5 handles compound terms by iterating over N, Fn's number of args. */
subst_args(0,_,_,_,_) :- !.
subst_args(N,Old,New,Val,NewVal) :-
    arg(N,Val,OldArg), arg(N,NewVal,NewArg), subst(n,Old,New,OldArg,NewArg),
    N1 is N - 1, subst_args(N1,Old,New,Val,NewVal).

/* s( + A, + B, + X, - result ):
/* Replace the 1st occurrence of A in X with B. Fail if no such occurrence. */
s(A,B,X,B) :- A == X, !.
s(_A,_B,X,_) :- (var(X) ; atomic(X)), !, fail.
s(A,B,[H|T],[NH|T]) :- s(A,B,H,NH), !.
s(A,B,[H|T],[H|NT]) :- !, s(A,B,T,NT).
s(A,B,X,Y) :- X =.. [F|Args], s(A,B,Args,NArgs), Y =.. [F|NArgs].

```



```

/*****
/* PROCESS_EMB (used by ext) */
/*
/* Controls the updating process of the symbolic table for inputs in which
/* embedded subtrees have been found. Notice that it is not the parse tree
/* which must be updated here but the 'symb_assoc' (a-list). This updating
/* takes place as a post-processing step in the definition of 'ext'. */

process_emb(_,A_List,[],A_List) :- !.
process_emb(ST_ID,A_List_in,Embl,A_List_out) :-
    eliminate_emb_sym(Embl,[],Embl_ok),
    update_A_List_with_emb(ST_ID,A_List_in,Embl_ok,A_List_out).

/* We can now eliminate the 'emb' functors: only a hindrance at this point! */
eliminate_emb_sym([],L,L) :- !.
eliminate_emb_sym([emb(X)|Y],L1,L2) :-
    !, eliminate_emb_sym(X,L1,L3), eliminate_emb_sym(Y,L3,L2).
eliminate_emb_sym([X|Y],L1,L2) :-
    append(L1,[X],L3), eliminate_emb_sym(Y,L3,L2).

/* The actual updating of the a-list, using the 'subst' predicate again. */
update_A_List_with_emb(_,A_List,[],A_List) :- !.
update_A_List_with_emb(ST_ID,A_List_in,[ST|R],A_List_out) :-
    gensym_''(ST_ID,Gen_ST_ID), subst(n,ST,Gen_ST_ID,A_List_in,A_List_temp),
    append(A_List_temp,[[Gen_ST_ID,ST]],A_List_ok),
    update_A_List_with_emb(ST_ID,A_List_ok,R,A_List_out).

/*****
/* FILTER_A_LIST: (used by ext) */
/*
/* Ensures that all symbolic tags contained in the a-list are actually used
/* in either the "new" PT or in the a-list itself (for embedded cases). This
/* predicate was designed to remove useless duplicates in the a-list.
/*
/* For example, ext(b_list,b,a(b(99),f(w),z(b(99))),K,L) returns
/* K = a('b3',f(w),z('b3')).
/* L = b_list(['b3',b(99)]).
*/

filter_A_List(NewPT,A_List,[],[]) :- !.
filter_A_List(NewPT,A_List,[[ID,ST]|Rest],[[ID,ST]|Rest2]) :-
    (f_mem(ID,NewPT);
     (delete(A_List,[ID,_],A_List_temp), f_mem(ID,A_List_temp))), !,
    filter_A_List(NewPT,A_List,Rest,Rest2).
filter_A_List(NewPT,A_List,[_|Rest],Rest2) :-
    filter_A_List(NewPT,A_List,Rest,Rest2).

/* F_MEM checks whether a certain symbol appears in an arbitrary Prolog term.*/
/* Remark: this predicate is also used in several places in 'HAIKU_CA.PL'. */
f_mem(_,S) :- var(S), !, fail.
f_mem(_,[]) :- !, fail.
f_mem(F,S) :- atomic(S), !, F == S.
f_mem(F,S) :- functor(S,F,_), !.
f_mem(F,[S_H|S_T]) :- f_mem(F,S_H); f_mem(F,S_T).
f_mem(F,S) :- functor(S,G,_), !, G \== '.', S =.. X, f_mem(F,X).

/*****
/* FAS: (used by top_st and several others) */
/*
/* "Find all symbols" is a predicate which finds (and collects in a list) all
/* symbols in an arbitrary Prolog structure T that begin with the prefix F. */

```

```

/*
/* Example: fas(a,w(1,33,a45.ert,a9i,k(1(p,a,z))),L) returns
/*      L = [a45,a9i,a]

fas(_,[],[])      :- !.
fas(_,T,[])      :- var(T), !.
fas(F,T,[T])     :- atomic(T), prefix_subchar(F,T), !.
fas(_F,T,[])     :- atomic(T), !.
fas(F,[TH|TT],L) :- fas(F,TH,T1), fas(F,TT,T2), append(T1,T2,L).
fas(F,T,L)       :- functor(T,G,_), !, G \== '.', T =.. TL, fas(F,TL,L).

prefix_subchar(F,T) :-
    name(F,A), length(A,L),
    name(T,B), atom_chars(T0,B), subchars(T0,A,0,L).

/*****
/* PRO_EXPLICITIZER (used by haiku_sa_0)
/*
/* This processor is used right after parsing and just before CA. Its role is:
/* to replace, whenever possible, occurrences of pronouns by explicit enti-
/* ties in order to avoid indefinite elements of knowledge during the KA pro-
/* cess. For instance, if an input contains an 'it', the "explicitizer" will
/* assist the user in identifying its referent which may be located in the
/* same input or, in the worst case considered here, in the previous input.
/*
/* The input DOES NOT contain a single pronoun ...
pro_explicitizer(In_PT,In_PT) :-
    fas('pers_pron',In_PT,L1), L1 == [], fas('poss_pron',In_PT,L2), L2 == [],
    fas('pron',In_PT,L3), L3 == [], fas('reflex_pron',In_PT,L4), L4 == [], !.

/* Otherwise, it does ... and explicitization will have to take place.
pro_explicitizer(In_PT,Out_PT) :-
    var(Out_PT), % filtering out a call from eliminate_ref_entities below.
    nlp, writep('Proceed with "pronoun explicitization" [n/Y]? '),
    read_answer(PRO),
    ((PRO == n, Out_PT = In_PT) ;
     (PRO \== n,
      collect_In_PT_Entities(In_PT.Entities_L,Pronouns_L),
      writep('ANAPHORA "EXPLICITIZATION" PRE-PROCESSOR ...'), nlp, nlp,
      writep('ENTITIES IN CURRENT PARSE TREE (OR CLAUSE): '), nlp,
      writep(Entities_L), nlp,
      writep('PRONOUNS IN CURRENT PARSE TREE (OR CLAUSE): '), nlp,
      writep(Pronouns_L), nlp,
      explicitization(In_PT,Entities_L,Pronouns_L,Out_PT,0))).

/* *****
/* Entities: looking for the 'entity' functor in the parse tree.
collect_In_PT_Entities(PT,L1,L2) :-
    ste(entity,PT,L0), eliminate_ref_entities(L0,L1,L2).

eliminate_ref_entities([],[],[])      :- !.
eliminate_ref_entities([emb(L)|T],T2,T3) :-
    !, append(L,T,T0), eliminate_ref_entities(T0,T2,T3).
eliminate_ref_entities([H|T],[H|T2],T3) :-
    H =.. [entity|R], R = [F|_], pro_explicitizer(F,F), !,
    eliminate_ref_entities(T,T2,T3).
eliminate_ref_entities([H|T],T2,[H|T3]) :- eliminate_ref_entities(T,T2,T3).

/*****
/* EXPLICITIZATION (used by pro_explicitizer)
/*
/*
/* High-level of explicitization itself. When the referent of a pronoun is

```

```

/* found, we replace its occurrence in the parse tree with the referent's */
/* sub-tree. Plus, we change the 'entity' functor name to a "special" name in */
/* order to indicate that this entity was found via pronoun explicitization. */
/* Here are the special functor names: entity_pro, entity_pro_generic, */
/* entity_pro_empty, entity_pro_new_NP, and entity_pro_unknown. */
/* */
/* The fifth parameter (F) indicates whether we are in a recursive call in */
/* which we are examining the previous parse tree. In that situation, F is */
/* set to 1. Notice that there is only one previous parse tree (no second */
/* previous parse tree for instance) and that this previous tree corresponds */
/* in fact to the most recent (successful) preceding parse tree. */

explicitization(Out_PT,_,[],Out_PT,_) :- !.

explicitization(In_PT,Entities_L,[P_H|P_T],Out_PT,F) :-
    F == 0,
    (((P_H = entity(ref(pers_pron(X)),_,_) ; P_H = entity(ref(pers_pron(X)),_)),
     (X == 'i' ; X == 'you' ; X == 'we')) ;
     (P_H = entity(ref(pron(one)),_,_) ; P_H = entity(ref(pron(one)),_))),
    nlp, writep(' Trying to find a referent for: '), writep(P_H),
    nlp, writep(' > FOR THIS PRONOUN, SUGGESTING "generic", OK [n/Y]? '),
    read_answer(GEN),
    ((GEN == n, fail) ;
     (GEN \== n,
      get_final_referent(2,P_H,REFERENT,NEW_FUNCT),
      nlp, writep(' >> anaphora resolved: new functor is '),
      writep(NEW_FUNCT), nlp,
      change_functor(REFERENT,_,NEW_REF,NEW_FUNCT,_),
      subst(1,P_H,NEW_REF,In_PT,Out_PT_temp),
      explicitization(Out_PT_temp,Entities_L,P_T,Out_PT,F))).

explicitization(In_PT,Entities_L,[P_H|P_T],Out_PT,F) :-
    ((F == 1,
     nlp, writep('Examining the previous parse tree ...'),
     nlp, writep('(previous parse tree = most recent preceding parse tree '),
     writep('or clause)'), nlp) ;
     F == 0),

    eliminate_impossible_referents(P_H,Entities_L,Entities_L_0),
    nlp, writep(' Trying to find a referent for: '), writep(P_H), nlp,
    writep(' List of candidate entities in this input: '), nlp,
    length(Entities_L_0,N),
    display_list_of_CPs(Entities_L_0,1,N), % re-using a predicate defined
    get_one_REF(Entities_L_0,N,Selected_Ent), % in HAIKU_CA.PL: no CPs here!

    ((Selected_Ent == nil, special_anaph_pro(P_H,F,REFERENT,NEW_FUNCT));
     (Selected_Ent \== nil, REFERENT = Selected_Ent, NEW_FUNCT = entity_pro)),

    ((REFERENT == nil, NEW_FUNCT == nil,
     previous_PT(PPT), collect_In_PT_Entities(PPT,Entities_L_PPT,_),
     explicitization(In_PT,Entities_L_PPT,[P_H],Out_PT_temp,1)) ;

     (REFERENT == nil, NEW_FUNCT == nil,
      \+ previous_PT(_),
      nlp, writep(' !? No previous parse tree ?!'), nlp,
      special_anaph_pro(P_H,1,REFERENT2,NEW_FUNCT2),
      nlp, writep(' >> anaphora resolved: new functor is '),
      writep(NEW_FUNCT2), nlp,
      change_functor(REFERENT2,_,NEW_REF,NEW_FUNCT2,_),
      subst(1,P_H,NEW_REF,In_PT,Out_PT_temp)) ;

     (REFERENT \== nil, NEW_FUNCT \== nil,
      nlp, writep(' >> anaphora resolved: new functor is '),
      writep(NEW_FUNCT), nlp,

```

```

change_function(REFERENT,_,NEW_REF,NEW_FUNCT,_),
subst(1,P_H,NEW_REF,In_PT,Out_PT_temp)),

explicitization(Out_PT_temp,Entities_L,P_T,Out_PT,F).

/* *****
/* IMPOSSIBLE: not very discriminating for now. No semantic restrictions for
/* instance. Only based on number agreement. Simplistic but cheap ...
eliminate_impossible_referents(_,[],[]) :- !.
eliminate_impossible_referents(P_H,[E_H|E_T],[E_H|E_T2]) :-
ste(number,P_H,[number(NUM1)]), ste(number,E_H,[number(NUM2)]),
(NUM1 == NUM2 ; var(NUM1) ; var(NUM2) ;
(NUM1 == sg, NUM2 == sg3) ; (NUM1 == sg3, NUM2 == sg)) !,
eliminate_impossible_referents(P_H,E_T,E_T2).
eliminate_impossible_referents(P_H,[_|E_T],E_T2) :-
eliminate_impossible_referents(P_H,E_T,E_T2).

get_one_REF([],_,nil) :- !.
get_one_REF(REFS,Upper_bound,REF) :-
nlp,
writep(' if appropriate, please select one of these referents:'), nlp,
writep(' enter a number between 1 and '), writep(Upper_bound),
writep(' (or 0 if none is appropriate): '),
read_answer(REFnum), verify_REFnum(REFS,Upper_bound,REFnum,REF).

verify_REFnum(_,_,0,nil) :- !.
verify_REFnum(L,Up,REFnum,REF_ok) :-
number(REFnum), REFnum <= Up, nth1(REFnum,L,REF_ok), !.
verify_REFnum(L,Up,_,REF_ok) :- get_one_REF(L,Up,REF_ok).

/* *****
/* SPECIAL_ANAPH_PRO (used by explicitization)
/*
/* When the referent of a pronoun is not one of those suggested initially,
/* see 'explicitization' above, we must resort to one of the following cases.
/* About the 2nd parameter (F): see details in 'explicitization' above.
/*

special_anaph_pro(P_H,F,REFERENT,NEW_FUNCT) :-
nlp, writep(' Looking for the referent of: '), writep(P_H), nlp,
writep(' Please, select one of these possibilities:'), nlp,
((F == 0, writep(' (0) check entities in the previous parse tree:'),
nlp) ;
F \== 0),
writep(' (1) empty subject (e.g. "IT is warm today.");'), nlp,
writep(' (2) generic (e.g. "ONE would think that ...");'), nlp,
writep(' (3) associate the referent with a new NP:'), nlp,
writep(' (4) unknown or unable to determine referent:'), nlp,
repeat,
writep(' Enter your choice [0,1,2,3,4]: '),
((F == 0, writep('0/1/')) ; (F \== 0, writep('1/'))),
writep('2/3/4: '), read_answer(C), C \== '', is_digit(_,C),
((F == 0, C >= 0, C <= 4) ; (F \== 0, C >= 1, C <= 4)), !,
get_final_referent(C,P_H,REFERENT,NEW_FUNCT).

/* *****
/* GET_FINAL_REFERENT (used by special_anaph_pro)
/*

get_final_referent(0,_,nil,nil).

get_final_referent(1,REF,REF,entity_pro_empty).

get_final_referent(2,REF,REF,entity_pro_generic).

```

```
get_final_referent(3,_,REF,entity_pro_new_NP) :-
    writep('      > enter a simple NP (e.g. "the data files."): '),
    get_NEXT_UNIT(Tokens,Tokens_Typogrph_Info,1), % call scanner!
    writep2(Tokens), nlp2, nlp,
    set_chrono,
    with_timing('lexical analysis',
        lexicalyser(Tokens_Typogrph_Info,Sent,_Err_msg)),
    with_timing('NP_parsing',
        np_equiv(_,1,NP0,_,Sent,[dct(punctuation:'.'/end_of_sentence,[])])),
    !, nlp, zap_nils(NP0,NP), writep(NP), nlp, ste(entity,NP,[REF]).

get_final_referent(3,_,REF,entity_pro_new_NP) :-
    writep('      !? INVALID NP ... please, resubmit a legal NP ?!'), nlp,
    get_final_referent(3,_,REF,entity_pro_new_NP).

get_final_referent(4,REF,REF,entity_pro_unknown).

/...../
```

F.2: The Individual-clause CA Processor

```

/*.....
/* file: HAIKU_CA.PL
/* version: July 17, 1993
/* author: Sylvain Delisle
/*.....

:- assert(haiku_version(17,7,93)).

/*.....
/* CLAUSAL_CASE_ANALYSIS (used by haiku_ca in HAIKU_SA)
/*
/* Top-level of CA: for every clause, find the subj, the main verb, and the
/* complement. Then, perform single clause CA, i.e. identify Cases for each.
/*

clausal_Case_Analysis(clauses(LIST),Out) :-
    nlp, writep('HAIKU: Case Analysis of current input's clauses ...'), nlp,
    loop_clausal_Case_Analysis(LIST,Out,Abort),
    (((Out == [] ; Abort == 1),
        nlp, writep('> Aborting Case Analysis: END OF ANALYSIS.'), nlp) ;
        Out \== []).
clausal_Case_Analysis(_,nil) :-
    writep('HAIKU: Case Analysis of current input's clauses ABORTED!'), nlp.

loop_clausal_Case_Analysis([],[],_) :- !.
loop_clausal_Case_Analysis([[Tag,MPT]|R1], [Tag,CS|R2],Abort) :-
    nlp, writep('CURRENT CLAUSE: '), writep(Tag), nlp,
    single_clause_CA(MPT,CS,CP),
    ((CP \== 0, loop_clausal_Case_Analysis(R1,R2,Abort)) ;
        (CP == 0, Abort = 1)).
loop_clausal_Case_Analysis([[Tag,_]|R1],R2,Abort) :-
    nlp, writep('!? Unable to properly analyse this clause: '), writep(Tag),
    writep(' (END OF ANALYSIS).'), nlp,
    loop_clausal_Case_Analysis(R1,R2,Abort).

/* ++++++ */

single_clause_CA(In_MPT,
    case_structure(MV,
        CMP_p,
        CP,
        SUBJ_Filler,
        COMPL_Fillers,
        MODIFS),
    CP) :-

/* Looking for MODIFIERS (subordinate clauses) ...
ext(n,modifs,initial,In_MPT,MPT000,A_List_modifs_0000),
(A_List_modifs_0000 =.. [modifs,[[_|MODIF0000]]] ; MODIF0000 = []).
ext(n,modifs,medial,MPT000,MPT00,A_List_modifs_000),
(A_List_modifs_000 =.. [modifs,[[_|MODIF000]]] ; MODIF000 = []).
ext(n,modifs,final,MPT00,MPT0,A_List_modifs_00),
(A_List_modifs_00 =.. [modifs,[[_|MODIF00]]] ; MODIF00 = []).
append(MODIF0000,MODIF000,MODIFA), append(MODIFA,MODIF00,MODIFB),
((MODIFB \== [], MODIF0 = subordinate_clauses(MODIFB)) ;
    MODIF0 = nil), !.

/* Looking for the COMPLEMENT ...
ext(n,ca_compl,complement,MPT0,MPT1,A_List_ca_compl1),
((A_List_ca_compl1 =.. [ca_compl,[[_|COMPL0]]], STAT0 = false) ;
    (ext(1,ca_compl,s_qualifier,MPT0,_,A_List_ca_compl2),

```

```

A_List_ca_compl2 =.. [ca_compl,[[_[[COMPL0]]]], STAT0 = true) ;
(fun('statement',MPT0,LSAT), LSAT \== [],
  LSAT = [COMPL00], COMPL0 = pseudo_compl(COMPL00), STAT0 = false) ;
(COMPL0 = nil, STAT0 = false)),
COMPL = COMPL0,
(((COMPL =.. [complement,X] ;
  COMPL =.. [s_qualifier,X] ;
  COMPL =.. [pseudo_compl,X]),
  X =.. [_ ,COMPL_Fillers)) ;
(COMPL == nil,
  COMPL_Fillers = nil)), !,

/* Looking for the SUBJECT ... */
member(S,[subj,wh_word,wh_subj,for,deter_pron]), % peculiarities for n.c.
ext(n,ca_subj,S,MPT1,MPT2,A_List_ca_subj),
((A_List_ca_subj == ca_subj([]), S == deter_pron) ;
  A_List_ca_subj \== ca_subj([])),
(A_List_ca_subj =.. [ca_subj,[[_[[SUBJ0]]]],
  ((S \== subj, SUBJ = subj(SUBJ0)) ; (S == subj, SUBJ = SUBJ0))) ;
  SUBJ = nil),
(SUBJ =.. [subj,SUBJ_Filler] ; SUBJ_Filler = nil), !,

/* Looking for the MAIN VERB ... */
ext(1,ca_main_verb,verb,MPT2,_,A_List_ca_main_verb),
((A_List_ca_main_verb =.. [ca_main_verb,[[_[[MV]]]],
  (MV =.. [verb|[double_quotes(MV_root)|_]] ; MV =.. [verb|[MV_root|_]]),
  ext(1,mv_adv,adv,MV,_,A_List_mv_adv),
  ((A_List_mv_adv =.. [mv_adv,[[_[[ADV]]]],
  (A_List_mv_adv =.. [mv_adv,[]], ADV = nil))) ;
  (A_List_ca_main_verb =.. [ca_main_verb,[]],
  MV = nil, MV_root = nil, ADV = nil)),
  ((MV_root == be, STAT0 == false, STAT = true) ;
  STAT = STAT0), !,

MODIFS = [main_verb_modif(ADV),clause_modif(MODIF0)],

writep('CURRENT SUBJ : '), writep(SUBJ), nlp,
writep('CURRENT VERB : '), writep(MV_root), nlp,
writep('CURRENT COMPL : '), writep(COMPL), nlp,
writep('CURRENT MODIFS : '), writep(MODIFS), nlp,

/* We're all set for Case analysis now ... */
((STAT == true,
  writep('> Aborting Case Analysis: main verb is STATIVE.'),
  nlp, CMP = nil, CMP_p = nil, CP = nil) ;
  (MV_root == nil,
  writep('> Aborting Case Analysis: main verb is missing.'),
  nlp, CMP = nil, CMP_p = nil, CP = nil) ;
  (STAT == false, MV_root \== nil,
  (COMPL == nil,
  nlp, writep('> Warning: Case Analysis may not be very useful; '),
  writep('DISCARD this clause [n/Y]? '),
  read_answer(AB),
  ((AB \== n, CMP = nil, CMP_p = nil, CP = nil,
  writep('> Case Analysis of this clause ABORTED: '),
  writep('no updating of the KB.'), nlp) ;
  (AB == n,
  determine_CMP(SUBJ,COMPL,MODIFS,CMP),
  ((CMP \== nil, CMP \== nil_abort,
  'CA_algo'(MV_root,SUBJ,COMPL,CMP,CMP_p,CP)) ;
  (CMP == nil_abort, CP = 0) ;
  (CMP == nil, CMP_p = nil, CP = nil)))))) ;
  (COMPL \== nil,
  determine_CMP(SUBJ,COMPL,MODIFS,CMP),

```

```

((CMP \== nil, CMP \== nil_abort,
  'CA_algo'(MV_root,SUBJ,COMPL,CMP,CMP_p,CP)) ;
(CMP == nil_abort, CP = 0) ;
(CMP == nil, CMP_p = nil, CP = nil))))).

/*****
/* DETERMINE_CMP (used by single_clause_CA)
/*
/* We here find the current clause's Case-Marker Pattern (CMP), based solely
/* on the associated (sub-) parse tree. The positional markers are 'psubj',
/* 'pobj', and 'piobj'. All others are introduced by prepositions & adverbs.
*****/

determine_CMP(SUBJ,COMPL,MODIFS,Approved_CMP) :-
  determine_subj_CMP(SUBJ,SUBJ_CM),
  determine_compl_CMP(COMPL,COMPL_CM),
  determine_modif_CMP(MODIFS,MODIFS_CM),
  join_with_dashes(SUBJ_CM,COMPL_CM,CMP0),
  join_with_dashes(CMP0,MODIFS_CM,CMP),
  nlp, writep('CMP found by HAIKU: '), writep(CMP), nlp,
  writep(' > Do you wish to overrule this CMP?'), nlp,
  writep('      (enter d to discard this clause, or)'), nlp,
  writep('      a to discard the entire input [y/N/d/a]? '),
  read_answer(Redef), verify_suggested_CMP(Redef,CMP,Approved_CMP).

determine_subj_CMP(nil,nil) :- !.
determine_subj_CMP(subj(_),psubj).

determine_compl_CMP(nil,nil) :- !.
determine_compl_CMP(complement(COMPL),COMPL_CM) :-
  (f_mem(svoo_svoc_svoa,COMPL) ; f_mem(svoo,COMPL)),
  f_mem(iobj,COMPL), !,
  find_PP_compl(COMPL,_,PP),
  join_with_dashes('pobj-piobj',PP,COMPL_CM).
determine_compl_CMP(complement(COMPL),COMPL_CM) :-
  (f_mem(svoo_svoc_svoa,COMPL) ; f_mem(svoo,COMPL)),
  find_PP_compl(COMPL,COMPL_rest,PP),
  fas(entity,COMPL_rest,List_of_entities), length(List_of_entities,2), !,
  join_with_dashes('pobj-piobj',PP,COMPL_CM).
determine_compl_CMP(complement(COMPL0),COMPL_CM) :-
  ext(n,temp1,conj_pps,COMPL0,COMPL1,_),
  ext(n,temp2,pp,COMPL1,COMPL2,_),
  ((member(E,[entity,
    entity_pro,
    entity_pro_generic,
    entity_pro_empty,
    entity_pro_new_NP,
    entity_pro_unknown]), f_mem(E,COMPL2)) ;
  f_mem(nominal_clause,COMPL2)), !,
  find_PP_compl(COMPL0,_,PP),
  join_with_dashes(pobj,PP,COMPL_CM).
determine_compl_CMP(complement(COMPL),COMPL_CM) :-
  fas('statement',COMPL,LSAT), LSAT == [], !,
  find_PP_compl(COMPL,_,PP),
  join_with_dashes(nil,PP,COMPL_CM).
determine_compl_CMP(_,pobj) :- !.

determine_modif_CMP([main_verb_modif(nil),clause_modif(nil)],nil) :- !.
determine_modif_CMP(_,adv).

/*****
/* FIND_PP_COMPL (used by determine_compl_CMP)
*****/

```



```

/* We must discard PPs attached to nouns as post-modifiers, as well as PPs
/* hidden in a passive 'iobj' (e.g. "The book was given TO BETH by Bob").
find_PP_compl(COMPL0,COMPL_r,COMPL_CM) :-
    ext(n,temp1,np_postmodifiers,COMPL0,COMPL1,_),
    ext(n,temp2,iobj,COMPL1,COMPL2,_),
    find_MV_PP(COMPL2,COMPL_r,COMPL_CM).

/* PPs: zero, one, or a conjunction of several.
find_MV_PP(COMPL,COMPL_r,COMPL_CM) :- \+ f_mem(pp,COMPL), \+ f_mem(pp_adv,COMPL),
!.
find_MV_PP(COMPL,COMPL_r,COMPL_CM) :-
    ext(n,temp0,'pp',COMPL,COMPL_0,temp0(PP_A_list_0)),
    ext(n,temp1,'pp_adv',COMPL_0,COMPL_r,temp1(PP_A_list_1)),
    append(PP_A_list_0,PP_A_list_1,PP_A_list_00), !,
    PP_A_list = temp(PP_A_list_00),
    ((PP_A_list =.. [_,[[_PP]]], collect_all_preps(PP,nil,COMPL_CM)) ;
    (PP_A_list =.. [_,PPs], collect_all_preps(PPs,nil,COMPL_CM))).

/* Note: for now, duplicate CMs are not allowed BUT this might not be jus-
/* tified as they may mark different Cases in the same sentence.
collect_all_preps([],COMPL_CM,COMPL_CM) :- !.
collect_all_preps(X,CM_In,CM_Out) :-
    (X = pp(conj_preps(_,[PREP|_]),_) ; X = pp(conj_preps(_,[PREP|_]),_,_) ;
    X = pp(PREP,_) ; X = pp(PREP,_,_) ;
    X = pp([_,PREP],_) ; X = pp([_,PREP],_,_) ;
    X = pp_adv(PREP,_) ; X = pp_adv(PREP,_,_)), !,
    join_with_dashes(CM_In,PREP,CM_Out).
collect_all_preps([[_X|T],CM_In,CM_Out) :-
    (X = pp(conj_preps(_,[PREP|_]),_) ; X = pp(conj_preps(_,[PREP|_]),_,_) ;
    X = pp(PREP,_) ; X = pp(PREP,_,_) ;
    X = pp([_,PREP],_) ; X = pp([_,PREP],_,_) ;
    X = pp_adv(PREP,_) ; X = pp_adv(PREP,_,_)),
    atom(CM_In), atom(PREP), name(CM_In,CM_In_ascii), name(PREP,PREP_ascii),
    ((append(_ ,PREP_ascii,CM_In_ascii), CM_temp = CM_In) ; % No duplicate CMs.
    join_with_dashes(CM_In,PREP,CM_temp)),
    collect_all_preps(T,CM_temp,CM_Out).

/*****
/* JOIN_WITH_DASHES (used by determine_CMP, determine_compl_CMP, &
/* collect_all_preps)
/*
/*
/* Joining individual Case Markers with dashes: psubj & pobj => psubj-pobj.
join_with_dashes(nil,Y,Y) :- !.
join_with_dashes(X,nil,X) :- !.
join_with_dashes(X,Y,Z) :-
    atom(X), atom(Y), name(X,X_ascii), name(Y,Y_ascii),
    append(X_ascii,[45],X0), append(X0,Y_ascii,Z0), name(Z,Z0). % 45 = '-'

/*****
/* VERIFY_SUGGESTED_CMP (by determine_CMP)
/*
/*
/* If the user wishes to change the CMP suggested by HAIKU, we must ensure
/* that the new one is valid ...
verify_suggested_CMP(a,_,nil_abort) :-
    !, nlp, writep('> Case Analysis of this entire input ABORTED: '),
    writep('no further updating of the KB. '), nlp.
verify_suggested_CMP(d,_,nil) :-
    !, writep('> Case Analysis of this clause ABORTED: '),
    writep('no further updating of the KB. '), nlp.

```

```

verify_suggested_CMP(y,CMP0,Approved_CMP)      :-
    writep(' > Please, enter the new CMP (e.g. psubj-pobj-with): '),
    read_answer(NEW_CMP), from_dashes_to_list(NEW_CMP,NEW_CMP_L),
    NEW_CMP_L \== [],
    validate_NEW_CMP_L(NEW_CMP_L,NEW_CMP,Approved_CMP0),
    writep(' > Replacing '''), writep(CMP0),
    writep(''' with '''), writep(Approved_CMP0), writep('' : OK [n-Y]? '),
    read_answer(O),
    ((O == n, verify_suggested_CMP(y,CMP0,Approved_CMP)) ;
     (O \== n, Approved_CMP = Approved_CMP0)).
verify_suggested_CMP(y,CMP0,Approved_CMP)      :-
    !, writep(' > INVALID CMP!'), nlp,
    verify_suggested_CMP(y,CMP0,Approved_CMP).
verify_suggested_CMP(_,Approved_CMP,Approved_CMP) :- !.

validate_NEW_CMP_L([],Approved_CMP,Approved_CMP) :- !.
validate_NEW_CMP_L([H|T],NEW_CMP,Approved_CMP) :-
    (memberchk(H,[psubj,pobj,piobj,adv]) ; dict(H,prep._,_)), !,
    validate_NEW_CMP_L(T,NEW_CMP,Approved_CMP).

/*****
/* CA_ALGO (used by single_clause_CA)
/*
/*
/* Following the CA algorithm, as described in the "IEEE paper", pp.14-15.
/* Remark: predicate and variable names are particularly self-explanatory.
*****/

'CA_algo'(Input_Verb,SUBJ,COMPL,Input_CMP_0,Input_CMP,Selected_CP) :-

/* Situations S1 to S3: the input verb is in mDict (known verb V).
mDict(Input_Verb,List_of_CMps,List_of_Cases_for_every_CM),

/* S1: Input_CMP is among CMps in mDict entry for Input_Verb.
/* {known CMP already associated with V; known CP or new CP with known CMP}
((from_dashes_to_list(Input_CMP_0,Input_CMP_L),
 permutation(Input_CMP_L,Input_CMP_00), % the KB may contain a permutation.
 from_list_to_dashes(Input_CMP_00,Input_CMP),
 memberchk(Input_CMP,List_of_CMps),
 cmpDict(Input_CMP,List_of_candidate_CPs_with_example_sentences), !,

 selection_dialogue('S1', Input_Verb, SUBJ, COMPL, Input_CMP,
                    List_of_Cases_for_every_CM,
                    List_of_candidate_CPs_with_example_sentences,
                    Selected_CP,
                    List_of_New_CM_Case_Pairs_Intro_by_Known_CM), !,
 update_HAIKU_Dicts('S1', Input_Verb, Input_CMP,
                    Selected_CP,
                    List_of_New_CM_Case_Pairs_Intro_by_Known_CM)) ;

/* S2: Input_CMP is in cmpDict.
/* {known CMP not yet associated with V; new or known CP}
(from_dashes_to_list(Input_CMP_0,Input_CMP_L),
 permutation(Input_CMP_L,Input_CMP_00), % the KB may contain a permutation.
 from_list_to_dashes(Input_CMP_00,Input_CMP),
 cmpDict(Input_CMP,List_of_candidate_CPs_with_example_sentences), !,

 selection_dialogue('S2', Input_Verb, SUBJ, COMPL, Input_CMP,
                    List_of_Cases_for_every_CM,
                    List_of_candidate_CPs_with_example_sentences,
                    Selected_CP,
                    List_of_New_CM_Case_Pairs_Intro_by_Known_CM), !,
 update_HAIKU_Dicts('S2', Input_Verb, Input_CMP,
                    Selected_CP,
                    List_of_New_CM_Case_Pairs_Intro_by_Known_CM)) ;

```

```

/* S3: (entirely new CMP; new or known CP) */
(Input_CMP = Input_CMP_0,
 find_close_CMPS(Input_CMP,Close_CMPS,Closest_CMP), !,

 selection_dialogue('S3', Input_Verb, SUBJ, COMPL,
 [Input_CMP,Close_CMPS,Closest_CMP],
 List_of_Cases_for_every_CM,
 [], % empty List_of_candidate_CPs_with_example_sent.
 Selected_CP,
 List_of_New_CM_Case_Pairs_Intro_by_Known_CM), !,
 update_HAIKU_Dicts('S3', Input_Verb, Input_CMP,
 Selected_CP,
 List_of_New_CM_Case_Pairs_Intro_by_Known_CM))).

'CA_algo'(Input_Verb,SUBJ,COMPL,Input_CMP_0,Input_CMP,Selected_CP) :-

/* Situations S4 to S5: the input verb is NOT in mDict (new verb). */

/* S4: Input_CMP is in cmpDict (known CMP; new CP (relative to this verb)). */
(from_dashes_to_list(Input_CMP_0,Input_CMP_L),
 permutation(Input_CMP_L,Input_CMP_00), % the KB may contain a permutation.
 from_list_to_dashes(Input_CMP_00,Input_CMP),
 cmpDict(Input_CMP,List_of_candidate_CPs_with_example_sentences), !,

 selection_dialogue('S4', Input_Verb, SUBJ, COMPL, Input_CMP,
 [], % empty List_of_Cases_for_every_CM
 List_of_candidate_CPs_with_example_sentences,
 Selected_CP,
 List_of_New_CM_Case_Pairs_Intro_by_Known_CM), !,
 update_HAIKU_Dicts('S4', Input_Verb, Input_CMP,
 Selected_CP,
 List_of_New_CM_Case_Pairs_Intro_by_Known_CM)) ;

/* S5: (entirely new CMP; new CP) */
(Input_CMP = Input_CMP_0,
 find_close_CMPS(Input_CMP,Close_CMPS,Closest_CMP), !,

 selection_dialogue('S5', Input_Verb, SUBJ, COMPL,
 [Input_CMP,Close_CMPS,Closest_CMP],
 [], % empty List_of_Cases_for_every_CM
 [], % empty List_of_candidate_CPs_with_example_sent.
 Selected_CP,
 List_of_New_CM_Case_Pairs_Intro_by_Known_CM), !,
 update_HAIKU_Dicts('S5', Input_Verb, Input_CMP,
 Selected_CP,
 List_of_New_CM_Case_Pairs_Intro_by_Known_CM))).

/*****
/* SELECTION_DIALOGUE (used by 'CA_algo') */
/*
/* The ultimate goal of this predicate is to lead the user to selecting the
/* most appropriate CP that can be associated with the current clause, con-
/* sidering its CMP. If this CP cannot be gleaned from HAIKU's past experi-
/* ence, the user will be invited to submit a new one, guided by HAIKU. */

selection_dialogue(SITU, Input_Verb, _SUBJ, _COMPL, Input_CMP, %+
 List_of_Cases_for_every_CM, %mDict
 List_of_candidate_CPs_with_example_sentences, %+cmpDict
 Selected_CP, %-
 List_of_New_CM_Case_Pairs_Intro_by_Known_CM) %-
:-
 nlp, writep_situation(SITU), nlp,

```

```

(memberchk(SITU,['S1','S2','S4']),
  Input_CMP0 = Input_CMP,
  nlp,
  writep(' Candidate CPs with example sentences:'), nlp,
  show_CPs_and_examples(List_of_candidate_CPs_with_example_sentences,
    List_of_candidate_CPs));

(memberchk(SITU,['S3','S5']),
  Input_CMP = [Input_CMP0,Close_CMps,CLOSEST_MATCH],
  writep(' entirely new CMP; no obvious candidate CPs to propose'), nlp,
  writep(' list of close CMps proposed by the "closeness metric": '),
  writep(Close_CMps), nlp,
  ((CLOSEST_MATCH = match(_,CLOSEST_CMP_L),
    from_list_to_dashes(CLOSEST_CMP_L,CLOSEST_CMP),
    cmpDict(CLOSEST_CMP,List_of_candidate_CPs_with_example_sentences0),
    nlp,
    writep(' Candidate CPs with example sentences for the closest CMP'),
    writep(' '). writep(CLOSEST_CMP), writep(')'), nlp,
    show_CPs_and_examples(List_of_candidate_CPs_with_example_sentences0,
      List_of_candidate_CPs));
  (CLOSEST_MATCH == [], % zero "close" CMps!
    List_of_candidate_CPs = [])), !,

nlp,
writep(' current clause''s main verb (infinitive): '),
writep(Input_Verb), nlp,
writep(' current clause''s CMP (or a permutation): '),
writep(Input_CMP0), nlp,

suggest_most_probable_CPs(Input_CMP0,
  List_of_Cases_for_every_CM,
  List_of_candidate_CPs,
  Suggested_CPs), !,

see_user_and_select_CP(SITU, Input_Verb, Input_CMP0, Suggested_CPs,
  List_of_Cases_for_every_CM,
  Selected_CP,
  List_of_New_CM_Case_Pairs_Intro_by_Known_CM).

/* ***** */
/* Tracing the CA algorithm: whatever the input, one of these 5 situations. */

writep_situation('S1') :-
  writep('S1: known CMP already associated with known verb; '),
  writep('CP is known or new.').
writep_situation('S2') :-
  writep('S2: known CMP not yet associated with known verb; '),
  writep('CP is known or new.').
writep_situation('S3') :-
  writep('S3: entirely new CMP with known verb; CP is known or new.').
writep_situation('S4') :-
  writep('S4: new verb with known CMP; new CP (relative to this verb)').
writep_situation('S5') :-
  writep('S5: new verb with (entirely) new CMP; new CP.').

/* ***** */
/* Used by 'selection_dialogue': listing CPs and example sentences in the */
/* list List_of_candidate_CPs_with_example_sentences (1st parameter). By the */
/* same token, we collect all the CPs in a separate list (2nd parameter). */

show_CPs_and_examples([],[]) :- !.
show_CPs_and_examples([[CP,Example_Sentence]|T],[CP|R]) :-
  writep(' CP: '), writep(CP), writep(' '),

```

```

        writop('Ex: '), writop(Example_Sentence), nlp, show_CPs_and_examples(T,R).

/* ***** */
/* Converting lists of patterns into atoms: [psubj,pobj] => psubj-pobj. This */
/* is needed when we must resort to the closeness metric which manipulates */
/* lists of CMPs. For details on the metric, see 'HAIKU_FC.PL'. */

from_list_to_dashes(In,Out) :-
    from_list_to_dashes_2(In,Out0), name(Out,Out0).

from_list_to_dashes_2([H],H_ascii) :- atom(H), name(H,H_ascii), !.
from_list_to_dashes_2([H|T1],T2) :-
    atom(H), name(H,H_ascii), append(H_ascii,[45],L0),
    from_list_to_dashes_2(T1,T3), append(L0,T3,T2).

/* ***** */
/* SUGGEST_MOST_PROBABLE_CPs (used by selection_dialogue) */
/* */
/* The "most probable CP" is the candidate CP with the highest 'rating' in */
/* List_of_candidate_CPs. Rating is computed below in 'compare_CMP_with_CP'. */

suggest_most_probable_CPs(Input_CMP,                                     %+
                          List_of_Cases_for_every_CM,                 %+mDict
                          List_of_candidate_CPs,                       %+cmpDict
                          Suggested_CPs)                              %-
:-
    from_dashes_to_list(Input_CMP,Input_CMP_L),

    ((List_of_candidate_CPs == [], Rated_CPs = []) ;
     (List_of_candidate_CPs \== [],
      compare_InputCMP_with_candidate_CPs(Input_CMP_L,
                                           List_of_candidate_CPs,
                                           List_of_Cases_for_every_CM,
                                           Rated_CPs))), !,

    select_most_probable_CPs(Rated_CPs,Suggested_CPs).

/* ***** */
/* COMPARE_INPUTCMP_WITH_CANDIDATE_CMPS (used by suggest_most_probable_CPs) */
/* */
/* This predicate recursively controls the rating computation. */

compare_InputCMP_with_candidate_CPs(_Input_CMP_L,
                                     [],
                                     _List_of_Cases_for_every_CM,
                                     [])
:- !.

compare_InputCMP_with_candidate_CPs(Input_CMP_L,
                                     [CP_H|CP_T],
                                     List_of_Cases_for_every_CM,
                                     [[Rating,CP_H]|CP_T_rated])
:-
    from_dashes_to_list(CP_H,CP_H_L),
    length(Input_CMP_L,N), length(CP_H_L,N), !, % Lengths must be equal.
    compare_CMP_with_CP(Input_CMP_L,
                        CP_H_L,
                        List_of_Cases_for_every_CM,
                        0, N, Rating),
    compare_InputCMP_with_candidate_CPs(Input_CMP_L,
                                         CP_T,
                                         List_of_Cases_for_every_CM,
                                         CP_T_rated).

```

```

compare_InputCMP_with_candidate_CPs(Input_CMP_L,
                                     [_|CP_T],
                                     List_of_Cases_for_every_CM,
                                     CP_T_rated)
:-
    compare_InputCMP_with_candidate_CPs(Input_CMP_L,
                                         CP_T,
                                         List_of_Cases_for_every_CM,
                                         CP_T_rated).

/*****
/* COMPARE_CMP_WITH_CP (used by compare_InputCMP_with_candidate_CPs) */
/*
/* We want to suggest CPs that have been associated with identical CMPs or
/* the closest possible CMPs, based on HAIKU's past experience. The notion
/* of 'rating' we're using here is defined as follows: the number of indivi-
/* dual Cases (in the candidate CP) that have already been associated with
/* the current input's individual CMs divided by the length of the input CMP.*/

compare_CMP_with_CP([],[],_,N1,N,Rating) :- !, Rating is N1 / N.

compare_CMP_with_CP([CM_H|CM_T],
                    [C_H|C_T],
                    List_of_Cases_for_every_CM,
                    N0, N, Rating) :-
    look_up_List_of_known_Cases(CM_H,C_H,List_of_Cases_for_every_CM,N0,N1),
    compare_CMP_with_CP(CM_T,C_T,List_of_Cases_for_every_CM,N1,N,Rating).

look_up_List_of_known_Cases(CM_H,C_H,L,N0,N1) :-
    memberchk([CM_H,L0],L), memberchk(C_H,L0), !, N1 is N0 + 1. % known Case
look_up_List_of_known_Cases(_,_,_,N0,N0).

/*****
/* SELECT_MOST_PROBABLE_CPS (used by suggest_most_probable_CPs) */
/*
/* Once the ratings have been computed for every candidate CP, we must find
/* the one with the highest rating, i.e. the "most probable" (or likely). */

select_most_probable_CPs([],[]) :- !.
select_most_probable_CPs([[Rating,CP_H_L]|CP_T],Suggested_CPs) :-
    find_max_emb_list(CP_T,[Rating,CP_H_L],Max),
    find_all_CPs_with_max_prob([Rating,CP_H_L]|CP_T,Max,Suggested_CPs).

find_max_emb_list([[_|T],[Rm,_],Max) :-
    R <= Rm, find_max_emb_list(T,[Rm,_],Max).
find_max_emb_list([[_|T],[Rm,_],Max) :-
    R > Rm, find_max_emb_list(T,[R,_],Max).
find_max_emb_list([],[_],Max).

find_all_CPs_with_max_prob([],_,[]) :- !.
find_all_CPs_with_max_prob([[_|CP_H]|CP_T,Max,[CP_H|T]) :-
    !, find_all_CPs_with_max_prob(CP_T,Max,T).
find_all_CPs_with_max_prob([_|CP_T],Max,T) :-
    find_all_CPs_with_max_prob(CP_T,Max,T).

/*****
/* FROM_DASHES_TO_LIST (used by suggest_most_probable_CPs &
/* compare_InputCMP_with_candidate_CPs)
/*
/* This predicate is the inverse of 'from_list_to_dashes': see above.
*/

```

```

from_dashes_to_list('',[]) :- !.
from_dashes_to_list(Input_CMP,Input_CMP_L) :-
    atom(Input_CMP), name(Input_CMP,In_ascii), memberchk(45,In_ascii),
    dashes_to_list(Input_CMP_L,In_ascii,"").
from_dashes_to_list(Input_CMP,[Input_CMP]).

dashes_to_list([H|T]) -->
    lowercase_CM(H_ascii), (name(H,H_ascii)), "--", dashes_to_list(T).
dashes_to_list([H]) --> lowercase_CM(H_ascii), (name(H,H_ascii)).

lowercase_CM([H|T]) --> [H], (is_lower(H)), (lowercase_CM(T) ; {T = []}).
lowercase_CM([H]) --> [H], (H == 63). % 63='?'

/*****
/* SEE_USER_AND_SELECT_CP (used by selection_dialogue) */
/*
/* Most likely (or probable) CPs are proposed to the user. Ultimately, only
/* one of these, or a new one, must be identified: 'Selected_CP'. */
*****/

see_user_and_select_CP(SITU, _Input_Verb, Input_CMP, Suggested_CPs, %+
    List_of_Cases_for_every_CM, %+
    Selected_CP_ok, %+
    List_of_New_CM_Case_Pairs_Intro_by_Known_CM) %+
:-
writep(' most probable or closest CPs, which are of compatible length: '),
((memberchk(SITU,['S1','S2','S3'])),
 display_list_of_CPs(Suggested_CPs,1,N),
 get_one_CP(Suggested_CPs,N,Selected_CP_0a),
 ((Selected_CP_0a \== nil,
  from_dashes_to_list(Input_CMP,Input_CMP_L),
  length(Input_CMP_L,L1),
  from_dashes_to_list(Selected_CP_0a,Selected_CP_0a_L),
  length(Selected_CP_0a_L,L2),
  check_syntax_of_CP(Input_CMP_L,Selected_CP_0a,L1,L2,Selected_CP_0)) ;
 (Selected_CP_0a == nil, Selected_CP_0 = nil))) ;
(memberchk(SITU,['S4','S5'])),
writep(Suggested_CPs, nlp)), !.

((memberchk(SITU,['S1','S2','S3'])), Selected_CP_0 \== nil, % KNOWN CP
 Selected_CP = Selected_CP_0,
 writep(' replace cmpDict's example with current clause [y/N]? '),
 read_answer(R),
 ((R == y, replace_example_sentence_in_cmpDict(Input_CMP,Selected_CP)) ;
  R \== y)) ;
((memberchk(SITU,['S4','S5']) ; Selected_CP_0 == nil), % NEW CP
 let_user_ident_new_CP(Input_CMP,Selected_CP)), !.

((Selected_CP \== nil,
 from_dashes_to_list(Input_CMP,I_CMP_L), length(I_CMP_L,L11),
 from_dashes_to_list(Selected_CP,S_CP_L), length(S_CP_L,L22),
 ((L11 \== L22,
  nlp,
  writep(' ILLEGAL CP: the CMP and the CP must both be of length '),
  writep(L11), nlp, let_user_ident_new_CP(Input_CMP,Selected_CP_ok)) ;
 (L11 == L22,
  Selected_CP_ok = Selected_CP)),
 ident_new_CM_Case_pairs(Input_CMP,
    List_of_Cases_for_every_CM,
    Selected_CP_ok,
    List_of_New_CM_Case_Pairs_Intro_by_Known_CM)) ;

(Selected_CP == nil,
 Selected_CP_ok = nil,
 List_of_New_CM_Case_Pairs_Intro_by_Known_CM = [])).

```

```

/* *****
/* The user may suggest that HAIKU replaces an example sentence with the
/* current input because it is considered more typical or representative.
/* *****

replace_example_sentence_in_cmpDict(Input_CMP,Selected_CP) :-
    writep(' REPLACING example in cmpDict for '), writep(Input_CMP),
    writep(' ...'), nlp, get_current_clause_tokens(List_of_Tokens),
    retract(cmpDict(Input_CMP,List_of_CPs_and_example_sentences)), !,
    delete(List_of_CPs_and_example_sentences,[Selected_CP,_],NEW_List0),
    append(NEW_List0,[[Selected_CP,List_of_Tokens]],NEW_List),
    assert(cmpDict(Input_CMP,NEW_List)),
    writep(' => '), writep(Input_CMP), writep(' '), writep(NEW_List), nlp.
replace_example_sentence_in_cmpDict(_,_):-
    writep(' !? unsuccessful updating of cmpDict ?!'), nlp.

/* Note: for the time being, we consider the whole input string instead of
/* just the current clause. The main menu's option 'D' may also be activated.*/
get_current_clause_tokens(List_of_Tokens) :-
    sentence_tokens(L), make_atom(L,L_ascii_0), name(L_ascii_0,L_ascii_1),
    subst(n,39,95,L_ascii_1,L_ascii_2), % subst quotes with underscores.
    append([39|L_ascii_2],[39],L0), name(List_of_Tokens,L0).

/* *****
/* Making sure that we're dealing with atoms ... Using a temporary file.
/* *****

make_atom(Term,Atom) :-
    tell('#####.HAIKU.make_atom.dump'), write(Term), nl, told,
    see('#####.HAIKU.make_atom.dump'), read_line(L), seen,
    name(Atom,L).

/*****
/* LET_USER_IDENT_NEW_CP (used by see_user_and_select_CP)
/* *****/

let_user_ident_new_CP(CMP,CP)
:-
    nlp,
    writep(' NEW CP: list of 28 legal Cases in HAIKU'),
    nlp,
    writep(' agt:agent; benf:beneficiary; expr:experiencer; inst:instrument;'),
    nlp,
    writep(' obj:object; recp:recipient; caus:cause; manr:manner; val:value;'),
    nlp,
    writep(' eff:effect; purp:purpose; freq:frequency; tat:time_at;'),
    nlp,
    writep(' tfrm:time_from; tto:time_to; ttru:time_through; dir:direction;'),
    nlp,
    writep(' lat:location_at; lfrm:location_from; lto:location_to; ord:order;'),
    nlp,
    writep(' ltru:location_through; ornt:orientation; cont:content;'),
    nlp,
    writep(' excl:exclusion; acmp:accompaniment; matr:material; meas:measure.'),
    nlp,
    writep(' REM: A CASE MAY BE SUFFIXED WITH A ''?' TO '),
    writep(' DENOTE UNCERTAINTY (e.g. lat?).'),
    nlp,
    from_dashes_to_list(CMP,CMP_L), length(CMP_L,L1), consult_cmDict(CMP_L),
    nlp,
    repeat,
    writep(' please enter the new CP (e.g. agt-obj-tat),'), nlp,
    writep(' or enter ''h'' to see the current input string and CMP,'), nlp,
    writep(' or CR to abort ["new CP"/h/CR]? '),
    read_answer(NewCP),

```



```

(NewCP \== '', NewCP \== h,
  from_dashes_to_list(NewCP,NewCP_L), length(NewCP_L,L2),
  check_syntax_of_CP(CMP_L,NewCP,L1,L2,CP), OK = true) ;
(NewCP == '', CP = nil, OK = true) ;
(NewCP == h,
  sentence_tokens(TOKENS),
  nlp, writep(TOKENS), nlp, writep(CMP), nlp, nlp, OK = false)),
OK == true, !.

consult_cmDict([])      :- !.
consult_cmDict([H|T]) :-
  H \== psubj, H \== pobj, cmDict(H,CCL),
  nlp,
  writep(' here is what the cmDict knows about this Case marker: '),
  upper(H,H_upper),
  writep(H_upper), nlp,
  writep(' {format: Case(Commonness,Example); CO=1:common, CO=2:uncommon}'),
  nlp, list_known_CM_Cases(CCL), consult_cmDict(T).
consult_cmDict([_|T]) :- consult_cmDict(T).

list_known_CM_Cases([])      :- !.
list_known_CM_Cases([H|T]) :-
  writep(' '), writep(H), nlp, list_known_CM_Cases(T).

/* ++++++ */
/* Checking the new CP's syntax: several constraints must be enforced. */

check_syntax_of_CP(CMP_L,CP0,L1,_,CP) :-
  from_dashes_to_list(CP0,CP0_L), \+ is_set(CP0_L), !,
  writep(' ILLEGAL CP! No duplicate Cases allowed: '),
  read_answer(NewCP), from_dashes_to_list(NewCP,NewCP_L),
  length(NewCP_L,L3), check_syntax_of_CP(CMP_L,NewCP,L1,L3,CP).

check_syntax_of_CP(CMP_L,_,L1,L2,CP) :-
  L1 \== L2, !, writep(' ILLEGAL CP! The new CP must contain exactly '),
  writep(L1), writep(' Case(s), i.e. length='), writep(L1), writep(': '),
  read_answer(NewCP), from_dashes_to_list(NewCP,NewCP_L),
  length(NewCP_L,L3), check_syntax_of_CP(CMP_L,NewCP,L1,L3,CP).

check_syntax_of_CP(CMP_L,CP,L,L,CP_ok) :-
  from_dashes_to_list(CP,CP_L), every_CP_is_ok(CP_L), !,
  writep(' CMP & CP will be paired as follows: '),
  display_CMP_CP_pairs(CMP_L,CP_L),
  writep(' ; Correct [n/Y]? '), read_answer(OK),
  ((OK \== n, CP_ok = CP) ;
   (OK == n,
    writep(' CP in wrong order, please re-enter in proper order: '),
    read_answer(NewCP), from_dashes_to_list(NewCP,NewCP_L),
    length(NewCP_L,L3), check_syntax_of_CP(CMP_L,NewCP,L,L3,CP_ok))).

check_syntax_of_CP(CMP_L,_,L1,_,CP) :-
  writep(' ILLEGAL CP! Please use only HAIKU's legal Case labels: '),
  read_answer(NewCP), from_dashes_to_list(NewCP,NewCP_L),
  length(NewCP_L,L3), check_syntax_of_CP(CMP_L,NewCP,L1,L3,CP).

/* LIST OF LEGAL CASES: currently, there are 28 distinct Cases (sem. tags). */
every_CP_is_ok([])      :- !.
every_CP_is_ok([H|T]) :-
  atom(H),
  ((name(H,H_ascii), rev(H_ascii,[63|H0_ascii_r]), % 63='?'
   rev(H0_ascii_r,H0_ascii), name(H_ok,H0_ascii)) ;

```

```

H_ok = H),
memberchk(H_ok, [agt, benef, expr, inst, obj, recip, caus, eff, purp, freq, tat, ttrm,
               tto, ttru, dir, lat, lfrm, lto, ltru, ornt, cont, excl, acmp, minr,
               matr, meas, ord, val]),
every_CP_is_ok(T).

display_CMP_CP_pairs([], []) :- !.
display_CMP_CP_pairs([H1|T1], [H2|T2]) :-
    writep(H1), writep('/'), writep(H2), writep(' '),
    display_CMP_CP_pairs(T1, T2).

/* ***** */
/* Once we have identified the new CP, we must keep track of the new CM-CP */
/* pairs in order to update the mDict appropriately (its 3rd parameter). */

ident_new_CM_Case_pairs(CMP, List_of_Cases_for_every_CM, CP, New_Pairs_L) :-
    from_dashes_to_list(CMP, CMP_L), from_dashes_to_list(CP, CP_L),
    spot_new_pairs(CMP_L, List_of_Cases_for_every_CM, CP_L, New_Pairs_L).

spot_new_pairs([], _, [], []) :- !.
spot_new_pairs([CM_H|CM_T], List_CM_Cases, [C_H|C_T], [[CM_H, C_H]|T]) :-
    look_up_List_of_known_Cases(CM_H, C_H, List_CM_Cases, 1, 1), !,
    spot_new_pairs(CM_T, List_CM_Cases, C_T, T).
spot_new_pairs([_|CM_T], List_CM_Cases, [_|C_T], T) :-
    spot_new_pairs(CM_T, List_CM_Cases, C_T, T).

/* ***** */

display_list_of_CPs([], _, _) :- !, writep(' []'), nlp.
display_list_of_CPs(X, Y, Z) :- display_list_of_PofS(X, Y, Z).

get_one_CP([], _, nil) :- !.
get_one_CP(CPs, Upper_bound, CP_ok) :-
    nlp,
    writep(' if appropriate, please select one of these CPs:'), nlp,
    writep(' enter a number between 1 and '), writep(Upper_bound),
    writep(' (or 0 if none is appropriate): '),
    read_answer(CPnum), verify_CPnum(CPs, Upper_bound, CPnum, CP_ok).

verify_CPnum(_, 0, nil) :- !.
verify_CPnum(L, Up, CPnum, CP_ok) :-
    number(CPnum), CPnum =< Up, nth1(CPnum, L, CP_ok), !.
verify_CPnum(L, Up, _, CP_ok) :- get_one_CP(L, Up, CP_ok).

/* ***** */
/* UPDATE_HAIKU_DICTS (used by 'CA_algo') */
/* */
/* At this point, we have all the required elements of info. about the cur- */
/* rent input: let's update HAIKU's dictionaries accordingly. */

update_HAIKU_Dicts(_, _Input_Verb, _Input_CMP, nil, [])
:-
    !, writep('> Case Analysis of this clause ABORTED: no updating of the KB.'),
    nlp.

update_HAIKU_Dicts(SITU, Input_Verb, Input_CMP, Selected_CP,
                  List_of_New_CM_Case_Pairs_Intro_by_Known_CM)
:-
    update_mDict(SITU, Input_Verb, Input_CMP,
                  List_of_New_CM_Case_Pairs_Intro_by_Known_CM),
    update_cmpDict(SITU, Input_CMP, Selected_CP),

```

```

update_cpDict(Selected_CP,Input_Verb).

/*.....*/
/* UPDATE_mDICT (used by update_HAIKU_Dicts) */
/* */
/* mDICT example: mDict(store,['psubj-pobj'],[[pobj,[obj]],{psubj,[agt]]}). */

update_mDict(SITU,Input_Verb,Input_CMP,
              List_of_New_CM_Case_Pairs_Intro_by_Known_CM)
:-
memberchk(SITU,['S1','S2','S3']),
mDict(Input_Verb,List_of_CMps,List_of_CM_Case_lists),
((SITU \== 'S1', append(List_of_CMps,[Input_CMP],NEW_List1a)) ;
 (SITU == 'S1', NEW_List1a = List_of_CMps)),
update_List_of_CM_Case_lists(List_of_CM_Case_lists,
                              List_of_New_CM_Case_Pairs_Intro_by_Known_CM,
                              NEW_List2a),
sort(NEW_List1a,NEW_List1), sort(NEW_List2a,NEW_List2),
(mDict(Input_Verb,NEW_List1,NEW_List2) ;
 (writep(' UPDATING mDict for '), writep(Input_Verb), writep(' ...'), nlp,
  retractall(mDict(Input_Verb,List_of_CMps,List_of_CM_Case_lists)),
  assert(mDict(Input_Verb,NEW_List1,NEW_List2)),
  writep(' => '), writep(Input_Verb), writep(' '), writep(NEW_List1),

  writep(' '), writep(NEW_List2), nlp)), !.

update_mDict(SITU,Input_Verb,Input_CMP,
              List_of_New_CM_Case_Pairs_Intro_by_Known_CM)
:-
memberchk(SITU,['S4','S5']),
writep(' UPDATING mDict for '), writep(Input_Verb), writep(' ...'), nlp,
create_List_of_CM_Case_lists(List_of_New_CM_Case_Pairs_Intro_by_Known_CM,La),
Input_CMP0a = [Input_CMP],
sort(Input_CMP0a,Input_CMP0), sort(La,L),
assert(mDict(Input_Verb,Input_CMP0,L)),
writep(' => '), writep(Input_Verb), writep(' '), writep(Input_CMP0),
writep(' '), writep(L), nlp, !.

update_mDict(.,.,.) :-
writep(' !? unsuccessful updating of mDict ?!'), nlp.

/* ..... */

update_List_of_CM_Case_lists(L,[],L) :- !.
update_List_of_CM_Case_lists(L1,[[CM0,C0]|T],L2) :-
delete(L1,[CM0,Cs0],L0), append(Cs0,[C0],L00), append(L0,[[CM0,L00]],L22),
update_List_of_CM_Case_lists(L22,T,L2).

create_List_of_CM_Case_lists([],[]) :- !.
create_List_of_CM_Case_lists([[CM_H,C_H]|T1],[[CM_H,[C_H]]|T2]) :-
create_List_of_CM_Case_lists(T1,T2).

/*.....*/
/* UPDATE_cmpDICT (used by update_HAIKU_Dicts) */
/* */
/* cmpDICT example: cmpDict('psubj-adv',[['agt-manr',[he,always,works]]]). */

update_cmpDict(SITU,Input_CMP,Selected_CP) :-
memberchk(SITU,['S1','S2','S4']), % No updating necessary!
cmpDict(Input_CMP,List_of_CPs_and_example_sentences),
memberchk([Selected_CP,_,_],List_of_CPs_and_example_sentences), !.

update_cmpDict(SITU,Input_CMP,Selected_CP) :-

```

```

((memberchk(SITU,['S1','S2','S4']),
  cmpDict(Input_CMP,List_of_CPs_and_example_sentences),
  \+ memberchk([Selected_CP,_],List_of_CPs_and_example_sentences)) ;

(memberchk(SITU,['S3','S5']),
  List_of_CPs_and_example_sentences = {})),

writep('  UPDATING cmpDict for '), writep(Input_CMP), writep(' ...'), nlp,
get_current_clause_tokens(List_of_Tokens),
NEW = [Selected_CP,List_of_Tokens],

((memberchk(SITU,['S1','S2','S4']),
  retractall(cmpDict(Input_CMP,List_of_CPs_and_example_sentences))) ;
 memberchk(SITU,['S3','S5'])),

append(List_of_CPs_and_example_sentences,[NEW],NEW_Lista),
sort(NEW_Lista,NEW_List),
assert(cmpDict(Input_CMP,NEW_List)),
writep(' => '), writep(Input_CMP), writep(' '), writep(NEW_List), nlp, !.

update_cmpDict(_,_,:) :-
  writep(' !? unsuccessful updating of cmpDict ?!'), nlp.

/*****
/* UPDATE_cpDICT (used by update_HAIKU_Dicts) */
/*
/* cpDict example: cpDict('agt-obj-inst-lfrm',[prepare]). */
*****/

update_cpDict(Selected_CP,Input_Verb) :-
  ((cpDict(Selected_CP,List_of_Verbs),
    append(List_of_Verbs,[Input_Verb],NEW_Lista)) ;
   NEW_Lista = [Input_Verb]),
  sort(NEW_Lista,NEW_List),
  (cpDict(Selected_CP,NEW_List) ;
   (writep('  UPDATING cpDict for '), writep(Selected_CP),
    writep(' ...'), nlp,
    retractall(cpDict(Selected_CP,List_of_Verbs)),
    assert(cpDict(Selected_CP,NEW_List)),
    writep(' => '), writep(Selected_CP), writep(' '),
    writep(NEW_List), nlp)).

update_cpDict(_,_,:) :-
  writep(' !? unsuccessful updating of mDict ?!'), nlp.

/*****/

```

Appendix G: Examples of Representative Semantic Analyses

This appendix shows an abridged log of a brief session with HAIKU. We analyze two of the five inputs for which we presented the parse trees produced by DIPETT in Appendix E, plus another one which is parsed as a series of fragments — we will not duplicate the parse trees here. These examples illustrate how KA through TP is actually accomplished in HAIKU.

G.1: Semantic Analysis #1

Unit #1: Refer to lines 304 and 305 in the 1990 General Tax Guide for more information on these non-refundable tax credits .
 Process this unit (Y/n), try another Version, or Exit [Y/n/v/e]?

lexical analysis (cpu time) : 0.25 second(s).
 First parse for input sentence:

{ ... }

parsing (cpu time) : 0.267 second(s).

```
parse_tree__decla_or_imper
  simple_sentence
    structure
      single_main_clause
    { ... }
```

HAIKU: initiating Case-based semantic analysis ...

UPDATED PARSE

```
TREE(parse_tree__decla_or_imper(simple_sentence(structure(single_main
_clause(imperative(*statement1*)))),tokens([refer,to,lines,304,and,305
,in,the,1990,general,tax,guide,for,more,information,on,these,non-
refundable,tax,credits])),end_of_input(period)))
```

LOGICAL CLAUSAL STRUCTURE

```
top_structure
  [*statement1*]
```

SYMBOLIC A-LISTS

```
coordinators
clauses
```

```
[*statement1*,statement(predicate(regular(verb(refer,tense([imperativ
e]),neg(yes),trans(tr_intr))),voice(active),complement(svo([conj_pps(
{ ... }
```

INPUT

```
STRING([refer,to,lines,304,and,305,in,the,1990,general,tax,guide,for,
more,information,on,these,non-refundable,tax,credits,.])
```

HAIKU: Case Analysis of current input's clauses ...

CURRENT CLAUSE: *statement1*

CURRENT SUBJ : nil

CURRENT VERB : refer

```
CURRENT COMPL : complement(svo([conj_pps(
{ ... }
```

```
CURRENT MODIFS : [main_verb_modif(nil),clause_modif(nil)]
```

{below: notice that HAIKU finds four CMs. In fact, the 'in' prepositional phrase is a modifier of the preceding 'to' prepositional phrase, and similarly for the 'on' and 'for' prepositional phrases. But since DIPETT could not disambiguate these, it left all prepositional phrases on the top-level, as modifiers of the main verb. This

ensures that no potential Case markers are overlooked. If the re-attachment facility were available, the user would have had the opportunity to reconsider the parse tree before Case analysis. Interestingly, in this example, the four CMs are semantically defensible.}

CMP found by HAIKU: to-in-for-on

```
> Do you wish to overrule this CMP
  (enter d to discard this clause, or
   a to discard the entire input) [y/N/d/a]?
```

S5: new verb with (entirely) new CMP; new CP.

```
entirely new CMP; no obvious candidate CPs to propose
list of close CMPs proposed by the "closeness metric":
[THE INPUT CMP PROPERLY UNDERMATCHES: ,
THE INPUT CMP NON-PROPERLY UNDERMATCHES: ,
match(0.15025, [psubj, pobj, on, adv]),
THE INPUT CMP PROPERLY OVERMATCHES: , match(0.416667, [to]),
THE INPUT CMP NON-PROPERLY OVERMATCHES: ,
match(1.11111, [psubj, on, in]), match(0.208333, [psubj, to]), match(0.20833
3, [psubj, for]), match(0.138889, [psubj, pobj, to]), match(0.138889, [psubj,
pobj, on])]
```

Candidate CPs with example sentences for the closest CMP (psubj-on-in):

```
CP: agt-lat-lto Ex:
[bob, printed, tom_s, report, on, the, printer, in, this, room, yesterday]
current clause's main verb (infinitive): refer
current clause's CMP (or a permutation): to-in-for-on
most probable or closest CPs, which are of compatible length: []
```

```
NEW CP: list of 28 legal Cases in HAIKU
agt:agent; benf:beneficiary; expr:experiencer; inst:instrument;
obj:object; recp:recipient; caus:cause; manr:manner; val:value;
eff:effect; purp:purpose; freq:frequency; tat:time_at;
tfrm:time_from; tto:time_to; ttru:time_through; dir:direction;
lat:location_at; lfrm:location_from; lto:location_to; ord:order;
ltru:location_through; ornt:orientation; cont:content;
excl:exclusion; acmp:accompaniment; matr:material; meas:measure.
REM: A CASE MAY BE SUFFIXED WITH A '?' TO DENOTE UNCERTAINTY
(e.g. lat?).
```

```
here is what the cmDict knows about this Case marker: to
(format: Case(Commonness, Example); CO=1:common, CO=2:uncommon)
tto(1, We worked from nine TO FIVE.)
lto(1, I nailed the picture TO THE WALL.)
dir(1, We drove TO THE MALL.)
recp(1, I gave it TO HIM.)
meas(1, The temperature has increased TO TEN DEGREES.)
purp(1, She came TO THE RESCUE.)
eff(1, We won TO MY SURPRISE.)
acmp(1, He added insult TO INJURY.)
benf(1, She gives money TO MANY CHARITIES.)
```

```
here is what the cmDict knows about this Case marker: in
(format: Case(Commonness, Example); CO=1:common, CO=2:uncommon)
lat(1, The car was parked IN THE GARAGE.)
lto(1, Jump IN THE POOL.)
```

```

tat(1,Come IN AN HOUR.)
ttru(1,We sleep IN THE NIGHT.)
cont(1,He is interested IN SCIENCE.)
matr(1,The statues were carved IN GRANITE.)
meas(1,I can usually drive to work IN FIFTEEN MINUTES.)
dir(1,She looked IN MY DIRECTION.)
freq(1,Buses arrive IN FIVE MINUTE INTERVALS.)
manr(1,He did it IN AN AWKWARD WAY.)
purp(2,We acted IN OUR BEST INTEREST.)
eff(2,The battle will end IN DEATH.)

```

```

here is what the cmDict knows about this Case marker: for
{format: Case(COMmonness,Example); CO=1:common, CO=2:uncommon}
lto(1,With a bear aim FOR THE HEART.)
dir(1,Run FOR THE TRAIN!)
cont(1,I stand FOR FISCAL PROBITY.)
ttru(1,They worked FOR THREE HOURS.)
benf(1,I'd do anything FOR YOU.)
purp(1,That drug is FOR PEOPLE WITH HEADACHES.)
meas(1,I bought a book FOR TEN DOLLARS.)
caus(1,He got a medal FOR COURAGE.)
recp(1,There's mail FOR EVERYONE.)
tat(2,Call him FOR TEN A.M.)
purp(2,I want a teddy FOR A PRESENT.)

```

```

here is what the cmDict knows about this Case marker: on
{format: Case(COMmonness,Example); CO=1:common, CO=2:uncommon}
cont(1,She was reading ON ECONOMICS.)
lat(1,We were sitting ON THE SOFA.)
tat(1,We returned ON MONDAY.)
purp(1,We spent the money ON BOOKS.)
caus(1,We acted ON HIS ADVICE.)
inst(1,I spoke to her ON THE TELEPHONE.)
ornt(1,The bottle lay ON ITS SIDE.)

```

```

please enter the new CP (e.g. agt-obj-tat),
or enter 'h' to see the current input string and CMP,
or CR to abort ["new CP"/h/CR]? lto-dir-purp-cont
CMP & CP will be paired as follows: to/lto in/dir for/purp on/cont
; Correct [n/Y]?

```

{below: the actual phase of knowledge acquisition ("learning"): HAIKU updates its dictionaries (KB).}

```

UPDATING mDict for refer ...
=> refer [to-in-for-on]
[[for,[purp]], [in,[dir]], [on,[cont]], [to,[lto]]]
UPDATING cmpDict for to-in-for-on ...
=> to-in-for-on [[lto-dir-purp-
cont, '[refer,to,lines,304,and,305,in,the,1990,general,tax,guide,for,m
ore,information,on,these,non-refundable,tax,credits,...]']]
UPDATING cpDict for lto-dir-purp-cont ...
=> lto-dir-purp-cont [refer]

```

```

CASE STRUCTURES
clauses
[*statement1*,
  case_structure(

```


Appendix G: Examples of Representative Semantic Analysis _____

```
verb(refer,tense([imperative]),neg(yes),trans(tr_intr)),  
to-in-for-on,  
lto-dir-purp-cont,  
nil,  
[conj_pps(  
  {...}  
  [main_verb_modif(nil),clause_modif(nil)]]]
```

HAIKU semantic analysis (cpu time) : 5.317 second(s).

Continue [n/Y]?

G.2: Semantic Analysis #2

Unit #2: Since Rosa is the supporting person with the lower net income , she would claim the child care expenses .
 Process this unit (Y/n), try another Version, or Exit [Y/n/v/e]?

lexical analysis (cpu time) : 0.1 second(s).
 First parse for input sentence:

{ ... }

parsing (cpu time) : 0.25 second(s).

```
parse_tree__decla_or_imper
  complex_sentence
    structure
      single_main_clause
    { ... }
```

HAIKU: initiating Case-based semantic analysis ...

Proceed with "pronoun explicitization" [n/Y]?
 ANAPHORA "EXPLICITIZATION" PRE-PROCESSOR ...

ENTITIES IN CURRENT PARSE TREE (OR CLAUSE):
 [entity(proper_noun(rosa),number(sg3),tokens([rosa])),

```
entity(determinatives(deter(the)),attributes([attrs(adj(support,pos))
,tokens([supporting]))),head_noun(noun(person),number(sg3),np_postmo
diflers([pp(with,
{ ... }
```

PRONOUNS IN CURRENT PARSE TREE (OR CLAUSE):
 [entity(ref(pers_pron(she)),number(sg3),tokens([she]))]

```
Trying to find a referent for:
entity(ref(pers_pron(she)),number(sg3),tokens([she]))
List of candidate entities in this input:
entity(proper_noun(rosa),number(sg3),tokens([rosa])) (1) ;
emb([entity(determinatives(deter(the)),attributes(conj_adj_p(implicit
_and,[attrs(adj(low,comp)),attrs(adj(net,pos))],tokens([lower,net]))
,head_noun(noun(income)),number(sg3),tokens([the,lower,net,income]))
) (2) ;
```

if appropriate, please select one of these referents:
 enter a number between 1 and 2 (or 0 if none is appropriate): 1

>> anaphora resolved: new functor is entity_pro

UPDATED PARSE

```
TREE(parse_tree__decla_or_imper(complex_sentence(structure(single_mai
n_clause(declarative(*statement2*))),tokens([since,rosa,is,the,suppor
ting,person,with,the,lower,net,income,,she,would,claim,the,child,car
e,expenses])),end_of_input(period)))
```

LOGICAL CLAUSAL STRUCTURE

```
top_structure
  [*statement2*,subordinator([since,tokens([since]))],*statement3*]
```

```
SYMBOLIC A-LISTS
  coordinators
  clauses
```

{below: this time, the input consists of two clauses: a main clause plus a subordinate clause. They will be analyzed separately.}

```
[*statement2*,statement(subordinate_clauses( ... *statement3* ... ,
{ ... )
```

```
[*statement3*,statement(subj(entity(proper_noun(rosa),
{ ... )
```

INPUT

```
STRING([since,rosa,is,the,supporting,person,with,the,lower,net,income
,,she,would,claim,the,child,care,expenses,.])
```

HAIKU: Case Analysis of current input's clauses ...

CURRENT CLAUSE: *statement2*

CURRENT SUBJ :

```
subj(entity_pro(proper_noun(rosa),number(sg3),tokens([rosa])))
```

CURRENT VERB : claim

CURRENT COMPL :

```
complement(svo(entity(determinatives(deter(the)),head_noun(noun(expen
se,noun_modifiers(pre_modif(conj_noun_pre_mod(implicit_and,[child,car
e])))),number(pl),tokens([the,child,care,expenses])))
```

CURRENT MODIFS :

```
[main_verb_modif(nil),clause_modif(subordinate_clauses([initial([subo
rdinate(adverbial(adv_clause([since,tokens([since]),*statement3*),to
kens([since,rosa,is,the,supporting,person,with,the,lower,net,income])
]),tokens([since,rosa,is,the,supporting,person,with,the,lower,net,inc
ome,,])])))]
```

CMP found by HAIKU: psubj-pobj-adv

> Do you wish to overrule this CMP

(enter d to discard this clause, or

a to discard the entire input) [y/N/d/a]?

S4: new verb with known CMP; new CP (relative to this verb).

candidate CPs with example sentences:

CP: agt-obj-manr Ex:

```
[bob,always,prints,and,reads,his,messages]
```

current clause's main verb (infinitive): claim

current clause's CMP (or a permutation): psubj-pobj-adv

most probable or closest CPs, which are of compatible length:

```
[agt-obj-manr]
```

NEW CP: list of 28 legal Cases in HAIKU

```
agt:agent; benf:beneficiary; expr:experiencer; inst:instrument;
obj:object; recp:recipient; caus:cause; manr:manner; val:value;
eff:effect; purp:purpose; freq:frequency; tat:time_at;
tfrm:time_from; tto:time_to; ttru:time_through; dir:direction;
lat:location_at; lfrm:location_from; lto:location_to; ord:order;
ltr:location_through; ornt:orientation; cont:content;
excl:exclusion; acmp:accompaniment; matr:material; meas:measure.
```

REM: A CASE MAY BE SUFFIXED WITH A '?' TO DENOTE UNCERTAINTY
(e.g. lat?).

please enter the new CP (e.g. agt-obj-tat),
or enter 'h' to see the current input string and CMP,
or CR to abort ["new CP"/h/CR]? agt-obj-caus
CMP & CP will be paired as follows: psubj/agt pobj/obj adv/caus ;
Correct [n/Y]?

UPDATING mDict for claim ...
=> claim [psubj-pobj-adv]
[[adv,[caus]], [pobj,[obj]], [psubj,[agt]]]
UPDATING cmpDict for psubj-pobj-adv ...
=> psubj-pobj-adv [[agt-obj-
caus, '[since,rosa,is,the,supporting,person,with,the,lower,net,income,
,,she,would,claim,the,child,care,expenses,.]'], [agt-obj-
manr,[bob,always,prints,and,reads,his,messages]]]
UPDATING cpDict for agt-obj-caus ...
=> agt-obj-caus [claim]

CURRENT CLAUSE: *statement3*
CURRENT SUBJ :
subj(entity(proper_noun(rosa),number(sg3),tokens([rosa])))
CURRENT VERB : be
CURRENT COMPL : s_qualifier(svc_sva(entity(
{ ... }
CURRENT MODIFS : [main_verb_modif(nil),clause_modif(nil)]
> Aborting Case Analysis: main verb is STATIVE.

{above: in HAIKU's current framework, stative verbs are not processed by the
Case Analyzer, but by the Mini-Network Builder.}

CASE STRUCTURES

```
clauses
  [*statement2*,
    case_structure(
      verb(claim,
        tense([conditional_present_simple]),neg(yes),trans(tr)),
      psubj-pobj-adv,
      agt-obj-caus,
      entity_pro(proper_noun(rosa),number(sg3),tokens([rosa])),
      entity(determinatives(deter(the)),
        head_noun(noun(expense,noun_modifiers(
          { ... })),
        [main_verb_modif(nil),
          clause_modif(subordinate_clauses(
            { ... }])
      [*statement3*,
        case_structure(
          verb(be,tense([present_simple]),neg(yes),trans(stat)),
          nil,
          nil,
          entity(proper_noun(rosa),number(sg3),tokens([rosa])),
          entity(determinatives(deter(the)),
            attributes([attrs(adj(support,pos)),tokens([supporting]))],
            head_noun(noun(person)),
            { ... },
            [main_verb_modif(nil),clause_modif(nil)])]
```

Appendix G: Examples of Representative Semantic Analysis _____

HAIKU semantic analysis (cpu time) : 4.984 second(s).

Continue [n/Y]?

G.3: Semantic Analysis #3

Unit #3: In the next few pages you can learn how to ask the right questions even if you've never used a computer before .
 Process this unit (Y/n), try another Version, or Exit [Y/n/v/e]?

lexical analysis (cpu time) : 0.3 second(s).
 First parse for input sentence:

PARSING STOPPED for the following reason:
 CPU TIME LIMIT EXCEEDED in ... complement(intr) ;
 > the current CPU time limit is: 5.0 second(s).
 At that moment, the input string was:
 [how,to,ask,the,right,questions,even_if,you,have,never,used,a,computer,before..]

** Sorry dear user, UNABLE TO PARSE THE INPUT **

{ ... }

** Reparse as a series of fragments [n/Y]?
 fragmentary parsing (cpu time) : 0.333 second(s).
 ** Number of fragments : 4
 ** Number of tokens not parsed : 1
 ** % of input string recognized : 95.6522%

parsing (cpu time) : 5.65 second(s).

parse_tree__frag_series
 sentence_fragment
 { ... }

HAIKU: initiating Case-based semantic analysis ...

** CA: the current input is a series of fragments ...

UPDATED PARSE
 TREE(sentence_fragment([pp(in,entity(determinatives(deter(the),postdeter(closed(ord(next))))),attributes([attrs(adj(few,pos)),tokens([few])]),head_noun(noun(page)),number(pl),tokens([the,next,few,pages])),tokens([in,the,next,few,pages])),tokens([in,the,next,few,pages])))

LOGICAL CLAUSAL STRUCTURE
 top_structure
 []

SYMBOLIC A-LISTS
 coordinators
 clauses

INPUT
 STRING([in,the,next,few,pages,you,can,learn,how,to,ask,the,right,questions,even,if,you've,never,used,a,computer,before..])

HAIKU: Case Analysis of current input's clauses ...

> Aborting Case Analysis: END OF ANALYSIS.

> EMPTY CLAUSE: CA cannot be performed on this partial input.

Proceed with "pronoun explicitization" [n/Y]?
 ANAPHORA "EXPLICITIZATION" PRE-PROCESSOR ...

ENTITIES IN CURRENT PARSE TREE (OR CLAUSE):
 [entity(determinatives(deter(the)),attributes([attrs(adj(right,pos)),
 tokens([right]))),head_noun(noun(question)),number(pl),tokens([the,ri
 ght,questions]))]
 PRONOUNS IN CURRENT PARSE TREE (OR CLAUSE):
 [entity(ref(pers_pron(you)),number(pl),tokens([you]))]

Trying to find a referent for:
 entity(ref(pers_pron(you)),number(pl),tokens([you]))
 > FOR THIS PRONOUN, SUGGESTING "generic", OK [n/Y]?

>> anaphora resolved: new functor is entity_pro_generic

UPDATED PARSE TREE(sentence_fragment(*statement4*))

LOGICAL CLAUSAL STRUCTURE
 top_structure
 [*statement4*]

SYMBOLIC A-LISTS
 coordinators
 clauses

[*statement4*,statement(subj(entity_pro_generic(ref(pers_pron(you)),n
 umber(pl),tokens([you]))),predicate(regular(verb(learn,tense([can_pre
 sent]),neg(yes),trans(tr_intr))),voice(active),complement(svo(nominal
 _clause(*to_infinitive_clause1*,tokens([how,to,ask,the,right,question
 s])))),tokens([can,learn,how,to,ask,the,right,questions]))]

[*to_infinitive_clause1*,to_infinitive_clause(wh_word(how),predicate(
 regular(verb(ask,tense([inf]),neg(yes),trans(tr_intr))),voice(active)
 ,complement(svo(entity(determinatives(deter(the)),attributes([attrs(a
 dj(right,pos)),tokens([right]))),head_noun(noun(question)),number(pl)
 ,tokens([the,right,questions])))))]

INPUT

STRING([in,the,next,few,pages,you,can,learn,how,to,ask,the,right,ques
 tions,even,if,you've,never,used,a,computer,before,.])

HAIKU: Case Analysis of current input's clauses ...

CURRENT CLAUSE: *statement4*
 CURRENT SUBJ :
 subj(entity_pro_generic(ref(pers_pron(you)),number(pl),tokens([you]))
)
 CURRENT VERB : learn
 CURRENT COMPL :
 complement(svo(nominal_clause(*to_infinitive_clause1*,tokens([how,to,
 ask,the,right,questions]))))
 CURRENT MODIFS : [main_verb_modif(nil),clause_modif(nil)]

```

CMP found by HAIKU: psubj-pobj
> Do you wish to overrule this CMP
  (enter d to discard this clause, or
   a to discard the entire input) [y/N/d/a]?

S4: new verb with known CMP: new CP (relative to this verb).

candidate CPs with example sentences:
  CP: agt-obj  Ex: [bob,printed,a,file]
current clause's main verb (infinitive): learn
current clause's CMP (or a permutation): psubj-pobj
most probable or closest CPs, which are of compatible length:
  [agt-obj]

NEW CP: list of 28 legal Cases in HAIKU
  agt:agent; benf:beneficiary; expr:experiencer; inst:instrument;
  obj:object; recp:recipient; caus:cause; manr:manner; val:value;
  eff:effect; purp:purpose; freq:frequency; tat:time_at;
  tfrm:time_from; tto:time_to; ttru:time_through; dir:direction;
  lat:location_at; lfrm:location_from; lto:location_to; ord:order;
  ltru:location_through; ornt:orientation; cont:content;
  excl:exclusion; acmp:accompaniment; matr:material; meas:measure.
  REM: A CASE MAY BE SUFFIXED WITH A '?' TO DENOTE UNCERTAINTY
(e.g. lat?).

  please enter the new CP (e.g. agt-obj-tat),
  or enter 'h' to see the current input string and CMP,
  or CR to abort ["new CP"/h/CR]? agt-obj
  CMP & CP will be paired as follows: psubj/agt pobj/obj ; Correct
[n/Y]?

  UPDATING mDict for learn ...
  => learn [psubj-pobj] [[pobj,[obj]], [psubj,[agt]]]
  UPDATING cpDict for agt-obj ...
  => agt-obj
[ask,complete,copy,crash,delete,erase,get,give,know,learn,like,look,p
ay,prefer,prepare,print,read,recreate,report,save,sort,store,think,tr
y,want,work,write]

CURRENT CLAUSE: *to_infinitive_clause1*
CURRENT SUBJ   : subj(wh_word(how))
CURRENT VERB   : ask
CURRENT COMPL  :
complement(svo(entity(determinatives(deter(the)),attributes([attrs(ad
j(right,pos)),tokens([right]])),head_noun(noun(question)),number(pl),
tokens([the,right,questions]))))
CURRENT MODIFS : [main_verb_modif(nil),clause_modif(nil)]

CMP found by HAIKU: psubj-pobj
> Do you wish to overrule this CMP
  (enter d to discard this clause, or
   a to discard the entire input) [y/N/d/a]? y
> Please, enter the new CMP (e.g. psubj-pobj-with): pobj
> Replacing 'psubj-pobj' with 'pobj' ; OK [n/Y]?

S2: known CMP not yet associated with known verb; CP is known or new.

candidate CPs with example sentences:

```



```

CP: obj Ex: [i,want,to,know,which,file,you,have,printed]
current clause's main verb (infinitive): ask
current clause's CMP (or a permutation): pobj
most probable or closest CPs, which are of compatible length:
  obj (1) ;

if appropriate, please select one of these CPs;
  enter a number between 1 and 1 (or 0 if none is appropriate): 1
CMP & CP will be paired as follows: pobj/obj ; Correct [n/Y]?
replace cmpDict's example with current clause [y/N]?

UPDATING mDict for ask ...
=> ask [pobj,psubj-pobj] [[pobj,[obj]],[psubj,[agt]]]
UPDATING cpDict for obj ...
=> obj
[access,ask,conclude,contain,get,give,happen,know,print,read,see,show
,think,try,write]

CASE STRUCTURES
clauses
[*statement4*,
  case_structure(
    verb(learn,tense([can_present]),neg(yes),trans(tr_intr)),
    psubj-pobj,
    agt-obj,
    entity_pro_generic(
      ref(pers_pron(you)),number(pl),tokens([you])),
    nominal_clause(*to_infinitive_clause1*,
      tokens([how,to,ask,the,right,questions])),
    [main_verb_modif(nil),clause_modif(nil)])]

[*to_infinitive_clause1*,
  case_structure(
    verb(ask,tense([inf]),neg(yes),trans(tr_intr)),
    pobj,
    obj,
    wh_word(how),
    entity(determinatives(deter(the)),
      attributes([attrs(adj(right,pos)),tokens([right])]),
      head_noun(noun(question)),number(pl),
      tokens([the,right,questions])),
    [main_verb_modif(nil),clause_modif(nil)])]

Proceed with "pronoun explicitization" [n/Y]?
ANAPHORA "EXPLICITIZATION" PRE-PROCESSOR ...

ENTITIES IN CURRENT PARSE TREE (OR CLAUSE):
[entity(determinatives(deter(a)),head_noun(noun(computer)),number(sg3
),tokens([a,computer]))]
PRONOUNS IN CURRENT PARSE TREE (OR CLAUSE):
[entity(ref(pers_pron(you)),number(pl),tokens([you]))]

Trying to find a referent for:
entity(ref(pers_pron(you)),number(pl),tokens([you]))
> FOR THIS PRONOUN, SUGGESTING "generic", OK [n/Y]?

>> anaphora resolved: new functor is entity_pro_generic

```

```

UPDATED PARSE TREE(sentence_fragment(*statement5*))

LOGICAL CLAUSAL STRUCTURE
  top_structure
    [*statement5*]

SYMBOLIC A-LISTS
  coordinators
  clauses

[*statement5*,statement(subj(entity_pro_generic(ref(pers_pron(you)),n
umber(pl),tokens([you]))),predicate(regular(verb(use,tense([present_p
erfect_simple]),neg(yes),adv([never,tokens([never]))),trans(tr_intr)
)),voice(active),complement(svo(entity(determinatives(deter(a)),head_
noun(noun(computer)),number(sg3),tokens([a,computer])))),tokens([hav
e,never,used,a,computer])))]

INPUT
STRING([in,the,next,few,pages,you,can,learn,how,to,ask,the,right,ques
tions,even,if,you've,never,used,a,computer,before,.])

HAIKU: Case Analysis of current input's clauses ...

CURRENT CLAUSE: *statement5*
CURRENT SUBJ   :
subj(entity_pro_generic(ref(pers_pron(you)),number(pl),tokens([you]))
)
CURRENT VERB   : use
CURRENT COMPL  :
complement(svo(entity(determinatives(deter(a)),head_noun(noun(compute
r)),number(sg3),tokens([a,computer]))))
CURRENT MODIFS :
[main_verb_modif(adv([never,tokens([never]))),clause_modif(nil)]

CMP found by HAIKU: psubj-pobj-adv
> Do you wish to overrule this CMP
  (enter d to discard this clause, or
   a to discard the entire input) [y/N/d/a]?

S4: new verb with known CMP; new CP (relative to this verb).

candidate CPs with example sentences:
  CP: agt-obj-caus  Ex:
' [since,rosa,is,the,supporting,person,with,the,lower,net,income,,,she
,would,claim,the,child,care,expenses,.] '
  CP: agt-obj-manr  Ex:
[bob,always,prints,and,reads,his,messages]
  current clause's main verb (infinitive): use
  current clause's CMP (or a permutation): psubj-pobj-adv
  most probable or closest CPs, which are of compatible length:
  [agt-obj-caus,agt-obj-manr]

NEW CP: list of 28 legal Cases in HAIKU
  agt:agent; benf:beneficiary; expr:experiencer; inst:instrument;
  obj:object; recp:recipient; caus:cause; manr:manner; val:value;
  eff:effect; purp:purpose; freq:frequency; tat:time_at;
  tfrm:time_from; tto:time_to; ttru:time_through; dir:direction;
  lat:location_at; lfrm:location_from; lto:location_to; ord:order;

```

```

ltru:location_through; ornt:orientation; cont:content;
excl:exclusion; acmp:accompaniment; matr:material; meas:measure.
REM: A CASE MAY BE SUFFIXED WITH A '?' TO DENOTE UNCERTAINTY
(e.g. lat?).

please enter the new CP (e.g. agt-obj-tat),
or enter 'h' to see the current input string and CMP,
or CR to abort ["new CP"/h/CR]? agt-obj-manr
CMP & CP will be paired as follows: psubj/agt pobj/obj adv/manr ;
Correct [n/Y]?

UPDATING mDict for use ...
=> use [psubj-pobj-adv] [[adv,[manr]], [pobj,[obj]], [psubj,[agt]]]
UPDATING cpDict for agt-obj-manr ...
=> agt-obj-manr [erase,need,print,use]

CASE STRUCTURES
clauses
[*statement5*,
  case_structure(
    verb(use,tense([present_perfect_simple]),neg(yes),
      advs([never,tokens([never]])],trans(tr_intr)),
    psubj-pobj-adv,
    agt-obj-manr,
    entity_pro_generic(ref(pers_pron(you)),number(pl),
      tokens([you])),
    entity(determinatives(deter(a)),head_noun(noun(computer)),
      number(sg3),tokens([a,computer])),
    [main_verb_modif(advs([never,tokens([never]])]),
      clause_modif(nil)]]])

UPDATED PARSE TREE(sentence_fragment([before,tokens([before])]))

LOGICAL CLAUSAL STRUCTURE
top_structure
[]

SYMBOLIC A-LISTS
coordinators
clauses

INPUT
STRING([in,the,next,few,pages,you,can,learn,how,to,ask,the,right,ques
tions,even,if,you've,never,used,a,computer,before,.])

HAIKU: Case Analysis of current input's clauses ...

> Aborting Case Analysis: END OF ANALYSIS.

> EMPTY CLAUSE: CA cannot be performed on this partial input.

HAIKU semantic analysis (cpu time) : 5.167 second(s).

```

Appendix H: KB Accumulated by HAIKU in Chapters 1 and 2

This appendix lists the full contents of HAIKU's KB after processing chapters 1 and 2 of the QUIZ manual, starting with empty dictionaries — see 4.8. At the end of chapter 1, HAIKU's KB contained 86 entries in its MDict, 38 entries in its CMPDict, and 38 entries in its CPDict. At the end of chapter 2, the number of entries were, respectively, 119 (+38%), 48 (+26%), and 50 (+32%).

H.1: MDict Entries

MDict (119 entries)

```

mDict(access,[pobj,'pobj-adv','psubj-pobj','psubj-pobj-adv','psubj-
to'],[[adv,[manr]], [pobj,[obj]], [psubj,[agt]], [to,[obj]]]).
mDict(add,[pobj,'pobj-to'],[[pobj,[obj]], [to,[lto]]]).
mDict(affect,['psubj-pobj'],[[pobj,[obj]], [psubj,[agt]]]).
mDict(agree,['on-within'],[[on,[cont]], [within,[lat]]]).
mDict(answer,['psubj-pobj'],[[pobj,[obj]], [psubj,[agt]]]).
mDict(apply,['pobj-adv'],[[adv,[manr]], [pobj,[obj]]]).
mDict(ask,[pobj,'pobj-piobj','psubj-for-like','psubj-
pobj'],[[for,[obj]], [like,[manr]], [piobj,[obj]], [pobj,[obj,recp]],
[psubj,[agt]]]).
mDict(assume,[pobj],[[pobj,[obj]]]).
mDict(avoid,[pobj],[[pobj,[obj]]]).
mDict(base,['pobj-on-for'],[[for,[purp]], [on,[cont]], [pobj,[obj]]]).
mDict(begin,[pobj],[[pobj,[obj]]]).
mDict(build,[pobj,'psubj-pobj-
adv'],[[adv,[manr]], [pobj,[obj]], [psubj,[agt]]]).
mDict(call,[pobj,'pobj-adv','psubj-
pobj'],[[adv,[caus]], [pobj,[obj]], [psubj,[agt]]]).
mDict(cancel,['psubj-pobj','psubj-pobj-
adv'],[[adv,[manr]], [pobj,[obj]], [psubj,[agt]]]).
mDict(cause,[pobj,'psubj-pobj','psubj-pobj-
adv'],[[adv,[manr]], [pobj,[obj]], [psubj,[agt]]]).
mDict(change,[pobj,'psubj-pobj'],[[pobj,[obj]], [psubj,[agt]]]).
mDict(check,['with-on'],[[on,[cont]], [with,[acmp]]]).
mDict(choose,[pobj,'psubj-pobj','psubj-pobj-
adv'],[[adv,[manr]], [pobj,[obj]], [psubj,[agt]]]).
mDict(clear,[pobj,'psubj-pobj-
adv'],[[adv,[ord]], [pobj,[obj]], [psubj,[agt]]]).
mDict(combine,[pobj],[[pobj,[obj]]]).
mDict(consist,['psubj-of-
to'],[[of,[obj]], [psubj,[agt]], [to,[recp]]]).
mDict(contain,['psubj-pobj','psubj-pobj-
adv'],[[adv,[manr]], [pobj,[obj]], [psubj,[agt]]]).
mDict(copy,[pobj,'psubj-pobj'],[[pobj,[obj]], [psubj,[agt]]]).
mDict(create,[pobj,'psubj-
pobj',with],[[pobj,[obj]], [psubj,[agt]], [with,[inst]]]).
mDict(describe,[pobj],[[pobj,[obj]]]).
mDict(design,['psubj-pobj'],[[pobj,[obj]], [psubj,[agt]]]).
mDict(designate,['pobj-as','psubj-pobj-
as'],[[as,[manr]], [pobj,[obj]], [psubj,[agt]]]).
mDict(display,[on,'psubj-
pobj'],[[on,[lat]], [pobj,[obj]], [psubj,[agt]]]).
mDict(do,[pobj,'psubj-on','psubj-
pobj'],[[on,[cont]], [pobj,[obj]], [psubj,[agt]]]).
mDict(eliminate,['psubj-pobj'],[[pobj,[obj]], [psubj,[agt]]]).
mDict(encounter,['psubj-pobj-
adv'],[[adv,[manr]], [pobj,[obj]], [psubj,[agt]]]).
mDict(enter,[pobj,'pobj-adv','pobj-in','pobj-into-as','pobj-
on','psubj-pobj','psubj-pobj-
in'],[[adv,[manr]], [as,[manr]], [in,[manr,lat]], [into,[lto]], [on,[l
at]], [pobj,[obj]], [psubj,[agt]]]).

```

```

mDict(establish, [pobj, 'pobj-between', 'psubj-pobj-
adv'], [[adv, [manr]], [between, [lat]], [pobj, [obj]], [psubj, [agt]]]).
mDict(execute, ['psubj-pobj'], [[pobj, [obj]], [psubj, [agt]]]).
mDict(explain, [pobj], [[pobj, [obj]]]).
mDict(find, [pobj, 'psubj-pobj'], [[pobj, [obj]], [psubj, [agt]]]).
mDict(fit, ['psubj-across'], [[across, [ltru]], [psubj, [agt]]]).
mDict(follow, [pobj, 'psubj-pobj', 'psubj-pobj-
adv'], [[adv, [manr]], [pobj, [obj]], [psubj, [agt]]]).
mDict(format, [pobj, 'psubj-pobj-
adv'], [[adv, [excl]], [pobj, [obj]], [psubj, [agt]]]).
mDict(get, [pobj, 'psubj-pobj'], [[pobj, [obj]], [psubj, [agt]]]).
mDict(give, ['psubj-pobj'], [[pobj, [obj]], [psubj, [agt]]]).
mDict(handle, ['psubj-pobj-
adv'], [[adv, [manr]], [pobj, [obj]], [psubj, [agt]]]).
mDict(have, [pobj, 'pobj-in', 'psubj', 'psubj-pobj', 'psubj-pobj-
adv'], [[adv, [manr]], [in, [lat]], [pobj, [obj]], [psubj, [agt]]]).
mDict(hold, ['down-adv'], [[adv, [tat]], [down, [obj]]]).
mDict(identify, ['psubj-pobj'], [[pobj, [obj]], [psubj, [agt]]]).
mDict(include, ['pobj-in', 'psubj-pobj-in-
adv'], [[adv, [manr]], [in, [lto]], [pobj, [obj]], [psubj, [agt]]]).
mDict(indicate, ['psubj-pobj'], [[pobj, [obj]], [psubj, [agt]]]).
mDict(initiate, ['psubj-pobj'], [[pobj, [obj]], [psubj, [agt]]]).
mDict(involve, ['psubj-pobj-
adv'], [[adv, [ord]], [pobj, [obj]], [psubj, [agt]]]).
mDict(issue, ['psubj-pobj'], [[pobj, [obj]], [psubj, [agt]]]).
mDict(join, ['psubj-pobj', 'psubj-pobj-
adv'], [[adv, [tfm]], [pobj, [obj]], [psubj, [agt]]]).
mDict(know, [pobj, 'psubj-pobj'], [[pobj, [obj]], [psubj, [agt]]]).
mDict(learn, ['psubj-pobj'], [[pobj, [obj]], [psubj, [agt]]]).
mDict(leave, [pobj, 'pobj-adv'], [[adv, [manr]], [pobj, [obj]]]).
mDict(let, [pobj, 'pobj-adv'], [[adv, [manr]], [pobj, [obj]]]).
mDict(lie, ['psubj-in-
with'], [[in, [cont]], [psubj, [agt]], [with, [acmp]]]).
mDict(like, ['psubj-pobj'], [[pobj, [obj]], [psubj, [agt]]]).
mDict(link, [pobj, 'pobj-in', 'psubj-pobj', 'psubj-pobj-piobj', 'through-
in', to], [[in, [lat, lto]], [piobj, [obj]], [pobj, [obj, tat]], [psubj, [agt
]], [through, [ltru]], [to, [lto]]]).
mDict(list, [pobj, 'psubj-pobj', 'psubj-pobj-
in'], [[in, [lto]], [pobj, [obj]], [psubj, [agt]]]).
mDict(look, ['psubj-at'], [[at, [lat]], [psubj, [agt]]]).
mDict(lose, ['pobj-adv'], [[adv, [manr]], [pobj, [obj]]]).
mDict(make, [to], [[to, [recp]]]).
mDict(mean, [pobj, 'psubj-pobj'], [[pobj, [obj]], [psubj, [agt]]]).
mDict(miss, ['psubj-pobj'], [[pobj, [obj]], [psubj, [agt]]]).
mDict(modify, ['psubj-pobj'], [[pobj, [obj]], [psubj, [agt]]]).
mDict(move, [pobj], [[pobj, [obj]]]).
mDict(name, [pobj, 'psubj-pobj', 'psubj-pobj-
in'], [[in, [lto]], [pobj, [obj]], [psubj, [agt]]]).
mDict(need, [pobj, 'psubj-pobj', 'psubj-pobj-
adv'], [[adv, [tfm, ord]], [pobj, [obj]], [psubj, [agt]]]).
mDict(notice, ['pobj-adv'], [[adv, [manr]], [pobj, [obj]]]).
mDict(number, ['psubj-pobj'], [[pobj, [obj]], [psubj, [agt]]]).
mDict(occupy, ['pobj-adv'], [[adv, [manr]], [pobj, [obj]]]).
mDict(omit, ['pobj-from'], [[from, [lfrm]], [pobj, [obj]]]).
mDict(organize, [with], [[with, [inst]]]).
mDict(override, ['psubj-pobj'], [[pobj, [obj]], [psubj, [agt]]]).
mDict(personalize, [pobj], [[pobj, [obj]]]).

```

```

mDict(place, ['psubj-pobj-
  adv'], [[adv, [manr]], [pobj, [obj]], [psubj, [agt]]]).
mDict(prepare, [pobj], [[pobj, [obj]]]).
mDict(press, [pobj, 'pobj-adv'], [[adv, [manr]], [pobj, [obj]]]).
mDict(print, ['psubj-on'], [[on, [lto]], [psubj, [agt]]]).
mDict(process, ['of-
  from', pobj], [[from, [lfrm]], [of, [obj]], [pobj, [obj]]]).
mDict(produce, [pobj, psubj, 'psubj-
  pobj'], [[pobj, [obj]], [psubj, [agt]]]).
mDict(provide, [pobj, 'psubj-pobj'], [[pobj, [recp, obj]], [psubj, [agt]]]).
mDict(re-enter, [pobj], [[pobj, [obj]]]).
mDict(read, [pobj, 'pobj-adv', 'psubj-pobj', 'psubj-
  with'], [[adv, [manr]], [pobj, [obj]], [psubj, [agt]], [with, [inst]]]).
mDict(recall, ['psubj-pobj'], [[pobj, [obj]], [psubj, [agt]]]).
mDict(recognize, ['psubj-pobj-
  adv'], [[adv, [tat]], [pobj, [obj]], [psubj, [agt]]]).
mDict(record, [from, pobj, 'pobj-
  on'], [[from, [lfrm]], [on, [lat]], [pobj, [obj]]]).
mDict(reduce, ['psubj-pobj'], [[pobj, [obj]], [psubj, [agt]]]).
mDict(refer, ['psubj-to', 'psubj-to-adv', 'to-
  in'], [[adv, [ltru]], [in, [lto]], [psubj, [agt]], [to, [cont, obj]]]).
mDict(report, [pobj, 'pobj-adv', 'pobj-for-adv', 'pobj-in', 'psubj-
  pobj', 'psubj-pobj-
  adv', with], [[adv, [manr]], [for, [lto]], [in, [lto]], [pobj, [obj]], [psub
  j, [agt]], [with, [inst]]]).
mDict(require, ['psubj-pobj'], [[pobj, [obj]], [psubj, [agt]]]).
mDict(respond, ['psubj-with', 'psubj-with-
  adv'], [[adv, [freq]], [psubj, [agt]], [with, [manr]]]).
mDict(restart, ['psubj-pobj'], [[pobj, [obj]], [psubj, [agt]]]).
mDict(review, [of, pobj], [[of, [obj]], [pobj, [obj]]]).
mDict(satisfy, ['psubj-with-
  adv'], [[adv, [manr]], [psubj, [agt]], [with, [obj]]]).
mDict(save, [pobj, 'pobj-with', 'psubj-
  pobj'], [[pobj, [obj]], [psubj, [agt]], [with, [inst]]]).
mDict(say, [pobj, 'psubj-pobj'], [[pobj, [obj]], [psubj, [agt]]]).
mDict(search, ['psubj-pobj'], [[pobj, [obj]], [psubj, [agt]]]).
mDict(see, [pobj, 'psubj-pobj-
  adv'], [[adv, [ord]], [pobj, [obj]], [psubj, [agt]]]).
mDict(select, [pobj], [[pobj, [obj]]]).
mDict(set, [pobj, 'psubj-pobj-
  adv'], [[adv, [lat]], [pobj, [obj]], [psubj, [agt]]]).
mDict(show, [pobj], [[pobj, [obj]]]).
mDict(sort, [pobj, 'pobj-on', 'psubj-
  adv'], [[adv, [manr]], [on, [cont]], [pobj, [obj]], [psubj, [agt]]]).
mDict(specify, [pobj], [[pobj, [obj]]]).
mDict(start, ['psubj-pobj', 'psubj-pobj-
  adv'], [[adv, [manr]], [pobj, [obj]], [psubj, [agt]]]).
mDict(stop, [pobj], [[pobj, [obj]]]).
mDict(store, [pobj, 'psubj-pobj', 'psubj-pobj-
  adv'], [[adv, [manr]], [pobj, [obj]], [psubj, [agt]]]).
mDict(suppose, [pobj], [[pobj, [obj]]]).
mDict(suspect, ['psubj-pobj'], [[pobj, [obj]], [psubj, [agt]]]).
mDict(take, [pobj], [[pobj, [obj]]]).
mDict(tell, [pobj, 'pobj-adv', 'psubj-pobj', 'psubj-pobj-adv', 'psubj-
  pobj-
  piobj'], [[adv, [manr]], [piobj, [caus, obj, recp]], [pobj, [obj, manr]], [p
  subj, [agt]]]).
mDict(try, [pobj, 'psubj-pobj'], [[pobj, [obj]], [psubj, [agt]]]).

```

```

mDict(turn, ['psubj-pobj-into-
  for'], [[for, [benf]], [into, [val]], [pobj, [obj]], [psubj, [agt]]]).
mDict(type, [pobj], [[pobj, [obj]]]).
mDict(understand, ['psubj-pobj'], [[pobj, [obj]], [psubj, [agt]]]).
mDict(use, [pobj, 'pobj-as-for-in-adv', 'pobj-piobj', 'psubj-adv', 'psubj-
  pobj', 'psubj-pobj-
  adv'], [[adv, [freq, manr]], [as, [manr]], [for, [purp]], [in, [lto]], [piob
  j, [purp]], [pobj, [obj]], [psubj, [agt]]]).
mDict(view, ['psubj-pobj'], [[pobj, [obj]], [psubj, [agt]]]).
mDict(want, [pobj, psubj, 'psubj-pobj', 'psubj-pobj-adv', 'psubj-pobj-
  piobj', 'psubj-pobj-
  to'], [[adv, [lat, tat]], [piobj, [obj]], [pobj, [obj, caus]], [psubj, [agt]
  ], [to, [lto]]]).
mDict(work, [for, in, 'psubj-pobj', 'psubj-pobj-adv', 'through-in-
  as', with], [[adv, [manr]], [as, [manr]], [for, [recp]], [in, [lto, lat]], [p
  obj, [obj]], [psubj, [agt]], [through, [ltru]], [with, [inst]]]).

```


H.2: CMPDict Entries

CMPDict (48 entries: to abbreviate the presentation, example sentences have been replaced with “... “ here)

```

cmpDict('down-adv',[['obj-tat', ... ]]).
cmpDict('of-from',[['obj-lfrm', ... ]]).
cmpDict('on-within',[['cont-lat', ... ]]).
cmpDict('pobj-adv',[['obj-caus', ... ],['obj-manr', ... ]]).
cmpDict('pobj-as',[['obj-manr', ... ]]).
cmpDict('pobj-as-for-in-adv',[['obj-manr-purp-lto-freq', ... ]]).
cmpDict('pobj-between',[['obj-lat', ... ]]).
cmpDict('pobj-for-adv',[['obj-lto-manr', ... ]]).
cmpDict('pobj-from',[['obj-lfrm', ... ]]).
cmpDict('pobj-in',[['obj-lat', ... ],['obj-lto', ... ]]).
cmpDict('pobj-into-as',[['obj-lto-manr', ... ]]).
cmpDict('pobj-on',[['obj-cont', ... ],['obj-lat', ... ]]).
cmpDict('pobj-on-for',[['obj-cont-purp', ... ]]).
cmpDict('pobj-piobj',[['obj-purp', ... ],['recp-obj', ... ]]).
cmpDict('pobj-to',[['obj-lto', ... ]]).
cmpDict('pobj-with',[['obj-inst', ... ]]).
cmpDict('psubj-across',[['agt-ltru', ... ]]).
cmpDict('psubj-adv',[['agt-manr', ... ]]).
cmpDict('psubj-at',[['agt-lat', ... ]]).
cmpDict('psubj-for-like',[['agt-obj-manr', ... ]]).
cmpDict('psubj-in-with',[['agt-cont-acmp', ... ]]).
cmpDict('psubj-of-to',[['agt-obj-recp', ... ]]).
cmpDict('psubj-on',[['agt-cont', ... ],['agt-lto', ... ]]).
cmpDict('psubj-pobj',[['agt-obj', ... ]]).
cmpDict('psubj-pobj-adv',[['agt-obj-excl', ... ],['agt-obj-freq', ... ],
    [['agt-obj-lat', ... ],['agt-obj-manr', ... ],['agt-obj-ord', ... ],
    [['agt-obj-tat', ... ],['agt-obj-tfrm', ... ]]).
cmpDict('psubj-pobj-as',[['agt-obj-manr', ... ]]).
cmpDict('psubj-pobj-in',[['agt-obj-lto', ... ],['agt-obj-manr', ... ]]).
cmpDict('psubj-pobj-in-adv',[['agt-obj-lto-manr', ... ]]).
cmpDict('psubj-pobj-into-for',[['agt-obj-val-benf', ... ]]).
cmpDict('psubj-pobj-piobj',[['agt-caus-obj', ... ],['agt-manr-obj', ... ],
    [['agt-obj-caus', ... ],['agt-obj-recp', ... ],['agt-tat-obj', ... ]]).
cmpDict('psubj-pobj-to',[['agt-obj-lto', ... ]]).
cmpDict('psubj-to',[['agt-obj', ... ]]).
cmpDict('psubj-to-adv',[['agt-obj-ltru', ... ]]).
cmpDict('psubj-with',[['agt-inst', ... ],['agt-manr', ... ]]).
cmpDict('psubj-with-adv',[['agt-manr-freq', ... ],['agt-obj-manr', ... ]]).
cmpDict('through-in',[['ltru-lto', ... ]]).
cmpDict('through-in-as',[['ltru-lto-manr', ... ]]).
cmpDict('to-in',[['cont-lto', ... ]]).
cmpDict('with-on',[['acmp-cont', ... ]]).
cmpDict(for,[['recp', ... ]]).
cmpDict(from,[['lfrm', ... ]]).
cmpDict(in,[['lat', ... ]]).
cmpDict(of,[['obj', ... ]]).
cmpDict(on,[['lat', ... ]]).
cmpDict(pobj,[['obj', ... ],['recp', ... ]]).

```

```
cmpDiet(psubj,[[agt, ... ]]).  
cmpDiet(to,[[lto, ... ],[recp, ... ]]).  
cmpDiet(with,[[inst, ... ]]).
```



```
cpDict('obj-purp', {use}).
cpDict('obj-tat', {hold}).
cpDict('recp-obj', {ask}).
cpDict(agt, {have, produce, want}).
cpDict(inst, {create, organize, report, work}).
cpDict(lat, {display, work}).
cpDict(lfrm, {record}).
cpDict(lto, {link}).
cpDict(obj, {access, add, ask, assume, avoid, begin, build, call, cause, change
, choose, clear, combine, copy, create, describe, do, enter, establish, expl
ain, find, follow, format, get, have, know, leave, let, link, list, mean, move
, name, need, personalize, prepare, press, process, produce, read, record, r
eport, review, save, say, see, select, set, show, sort, specify, stop, store,
suppose, take, tell, try, type, use, want, re-enter}).
cpDict(recp, {make, provide, work}).
```

Appendix I: HAIKU's Mini-Network Builder (MNB)

This appendix briefly describes the MNB, one of HAIKU's components that has been mentioned several times in chapters 1 and 4, and identified as the post-Case-analysis processor. The following description was prepared by Ken Barker, the person in the TANKA research group responsible for the MNB: it is included here with the kind permission of its author. We remind the reader that this module is still under development at the time of writing: what follows is basically design considerations. Nevertheless, this information is very helpful since it provides complementary details of HAIKU and supports our claim that DIPETT and HAIKU constitute a KA-productive linguistic module.

I.1: From Augmented Parse Trees to Conceptual Network Frames

The MNB is the module in TANKA responsible for constructing Conceptual Network (CN) frames¹, also called concepts, from parse trees augmented by Case analysis. The input to the MNB is a list of Case Structures (one per clause) and clausal connectives from a given sentence. Output from the MNB is a set of concept frames connected by links corresponding to semantic relationships between constituents of the sentence.

Three levels of semantic links exist: Clause Level links representing semantic relationships between clauses; Case links representing semantic relationships between a verb and its arguments; and Noun Modifier links representing semantic relationships between the head noun of a noun phrase and its modifiers.

One of the main tasks of the MNB is to augment the output of HAIKU's Case analysis to include the semantic information from Clause Level and Noun Modifier relationships. In TANKA, the MNB comes right after the Case analysis module and before the CN matching module.

¹ For the most recent details on the CN formalism, see [Yang 93].

I.2: Clause Level Relationships

Clause Level Relationships (CLRs) are semantic relationships between two clauses joined by a conjunction. The conjunction acts as a marker heralding the presence of a CLR. An exhaustive dictionary search of conjunctions has provided a list of "CLR markers" and the particular CLRs marked by each conjunction. The following table lists the CLRs and the corresponding link names. Since CLR links are directional, an inverse link name is also required for each CLR.

CLR	Link Name	Inverse Link Name
Cause	causes	caused_by
Enablement	enables	enabled_by
Entailment	entails	entailed_by
Prevention	prevents	prevented_by
Opposition	opposes	opposed_by
Conjunction	conj	conj
Disjunction	disj	disj
Temporal Precedence	before	after
Time_at	time	time
Location	location	location

In the following CLR definitions, A1 and A2 refer to acts or states represented by clauses.

- **Cause**

The Cause relationship is represented by a link `causes(A1,A2)` where A1 makes A2 occur or exist. A1 is probably sufficient to cause A2, but is not necessarily necessary.

INVERSE: `caused_by(A2,A1)`

- **Enablement**

The Enablement relationship is represented by a link `enables(A1,A2)` where A1 makes A2 possible. A1 is necessary to enable A2 but is not sufficient. This CLR covers the relationship that is sometimes referred to as Instrumental-Causation.

INVERSE: `enabled_by(A2,A1)`

- **Entailment**

A link `entails(A1,A2)` represents an Entailment relationship. If A1 exists or occurs then A2 must also exist or occur.

INVERSE: `entailed_by(A2,A1)`

- **Prevention**

The Prevention relationship is represented by a link `prevents(A1,A2)` where A1 is meant to keep A2 from occurring or existing. A1 is sufficient to prevent A2 but is not necessarily necessary.

INVERSE: `prevented_by(A2,A1)`

- **Opposition**

The Opposition relationship is represented by a link `opposes(A1,A2)`. A1 detracts from A2 but is insufficient to keep A2 from occurring or existing

INVERSE: `opposed_by(A2,A1)`

- **Conjunction**

A Conjunction relationship exists between two acts or states about which no more can be said than that they both occur or exist. This relationship is represented by a link `conj(A1,A2)`.

INVERSE: `conj(A2,A1)`

- Disjunction

A Disjunction relationship exists between two acts or states about which no more can be said than that one or both occur or exist. This relationship is represented by a link `conj(A1,A2)`.

INVERSE: `disj(A2,A1)`

- Location

The Location relationship is represented by a link `location(A1,A2)` where A2 indicates the position (physical or abstract) where A1 occurs or exists.

INVERSE: `location(A2,A1)`

- Temporal Precedence

Temporal Precedence relationships are represented by a link `before(A1,A2)` where A1 occurs or exists before A2.

INVERSE: `after(A2,A1)`

- Time

A link `time(A1,A2)` represents the relationship in which A1 and A2 occur or exist at the same time.

INVERSE: `time(A2,A1)`

1.3: Case Relationships

The Case Relationships will typically be used to construct links between an ACTIVITY type concept (representing the central verb within a clause) and OBJECT type (possibly also PROPERTY type) concepts (representing the Case fillers). The following list gives the link names and inverse link names corresponding to the 28 Cases.

Case Name	Link Name	Inverse Link Name
Agent	agent	agent_of
Beneficiary	beneficiary	beneficiary_of
Experiencer	experiencer	experiencer_of
Instrument	instrument	instrument_of
Object	object	object_of
Recipient	recipient	recipient_of
Cause	cause	cause_of
Effect	effect	effect_of
Opposition	opposition	opposition_of
Purpose	purpose	purpose_of
Frequency	frequency	frequency_of
Time_at	time_at	time_of
Time_from	time_from	start_time_of
Time_to	time_to	end_time_of
Time_through	time_through	duration_of
Direction	direction	direction_of
Location_at	location_at	location_of

Location_from	location_from	source_location_of
Location_to	location_to	dest_location_of
Location_through	location_through	through_location_of
Orientation	orientation	orientation_of
Accompaniment	accompaniment	accompaniment_of
Content	content	content_of
Exclusion	excludes	excluded_from
Manner	manner	manner_of
Material	material	material_of
Measure	measure	measure_of
Order	order	order_of

1.4: Noun Modifier Relationships

The Noun Modifier Relationships (NMRs) are semantic relationships between a noun and nominal modifiers (including nouns, nominal adjectives and post-nominal prepositional phrases). Since clauses, verb arguments and noun modifiers often represent the same underlying semantic relationships, many of the corresponding CLR, Case and NMR names overlap. Where NMR link names collide with CLR and Case link names, the NMR name has a prefixed 'n-'. Although the underlying semantic relationship may be the same, the network link names should be kept distinct.

NMR	Link Name	Inverse Link Name
Causative	n-causes	n-caused_by
Possessive	possesses	possessed_by
Productive	produces	produced_by
Compositional	composes	composed_of
Instrumental	uses	used_by
Essive	is	is
Locational	n-location	n-location_of
Temporal	n-time	n-time_of
Source	source	source_of
Destination	destination	destination_of
Purposive	n-purpose	n-purpose_of
Benefactive	n-beneficiary	n-beneficiary_of
Topic	topic	topic_of
Agentive	n-agent	n-agent_of
Objective	n-object	n-object_of

1.5: Building Mini-Networks

The MNB converts the semantic representation of a sentence into a set of CN concept frames. The basic process is as follows.

- 1) The MNB creates a concept frame for the main verb in each clause in a sentence. Links corresponding to the Case Relationships between the verb and its arguments are inserted into the verb frame. Concept frames are also created for each of the Case fillers. These Case Filler frames are given Inverse Case links to match the Case links in the verb frame. For the clause "printers print documents", the MNB creates a concept frame for the verb "print" with the links agent(printers) and

object(documents). The concept frame for "printers" has a link agent_of(printing) and the frame for "documents" has a link object_of(printing).

- 2) For each Case Filler frame corresponding to a noun phrase, links are inserted corresponding to the Noun Modifier relationships between the current noun and its modifiers. For example, the concept frame corresponding to the noun phrase "laser printer" would have a link uses(laser) and the frame for "laser" would have a link used_by(laser printer).
- 3) Finally, Clause Level Relationships are represented by inserting CLR links into the verb frames of each of the clauses. For the sentence "If you finish the work then you can leave" a link enables(leaving) will be inserted into the "finish" concept frame and a link enabled_by(finishing) will be inserted into the frame corresponding to "leave".

The following examples show the results of the three levels of semantic analysis (Clause Level, Case and Noun Modifier) as well as the final MNB Output: a set of (slightly simplified) CN concept frames. The examples are intended to partially illustrate how the MNB will do further semantic analysis on the Case Analyzer output and convert the augmented semantic representation into CN representation.

- **Example 1: "A solar generator produces electricity from the sun's energy."**

Clausal Analysis

<no clausal connectives>

Case Analysis

producing:

agent: 'a solar generator'

object: 'electricity'

instrument: 'the sun's energy'

Noun Modifier Analysis

'a solar generator':

concept name: solar_generator (by concatenation)

superconcept: generator (by modifier deletion)

instrumental: sun (by nominalization of modifier 'solar')

'electricity':

concept name: electricity

'the sun's energy':

concept name: energy_of_sun (by possessive inversion)

superconcept: energy

source: sun (by deletion of possessive)

MNB output

OBJECT solar_generator.

is_a: generator.

uses: sun.

agent_of: producing.

end solar_generator.

OBJECT sun.

used_by: solar_generator.

source_of: energy_of_sun.

end sun.

OBJECT generator.

kind: solar_generator.

```

end generator.
ACTIVITY producing.
  agent: solar_generator.
  object: electricity.
  instrument: energy_of_sun.
end producing.
OBJECT electricity.
  object_of: producing.
end electricity.
OBJECT energy_of_sun.
  source: sun.
  instrument_of: producing.
end energy_of_sun.

```

- **Example 2: “A taxpayer can claim childcare expenses if he paid money to a nursery.”**

Clausal Analysis

```

*statement1* if *statement2*:
CLR: enables(*statement2*, *statement1*)

```

Case Analysis

```

claiming:
  agent: 'a taxpayer'
  object: 'childcare expenses'
paying:
  agent: 'a taxpayer'
  object: 'money'
  recipient: 'a nursery school'

```

(by pronoun explicitization module)

Noun Modifier Analysis

```

'a taxpayer':
  concept name: taxpayer
'childcare expenses':
  concept name: childcare_expense
  superconcept: expense
  purposive: childcare
'money':
  concept name: money
'nursery':
  concept name: nursery

```

MNB Output

```

OBJECT taxpayer.
  agent_of: {claiming, paying}.
end taxpayer.
ACTIVITY claiming.
  agent: taxpayer.
  object: childcare_expense.
  enabled_by: paying.
end claiming.
OBJECT childcare_expense.
  is_a: expense.
  for_purpose_of: childcare.
  object_of: claiming.
end childcare_expense.

```

```
OBJECT expense.  
  kind: childcare_expense.  
end expense.  
OBJECT childcare.  
  purpose_of: childcare_expense.  
end childcare.  
ACTIVITY paying.  
  agent: taxpayer.  
  object: money.  
  recipient: nursery_school.  
  enables: claiming.  
end paying.  
OBJECT money.  
  object_of: paying.  
end money.  
OBJECT nursery.  
  recipient_of: paying.  
end nursery.
```

INDEX

- a priori domain knowledge 59; 188
- a priori knowledge i; 5; 46; 49; 202
- a priori semantic knowledge 195
- abbreviations 74
- acquiring fragments of knowledge 101
- actual clauses 197
- actual parse trees 331
- adaptation effort 177
- ADJECTIVES 71; 95
- ADVERBIAL PARTICLES 71
- Adverbial clause 83
- adverbials 138; 139
- ADVERBS 71
- agreement and ambiguity. 61
- agreement grammars 61
- AI xii
- algorithmic definition of the CA process 145
- ALICE 38
- Alvey project 26
- ambiguity 62
- anaphora 31; 104
- anytime module strategy 205
- Appendage 82
- Apposition 82
- architecture of the prototype 9
- arguments 50; 51; 121
- association between individual verbs and CPs 159
- attachment 28; 91; 145
- attachment heuristics 91; 186
- Avoiding Complete Rejection 157

- batch parsing mode 93; 113
- BEATRIX 23
- beliefs 59
- best analysis 91
- best first parse trees 183
- best neighbours 169
- best-first expectation-driven analysis 193
- bipartite constructions 95
- broad coverage 200
- broad syntactic coverage 25
- broad-coverage parser 59; 108; 114
- broad-coverage parsing 13

- CA algorithm 148; 150; 160; 161; 162
- CA driver 145; 148
- CA driver's main algorithm 145
- caption 136
- Cardinality 164
- Case 121; 131
- Case analysis 127; 128; 193; 412
- Case analysis of a single clause 148
- Case analysis process 188
- Case analysis situations 346
- Case Analyzer 162; 194
- Case fillers 122; 158; 204; 208; 416
- Case frames 23; 50
- Case links 412
- Case marker identification phase 91
- Case markers 121; 122; 158
- Case names 127
- Case Pattern (CP) 141
- Case Pattern Dictionary (CPDiet) 142
- Case pattern 51; 122
- Case Relationships 414
- Case structures 180; 412
- Case systems 128
- Case Theory 120; 127
- Case-Analyzer driver 359
- Case-based semantic analysis 122; 140; 166; 206
- Case-based semantic theory 128
- Case-based text understanding 120
- Case-marker Pattern (CMP) 141
- Case-marker Pattern Dictionary (CMPDiet) 142
- Case-marker Patterns 51; 162
- case_structure 126
- Cause 413
- central domain-specific verbs 179
- central elements 196
- characterization 180
- Classifier 82
- clausal connectors 145
- Clause Level links 412
- Clause Level Relationships 413
- clauses 120; 145
- closeness algorithm 174
- closeness factor CF 165
- closeness metric 149; 158; 160; 168; 359
- CLR definitions 413
- cluster 197
- CM-Case mappings 151
- CM-ranking 167; 174
- CMP 177; 193
- CMPDiet (Case-marker Pattern dictionary) 140
- CMPDiet 179; 194
- CMPs 149; 187; 205
- cognitive science claims 22
- collocations 47
- Common-sense knowledge 4
- Comparatives 133
- Comparison Coefficient 165; 166
- Complex sentences 145
- complex taggers 89

complicated input strings 33
 complicated sentences 204
 composite object 138
 compositional approach 18; 203
COMPOSITIONALITY 15
 compound 98
 compound sentences 96
 computational linguistics 22
 computational paradigm 64
 computerized model 201
 computing distances between patterns 162
 concepts 9
 Conceptual Analyzer 120
 Conceptual Knowledge Processor. 176
 Conceptual Network 9; 178; 197; 412
 conceptual categories 187
 conceptual data model 207
 conceptual distance 197
 conceptual domain model 17
 conceptual model 7; 140; 201
 conceptual ontology 188
 Conjoined adjectivals 139
 Conjoined noun phrases and nominal clauses 138
 Conjoined prepositional phrases 138
 Conjoined sentences 98; 99
 Conjoined subordinate clauses 138
 conjoined items 94
 conjoined sentences with elided object 192
 conjoined structures 94; 116
 conjoined verb phrases 98; 137; 145
 conjoined verbs 134; 145; 191
 conjoined VPs 137
 conjunction 136; 413
CONJUNCTIONS 71
 construction of lexicons 89
 context 31
 contractions 74
 Coordinated clauses 137
 coordination 80
 coordination demons 80
 corpora 46; 202
 corpus 46
 correlative coordination 94
 coverage 115
 coverage and performance 110; 111; 117
 coverage of English grammar 189
 CP 149; 162; 177
 CPDiet (Case Pattern dictionary) 140
 CPDiet 180
 CPs 187; 205
 CPU time limit 100; 112; 114; 115; 324
 customized lexicon 200

DAM 54
 data base schema 35
 data base system 24
 data model 207
 data structures 140

 database 50; 207
 DCG rules 64; 67; 69; 78; 85; 94; 108
 DCG xii; 62; 64; 81
 definite reference 107
 definition of DIPETT's grammar 78
 deictics 107
 density of verbs 96
DETERMINERS 71
 DGs 80
 dictionary of Case-marker Patterns 174
 DIPETT 58; 63; 116; 194
 DIPETT parser 10; 108; 200
 DIPETT's driver 234
 DIPETT's grammar of English 117; 242
 DIPETT's lexicon 71
 DIPETT's non-standard mechanisms 84
 Discontinuous Grammars (DGs) 79
 Discourse relations 137
 Discourse Representation Structures 37
 discourse entities 104
 discourse processing 31; 104; 210
 discourse processing aspects 209
 discourse relation 136
 discriminating set 164
 Disjunction 414
 domain-independent 141
 domain-independent knowledge 51
 Domain-specific knowledge 4
 domain-specific information 35
 driver 69
 dynamic acquisition of knowledge 8
 dynamic dictionary expansion 69

EBL xii; 37; 206
 editor module 185
 efficiency 85
 efficient parsers 26
 elided object 80
ELIZA 44
 ellipsis 32
 embedded relativizers 192
 emphasizers 139
 empty determiner 117
 empty dictionaries 178
 empty HAIKU dictionaries 176
 empty KB 177
 Enablement 413
 endless computation 87
 engineering 7; 18; 202
 English function words 70
 English grammars 62
 English technical texts 140
 Entailment 413
 entirely new pattern 148
 enumerations 116; 190
EPISTOS 40
 erroneous analysis 183
 erroneous full analysis 189

error explanation mechanism 69
 ES xii
 existential-There 135
 expectation-driven parser 205
 expectation-driven parsing 194; 196
 expectation-driven processor 188
 Explanation-based learning xiii
 explicitization 104
 expository prose 14
 expository technical texts 3
 expository texts 196
 extra-grammatical 101; 115
 extra-grammatical input strings 88
 Extraposition Grammars (XGs) 79
 extraposition 79

failure-driven acquisition 53
 feature structures 63
 Fillmore's Case Theory 120
 find_close 149; 151
 fitted parse 33
 focus 31; 197
 fragment processing 33
 fragmentary analysis 189
 fragmentary parsing 92; 100; 101; 102; 115
 fragments 32; 33; 99; 100; 101; 136; 185; 388
 frame instantiation 202
 frequency distributions 47
 frequency statistics 204
 FRUMP 36; 196
 full parsing 49; 114
 functional grammar 64
 functional syntactic representation 67
 fundamental syntactic structure 61

general purpose explanatory verbs 179
 general textual material 205
 general-purpose NL processor 25
 generalization process 36
 generic 29
 generic knowledge 9; 45
 GENESIS 37
 genitives with implicit conjunction 189
 global text analysis 39; 196
 grammar 116
 grammar design 62
 grammar for questions 242
 grammar formalism 61
 granularity parameter 197
 groupings 34; 73; 75; 113

HAIKU 10; 45; 51; 141; 173; 177; 194
 HAIKU Case-based semantic analyzer 200
 HAIKU semantic processor 120
 HAIKU's dictionaries 143
 HAIKU's KB 140
 helper mechanism 85; 87
 Here+be 135; 136

hierarchical knowledge-base assertions 134
 Horn clauses 206
 hybrid pattern 160
 hypernyms 180; 188

idiomatic constructions 70; 73
 idioms 32; 33; 34; 113
 illocutionary markers 210
 imperative statement 135
 implicit subjects and complements 99
 incremental KB 140
 incremental knowledge acquisition 7; 59
 individual-clause Case Analyzer 359
 inference 59
 inflected forms 75
 information-retrieval approach 202
 initial domain-specific KB 16
 initial learning phase 18; 203
 input CMP 148
 input verb 148
 inserted optional modifiers 190
 intelligent assistant i; 8
 intentions 59
 inter-clausal decomposition 148
 Inter-sentential conjunctions 136
 inter-sentential discourse operators 136
 Inter-verbal intervals 96
 interactive parse tree editor 183
 interlinked clauses 145
 INTERPRETATION RULE CADJ 139
 INTERPRETATION RULE CCS 137
 INTERPRETATION RULE CMV 134
 INTERPRETATION RULE CNPNC 138
 INTERPRETATION RULE CPP 138
 INTERPRETATION RULE CPT 133
 INTERPRETATION RULE CSCA 138
 INTERPRETATION RULE CVP 137
 INTERPRETATION RULE HBET 135
 INTERPRETATION RULE IMP 135
 INTERPRETATION RULE ISCJ 136
 INTERPRETATION RULE NC 133
 INTERPRETATION RULE NPPM 132
 INTERPRETATION RULE ST 134
 INTERPRETATION RULE TC 136
 INTERPRETATION RULE VLF 136
 interpretation of the metric's output 172
 intersentential relations 196
 interval-based segmentation 97
 intra-clausal decomposition 148
 Intra-sentential conjunctions 137
 intractability 61
 inverse link name 413
 IRACQ 50
 is-a hierarchy 23
 is-a model 45
 is-a relation 41

joint probability 166

Jurafsky's classification 64

KA bottleneck 2; 13

KA methodology 92

KA xii; xiii

KB accumulated by HAIKU 178

KB text understanding 13

KB xii

KBS xii

KLAUS 36

Knowledge Processing Module 11

knowledge 203

knowledge acquisition bottleneck i

knowledge engineer i; 200

knowledge extraction from texts 180

knowledge-based text understanding 140

knowledge-intensive approach 53

KR xii

lab experiment 176

language generation 204

large-coverage parser 116

LDOCE 54

learning 16

learning curve 177

lexemes 70

lexical acquisition 55; 75

lexical ambiguity 89; 183; 185; 190

lexical analysis 75; 96

lexical analyzer 69

lexical Case marker 121

lexical categories 98

lexical database 74

lexical disambiguation 75; 90

lexical entries 70

lexical knowledge 49; 54

lexical marker 151

lexical processing 74

lexicon 69

LILOG project 26

Linguistic knowledge 4; 5

Linguistic processing 15

Linguistic String Project 26; 35

linguistic coverage 62

linguistic coverage of NLP systems 26

linguistic expressiveness 79

linguistic processing in TANKA 9

list of Cases 129

Location 414

logic grammar formalisms 79

logic grammars 88

LOGOS 42

longest possible substrings 103

look-ahead mechanisms 98

LUKES 42

main goals of this research 12

main limitations 116

main non-terminals 82

maintenance documentation 108

major syntactic types 82

MaLT_e 206

mandative subjunctive 192

Marcus parser 26; 30; 53

match perfectly (PM) 163

MDict (Meaning dictionary) 140

MDict 179; 194

Meaning Dictionary (MDict) 141

meaning triangle 43

memorizer 85

memorizing device 69

meta-comments 139

meta-connector 137

meta-textual element 136

metamorphosis grammars 80

Mini-Net Builder 134

Mini-Network Builder 136

Minimal Attachment 28; 29

minimal knowledge 177

minimal learning phase 177

misattachments 91; 183; 189

misparses 183

ML xii; xiii

MNB 176; 177; 412

modern linguistic theories 63

Modifier Structure Grammars (MSGs) 79

Modifiers 95; 139

morphological analysis 75

morphological processing 74

MSGs 80

MT xii

multiple coordinated sentences 96

multiple coordination 94

n-grams 46; 202

NANOKLAUS 36

Natural language interfaces 24

new elements of knowledge 140

new pattern 148

new_CP_dialogue 151

NL grammars 78

NL xii

NLP and ML techniques 206

NLP xiii

NLU xii

nominal clauses 82; 95; 133

nominal fillers 208

non-DCG formalisms 81

non-discriminating syntactic patterns 180

non-empty intersection set (NPUM) 163

non-proper matching 169

non-proper undermatch 164

non-technical text 208

non-textual material 15

noncommittal surface-syntactic parser 91

not parsable 87

Notes 85
 Noun Modifier links 412
 Noun Modifier Relationships 415
 noun classifiers 95
 noun fillers 187
 Noun-Noun Relationships 128
 NOUNS 71
 NP post-modifiers 132

 off-the-shelf dictionaries 74
 one-Case-per-argument constraint 127
 ontological 29
 ontological knowledge 29
 ontological markers 208
 Opposition 413
 overall coverage 115
 overmatching 162; 168; 173

 PALKA 49
 parse tree 67; 177
 parse tree editor 145; 183
 parse tree repairs 186
 parser 58
 Parsing Statistics 111
 parsing 203
 parsing environment 69; 84
 parsing errors 185
 parsing of conjoined constituents 94
 parsing techniques 61
 parsing time 111; 324
 parsing-related issues 25
 part of speech 70
 partial KA 204
 partial parsing 33; 202
 partial understanding 14
 partially-matching CMPs 162
 particle table 70; 72
 parts of speech 71; 89
 passive chart 69; 85
 past participle constructions 191
 pattern recognition 162
 PC-KIMMO 74
 perfect match 163
 perfect parse tree 91
 perfectly specific CMP 167
 performance 324
 PETRARCA 54
 phrasal verbs 70; 190
 phrase structure rules 79
 plans and goals 59
 portability of NLP systems 5
 positional Case marker 121
 post-Case-analysis processor 412
 Post-classifier 82
 Post-determiner 82
 post-parsing module 120
 post-parsing processes 116
 post-parsing processor 91

 post-tagging filter 90
 post-verbal adverbs 191
 postposed pronoun 191
 PowerHouse 110
 PP attachment 28; 29
 Pre-determiner 82
 pre-parsing tasks 74
 predicate 14
 predicate-argument structure 122
 PREPOSITIONS 71
 Prevention 413
 problematic input strings 92
 processing of conjoined elements 29
 processing of text 22
 production of apt example sentences 158
 pronoun antecedents 31
 pronoun explicitization mechanism 31
 pronoun explicitizer 104; 106; 359
 pronoun reference 107
 PRONOUNS 71
 Proper matching 169
 proper subset (PUM) 163
 proper undermatch 163
 PROPOS 40
 proposition-based analysis 13
 propositions 13
 punctuation 116; 189
 punctuation marks 74
 PUNDIT system 26; 33; 45
 purely linguistic applications 204

 questions 190
 QUIRK 83; 84
 QUIZ 12; 16

 rate of learning 9
 re-attachment facility 100
 real technical texts 58
 realistic sentences 128
 reference 31
 reference in technical prose 32
 referents 104
 reflexive pronouns 189
 regular conjunction 95
 resolution of pronoun anaphora 104
 Right Association 28; 29
 root 70; 75
 rule packet 78

 SATO 55
 scanner 69; 74
 scanning 73; 74
 schema 36
 scientific texts 44
 SCISOR 23
 scope of the semantic analysis 128
 Scoping of negation and adverbs 99
 scripts 196

segment parsing 33
 selectional constraints 45; 48
 selectional patterns 205
 selectional restrictions 47
 semantic analysis 127
 semantic categories 187; 194
 semantic interpretation rules 132
 semantic knowledge 5
 semantic markers 180
 semantic network 122
 semantic pattern 140
 semantic phenomena 59
 semantic relations 127
 semantic role 121
 semantic similarity via syntactic congruence 175
 semantics 5
 semi-automatic approach 8; 200
 semi-automatic assistants for KA 202
 semi-automatic ii
 semi-automatic learning system 14
 sensible parse tree 92
 Sentence modifier 83
 sentence modifiers 138
 sentential relative clause 191
 set of 28 Cases 129
 set of Cases 127
 SEXTANT 47
 short stories 37
 simple and complex sentences 96
 simple coordination 94
 single clauses 145
 SINTESE 47
 situations (#1-#5) 148
 skip 81
 SNePS 52
 SORTAL 43
 sortal analysis 43
 source text 16
 SPECIAL 71
 speech acts 31; 59; 209
 speed of semantic analysis 177
 SPQR 45
 statistical approach 202
 statistical basis 46
 statistical models 47
 stative main verb 134
 stories 8; 196; 208
 story grammar 196
 structural ambiguity 29; 91
 stylistic constructions 192
 subcategorization frames 47
 sublanguage 48
 substantial subset of a NL 61
 successful coverage 114
 surface-syntactic 59
 surface-syntactic coverage 82
 surface-syntactic lexicon 70
 symbolic pattern matching 162
 syntactic analyzer 69
 syntactic clues 49; 51
 syntactic indicators of semantic relations 162
 syntactic knowledge 52
 syntactic module 78; 82
 syntactic units of English 82
 syntactico-semantic clues 193
 syntactico-semantic similarity 166
 syntax 5; 59
 syntax-first 15; 48
 systematic methodology 202
 T4043 text 112
 table of CM values 167
 tagger 69; 89
 TANKA i; 7; 36; 39; 43; 45; 48; 49; 62; 120; 122; 176; 196; 197; 200; 206; 412
 TANKA system 7
 target CP 142
 Tax Guide 111
 taxonomic knowledge 41
 TEAM 50
 technical documentation 8
 technical English 59; 108; 116
 technical manual 200
 technical prose 8; 14
 technical texts 180; 189
 TELI 49
 Temporal Precedence 414
 temporal ordering 134
 tense table 78; 242
 test cases 108
 testing and fine-tuning 88
 testing of HAIKU 176
 Text Base Manager 10
 Text summarization 196
 text fragments 128
 text retrieval system 24
 text segmentation 33; 49
 text skimming 33
 The 1990 Canadian T4043E Tax Guide 111
 The Collins 12; 54
 The Collins dictionary 70; 74; 89
 the DIPET¹ parser 64
 the QUIZ manual 63; 110; 112; 331
 the QUIZ text. 99
 the T4043 Tax Guide 331
 the Tax Guide 63
 thematic expectations 51
 thematic knowledge 51
 thematic roles 31; 51
 theoretical foundation of this research 12
 theory-neutral 64
 Time 414
 timing out mechanism 94
 title 136
 Tomita's parsing algorithm 61
 top-level rule selection mechanism 96; 97

- top-level rules 96
- topic 197
- topic analyzer 24
- topic attributes 196
- topic indicators 196
- topic-focus 31
- TP systems 22
- TP xii; xiii
- trace 79
- trace mechanism 69; 88
- type-0 languages 79

- ubiquitous CMP and CP patterns 180
- undermatch 162
- undermatching 162; 168; 173
- ungrammatical input strings 88
- universal principles 63
- universal recognition problem 61
- unknown words 75; 76; 113
- unparsable input strings 92
- useful parsers 87
- user 177; 183
- user interaction 69
- user interface 70; 122; 183; 204
- user of TANKA 16

- verbal density 96
- VERBS 71
- very general knowledge base 205
- very large corpora 47
- vocabulary 200

- weights 172; 173
- well-formed substring table 69; 85
- well-formed substrings 85
- well-formedness 117
- word collocations 46; 202
- WordNet 12; 74; 180; 188

- XGs 79

