Internal use only

002

TR-IT-0129

An Interactive Disambiguation Module for English Input: an Engine and the Associated Lingware

Hervé BLANCHON

1995.8.25

An interactive disambiguation methodology has been proposed and implemented at the GETA lab. in the framework of Dialogue-Based MachineTranslation. This methodology has been generalized and re-engineered at ATR-ITL in the framework of spoken language transtlation and the MIDDIM project, a joint research between ATR-ITL and the GETA-CNRS aimed to study Multimodal Interactive Disambiguation.

The proposed disambiguation methodology is based on the manipulation of tree structures. A kind of ambiguity is described with a set of patterns called a beam. A pattern contains variables and descibes a tree structure with constraints on its geometry and labelling. Once a beam has been recognized, a question is prepared. The question items are produced through the manipulation, with a set of basic operators, of the values given to the variables instanciated during the recognition of the beam.

A particular disambiguation module is described with a lingware which is language and analyzer dependant. This lingware is then used as input data to an interactive disambiguation engine so as to describe a particular instance of a running disambiguation module. In this paper we are going to describe the engine and the lingware we have developed at ATR-ITL for English input.

ATR – Interpreting Telecommunications Research Laboratories
2–2 Hikari-dai, Seika-cho, Soraku-gun, Kyoto 619-02, Japan

© 1995 by ATR Interpreting Telecommunications Research Laboratories

Content

INTRODUCTION 1

PART I THE DISAMBIGUATION ENGINE

INTRODUCTION 13

CHAPTER 1. PATTERN MATCHING & BEAM MATCHING MECHANISMS 15

1.1. Pattern matching 16
1.1.1. Patterns description language 16
1.1.2. Pattern recognition mechanism 17

1.2. The beam matching mechanism 19
1.2.1. Beams and stacks 19
1.2.2. Beam matching: formal description 20
1.2.3. Beam matching: implementation 21

1.3. Comments 24

CHAPTER 2. CONSTRUCTION AND PRESENTATION OF A QUESTION TREE 27

2.1. Disambiguation automaton 28

2.2. Question tree construction 30
2.2.1. Strategy 30
2.2.2. Implementation 31
2.2.3. Construction of a question 32

2.3. Question tree presentation 33

2.4. Comments 33

CHAPTER 3. DIALOGUE & QUESTION CLASSES 35

3.1. A generic disambiguation question class 36

3.2. A generic textual disambiguation dialogue class 36

3.3. Comments 38

CHAPTER 4. OPERATORS 39

4.1. Selective projection 40
4.1.1. Definitions (selection) 40
4.1.2. Implementation (selection) 40

4.2. Acces to the Multilingual Lexical Data Base 42
4.2.1. Definition 42
4.2.2. Implementation 42

4.3. Other operations 42
4.3.1. Definition 42
4.3.2. Implementation 42
4.5. Comments 43

PART II THE ENGLISH LINGWARE

INTRODUCTION 47

CHAPTER 5. AMBIGUITIES, PATTERNS & PATTERN BEAMS 49

5.1. Situation 50

5.2. English pattern-defined ambiguities 50
5.2.1. Second verbal-phrase prepositional attachment 50
5.2.2. Simple adverbial attachment 51
5.2.3. Verbal Phrase prepositional attachment 51
5.2.4. Relative verbal phrase adverbial attachment 52
5.2.5. Verbal-phrase conjunction attachment 53
5.2.6. Non-verbal-phrase prepositional attachment 54

5.3. English listed but not solved ambiguities 56
5.3.1. noun-adjective ambiguity 56
5.3.3. Other syntactic class ambiguities 56
5.3.2. Phrasal verb ambiguity 57

5.4. Comments 58

CHAPTER 6. CLARIFICATION AUTOMATON 59

6.1. The beam matching ambiguity recognition states 60

6.3. The implemented automaton 61

6.3. Comments 61

CHAPTER 7. DIALOGUES CLASSES 63

7.1. Language dependent constraints 64

7.2. Dialogue classes 64

7.3. Comments 65

CHAPTER 8. DIALOGUE ITEM PRODUCTION METHODS 67

8.1. Principle 68

8.2. Methods skeleton 68

8.3. Comments 69

CONCLUSION 71

REFERENCES 75

APPENDIX A PATTERNS, BEAMS, & STACKS DEFINED 81

APPENDIX B METHODS 91

APPENDIX C PRODUCED DIALOGUES 101

APPENDIX D ORGANIZATION OF THE SOFTWARE 107

Figures

Figure 1: General idea of an archirecture for clarification modules 4
Figure 2: A particular instance of a disambiguation module 4
Figure 3: An example of an mc-structure (multilevel, concrete) 5
Figure 4: A disambiguation moduke's general architecture 7
Figure 5 : A disambiguation module's engine-defined part 13
Figure 6: An example of an mc-structure (multilevel, concrete) 14

Figure 1.1: Matching mechanisms in the global architecture 15
Figure 1.2: The pattern description grammar 16
Figure 1.3: The engine class pattern 17
Figure 1.4: 2 patterns forming a beam 17
Figure 1.5: Some examples of pattern matching results 18
Figure 1.6: The engine class pattern-beam 19
Figure 1.7: The engine class beam-stack 19
Figure 1.8: Beam matching definition 20
Figure 1.9: Matching distance definition 20
Figure 1.10: match-beam input and output 21
Figure 1.11: The synchronisation of the different pattern matchings 22
Figure 1.12: The engine method match-beam 23
Figure 1.13: A multiple data-structure disambiguation module architecture 24
Figure 1.14: Automatic learnig of new beams 25

Figure 2.1: Construction & presentation of a question tree in the global architecture 27
Figure 2.2: A question tree to discriminate 5 solutions 28
Figure 2.3: The engine method automaton-scheduler 28
Figure 2.4: General organisation of a disambiguation automaton 29
Figure 2.5: The engine method same-categories-p-state 30
Figure 2.6: The engine method same-geometry-p-state 30
Figure 2.7: The engine method prepare-question-tree 31
Figure 2.8: The engine method prepare-question-list 32

Figure 2.9: The engine method prepare-question 32
Figure 2.10: The engine function produce-item 32
Figure 2.11: The engine method question-tree-presentation 33
Figure 2.12: The engine method ask-question 33
Figure 2.13: Better organisation of the reentry in the desambiguation automaton 34

Figure 3.1: Dialogue & question classes in the global architecture 35
Figure 3.2: The engine class clarification-question-class 36
Figure 3.3: The engine class clarification-question-class 36
Figure 3.4: The engine class generic-textual-clarif-dialogue-class 37

Figure 4.1: The operators in the global architecture 39
Figure 4.2: The engine operator text 40
Figure 4.3: The engine sub-operator text-dans-stream 40
Figure 4.4: The engine operator coord 41
Figure 4.5: The engine operator but-coord 41
Figure 4.6: The engine sub-operator moins-coordonnant-dans -stream 41
Figure 4.7: The engine sub-operator disgard-coordonnant 41
Figure 4.8: The engine operator distribute 42
Figure 4.9: The engine sub-operator distribue-dans-stream 42
Figure 4.10: The engine sub-operator distribue-pattern-dans-stream 43
Figure 4.11: The engine operator bracket 43
Figure 4.12: The engine sub-operator parenthese-dans-stream 43
Figure 4.13: A language independant operator coord 43

Figure 7: Components of the English disambiguation lingware 47

Figure 5.1: The English patterns, beams and stacks in the global architecture 49
Figure 5.2: Mmc-strctuture for "Let me pull out my maps to help you." 50
Figure 5.3: The patterns for a Second phvb prepositional attachment ambiguity 51
Figure 5.4: Mmc-structure for "You can pay for it right on the bus." 51
Figure 5.5: Mmc-structure for "It says that here on my flyer." 51
Figure 5.6: The patterns for a Simple adverbial attachment ambiguity 51
Figure 5.7: Mmc-structure for "Where can I catch a taxi from Kyoto station." 52
Figure 5.8: The patterns for a phvb prepositional attachment type 1 ambiguity 52
Figure 5.9: Mmc-structure for "Go across the street to the North of the station." 52
Figure 5.10: The patterns for a phvb prepositional attachment type 2 ambiguity 52
Figure 5.11: Mmc-structure for "That is where you can pick up a taxi as well." 53
Figure 5.12: The patterns for a relative phvb adverbial attachment type 1 ambiguity 53
Figure 5.13: Mmc-structure for "I will show you where you are located right now." 53
Figure 5.14: The patterns for a phvb prepositional attachment type 2 ambiguity 53
Figure 5.15: Mmc-structure for "You can tell him that you are going to the international conference center and it should be a twenty minute ride." 54
Figure 5.16: The patterns for a Verbal-phrase conjunction attachment ambiguity 54
Figure 5.17: Mmc-structure for "You are going to the international conference center."55

Figure 5.18: The patterns for a non phvb prepositional attachment type 1 ambiguity ..55
Figure 5.19: Mmc-structure for "I want the symposium on interpreting telecommunication at the international conference center." 55
Figure 5.20: The patterns for a non phvb prepositional attachment type 2 ambiguity ..56
Figure 5.21: Mmc-structure for "This is an English speaking agent." 56
Figure 5.22: Mmc-structure for "You can catch a taxi at the second level platform." ..56
Figure 5.23: Mmc-structure for "The quickest route would be taking a taxi." 57
Figure 5.24: Mmc-structure for "You can either travel by subway, but or taxi." 57
Figure 5.25: Mmc-structure for "It is difficult to get out of Kyoto station." 57
Figure 5.26: Mmc-structure for "Do you want to go over that again." 57

Figure 6.1: The clarification automaton in the global architecture 59
Figure 6.2: Skeleton of an ambiguity recognition state 60
Figure 6.3: The implemented disambiguation automaton 61
Figure 7.1: The English dialogue classes in the general architecture 63
Figure 7.2: Some dialogues' slots 64
Figure 7.3: The lingware class english-general-textual-dialog-class 64
Figure 7.4: The lingware class english-polysemy-textual-dialog-class 65

Figure 8.1: The dialogue item production methods in the global architecture 67
Figure 8.2: A typical item-production-method method 68

Figure 8: Evolution of the coverage of a given disambiguation module 73

Figure A.1: The pattern *2phvbadvatt-1* 82
Figure A.2: The pattern *2phvbadvatt-1* 82
Figure A.3: The beam *2phvbadvatt_set_1* 82
Figure A.4: The stack *2phvbadvatt_beam_stack* 82
Figure A.5: The pattern *2phvbadvatt-1* 83
Figure A.6: The pattern *2phvbadvatt-1* 83
Figure A.7: The beam *2phvbadvatt_set_1* 83
Figure A.8: The stack *2phvbadvatt_beam_stack* 83
Figure A.9: The pattern *2phvbadvatt-1* 84
Figure A.10: The pattern *2phvbadvatt-1* 84
Figure A.11: The beam *2phvbadvatt_set_1* 84
Figure A.12: The pattern *2phvbadvatt-1* 85
Figure A.13: The pattern *2phvbadvatt-1* 85
Figure A.14: The beam *2phvbadvatt_set_1* 85
Figure A.15: The stack *2phvbadvatt_beam_stack* 85
Figure A.16: The pattern *2phvbadvatt-1* 86
Figure A.17: The pattern *2phvbadvatt-1* 86
Figure A.18: The beam *2phvbadvatt_set_1* 86
Figure A.19: The pattern *2phvbadvatt-1* 87
Figure A.20: The pattern *2phvbadvatt-1* 87
Figure A.21: The beam *2phvbadvatt_set_1* 87

Figure A.22: The stack *2phvbadvatt_beam_stack* 87
Figure A.23: The pattern *2phvbadvatt-1* 88
Figure A.24: The pattern *2phvbadvatt-1* 88
Figure A.25: The beam *2phvbadvatt_set_1* 88
Figure A.26: The stack *2phvbadvatt_beam_stack* 88
Figure A.27: The pattern *2phvbadvatt-1* 89
Figure A.28: The pattern *2phvbadvatt-1* 89
Figure A.29: The beam *2phvbadvatt_set_1* 89
Figure A.30: The pattern *2phvbadvatt-1* 90
Figure A.31: The pattern *2phvbadvatt-1* 90
Figure A.32: The beam *2phvbadvatt_set_1* 90
Figure A.33: The stack *2phvbadvatt_beam_stack* 90

Figure B.1: Item-prod-method ((pat-name (eql '*2phvbadvatt-1*)) binding) 92
Figure B.2: Item-prod-method ((pat-name (eql '*2phvbadvatt-2*)) binding) 92
Figure B.3: Item-prod-method ((pat-name (eql '*spladvatt-1*)) binding)....................93
Figure B.4: Item-prod-method ((pat-name (eql '*spladvatt-2*)) binding)....................93
Figure B.5: Item-prod-method ((pat-name (eql '*phvbprepatt-t1-1*)) binding) 94
Figure B.6: Item-prod-method ((pat-name (eql '*phvbprepatt-t1-2*)) binding) 94
Figure B.7: Item-prod-method ((pat-name (eql '*phvbprepatt-t2-1*)) binding) 95
Figure B.8: Item-prod-method ((pat-name (eql '*phvbprepatt-t2-2*)) binding) 95
Figure B.9: Item-prod-method ((pat-name (eql '*relphvbadvatt-t1-1*)) binding)....................96
Figure B.10: Item-prod-method ((pat-name (eql '*relphvbadvatt-t1-2*)) binding)....................96
Figure B.11: Item-prod-method ((pat-name (eql '*relphvbadvatt-t2-1*)) binding)....................97
Figure B.12: Item-prod-method ((pat-name (eql '*relphvbadvatt-t2-2*)) binding)....................97
Figure B.13: Item-prod-method ((pat-name (eql '*phvbconjatt-1*)) binding) 98
Figure B.14: Item-prod-method ((pat-name (eql '*phvbconjatt-2*)) binding) 98
Figure B.15: Item-prod-method ((pat-name (eql '*nphvbprepatt-t1-1*)) binding) 99
Figure B.16: Item-prod-method ((pat-name (eql '*nphvbprepatt-t1-2*)) binding) 99
Figure B.17: Item-prod-method ((pat-name (eql '*nphvbprepatt-t2-1*)) binding) 100
Figure B.18: Item-prod-method ((pat-name (eql '*nphvbprepatt-t2-2*)) binding) 100

Figure C.1 : Dialogue for "Let me pull up my maps to help you." 101
Figure C.2 : Dialogue for "You can pay for it right on the bus." 102
Figure C.3 : Dialogue for "It says that here on my flyer."....................102
Figure C.4 : Dialogue for "Where can I catch a taxi form Kyoto station." 102
Figure C.5 : Dialogue for "Go across the stree to the North of the station." 103
Figure C.6 : Dialogue for "This is where you can pick up a taxi as well." 103
Figure C.7 : Dialogue for "I wil show you where you are located right now."....................103
Figure C.8 : Dialogue for "You can tell hin that you are going to the international conference center and it should be a twenty minutes ride." 104
Figure C.9 : Dialogue for "You are going to the international conference center." 104
Figure C.10 : Dialogue for "I want the symposium on interpreting telecommunications at the international conference center." 105

Introduction

Situation, motivation

Natural language processing components are introduced in a growing number of software. Natural language can be seen as:

- the core data to be manipulated as in machine translation system, spelling and grammar checker, etc.
- a communication modality between the user and the system itself as in multi-modal drawing tools [Caelen 1994 ; Hiyoshi & Shimazu 1994 ; Nishimoto, *et al.* 1994], oral control systems, etc.

In some system the use of natural language may fall in both categories as, for example, in on-line information retrieval [Haddock 1992 ; Zue, *et al.* 1993 ; Goddeau, *et al.* 1994], and face to face translation systems [Morimoto, *et al.* 1992 ; Kay, *et al.* 1994]

Natural language processing techniques and tools have made many progress in the last few years. But so far they are facing major issues in two areas: speech recognition and natural language analysis. Central issue in speech recognition research is recognition accuracy according to vocabulary size, speaker-dependency, continuous speech, and task characteristics. Central issue in natural language analysis is the development of broad coverage analyzers able to produce accurate representations of *real* utterances.

For real applications, the bests results are achieved when speech, vocabulary (size and meaning), syntax, semantics, and pragmatics are well controlled. Experience has shown that, even when those parameters are well controlled, the data to be manipulated and analyzed could be ambiguous. Thus, it can be impossible for a system to compute the actual "meaning" of a natural language input. In this situation the system's answer is likely not to be the expected one.

Interactive disambiguation is seen as a solution to overcome the difficulties the analysis modules to be used in **real running systems**, and not small mockups, will ever face. It is often argued that interactive disambiguation is not necessary because heuristics and statistics are to be used to build analyzers producing an accurate enougt output . Is is has been shown that it is not the case (cf. discussion and pointers in [Boitet & Tomokiyo 1995]).

Of course we do not mean that interactive disambiguation has to be mandatory for each sentence or phrase to be analyzed but we consider this interactive process more as a *safety net* to be used when is it necessary to avoid any error in the system answers. Thus, five years ago we began a study on interactive disambiguation. Taking into account the preceeding results and failures in the field we gave ourself two constaints:

- interactive disambiguation dialogue can be presented only to the "author" of the utterance to be disambiguated.
- to be answered, interactive disambiguation dialogues should require neither linguistic knowledge, nor knowledge of the internal repersentations used by system.

In the architecture we propose, an interactive disambiguation module is made of two component:

- An engine, that is the core of the module and is language independant. This component will be used in all the disambiguation modules to be developped.
- A lingware, that is language dependant. This component is considered as input data to the engine so as to instanciate a particular disambiguation module for a given language.

The following figure gives an idea of the global architecture.

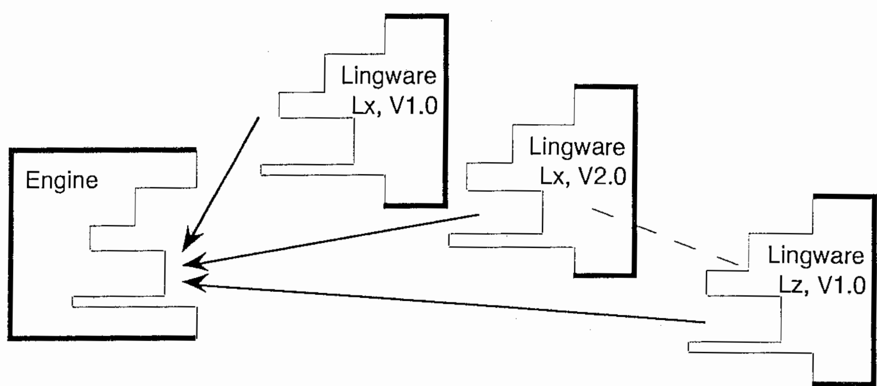


Figure 1: General idea of an archirecture for clarification modules

As shown bellow, an instance of a disambiguation module is the association of a lingware and the engine to process a representation of an ambiguous utterance and to produce a set of disambiguation questions.

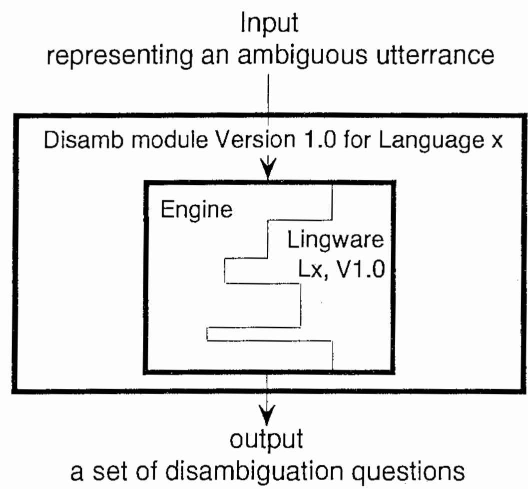


Figure 2: A particular instance of a disambiguation module

Idealy, we would like to provide the designer of a disambiguation module with a set of tools allowing him, at least, to describe:

- the ambiguities to be solved
- the labelling of the dialogues to be proposed to solve these ambiguities
- the order in which the ambiguities have to be solved, if several are present in the same utterance,
- the modalities to be used to solve each ambiguity,
- the modalities to be used to answer the questions about each ambigity,
- the way questions should be prepared and answered.

These descriptions are called the lingware.

The engine is then supposed to use the lingware to realize an interactive disambiguation process. Idealy it should provide:

- an ambiguity recognition mechanism to be used to recognize the ambiguities descibed by the designer,
- a set of operators to be used to describe the construction of the labelling of the dialogues,
- a kind of automaton mechanism to realize the ordering in the ambiguity recognition process,
- predefined dialog classes corresponding to the possible modalities,
- a question presentation mechanism,
- a set of question preparation and display strategies.

In the framework of this architecture, a first interactive disambiguation module has been proposed and implemented at the GETA lab. as a part of the LIDIA-1.0 mockup of a Dialogue-Based Machine Translation system [Blanchon 1994c ; Boitet & Blanchon 1995]. This first interactive disambiguation module was made of:

- a first version of an interactive disambiguation engine [Blanchon 1992 ; Blanchon 1994b],
- a first version of a lingware for French written input [Blanchon 1994b].

The analyzer used in the LIDIA project produces tree structures called mmc-structures (multisolution, multilevel, and conctrete). Multisolution means that the analyser produces every analysis fitting with the syntagmatic, syntactic and logico-semantic model of the grammar. Fig. 3 shows an example for the sentence "The student calcul this integral by the method of residue."

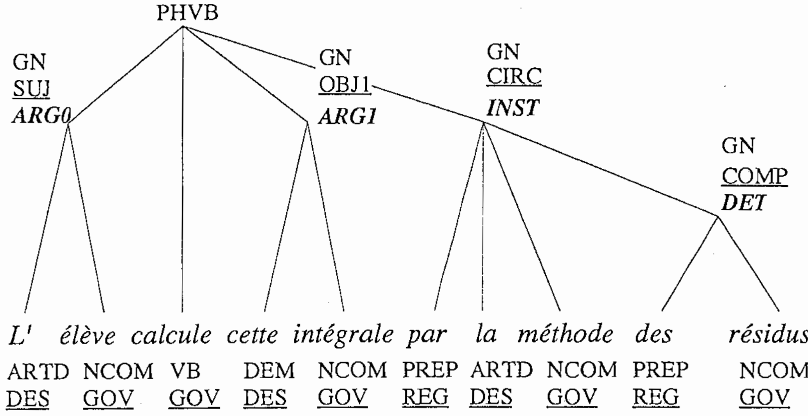


Figure 3: An example of an mc-structure (multilevel, concrete)

Multilevel means that the same structure consists of three levels of linguistic interpretation, namely the level of syntactic and syntagmatic classes, the level of syntactic functions and the level of logic and semantic relations. Finally, the structure is said to be concrete because the original utterance can be found back by a simple left-to-right reading of the structure.

The ambiguities are then descibed in terms of tree structures and the ambiguity recognition mechanism proposed is manipulating tree structures. More precisely, a kind of ambiguity is described with a set of patterns called a beam. A pattern contains variables and descibes a tree structure with constraints on its geometry and labelling. Once a beam has been recognized, a question is prepared. The question items are produced through the manipulation, with a set of basic operators, of the values given to the variables instanciated during the recognition of the beam.

The methodology has been generalized and re-engineered at ATR-ITL in the framework of spoken language transtlation and the MIDDIM project, a joint research between ATR-ITL and the GETA-CNRS aimed to study Multimodal Interactive Disambiguation. The first version of an interactive disambiguation module (cf also [Blanchon & Loken-Kim 1994 ; Blanchon, *et al.* 1995b]) has been developped which is made of:

- a new version of an interactive disambiguation engine,
- the first version of a lingware for English input.

The following figure, describing the implementation of the English disambiguation module, will be presented at the begining of each chapter with the irrelevant parts shaded. This is to help the reading by pointing out the integration in the overall process of each chapter's topic.

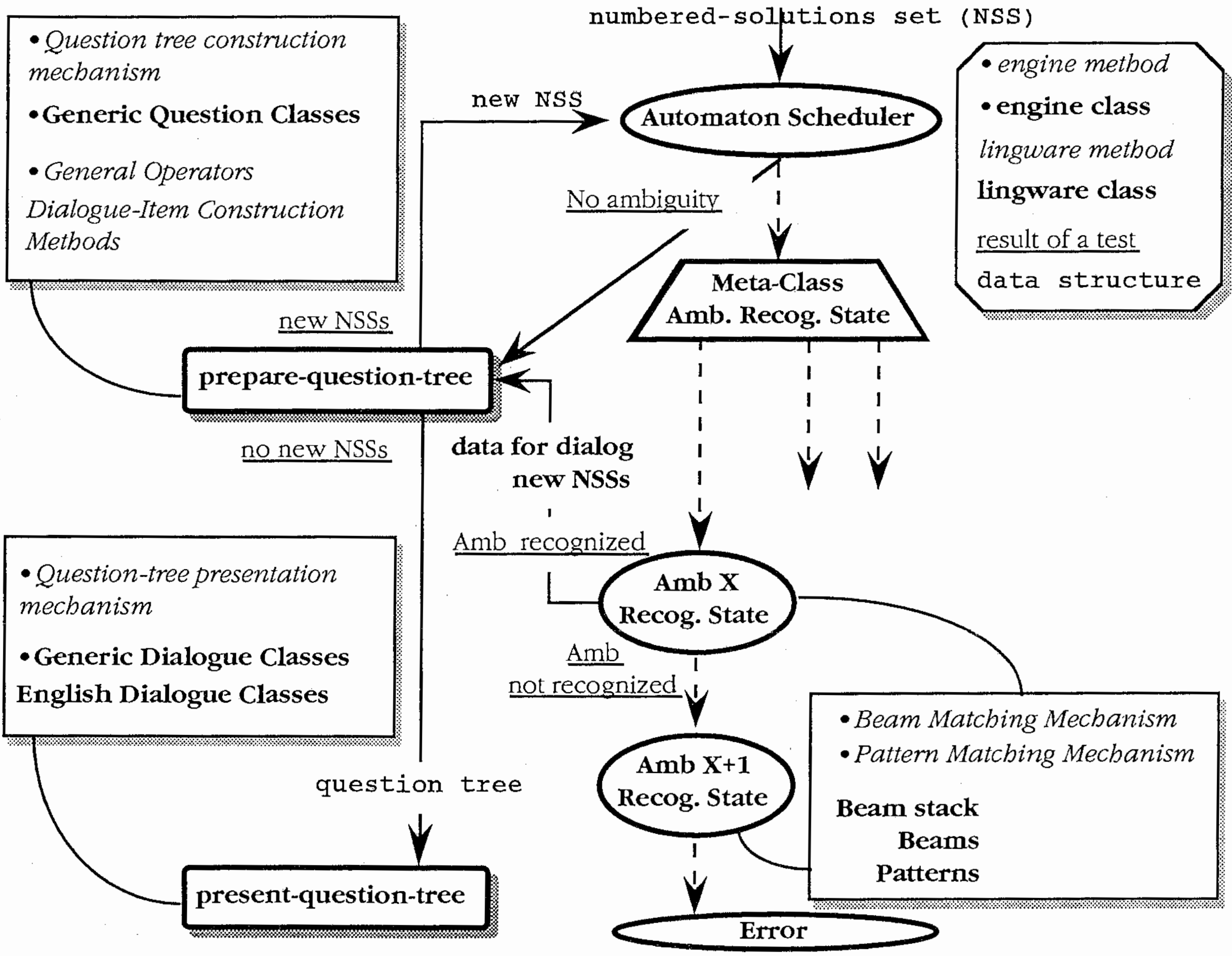


Figure 4: A disambiguation moduke's general architecture

As far as the overall disambiguation process is concerned, the core component is an automaton used to organize the detection of all the ambiguities occuring in the utterances to be disambiguated. This automaton is made of tree kind of states:

- an automaton scheduler,
- ambiguity-meta-class recognition states,
- ambiguity-class recognition states.

The ambiguity recognition states are defined by the designer of the lingware. They use the following services:

- a beam-matching mechanism, which uses:
- a pattern matching mechanism operating on tree structures, and
- a set of
  - patterns, grouped to from
  - pattern-beams, grouped to form
  - beam-stacks.

The automaton is recursivelly re-entered through the automaton-scheduler state until there is no more ambiguity to be solved. The recursive re-entrence is organized by the prepare question tree module. This module uses the following services:

- a question tree construction mechanism,
- a set of generic question classes,
- a set of general operators, and
- a set of dialogue-item construction methods.

When there is no more ambiguity to be solved tha question tree has been completed. This question tree is then presented to the user by the question tree presentation module. This module uses the following services:

- a question-tree presentation mechanism,
- a set of generic dialogue classes, and
- a set of English dialogue classes.

As far as the implementation is concerned, the described engine and English lingware have been realized in Macintosh Common Lisp V 2.0.1, an implementation of the Common Lisp Object System [Steele 1990 ; Keene 1989]. The only Macintosh dependant source code is the one used for the display of the dialogues. This code is calling the Macintosh toolbox routines.

Organisation of the document

In its current state, the engine is made thus up of:

- a pattern matching mechanism and a beam matching mechanism,
- a question tree construction mechanism and a question tree presentation mechanism,
- a set of generic dialogue and question classes, and
- a set of operators to be used to produce the dialogue items.

As far as the English lingware is concerned, it is made up of:

- patterns and patterns beams describing the handled ambiguities,
- an automaton organizing the order the ambiguities are searched in the input,
- a set of English specific dialogue classes, and
- a set of dialogue item construction methods used to produce the disambiguation dialogues.

The first part of this document, dedicated to the desciption of the engine, consists of four chapters decicated respectively to the components of the engine listed above. The second part, deficated to the English lingware, consists of four chapters dedicated respectively to the components of the lingware listed above. An exhaustive desciption of the lingware is given in the appendices.

Part I

The disambiguation engine

The disambiguation engine components are shown in plain style in the following figure.

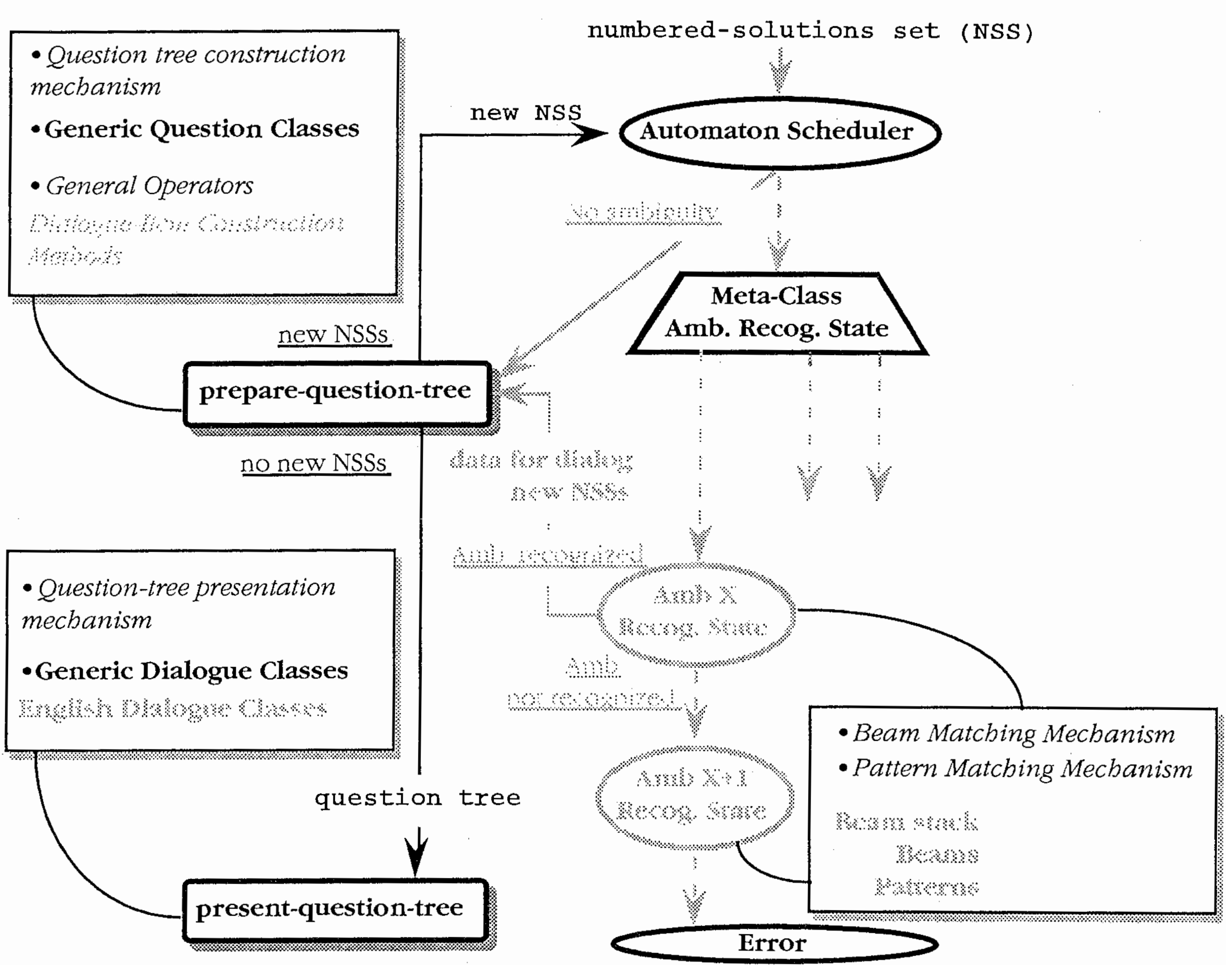


Figure 5 : A disambiguation module's engine-defined part

The disambiguation engine, is the language independent part of an interactive disambiguation module. This engine should thus be reused in each disambiguation module to be developped with the proposed methodology. It provides ambiguity recognition facilities as well as

disambiguation question construction and presentation facilities.

The ambiguities are represented in terms of tree strucures. Indeed, as far as the structure produced by the parser is concerned, in the study carried out so far we have been using a multisolution, mulltilevel and concrete tree structure.

Multisolution means that the analyzer produces every analysis fitting with the syntagmatic, syntactic and logico-semantic model of the grammar.

Multilevel means that the same structure consists of three levels of linguistic interpretation, namely the level of syntactic and syntagmatic classes, the level of syntactic functions and the level of logic and semantic relations.

Finally, the structure is said to be concrete because the original utterances can be reconstructed by a simple left-to-right reading of each analysis tree (mc-structure) occurring in the mmc-structure.

The following figure is the multilevel and concrete structure produced for the sentence "L'élève calcule cette intégrale par le méthode des résidus[1]."

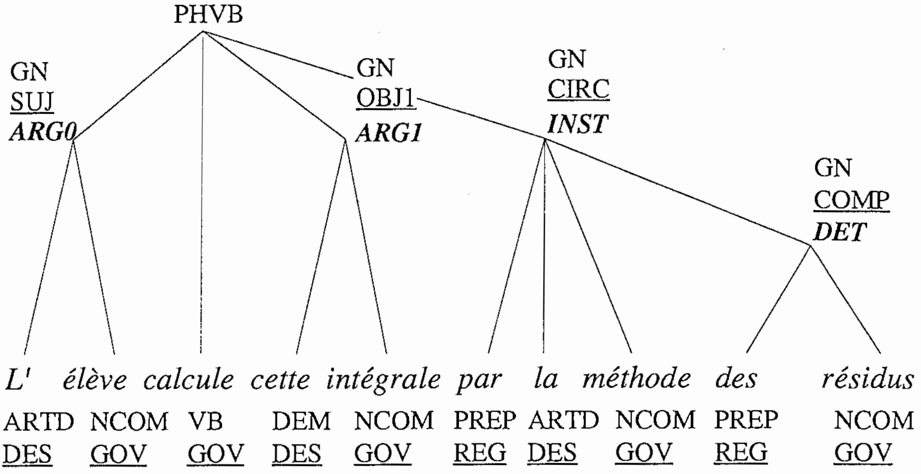


Figure 6: an example of an mc-structure (multilevel, concrete)

We will discuss the possible use of other structures in the §1.4.

The disambiguation engine is made of:

- a pattern matching mechanism and a beam matching mechanism which are the ambiguity recognition facilities,
- a question tree construction mechanism and a question tree presentation mechanism,
- a set of generic dialogue and question classes, which are the representation of the objects involved in the disambiguation process, and
- a set of operators to be used to produce the dialogue items, which are the dialogue item construction facilities.

Thoses elements will be described in the four following chapters. At the end of each chapter, we will discuss the ideas and the implementation proposed. We will also present new ideas and possible or required improvments of the current implementation.

---

1 The student calculate this integral by the method of the residue.

Chapter 1

Pattern matching & beam matching mechanisms

As shown bellow, the pattern matching and the beam matching mechanisms are used as facilities by the ambiguity recognition states to actually recognize the presence of an ambiguity.

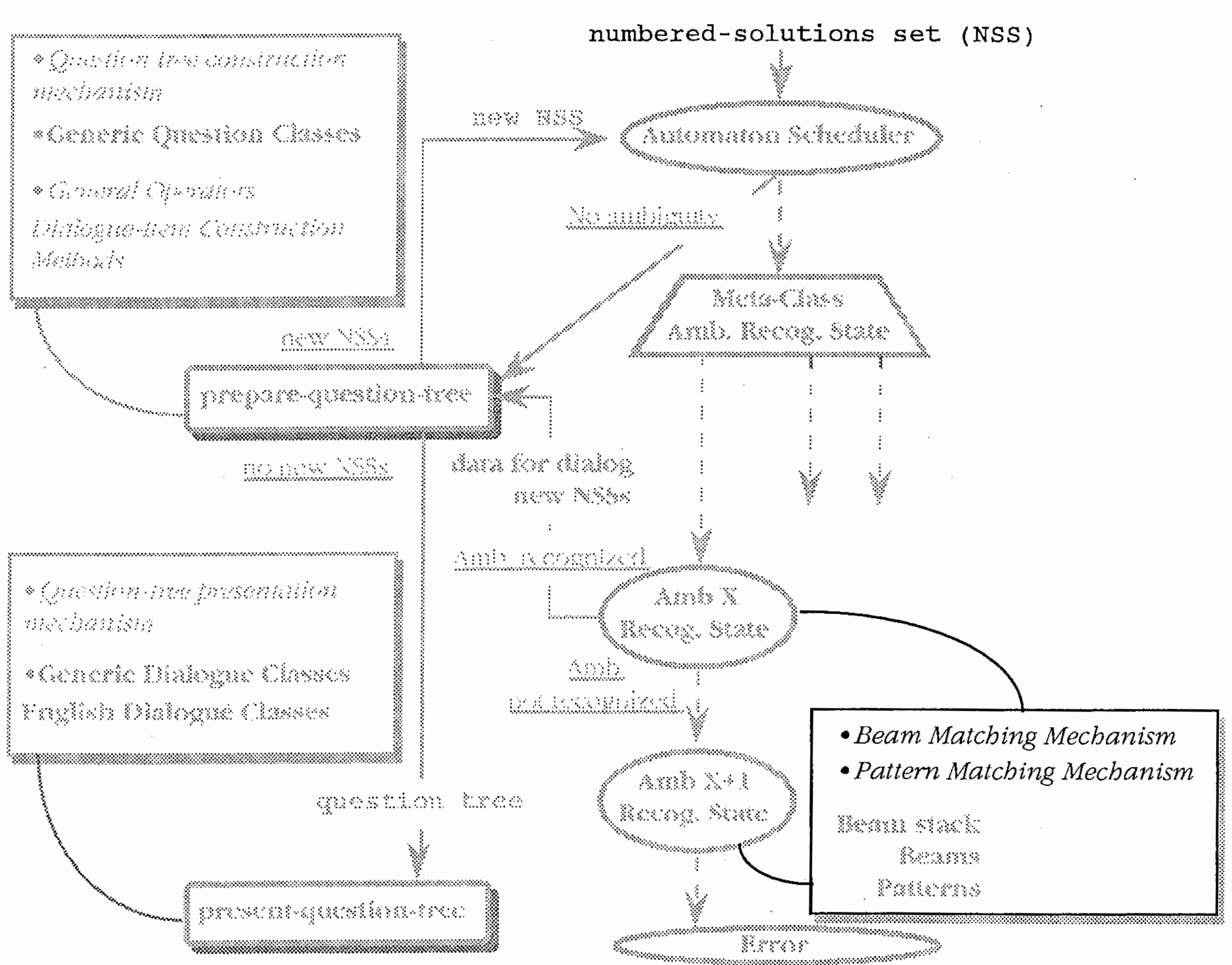


Figure 1.1: Matching mechanisms in the global architecture

1.1. Pattern matching

The recognition of an ambiguity is based, at the lower level, on the recognition of tree structures. Those tree structures are called patterns. A pattern, is the description with a pattern description language, of a tree structure with constraints on the geometry and the labelling of the nodes.

1.1.1. Patterns description language

The patterns are described with a language derived from the one proposed in [Norvig 1992]. Here is the BNF description of the pattern description language.

i. Grammar

```
pattern          ::=  variable                |
                      ; match every expression
                      constant                |
                      ; match only the atom (constant)
                      segment-pattern         |
                      ; match a segment
                      simple-pattern          |
                      ; match an expression
                      (pattern . pattern)
                      ; recognize the first characterthen the rest
simple-pattern   ::=  (?is variable pred args)    |
                      ; check the predicate pred (cf ii. comments on the grammar)
                      (?or pattern...)            |
                      ; match one of the different character
                      (?and pattern...)           |
                      ; match all the patterns
                      (?not pattern...)
                      ; match if the patterns are not recognized
segment-pattern  ::=  ((?* variable) ...)         |
                      ; match zero or more expressions
                      ((?+ variable) ...)         |
                      ; match one or more expression
                      ((?? variable) ...)         |
                      ; match zero or one expression
                      ((?if expression) ...)
                      ; check if the expression (which may contain variables) is true
variable         ::=  ?character+
                      ; ? followed by letters
constante        ::=  atom
                      ; an atom
```

Figure1.2: The pattern description grammar

For the simple-pattern "(?is variable pred args)" the predicate "pred" is applied on the arguments "variable" and "args" namely: pred (variable args).

ii. Internal representation

For the disambiguation process, a particular pattern is an instance of class called **pattern**. This class is defined as follows:

```
(defclass pattern ()
  ((pattern-name :initarg :pattern-name
                 :accessor pattern-name
                 :documentation "")
   (pattern-value :type list
                  :initarg :pattern-value
                  :initform nil
                  :accessor pattern-value
                  :documentation "")
   (pattern-method :type function
                   :initarg :pattern-method
                   :accessor pattern-method
                   :documentation "item production method associated with the pattern")))
```

Figure 1.3: The engine class pattern

An object **pattern** is defined with 3 slots:

- a pattern-name that is the external name of the pattern,
- a pattern-value that is the actual description of the pattern, in terms of its structure and labelling constraints, that will be used in the matching step, and,
- a pattern-method is a clos method that will be used to produce the dialogue item associated with recognized pattern in the dialogue to be presented to the user.

iii. Illustration

A **pattern** (as show in the figure bellow) describes a family of trees, with constraints on their geometry and labelling.

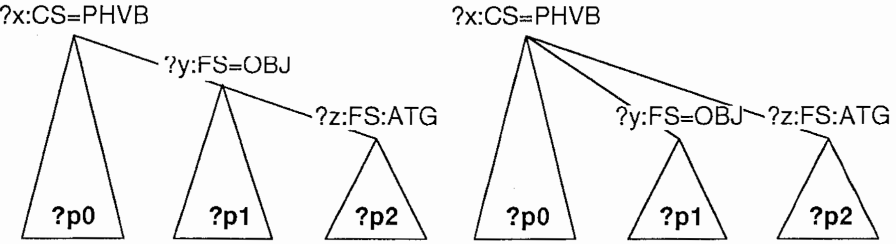


Figure 1.4: 2 patterns forming a beam

A **pattern** contains two kinds of variable: node variables (?x, ?y, ?z) describing constrains on nodes (CS=PHVB), and forest variables that can be sets of trees (?p0, ?p1, ?p2).

1.1.2. Pattern regognition mechanism

The pattern matching mechanism is also inspired by [Norvig 1992]'s proposal.

The result of the pattern matching mechanism is a list whose first element is `t` if matched, `nil` if not, and whose second element is a binding list containing the value of each variable in the pattern.

Here is some example of the produced results with the first version of the pattern matcher used at the GETA lab. in the framework of the LIDIA projet [Blanchon 1994a]. The matcher currently used is just slightly different. The matching can be said to be a lazy one because, a pattern has only to superpose itself on a prefix of the matched structure, and not on the whole structure.

```
;;;-----------------------------------------------------------------------
;;; Title       : Pattern-matcher
;;;-----------------------------------------------------------------------
;;; Authors      : Mathieu Lafourcade & Hervé Blanchon
;;;-----------------------------------------------------------------------
;;; Filename : pat-match.lisp
;;; Version  : 1.0
;;; Examples : (pat-match '(a (?* ?x) d) '(a b c d))
;;;          :  -> ((?X B C))
;;;          : (pat-match '(a (?* ?x) (?* ?y) d) '(a b c d))
;;;          : -> ((?Y B C) (?X))
;;;          : (pat-match '(a (?* ?x) (?* ?y) ?x ?y) '(a b c d (b c) (d)))
;;;          : -> ((?Y D) (?X B C))
;;;          : (pat-match '(?x ?op ?y is ?z (?if (eql (?op ?x ?y) ?z))) '(3 + 4 is 7))
;;;          : -> ((?Z . 7) (?Y . 4) (?OP . +) (?X . 3))
;;;          : (pat-match '(?x ?op ?y (?if (?op ?x ?y))) '(4 > 3))
;;;          : -> ((?Y . 3) (?OP . >) (?X . 4))
;;;          : (setf (get 'lidia-patterns 'cv-1)
;;;          :       '((?is ?x node-prop-equal-p 'CS 'PHVB)
;;;          :         (?* ?w)
;;;          :         ((?is ?y node-prop-equal-p 'FS 'OBJ) (?* ?p1))
;;;          :         ((?is ?z node-prop-equal-p 'CS 'PHVB) (?* ?p2))
;;;          :         ?punc
;;;          :         )
;;;          :       )
;;;          : (match (get 'lidia-patterns 'cv-1)
;;;          :       (get-solution (get 'lidia-test 'boule) 2)
;;;          :       )
;;;          : -->
;;;          : (
;;;          : (?PUNC ("." "." (CAT P)))
;;;          : (?P2 (("et" "ET" (FS REG CAT C))) ((NIL "GPRON" (K GN FS OBJ RL ARG1
;;;          :  CAT R GNR FEM PERS 3 NBR SING VALET N)) (("la" "BOULE" (FS GOV CAT
;;;          :  (R N) GNR FEM PERS 3 NBR SING SENS 1)))) ((NIL "NV" (K PHVB FS COORD
;;;          :  RL ID CAT V ENONCP DECL VOIX ACT MT IPR PHASE NONACC SUBV VF PERS 3
;;;          :  LINKS OUI NBR SING VALET Q ARGS A1)) (("lance" "LANCER" (FS GOV CAT V
;;;          :  MT IPR SUBV VF PERS 3 NBR SING SENS (6 5 4 3 2 1))))))
;;;          : (?Z NIL "PHVB" (K PHVB FS COORD RL ID CAT V ENONCP DECL VOIX ACT MT IPR
;;;          :  PHASE NONACC SUBV VF PERS 3 LINKS OUI NBR SING VALET Q ARGS A1))
;;;          : (?P1 (("la" "LE" (FS DES CAT D GNR FEM NBR SING))) (("boule" "BOULE"
;;;          :  (FS GOV CAT N SUBN NC GNR FEM NBR SING SENS 1))))
;;;          : (?Y NIL "DGN" (K GN FS OBJ RL ARG1 CAT (D N) SUBN NC GNR FEM PERS 3
;;;          :  NBR SING VALET N))
;;;          : (?W ((NIL "GN" (K GN FS SUJ RL ARG0 CAT N SUBN NP GNR MAS PERS 3
;;;          :  NBR SING VALET N)) (("Pierre" "*PIERRE" (FS GOV CAT N SUBN NP GNR MAS
;;;          :  PERS 3 NBR SING SENS 1)))) ((NIL "NV" (K PHVB CAT V ENONCP DECL
;;;          :  VOIX ACT MT IPR PHASE NONACC SUBV VF PERS 3 LINKS OUI RECHTS OUI
;;;          :  VCOOR OUI NBR SING VALET Q ARGS (A1 A0))) (("prend" "PRENDRE" (FS GOV
;;;          :  CAT V MT IPR SUBV VF PERS 3 NBR SING SENS (4 2 1))))))
;;;          : (?X NIL "PHVB" (K PHVB CAT V ENONCP DECL VOIX ACT MT IPR PHASE NONACC
;;;          :  SUBV VF PERS 3 LINKS OUI RECHTS OUI VCOOR OUI NBR SING VALET Q
;;;          :  ARGS (A1 A0)))
;;;          : )
;;;-----------------------------------------------------------------------
```

Figure 1.5: Some examples of pattern matching results

1.2. The beam matching mechanism

An ambiguity is described in terms of a co-occurrence of several patterns, sharing a set of variables, in the solutions produced by the analyzer. These patterns are groupped into what is called a beam.

Thus, the beam matching step is the actual step in the disambiguation process where an ambiguity is recognized. Not only is the recognition performed, but the data to be used to produce the disambiguation dialogue are also extracted from the structure produced by the analyzer. These data are the bindings of the variables used in the patterns.

1.2.1. Beams and stacks

An ambiguity is descibed as the simultaneous co-occurrence of several different patterns in the different solutions produced for an input. Those different patterns are grouped a set of patterns called a beam. A beam is an instance of the class **pattern-beam** defined as follows:

```
(defclass pattern-beam ()
 ((beam-name :type string
          :initarg :beam-name
          :initform ""
          :accessor beam-name
          :documentation "name of the beam")
  (beam-value :type list
          :initarg :beam-value
          :initform nil
          :accessor beam-value
          :documentation "list of the patterns of the beam")))
```

Figure 1.6: The engine class patern-beam

A **beam** is made of two slots:

- a beam-name that is used to refer to the beam within the disambiguation module, and,
- a beam-value that is the list of the patterns making up the beam.

A class of ambiguity can be defined (specified) with several beams. Thus beams are grouped into beam-stacks. A beam-stack is an instance of the class beam-stack defined as follows:

```
(defclass beam-stack ()
 ((beam-stack-name :type list
          :initarg :beam-stack-name
          :initform nil
          :accessor beam-stack-name
          :documentation "")
  (beam-stack-value :type list
          :initarg :beam-stack-value
          :initform nil
          :accessor beam-stack-value
          :documentation "")))
```

Figure 1.7: The engine class beam-stack

A **beam-stack** is made of two slots:

- a beam-stack-name that is used to refer to the stack within the application,

- a beam-stack-value that is the list of the beams making up the stack.

1.2.2. Beam matching: formal description

For a better understanding of the implementation, let's see the formal description of the beam matching mechanism.

A sentence S, with s solutions Sol_i, contains the ambiguity described by the beam B made of b patterns P_j if and only if:

- the number of solutions (s) is strictly greater than the number of pattern (b),
- for each solution Sol_i there is an unique pattern P_j that match that solution,
- each pattern P_j match at least one solution Sol_i,
- the distance fd between the bindings of the forest variables is null.

That is:

```
– ①  b < s
– ②  ∀i, ∃!j / match-p(Si, Pj)=t
– ③  ∀j, ∃i / match-p(Si, Pj)=t
– ④  ∀i, i', ∀j, j'
              match-p(Si, Pj)=t
         and  match-p(Si', Pj')=t
     ⇔      fd ( binding(Si, Pj),
                 binding(Si', Pj'))=0
```

Figure 1.8: Beam matching definition

fd the distance between the forest variables of two bindings is null if and only if:

- the coverage of each forest variable, except the last one, is the same in each binding, and
- for the last forest variable of the patterns, if the coverage are not the same, then one coverage has to be a prefix of the others.

That is:

Let $?p_{j,k}$, $1<k\leq l$, be the forest variables used in pattern P_j.

```
     d( binding(Si, Pj),
        binding(Si', Pj'))=0
⇔    ∀k, 1<k<l,
           coverage(?pj,k)=coverage(?pj',k)
     and
           coverage(?pj,l)=coverage(?pj',l)
        or prefix-p(?pj,l, ?pj',l)
        or prefix-p(?pj',l, ?pj,l)
```

Figure 1.9: Matching distance definition

The coverage of a variable is the projection of the leaves of the subtree this variable represents.

1.2.3. Beam matching: implementation

i. overview

In practice, the beam matching mechanism is realized by the method **match-beam** schematized bellow.

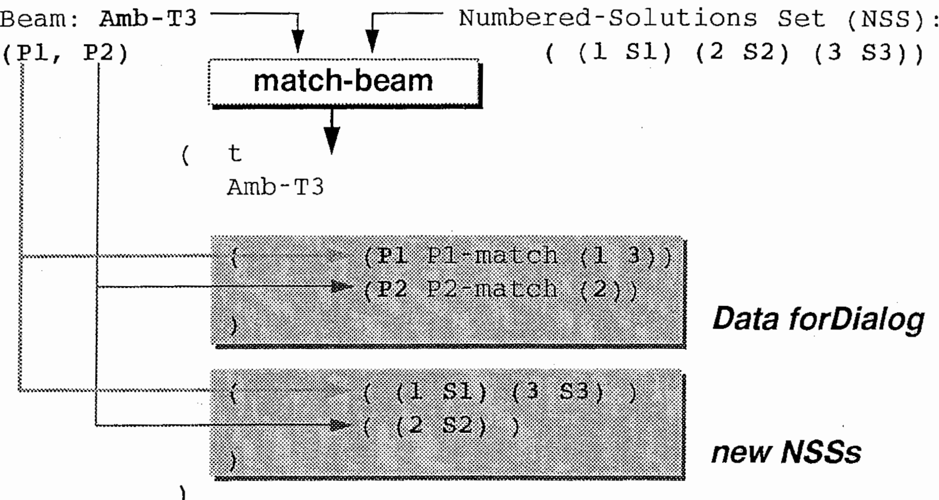


Figure 1.10: match-beam input and output

The input paramaters for **match-beam** are

- a pattern-beam that is a list of patterns used to descibe the different forms (realization) of ambiguity to be looked for.
- a numbered-solutions set that is a list of couples: ((number solutions)$^+$). The solutions produced by the analyzer are numbered and the disambiguation process allows the user to select the number of the solution he meant.

The output of **match-beam** is a list of four data:

- t or nil, if the beam has matched or not. In the latter case the other data are irrelevant,
- the name of the matched beam,
- a list of triplets which will enable the engine to construct the dialogue items to be used to solve the ambiguity,
- a list of new numbered-solutions sets in which, if necessary, other ambiguity will be searched to produce follow up disambiguation questions.

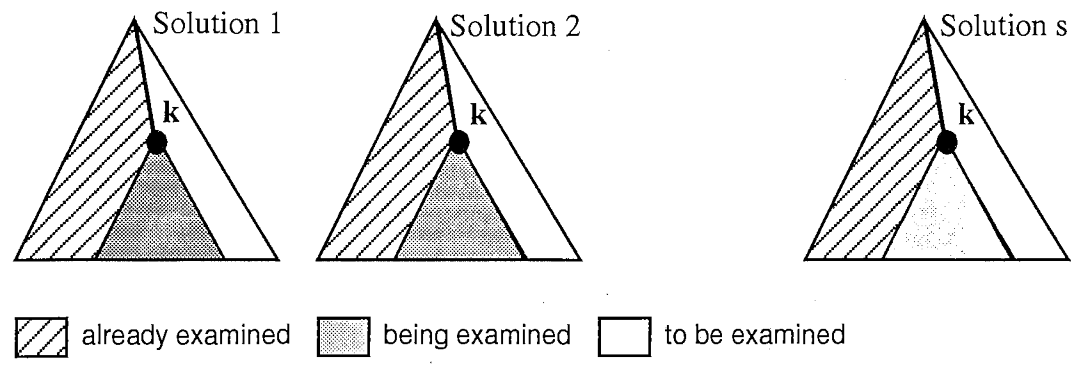
The core of the beam matching process is the filling up of a matrix.

ii. Matrix filling up and satisfaction of the required properties

The match-beam method uses a method called fill-the-matrix. This method is in charge to check that a given beam in the a beam-stack is able to match the set of solutions provided for a given utterance. For e given class of ambiguities to be recognized, each beam in the associated stack is matched is sequence against the set of solutions until a success is reached or there is no more beam to be matched.

The matrix is organized as follows: the patterns are the columns and the solutions are the lines.

An indix **k** is used for the matching of the patterns to be performed on the "same[2]" node in each solution as shown in the following figure:



already examined being examined to be examined

Figure 1.11: The synchronisation of the different pattern matchings

Indeed, an ambiguity is to be regognized in parts of a trees that are different as far as the geometry and/or the labelling is concerned. This means that in the different solutions there is always a common subtree (empty or not) that is shared, as a prefix, by all the solutions. In the above figure, the ▨ stripped subtree is this common subtree, the ▩ dotted ones are those representing the ambiguity. The ▩ subtrees are called the scope of the ambiguity in [Boitet & Tomokiyo 1995].

Thus the beam matching process is trying to find the smallest **k** allowing to fill correctly the matrix.

The squares of the matrix are filled with two values. The first value (**match-p**(S_i, P_j)) equal t if P_j has matched S_i starting at node **k**. If there was no matching starting at node **k**, then the value is nil. If there has been a match, the second value (**binding**(S_i, P_j)) representes the binding produced by the matching. This binding contains the value of each variable used in the pattern.

solution \ pattern	P_1	...	P_j	...	P_b
S_1					
...					
S_i			**match-p**(S_i, P_j), **binding**(S_i, P_j)		
...					
S_s					

The matrix is correctly filled if the required properties defined figure 1.8 are satisfied that is:

- to satisfy property ② there should be no empty lines or lines with more than one box filled, each solution must be matched by exactly one pattern.
- to satisfiy property ③ there should be no empty columns, each pattern should match at least one solution.

2 It is the same node as far as its indix in a breath first traversal of the tree is concerned. We call that multtiple pattern matching a synchronized one.

Property ① is satisfied before the matching begins and property ④ is satisfied because of the lazy synchronized pattern matching.

iii. Implementation

The actual match-beam method is defined as follows:

```
(defmethod match-beam ((self pattern-beam) the_numbered_analysis_list)
  " out: a 4 items list
       : First Item -> t or nil (t if matched, nil if not)
       : Second Item -> the_beam_name
       : Third Item ->  list of triplets (pattern-name binding concerned-solutions)
       : Fourth Item -> the_new_solution_sets
  "
  (let* ((the_beam_name (beam-name self))
         (the_pattern_list (beam-value self))            ;the patterns of the beam
         (the_patterns_number (length the_pattern_list)) ;the number of patterns in the beam
         (the_analysis_number (length the_numbered_analysis_list)) ;the number of solutions
         (the_fill_in_result (fill-the-matrix self the_numbered_analysis_list))
         (the_fill_in_success (car the_fill_in_result))
         (the_filled_matrix (cadr the_fill_in_result)))
    ;if the matrix has been filled correctly
    (if the_fill_in_success
      (let*
          (;tor each binding, the values of the last forest variables may have to be reduced.
           ;they may have different values because of the lazy matching
           ;the variables are reduced to the prefix value shared by the different instance
           (the_reduced_list
            (if (= the_patterns_number the_analysis_number)
              ;if the matrix is a square only the lines have to be reduced
              (project-square-matrix-into-list the_patterns_number
                                               the_analysis_number
                                               the_filled_matrix)
              ;if the matrix is not a square the lines and columns have to be reduced
              (project-rectangle-matrix-into-list the_patterns_number
                                                  the_analysis_number
                                                  the_filled_matrix)))
           (the_normalized_list (normalize-list the_reduced_list the_numbered_analysis_list))
           ;the triplets used to build the question (cf Fig. xx)
           (the_named_binding_list (construct-named-binding-list self the_normalized_list))
           ;the new numbered-solutions-sets for the follow up questions (cf Fig.  )
           (the_new_solution_sets (construct-new-solution-sets the_normalized_list
                                                               the_numbered_analysis_list)))
         ;construction of the result
         (list t the_beam_name the_named_binding_list the_new_solution_sets))
      ;the regognition failed
      '(nil nil nil nil))))
```

Figure 1.12: The engine method match-beam

1.3. Comments

- The reduction process is not very orthogonal with the global implementation, as solution has to be found to avoid this inelegance.
- It may be interesting to be able to define more complex constraints on the variables of the patterns. In the current implementation we have a one to one correspondance using equality.
- As it will shown later, is it also necessay to provide a property recognition mechanism working on lists because all the ambiguities can not be defined in terms of tree-patterns.
- How to reduce the number of unsuccessfull match while trying to recognize an ambiguity?
- The current data structure used to represent the analysis is a tree. Thus, the description of an ambiguity is made in terms of trees and the recognition mechanism is maniplulating tree structures.
  - As a first solution, we can imagine a more flexile engine providing several recognition modules to deal with differents representation of the analysis.

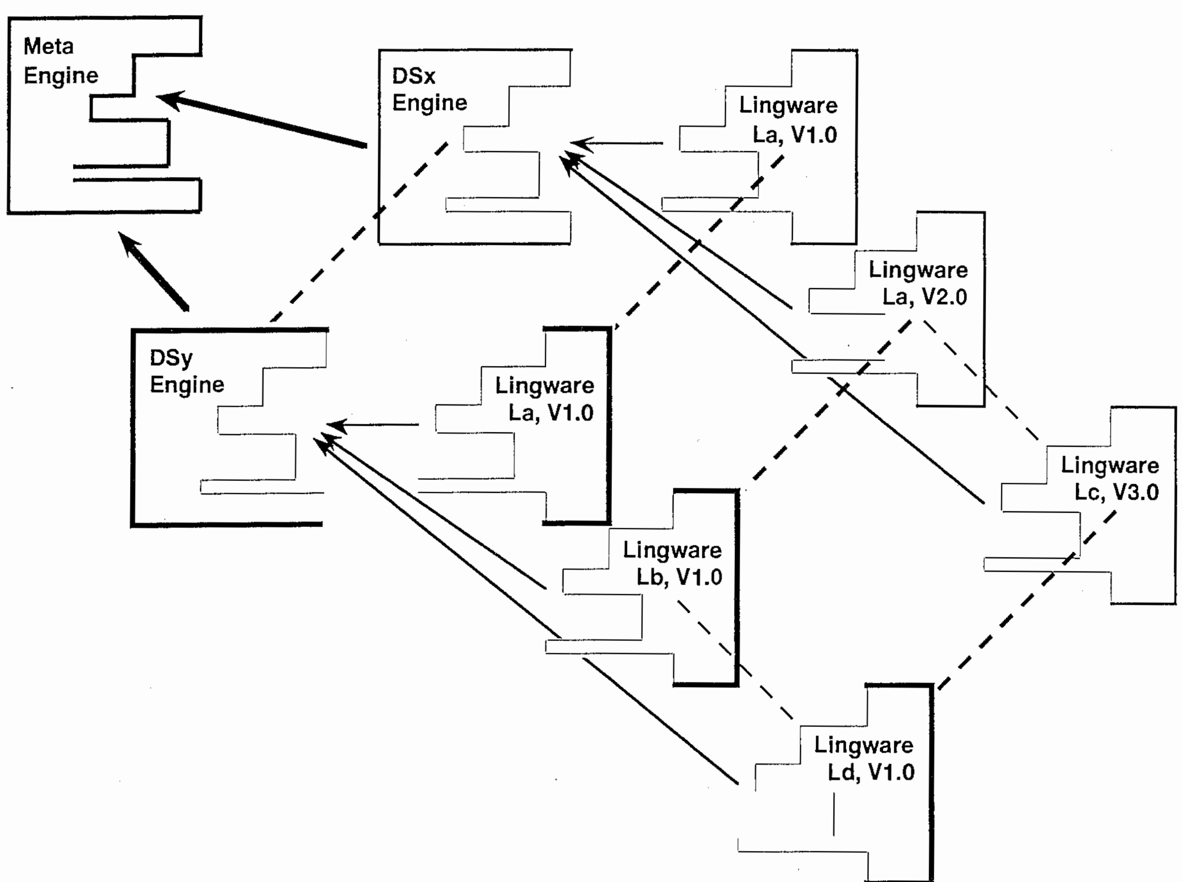


Figure 1.13: A multiple data-structure disambiguation module architecture

In this solution, serveral data-structure-dependant (DS) engine would inherit serveral modules from a data-structure-independant engine.

- An other solution would be to provide a family of transducers to transform different data structures into tree-strcuctures. With this solution the description of the ambiguities is not made in terms of the original data structure.

- The manipluated data structures have to be weighted so that the module can learn from the history of the dialogue (adapt itself to the user), and be tunable.
  - Weighted patterns
    - Reorder the patterns inside a beam. The most likely selected item should be selected by default
    - If likelihood of answer > value then question not asked
  - Weighted beams
    - Reorder the beams describing a kind of ambiguity. Speed up the system
    - If likelihood of beam < value then beam not tested
  - Weighted ambiguities
    - Reorder the states of the automaton. Speed up the system
    - If likelihood or importance of ambiguity < value then automatic selection
- New patterns have to be constructed automatically. When an ambiguity is not recognized by the module, it should prepare the patterns to be used to recognize it. The module will learn to recognize new ambiguities. Of course, the dialogue items production methods to be associated with the new patterns will have to be prepared by hand. The following figure shows one possible realization.

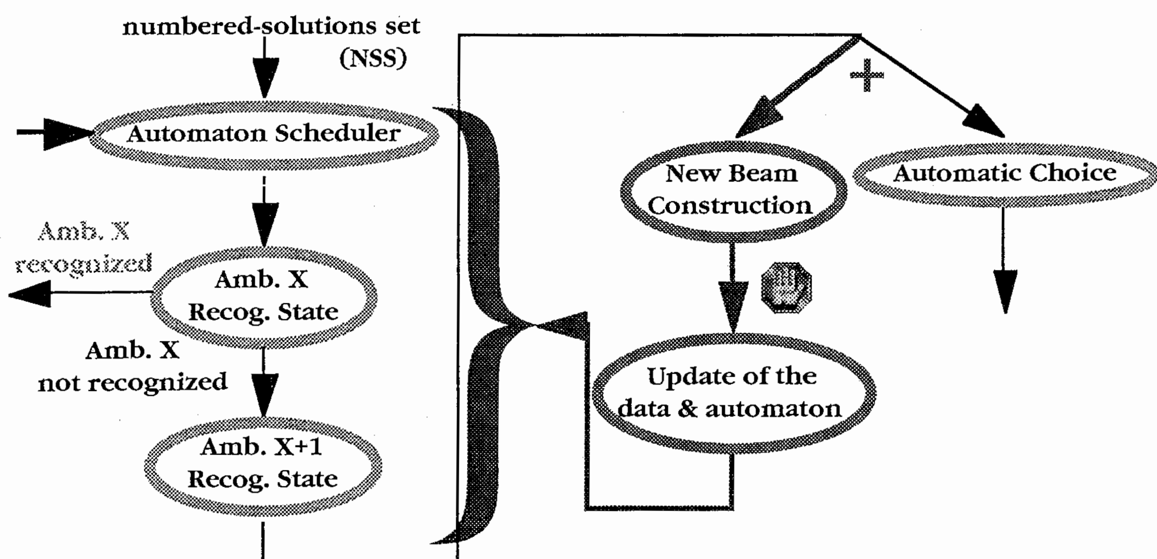


Figure 1.14: Automatic learnig of new beams

Chapter 2

Construction and presentation of a question tree

The construction of the question tree is a loop in the disambiguation automaton organized by the question-tree-constrction module. When a question tree (cf Fig. 2.2) has been built it is presented to the user by the question-tree-presentation module.

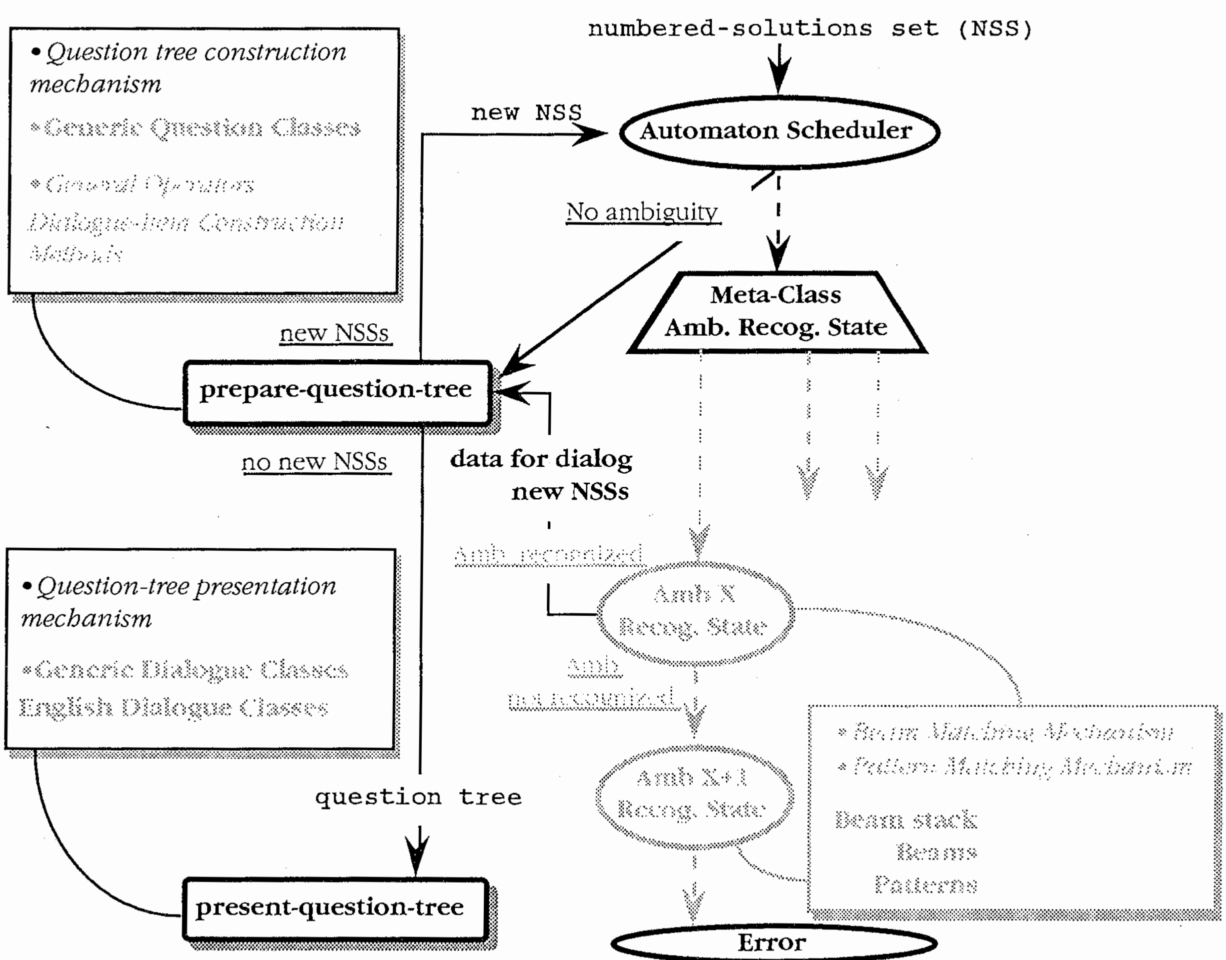


Figure 2.1: Construction & presentation of a question tree in the global architecture

The disambiguation automaton organizes the order the presence of the ambiguities has to be ckecked. Among other states this autonamon contains one distinguished state per class of ambiguity defined in the lingware.

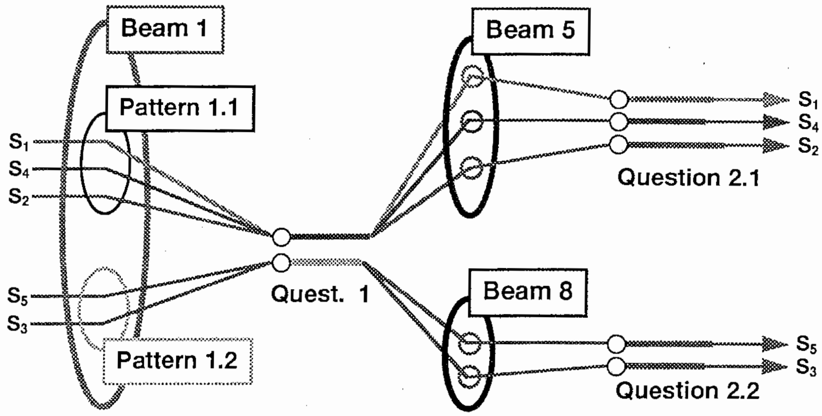


Figure 2.2: A question tree to discriminate 5 solutions

Figure 2.2 shows a example of a question tree. The nodes of the tree are representing questions and the leaves are representing the set of analysis from which the right interpretation have to be choosen.

2.1. Disambiguation automaton

Each state of the automaton is defined as a CLOS method [Keene 1989]. There is basically 3 kind of states:

- an entry point, that is the first state of the automaton. It is called automaton-scheduler and provided by the engine,
- ambiguity meta-class recognition states provided by the engine, and,
- ambiguity class recognition states provided by the lingware.

i. Automaton scheduler

As the entry point of a disambiguation automaton the `automaton-scheduler` (fig 2.3) is also the exit point of the automaton when the whole question tree has been produced. The first action completed is thus to test whether or not the `the_numbered_analysis_list` contains several analysis. If not, an empty question is prepared that will be a leaf of the question tree.

```
(defmethod automaton-scheduler ((the_language (eql 'english))
                                 the_sentence
                                 the_numbered_analysis_list)
  "  in: the_numbered_analysis_list is a list of indexed-solution-sets,
         question if <> nil is an object of type clarification_question_class
    out: a question tree
effects: produce a question tree
"
  (if (= 1 (length the_numbered_analysis_list))
    (list (make-instance 'empty-question
            :concerned-solution (first (first the_numbered_analysis_list))
            :concerned-tree (second (first the_numbered_analysis_list))))
    (same-categories-p-state the_language the_sentence the_numbered_analysis_list)))
```

Figure 2.3: The engine method automaton-scheduler

This method is specialised on the language parameter so that, the entry point name of each automaton is the same.

ii. ambiguity meta-class recognition states

An ambiguity meta-class recognition state is a predicative state used as a branching state in the automaton. So far we have proposed three ambiguity meta-classes defined in § 5.1. These classes are called: lexical-ambiguity, geometrical-ambiguity and labelling-ambiguity, they are supposed to be refined by the designer of the lingware into serveral designer-defined classes of ambiguity.

Thus, a disambiguation automaton should be shaped as follows:

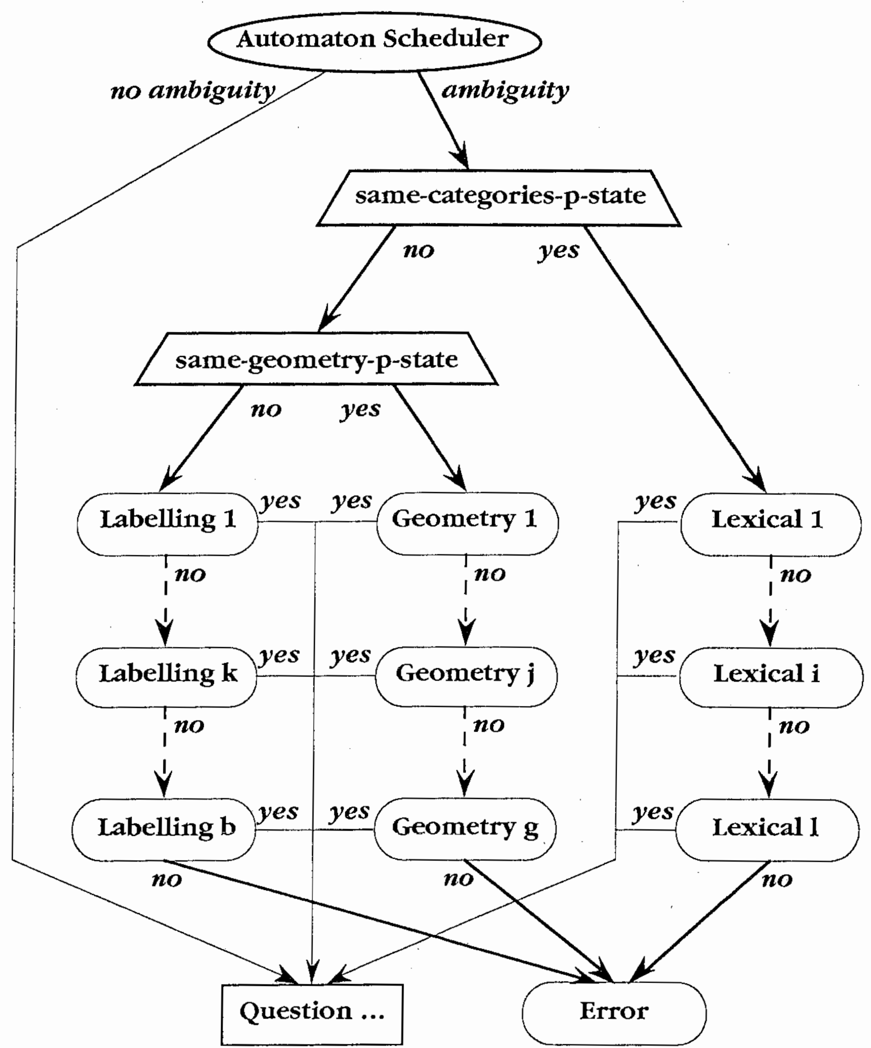


Figure 2.4: General organisation of a disambiguation automaton

The lexical ambiguities are to be solved first, then the geometrical ones, and finally the labelling ones. This strategy is guided by the following principles:

1. first, find the right simple phrases (ie. solve the lexical ambiguities),
2. second, find the construction of the verbs (ie. solve the labelling and some of the geometry ambiguities),
3. third, find the structure of the dependents of the verbs (ie. solve the geometry ambiguities which did not fall in the previous case),
4. last, find the word senses.

Pragmatic considerations led us to define them:

- The simple phrases are the basic bricks of the sentence, producing the sense;
- Then, we want to find the constituents of the sentence at the higher level, the level of the construction of the verb;
- After that, we want to find the right organization for the constituants of the verbs;
- Finally, it is time to find the sense of each occurrence as the whole set of senses of an occurrence has been reduced once the previous steps allowed to find its use in the sentence.

Those criteria seem reasonable and natural. Moreover, the order of the kinds of question will not be changed to improve the usability of the system.

The implemented ambiguity metaclass regognition states are defined as follows:

```
(defmethod same-categories-p-state ((the_language (eql 'english))
                                     the_sentence
                                     the_numbered_analysis_list)
        ;the same number of leaves and different lexical classes
  (let ((same_lex_class (same-categories-p the_numbered_analysis_list))
        ;the number of leaves is different (ie. not the same lexical units)
        (same_number_of_leaves (same-number-of-leaves-p the_numbered_analysis_list)))
    (Cond
     ((not same_number_of_leaves) (phrasal-verb-ambiguity-state the_language
                                                                the_sentence
                                                                the_numbered_analysis_list))
     ((not same_categories) (noun-adjective-ambiguity-state the_language
                                                            the_sentence
                                                            the_numbered_analysis_list))
     (t (same-geometry-p-state the_language
                               the_sentence
                               the_numbered_analysis_list)))))
```

Figure 2.5: The engine method same-categories-p-state

```
(defmethod same-geometry-p-state ((the_language (eql 'english))
                                   the_sentence
                                   the_numbered_analysis_list)
  (if (same-geometry-p the_numbered_analysis_list)
    "labelling ambiguity not handled yet"
    (second-phvb-adv-att-state the_language the_sentence the_numbered_analysis_list)))
```

Figure 2.6: The engine method same-geometry-p-state

iii. ambiguity class recognition states.

Provided by the lingware, the ambiguity recognition states are descibed in chapter 6.

2.2. Question tree construction

2.2.1. Strategy

Several question tree construction strategies can be designed according to the global

constraints a disambiguation module has to fullfil.

In the context of the LIDIA project the translation process is realized in a batch mode [Blanchon 1990 ; Boitet 1989 ; Boitet 1990]. The user of the system is not disturbed in the conception of the document he is writing and translating somehow simultaneously.

Question trees are also produced in batch mode without time constraints. When a question tree is ready, the user is told that new questions are waiting to be answered.

In the current implementation we did not propose a new strategy. That is why the whole question tree is build in one blow.

In the current clarification engine, this strategy is available by means of the `prepare-question-tree` method.

2.2.2. Implementation

The `prepare-question-tree` method is defined as follows:

```
(defmethod prepare-question-tree ((the_language (eql 'english))
                                   the_type
                                   (the_modality (eql 'textual))
                                   the_sentence
                                   the_list_of_triplets
                                   the_new_solution_sets)
        ;the current question is prepared
  (let* ((the_first_question (prepare-question the_language
                                                the_type
                                                the_modality
                                                the_sentence
                                                the_list_of_triplets))
        ;the follow up questions are prepared
        (the_next_questions (prepare-question-list the_language
                                                    the_sentence
                                                    the_new_solution_sets))
       ;the question tree is constructed
       (the_result (list the_first_question the_next_questions)))
   the_result))
```

Figure 2.7: The engine method prepare-question-tree

The `prepare-question-tree` method is first called when a first ambiguity has been recognized in the disambiguation automaton. This method is called with the result of the method beam-match:

- the data to construct the question for the regognized ambiguity (`the_list_of_triplets`) with the method `prepare-question`,
- the data to construct the questions to follow (`the_new_solution_sets`) with the method `prepare-question-list`.

The questions to follow are prepared by the method `prepare-question-list` reentering the automaton through the `automaton-sceduler` state as shown bellow:

```
(defmethod prepare-question-list ((the_language (eql 'english))
                                   the_sentence
                                   the_new_solution_sets)
  (if (= 1 (length the_new_solution_sets))
    ;there is only one question sub-tree to built
    (list (automaton-scheduler the_language the_sentence (car the_new_solution_sets)))
    ;there is several question sub-trees to be built
    (cons (automaton-scheduler the_language the_sentence (car the_new_solution_sets))
          (prepare-question-list the_language the_sentence (cdr the_new_solution_sets)))))
```

Figure 2.8: The engine method prepare-question-list

Thus, the construction of the question tree is made through a recursive process.

2.2.3. Construction of a question

Questions are actualy prepared with the method prepare-question defined as follows:

```
(defmethod prepare-question ((the_language (eql 'english))
                              the_type
                              (the_modality (eql 'textual))
                              the_sentence
                              the_list_of_triplets)
      ;construction of the list of dialogue items
  (let ((the_items (mapcar #'produce-item the_list_of_triplets)))
    ;creation of an object of the class clarification-question-class to be intertered
    ;in the current question tree
    (make-instance 'clarification-question-class
          :question-language the_language     ;language desmabiguated (for the meta-language)
          :question-type the_type             ;type of the question (for the labelling)
          :question-modality the_modality     ;modality(ies) of the presention
          :ambiguous-item the_sentence        ;tha ambiguaous utterance
          :question-items-list the_items)))   ;list of the rephasing items to be proposed
```

Figure 2.9: The engine method prepare-question

A triplet is a list of three elements: the name of the pattern which has matched, the resulting binding, and, a list of the indix numbers of the solutions concerned with the matching of the pattern. This triplet is used to construct a dialogue item.

The `produce-item` is implemented a follows:

```
(defun produce-item (self)
  (let ((the_pattern (first self))
        (the_binding (second self))
        (the_concerned_solutions (third self)))
      ;the pattern-method associated with the pattern is applied to the binding
    (list (apply (pattern-method the_pattern)
                 (list (pattern-name the_pattern) the_binding))
          the_concerned_solutions)))
```

Figure 2.10: The engine function produce-item

2.3. Question tree presentation

For a given ambigutous utterance, the disambiguation automaton produces a question tree. The question tree is covered by the `question-tree-presentation` function (cf Fig. 2.11) until no more question is to be asked. The method `ask-question` (cf Fig. 2.12) proposes the question to the user.

```
(defun question-tree-presentation (the_question_tree)
  (if (= 1 (length the_question_tree)) ;it is a leaf
    (progn
      (concerned-solution (first the_question_tree))  ;indix number the choosen analysis
      (geta-grapher:browse (concerned-tree (first the_question_tree))
                           'geta-grapher::ariane-tree))  ;display of the chosen tree for demo
    ;several questions are to be asked
          ;answer to the root question of the current question tree
    (let ((the_choice (ask-question (first the_question_tree)))
          ;list of the possible follow up sub- question-trees
          (the_other_questions (second the_question_tree)))
      ;the relevant sub-question-tree is going to be presented to the user
      (question-tree-presentation (nth (- the_choice 1) the_other_questions)))))
```

Figure 2.11: The engine method question-tree-presentation

The method `ask-question` is itself defined as follows:

```
(defmethod ask-question ((self clarification-question-class))
  (let ((the_question_language (question-language self))
        (the_question_type (question-type self))
        (the_question_modality (question-modality self))
        (the_ambiguous_item (ambiguous-item self))
        (the_question_items_list (question-items-list self)))
    (progn
      (purge-lidia-dialog-answer)
      ;actual presentation of the dialogue
      (do-dialog the_question_type
                 the_question_language
                 the_question_modality
                 the_ambiguous_item
                 (mapcar #'first the_question_items_list)
                 )
      ;waiting for the answer
      (wait-for-an-answer))))
```

Figure 2.12: The engine method ask-question

2.4. Comments

- We did not mention the ambiguities of polysemy. In our opinion, they should be the lasts to be solved. The door have to be left open to describe the different option.
- In the current implementation, the question tree is first completly constructed and then the questions are asked. This means that unused questions have been prepared. If time efficiency is crucial, at least, two other stategies can be proposed:
  - When a question has been prepared, wait for the answer to that question to prepare and present the next relevant one.

- When a question has been prepared, ask the question to the user. While the user is providing the answer to the current question, prepare the follow up questions. When the answer has been given present the user with the relevant question.

These strateges, or other ones can be easily implemented specializing the current prepare-question-tree method on an new argument called strategy that would be a global parameter of a given module.

- The current implentation of the `same-category-p-state` is not able to cope with the 3 families of lexical ambiguities defined in the §5.1.
- When an ambiguity has been detected and a question prepared, the reentry point in the automaton should not be the first state (automaton scheduler). It should be the state where the ambiguity was discovered. Indeed, no ambiguity can be found in the the states preceding this last one. The new organization would look as follows:

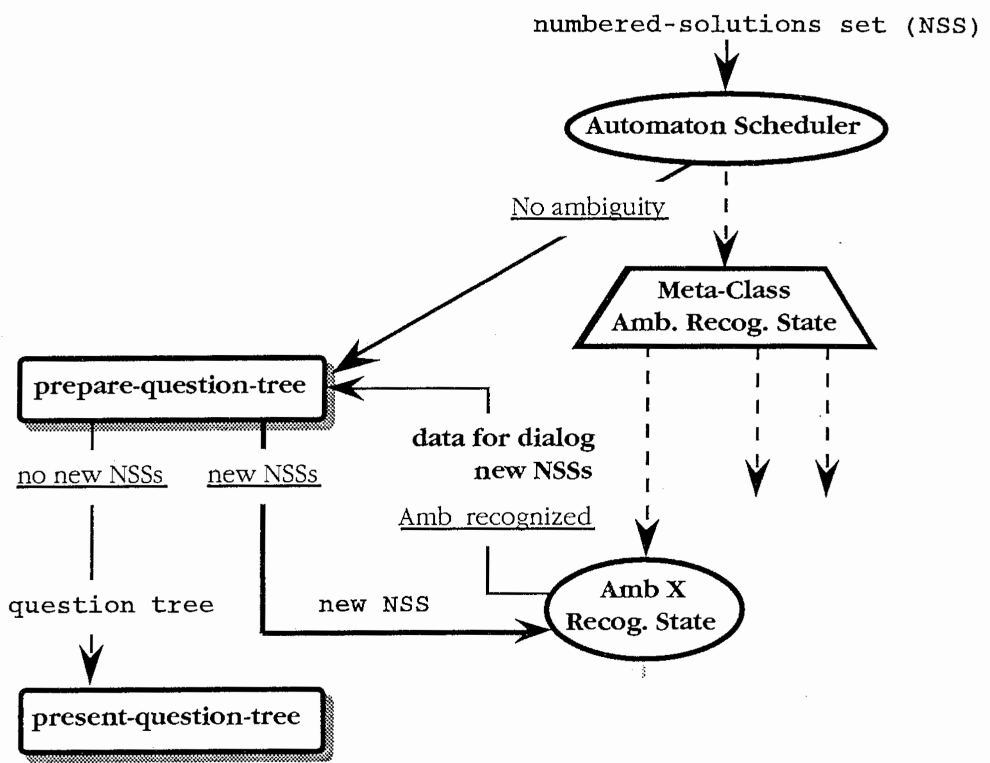


Figure 2.13: Better organisation of the reentry in the desambiguation automaton

- Can the defintion of the ambiguity meta-class recognition state be done by the designer of the lingware? It is necessary, usefull?
- Is-it important and desirable to broaden the tree, thus adding more less specific tests, so as to reduce the depth of the automaton.

Chapter 3

Dialogue & Question classes

Two kinds of generic classes are defined in the kernel: questions classes specifiying the diambiguation questions' content and dialogue classes specifying the presentation of the diambiguation dialogues to the user.

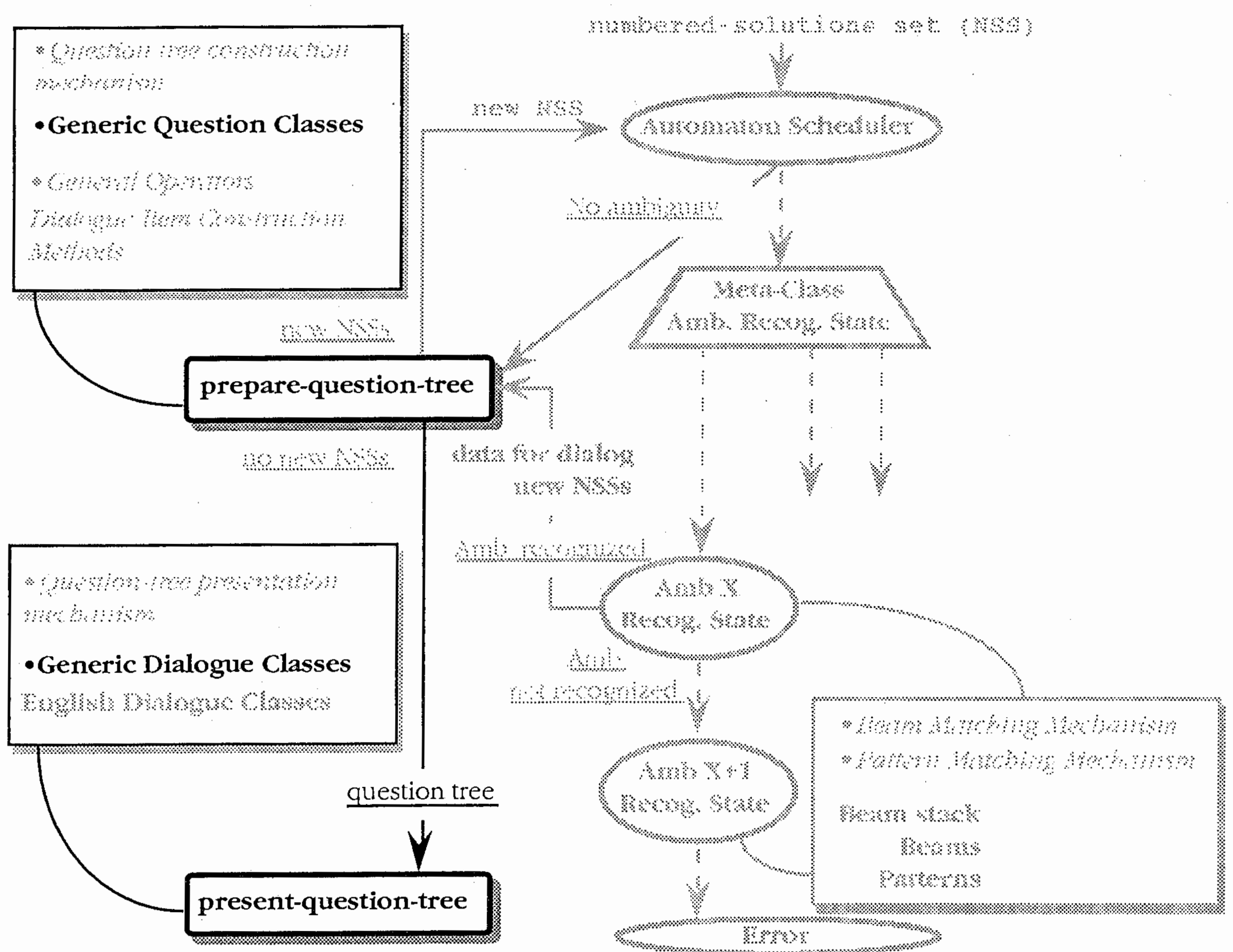


Figure 3.1: Dialogue & question classes in the global architecture

3.1. A generic disambiguation question class

The disambiguation process is producing a question-tree made of disambiguation questions that are to be displayed as dialogues on the screen. Each disambiguation question is an instance of the predefined class clarification-question-class defined as follows:

```
(defclass clarification-question-class ()
 ((question-language :initarg :question-language
              :accessor question-language)
  (question-type :initarg :question-type
             :accessor question-type)
  (question-modality :initarg :question-modality
              :accessor question-modality)
  (ambiguous-item :initarg :ambiguous-item
             :accessor ambiguous-item
  (question-items-list :initarg :question-items-list
             :accessor question-items-list)))
```

Figure 3.2: The engine class clarification-question-class

The slots have the following meanings:

- question-language is the language disambiguated and the metalanguage used to present the question. For an English module it is English, for a French module is French, etc...
- question-type is the type (the class) of ambiguity to be solved with the question.
- question-modality is the modality(ies) to be used to present the question to the user. So far it is textual. In the futur it may be also spoken, textual+spoken, spoken+drawn, etc...
- ambiguous-item is the utterance to be disambiguated with the question.
- question-items-list: is the list of the dialogue items to be proposed to the user.

The is also a particular class called empty-question that is a subclass of the clarification-question-class. This particular class of question is used to construct the leaves of the question trees. It is defined a follows:

```
(defclass empty-question (clarification-question-class)
 ((concerned-solution :type integer
                :initarg :concerned-solution
                :initform 0
                :accessor concerned-solution)
  (concerned-tree :type list
             :initarg :concerned-tree
             :initform nil
             :accessor concerned-tree)))
```

Figure 3.3: The engine class clarification-question-class

The slots have the following meanings:

- concerned-solution is the number of the chosen solution.
- concerned-tree is the analysis tree associated with the chosen solution.

3.2. A generic textual disambiguation dialogue class

As the presentation modality for the disambiguation have been so far only textual, we have

defined a generic-textual-clarif-dialog-class. All the textual clarification dialogues classes to be defined will inherit from this class defined as follows.

```
(defclass generic-textual-clarif-dialog-class (dialog)
 ((window-length
  :initarg :window-length                  :accessor window-length)
  (invitation-string
  :initarg :invitation-string              :accessor invitation-string)
  (invitation-string-font
  :initarg :invitation-string-font         :accessor invitation-string-font)
  (ambiguous-string
  :initarg :ambiguous-string               :accessor ambiguous-string)
  (ambiguous-string-font
  :initarg :ambiguous-string-font          :accessor ambiguous-string-font)
  (separation-line-position
  :initform :separation-line-position      :accessor separation-line-position
  :initform 0)
  (prompt-string
  :initarg :prompt-string                  :accessor prompt-string)
  (prompt-string-font
  :initarg :prompt-string-font             :accessor prompt-string-font)
  (items
  :initarg :items                          :accessor items)
  (items-font
  :initarg :items-font                     :accessor items-font)
  (current-choice
  :initarg :current-choice                 :accessor current-choice)
  :initform 1)
 (:default-initargs
  :close-box-p nil))
```

Figure 3.4: The engine class generic-textual-clarif-dialogue-class

The slots have the following meaning:

- window-length is the length of the disambiguation dialogue. It may change according to the dialogue to be disambiguated.
- invitation-string tells the user about the kind of problem to be solved in the ambiguous-string.
- invitation-string-font is the font, size and style to be used to display the invitation-string.
- ambiguous-string is the utterance the disambiguation dialogue is about.
- ambiguous-string-font is the font, size and style to be used to display the ambiguous-string.
- separation-line-position is the relaive position of a sepation line displayed in the dialogue between the ambiguous-string and the prompt-string.
- prompt-string is the text inviting the user to choose one of the proposed items.
- prompt-string-font is the font, size and style to be used to display the prompt-string.
- items is the list of the items to be proposed among which the user is asked to choose one.
- items-font is the font, size and style to be used to display the items.
- current-choice is the choice selected by default.

- (:default-initargs :close-box-p nil)) says the the only means to close the dialogue is to validate a chosen item by clicking the OK button.

3.3. Comments

- as we are going towards the use of several modalities, new classes will have to be developped.
- if serveral modalities are to be used simultaneousely, the desciption of the presentation of the dialogues may have to be descibed with a markup language (synchronizations, successions of the different messages in the dialogue). This is the meta-dialogue (the fixed part of the question).
- the desciption of the dialogue items has to be revised also to allow the manipulation of the offered modalities with the same problems of synchonization of the events.

Chapter 4

Operators

Operators are used to describe the dialogue items' construction. They are used by the prepare-question-tree module and allow to perform several operations on the binding of the variables. Three families of operators are defined to perform: selective projection, access to the lexical database and formating operations.

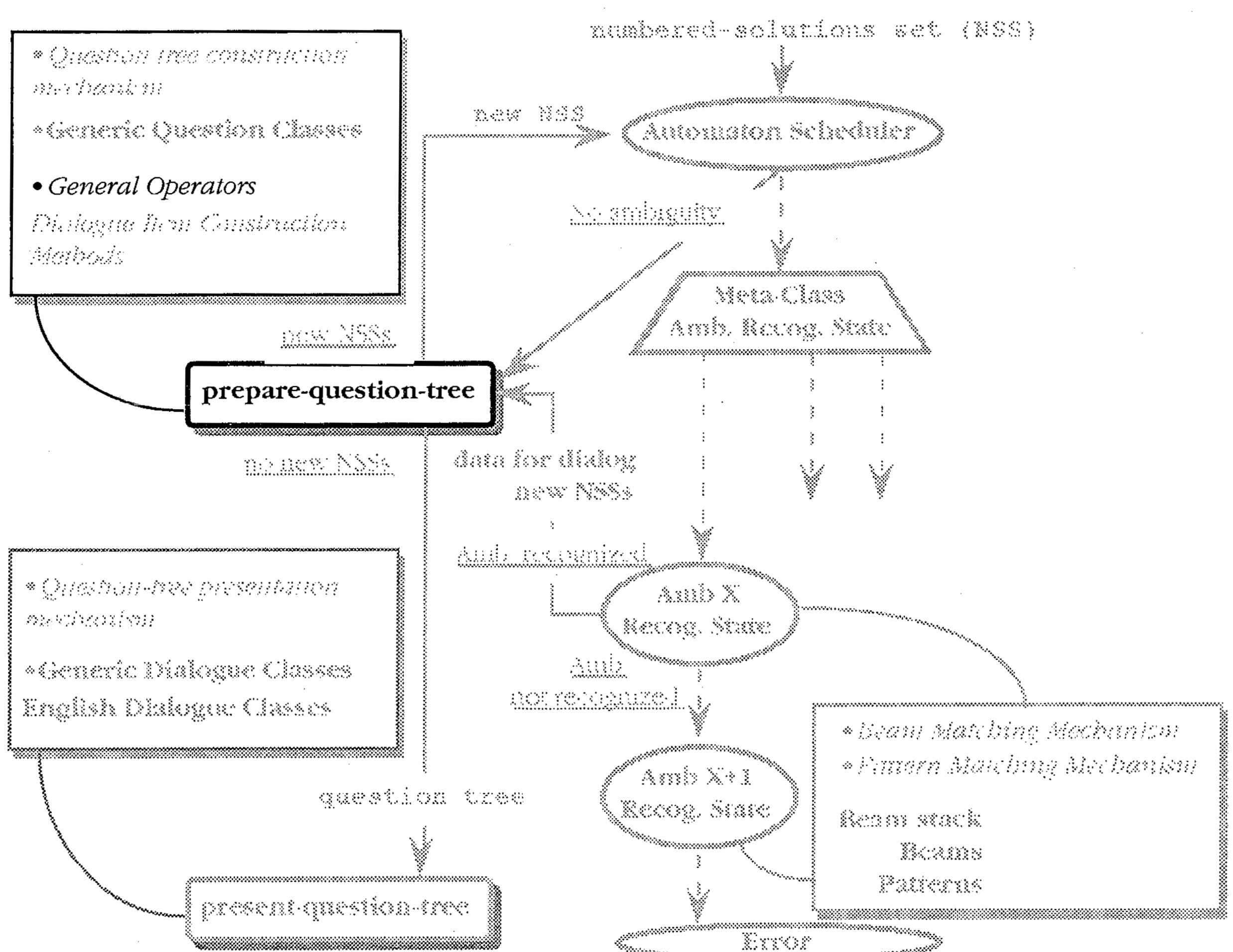


Figure 4.1: The operators in the global architecture

4.1. Selective projection

Operators of the first family describe some manipulations of subtree structures, basicaly the selection or the suppression of some part of the trees.

4.1.1. Definitions (selection)

`Text` produce the text of the linguistic trees given as parameter.

`Subject` produce the text of the subject of the linguistic trees given as parameter.

– **note:** There is such a function for each syntactic function and syntagmatic class. Example: `VerbalGroup`, `CircComp`...

`Coord` produce the coordinating occurrence of the linguistic trees given as parameter.

`But_Coord` produce the text of the linguistic trees given as parameter without the coordinating occurrence.

`But_Sub` produce the text of the linguistic trees given as parameter without the subordinating particle.

`But_Det` produce the text of the linguistic trees given as parameter without the determiner.

`Project` applied to a leaf of the mmc-structure, project the occurrence and the syntactic class of the occurrence. In the future some other information may also be proposed.

4.1.2. Implementation (selection)

Here is some of the implemented operators. The operators entry point function names are in bold.

```
(defun text (&rest more-trees)
  (if more-trees
    (let ((result (with-output-to-string (str)
                    (string-trim '(#\space)
                                 (apply #'texte-dans-stream str
                                        (first more-trees)
                                        (rest more-trees))))))
      (if result result ""))
    ""))
```

Figure 4.2: The engine operator text

```
(defun texte-dans-stream (str tree &rest more-trees)
  (if more-trees
    (progn
      (texte-dans-stream str tree)
      (apply #'texte-dans-stream str more-trees))
    (if (est-feuille? tree)
      (when (surface (racine tree))
        (format str "~A " (surface (racine tree))))
      (apply #'texte-dans-stream str (fils tree)))))
```

Figure 4.3: The engine sub-operator text-dans-stream

```
(defun coord (tree &rest more-trees)
  (declare (ignore more-trees))
   (if (est-feuille? tree)
    (when (or (string-equal (ul (racine tree)) "ET")
              (string-equal (ul (racine tree)) "OU")   ;for French
              (string-equal (ul (racine tree)) "AND")
              (string-equal (ul (racine tree)) "OR"))  ;for English
      (surface (racine tree))
      )
    (let ((coord nil)
          (the-sons (fils tree)))
      (do ((sons the-sons))
          ((or coord
               (not sons)))
        (setf coord (coordonnant (first sons)))
        (setf sons (cdr sons)))
      coord)))
```

Figure 4.4: The engine operator coord

```
(defun but-coord (tree &rest more-trees)
  (with-output-to-string (str)
    (apply #'moins-coordonnant-dans-stream str tree more-trees)))
```

Figure 4.5: The engine operator but-coord

```
(defun moins-coordonnant-dans-stream (str tree &rest more-trees)
  (texte-dans-stream str (discard-coordonnant tree nil))
  (if more-trees
    (apply #'texte-dans-stream str more-trees)))
```

Figure 4.6: The engine sub-operator moins-coordonnant-dans -stream

```
(defun discard-coordonnant (tree found?)
  (if (est-feuille? tree)
    (if (and (not found?)
             (or (string-equal (ul (racine tree)) "ET")
                 (string-equal (ul (racine tree)) "OU")   ;for French
                 (string-equal (ul (racine tree)) "AND")
                 (string-equal (ul (racine tree)) "OR"))) ;for English
      (values nil t)
      (values tree found?)
      )
    (let ((result nil))
      (do ((sons (fils tree)))
          ((not sons))
        (multiple-value-bind (discarded-tree discarded?)
                             (discard-coordonnant (first sons) found?)
          (setf found? discarded?)
          (setf sons (cdr sons))
          (setf result (concatenate 'list result (list discarded-tree))))))
      (values (concatenate 'list (list (racine tree)) result ) found?))
    ))
```

Figure 4.7: The engine sub-operator disgard-coordonnant

4.2. Acces to the Multilingual Lexical Data Base

4.2.1. Definition

`Agreement` produce the form of an adjective according to gender and number constraints.

```
ex :      Agreement("noirs",(("gn" [...] (gnr fem nbr plu)) ([...]))) -> "noires"
```

`Substitute` replace an ambiguous preposition by a non-ambiguous one (in the context) according to several properties: syntactic function or logico-semantic relation.

```
ex :      Substitute("de", #Objet_1) -> "à propos de"
```

`Definition` produce the definition of the occurrence given as the parameter.

4.2.2. Implementation

No English adaptation yet.

4.3. Other operations

Operators of the third family describe some more complex operations of distribution and bracketing of subtrees.

4.3.1. Definition

`Distribute` distribute an occurrence or a groupe of occurrences over other groups of occurrences and link the new groups with a preposition of coordination.

– **note:** The distribution is done agreeing the gender and the number of the adjective with the gender an the number of the different substentives

```
ex :      Distribute((A, B C, D), ou ,(1, 2), (1, 3)) -> A B C ou A D
```

`Bracket` brackets the text of the arguments.

```
ex :      Bracket("classeurs" ,"noirs") -> "(classeurs noirs)"
```

4.3.2. Implementation

```
(defun distribute (list-of-strings link pattern &rest more-patterns)
  (with-output-to-string (str)
    (apply #'distribue-dans-stream str list-of-strings link pattern more-patterns)))
```

Figure 4.8: The engine operator distribute

```
(defun distribue-dans-stream (str list-of-strings link pattern &rest more-patterns)
  (distribue-pattern-dans-stream str list-of-strings pattern)
  (format str "~A " link)
  (if more-patterns
    (apply #'distribue-pattern-dans-stream str list-of-strings more-patterns)))
```

Figure 4.9: The engine sub-operator distribue-dans-stream

```
(defun distribue-pattern-dans-stream (str list-of-strings pattern &rest more-patterns)
  (if more-patterns
    (progn (distribue-pattern-dans-stream str list-of-strings pattern)
           (apply #'distribue-pattern-dans-stream str list-of-strings more-patterns))
    (progn
      (dolist (num pattern)
        (format str "~A " (nth (- num 1) list-of-strings))))))
```

Figure 4.10: The engine sub-operator distribue-pattern-dans-stream

```
(defun bracket (tree &rest more-trees)
  (with-output-to-string (str)
    (apply #'parenthese-dans-stream str tree more-trees)))
```

Figure 4.11: The engine operator bracket

```
(defun parenthese-dans-stream (str tree &rest more-trees)
  (format str "(")
  (string-trim '(#\space) (apply #'texte-dans-stream str tree more-trees))
  (format str ")"))
```

Figure 4.12: The engine sub-operator parenthese-dans-stream

4.5. Comments

- some of the operators are language independant and tree-labelling independant. Those operators do not create any problem.
- some of the operators are language dependant and/or tree-labelling dependent. Some problems have to be solved as far as operators are concerned.

  Instead of using an explicit description of the values to be looked for (as the conjunctions of coordinations) directly into the programs of the operators themselves (cf. Fig. 4.4), there should be an indirection.

  For this indirection to be organized, there is serveral requirements:

  - the language and/or tree-labelling dependant elements have to be variabilized, and
  - the operators have to use those variables instead and not direct encoding of the values.

  Here is an example for the operator coordination

In an English lingware: `(defvar conjunctions_of_coordination_set '("and" "or"))`

In a French lingware: `(defvar conjunctions_of_coordination_set '("et" "ou"))`

In the engine:

```
(defun coord (tree &rest more-trees)
  (declare (ignore more-trees))
   (if (est-feuille? tree)
    (when (menber (ul (racine tree)) conjunctions_of_coordination_set)
      (surface (racine tree))
      )
    (...)))
```

Figure 4.13: A language independant operator coord

Part II

The English lingware

Introduction

The English disambiguation lingware components are shown in plain style in the following figure.

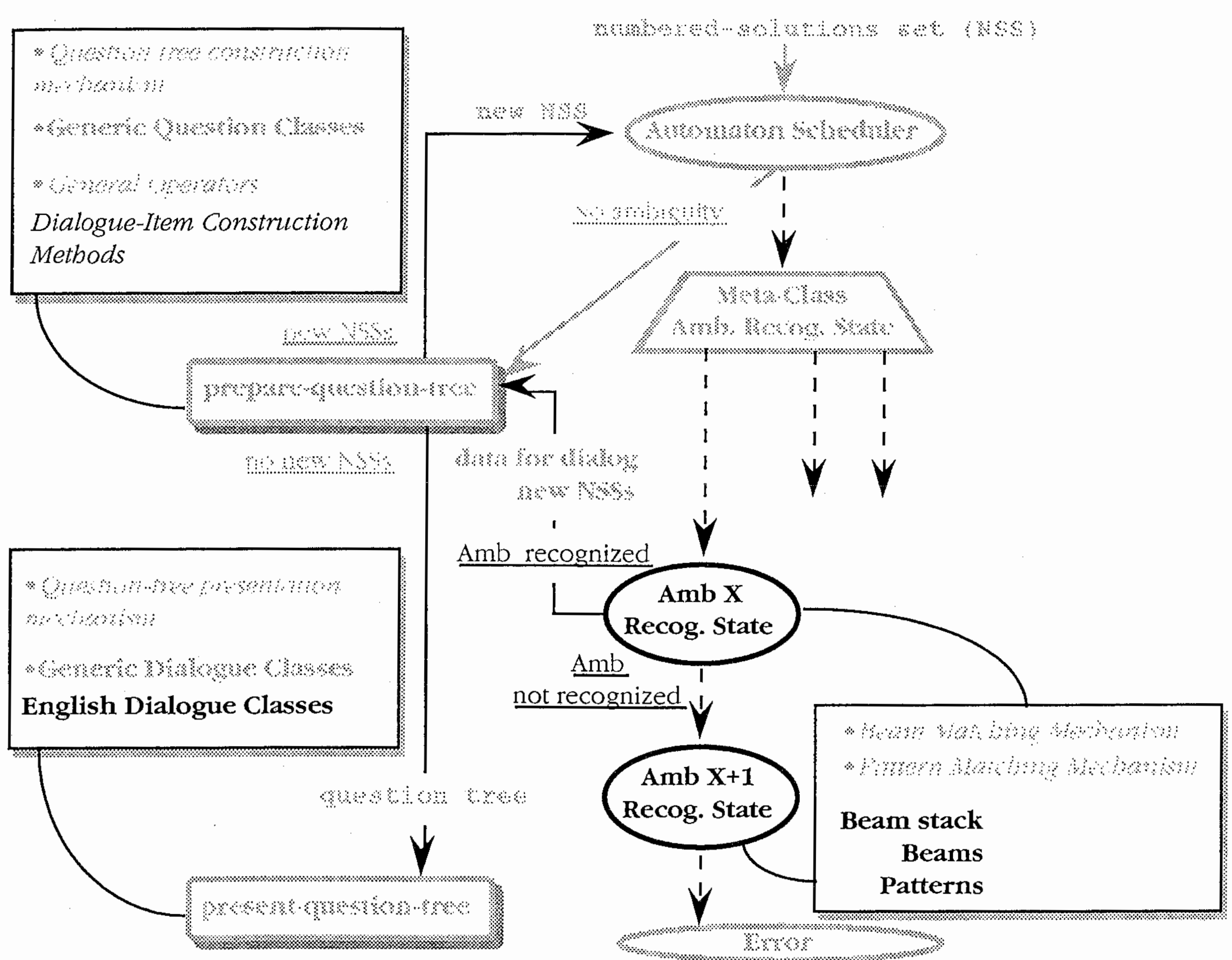


Figure 7: Components of the English disambiguation lingware

The lingware is used to describe all the language dependant parts of a disambiguation module. It should provide the disambiguation engine with a description of the ambiguities to be solved, the

order in which those ambiguities have to be solved, and the way the disambiguation dialogues should be labelled to solve them.

Thus, the lingware consists of:

- a set of beams; made of patterns, describing the ambiguities to be solved,
- a set of beam stacks grouping ambiguities of the same class,
- a set of ambiguity recognition states organized and ordered within a disambiguation automaton,
- a set of dialogue items production methods — one per defined pattern— each one describing what to do in order to construct a rephrasing of the original utterance to be disambiguated,
- a set of dialogue classes describing the language-to-be-disambiguated specific parts of the dialogues to be presented to the user.

Those components are going to be described in the four following chapters. At the end of each chapter, we will discuss the ideas and the implementation proposed. We will also present new ideas and possible or required improvements of the current implementation.

Chapter 5

Ambiguities, Patterns & pattern beams

As shown in the figure bellow, patterns, beams, and stacks are used inside the ambiguity recognition states of the disambiguation automaton.

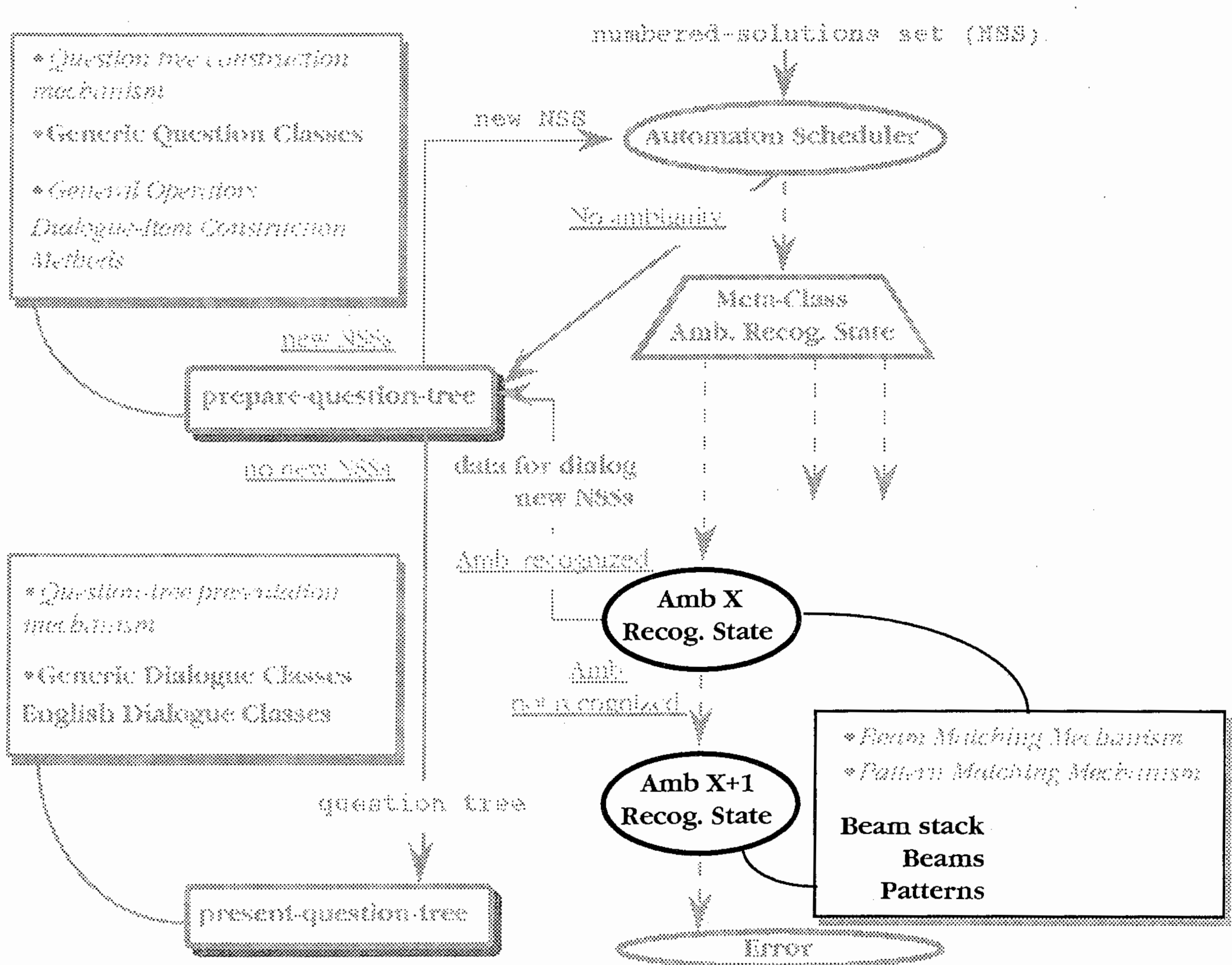


Figure 5.1: The English patterns, beams and stacks in the global architecture

In this chapter we are going to list the ambiguity defined for this first version of an English disambiguation module for English input.

5.1. Situation

We have defined a set of three meta-classes of ambiguities to be used and refined in each instance of a particular disambiguation module. Those meta-classes — lexical ambiguity, geometrical ambiguity, labeling ambiguity — are defined as follows:

- There is a lexical ambiguity when the analyzer is unable to operate a unique segmentation into words or terms ([right here] *vs.* [right] [here]), or unable to choose a word among homophones (to *vs.* too *vs.* two), or unable to choose a syntactic class among homographs (conduct noun *vs.* verb).
- There is a geometrical ambiguity when the analyzer produces several solutions with different geometry without a lexical ambiguity.
- There is a labeling ambiguity when the analyzer produces several solutions with the same geometry without a lexical ambiguity.

The ambiguities which make up the first corpus upon which the disambiguation mechanism was based were taken from a data base of spontaneous speech collected at ATR-ITL.

The conversations, between native speakers of American English, were recorded [Loken-Kim, *et al.* 1993a] during an experiment conducted in the Environment for MultiModal Interaction (EMMI) [Loken-Kim, *et al.* 1993b], and took place via both telephone and multimedia communication contexts [Fais 1994]. The 17 conversations from the experiment, comprising over 8000 words, were examined by hand, and all detected ambiguities were extracted. Ambiguities due solely to polysemy were disregarded; typical examples of all other types of ambiguity were selected to form the final corpus. The chosen sentences and their multiple, multilevel and concrete representations can be found in [Blanchon, *et al.* 1995a].

As far as the description of an ambiguity is concerned, it is somehow an approximation to say that they are described by sets of patterns called beams. Indeed, some ambiguities, namely the lexical ambiguities (at least most of them) can not be described effectively with patterns. They should be described by means of lists' properties. Much work has been done in the field of pattern-defined recognizable ambiguities, and far less for the other kind. We are going to give the results we get For English for the first corpus we used.

5.2. English pattern-defined ambiguities

So far we have defined six categories of pattern-defined ambiguities. The objects defined for the English module are listed in Appendix A.

5.2.1. Second verbal-phrase prepositional attachment

- Example sentence

– Let me pull out my maps to help you.

- Possible analysis trees

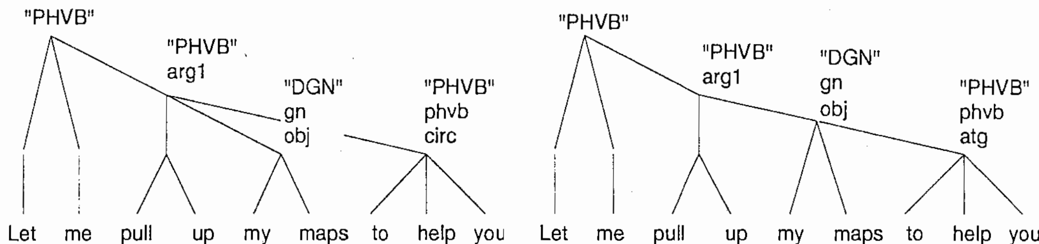


Figure 5.2: Mmc-strctuture for "Let me pull out my maps to help you."

- Patterns

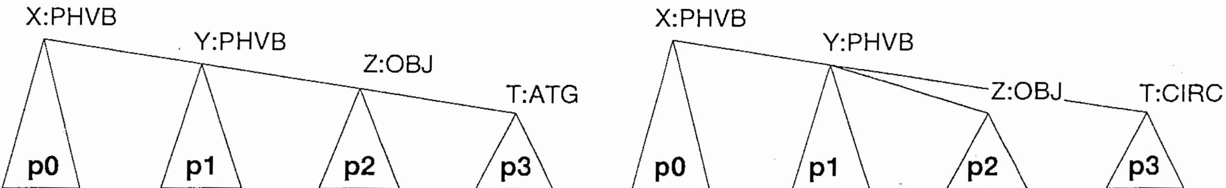


Figure 5.3: The patterns for a Second phvb prepositional attachment ambiguity

5.2.2. Simple adverbial attachment

- Example sentence

– You can pay for it right on the bus.

– It says that here on my flyer.

- Possible analysis trees

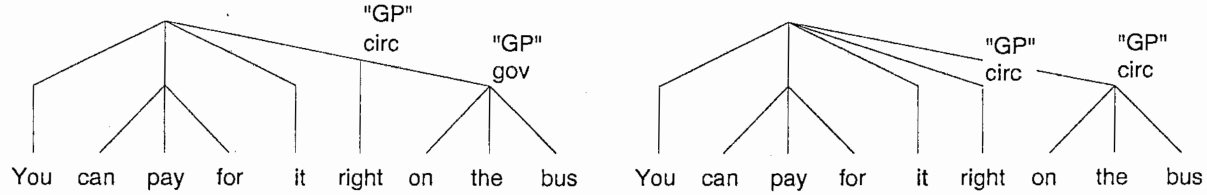


Figure 5.4: Mmc-structure for "You can pay for it right on the bus."

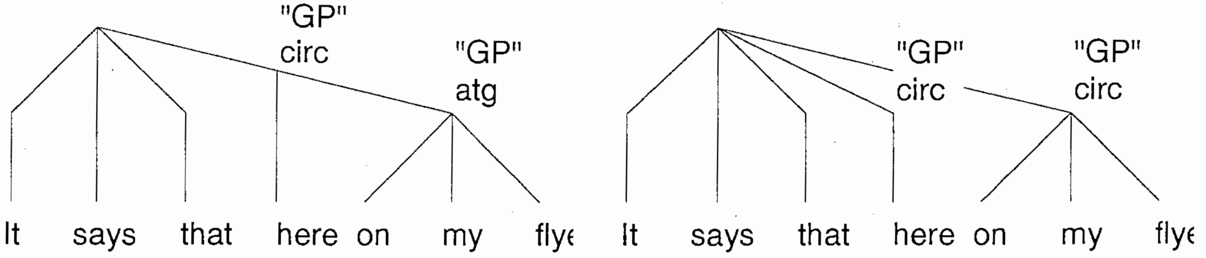


Figure 5.5: Mmc-structure for "It says that here on my flyer."

- Patterns

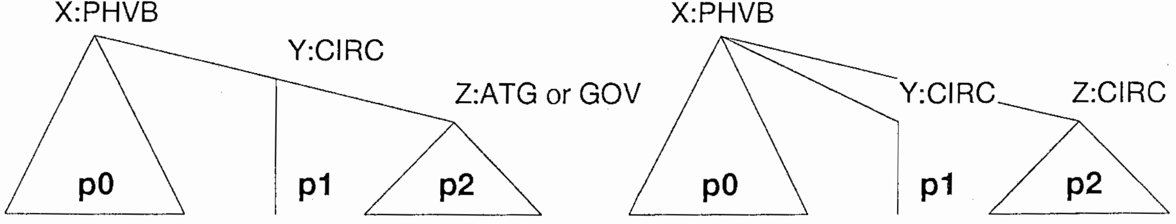


Figure 5.6: The patterns for a Simple adverbial attachment ambiguity

5.2.3. Verbal-phrase prepositional attachment

i. phvb prepositional attachment type 1

- Example sentence

– Where can I catch a taxi from Kyoto station.

- Possible analysis trees

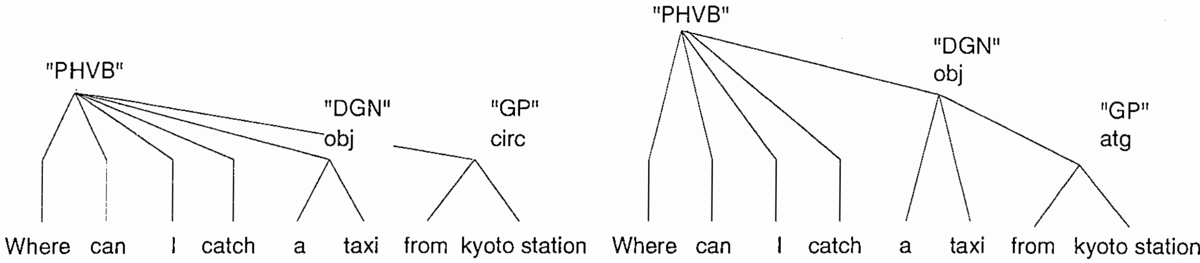


Figure 5.7: Mmc-structure for "Where can I catch a taxi from Kyoto station."

- Patterns

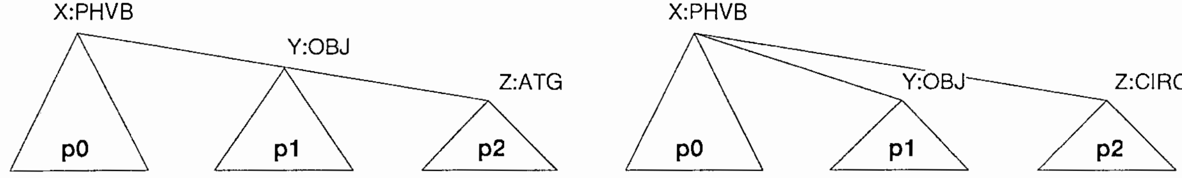


Figure 5.8: The patterns for a phvb prepositional attachment type 1 ambiguity

ii. phvb prepositional attachment type 2

- Example sentence

– Go across the street to the North of the station.

- Possible analysis trees

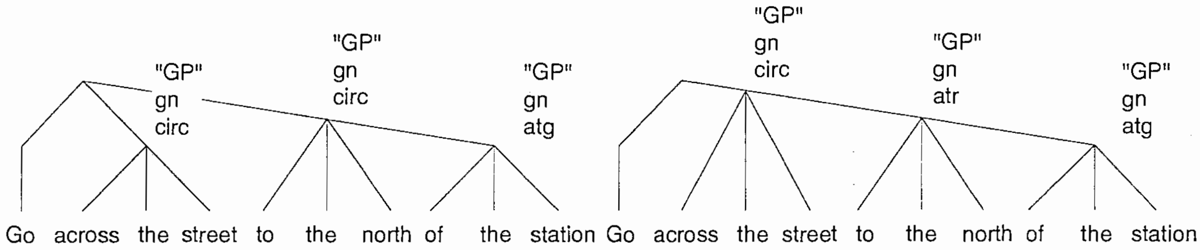


Figure 5.9: Mmc-structure for "Go across the street to the North of the station."

- Patterns

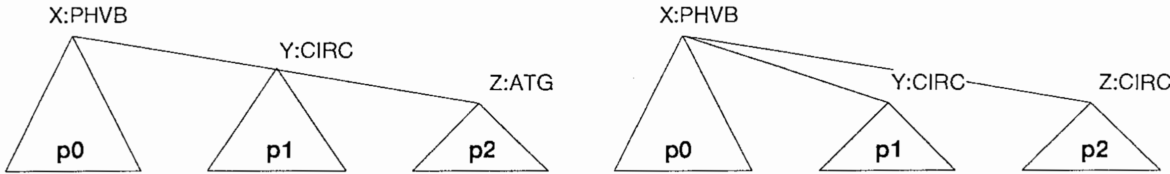


Figure 5.10: The patterns for a phvb prepositional attachment type 2 ambiguity

5.2.4. Relative verbal-phrase adverbial attachment

i. relative phvb adverbial attachment type 1

- Example sentence

– That is where you can pick up a taxi as well.

- Possible analysis trees

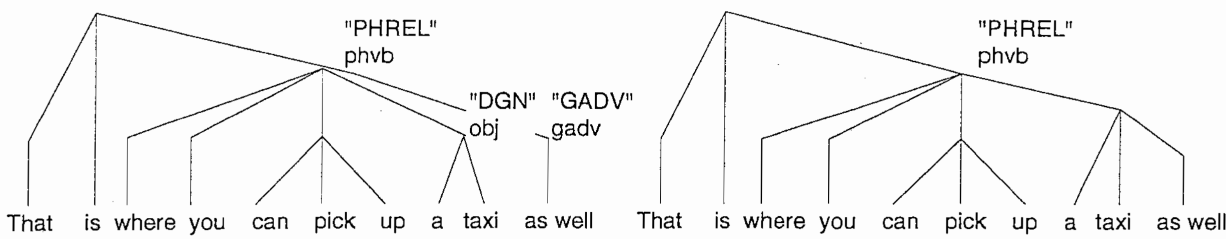


Figure 5.11: Mmc-structure for "That is where you can pick up a taxi as well."

- Patterns

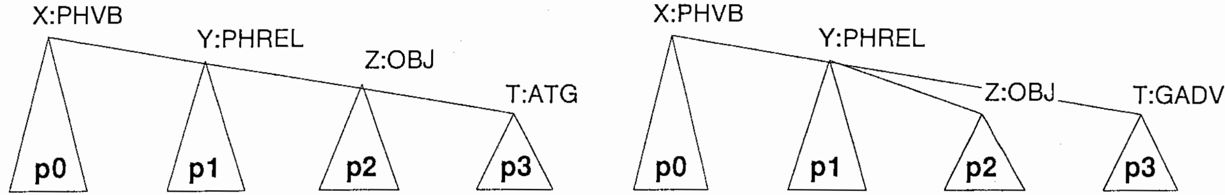


Figure 5.12: The patterns for a relative phvb adverbial attachment type 1 ambiguity

ii. relative phvb adverbial attachment type 2

- Example sentence
  – I will show you where you are located right now.

- Possible analysis trees

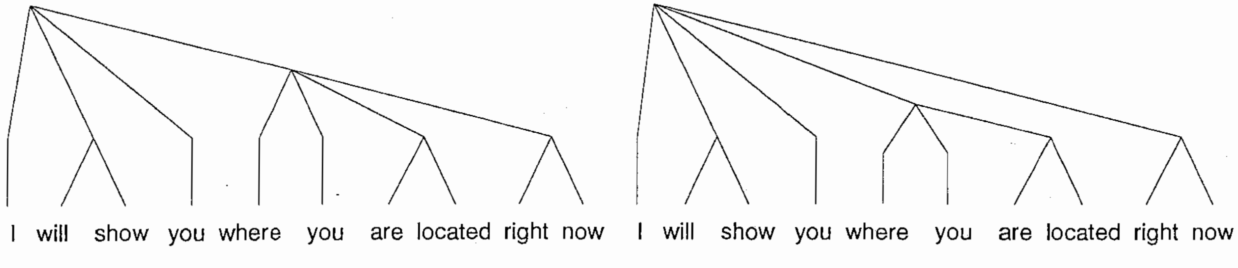


Figure 5.13: Mmc-structure for "I will show you where you are located right now."

- Patterns

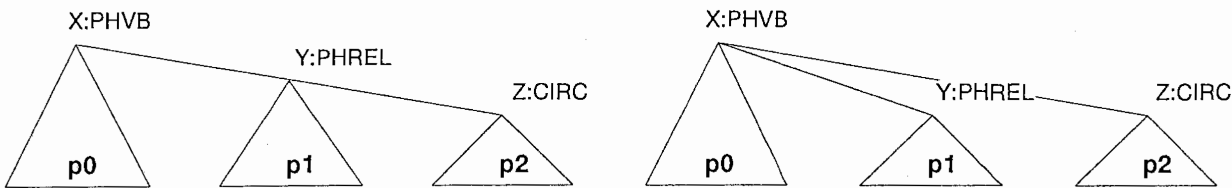


Figure 5.14: The patterns for a phvb prepositional attachment type 2 ambiguity

5.2.5. Verbal-phrase conjunction attachment

- Example sentence
  – You can tell him that you are going to the international conference center and it should be a twenty minute ride.

- Possible analysis trees

You can tell him that you are going to the international conference center and it should be a twenty minute ride

You can tell him that you are going to the international conference center and it should be a twenty minute ride

You can tell him that you are going to the international conference center and it should be a twenty minute ride

You can tell him that you are going to the international conference center and it should be a twenty minute ride

Figure 5.15: Mmc-structure for "You can tell him that you are going to the international conference center and it should be a twenty minute ride."

- Patterns

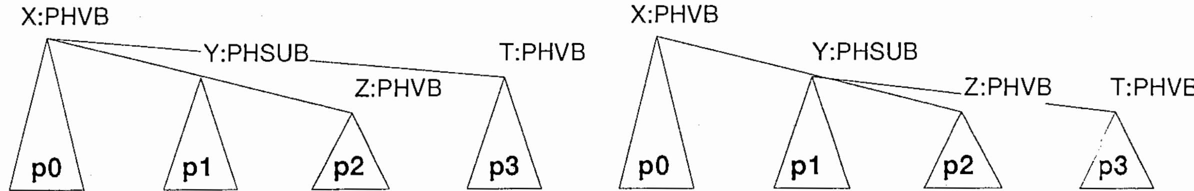


Figure 5.16: The patterns for a Verbal-phrase conjunction attachment ambiguity

5.2.6. Non-verbal-phrase prepositional attachment

i. non phvb prepositional attachment type 1

- Example sentence

– You are going to the international conference center.

- Possible analysis trees

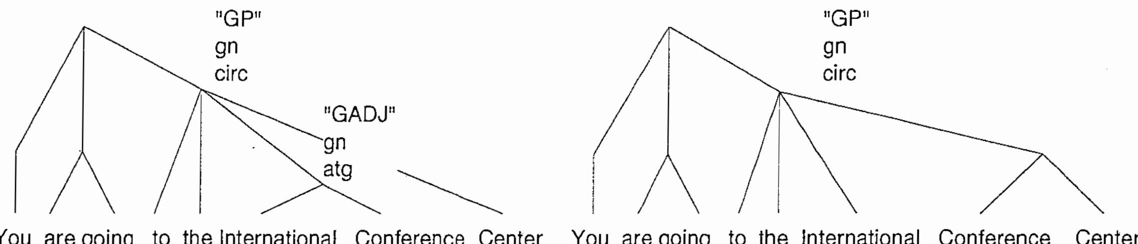


Figure 5.17: Mmc-structure for "You are going to the international conference center."

- Patterns

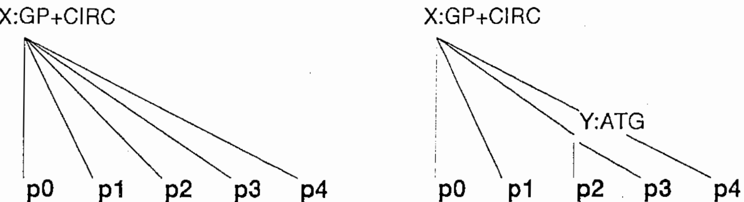


Figure 5.18: The patterns for a non phvb prepositional attachment type 1 ambiguity

ii. non phvb prepositional attachment type 2

- Example sentence

– I want the symposium on interpreting telecommunication at the international conference center.

- Possible analysis trees

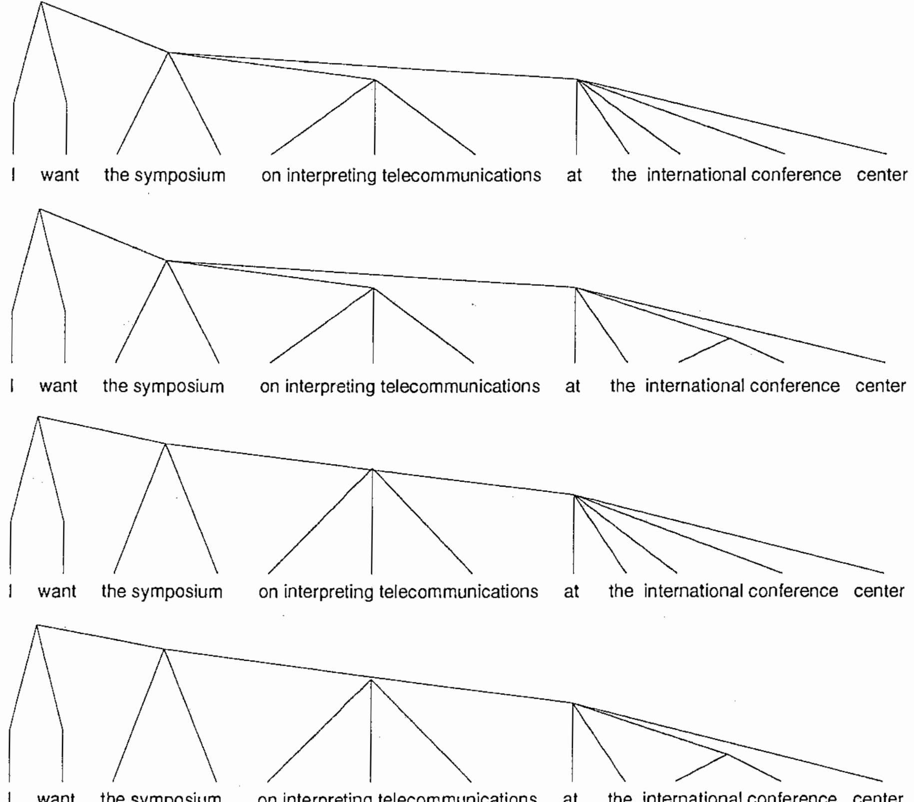


Figure 5.19: Mmc-structure for "I want the symposium on interpreting telecommunication at the international conference center."

- Patterns

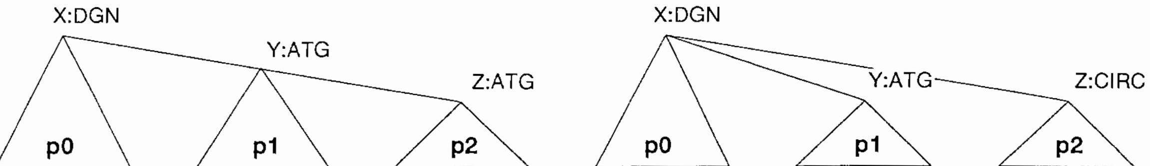


Figure 5.20: The patterns for a non phvb prepositional attachment type 2 ambiguity

5.3. English listed but not solved ambiguities

These ambiguities are those to be recognized by list property recognition mechanism that has not been studied enough for results to be given.

5.3.1. Noun-adjective ambiguity

- Example sentence

– This is an English speaking agent.

– You can catch a taxi at the second level platform.

- Possible analysis trees

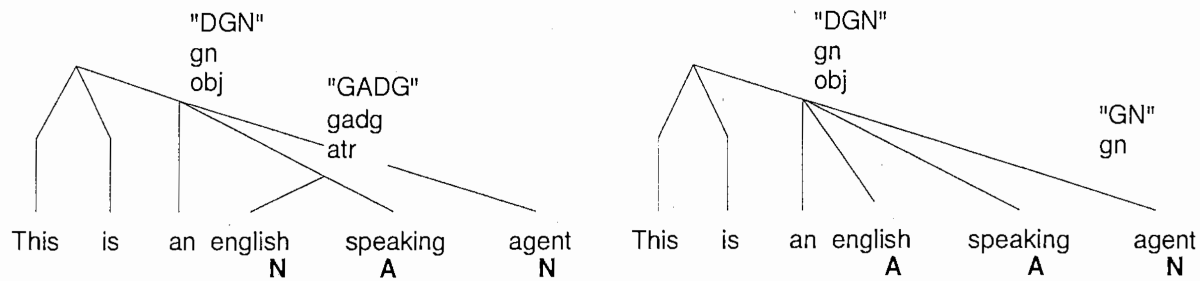


Figure 5.21: Mmc-structure for "This is an English speaking agent."

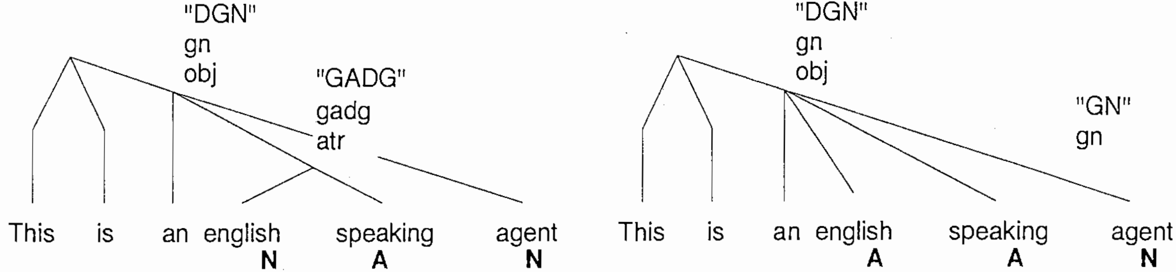


Figure 5.22: Mmc-structure for "You can catch a taxi at the second level platform."

5.3.3. Other syntactic class ambiguities

- Example sentence

– The quickest route would be taking a taxi.

– You can either travel by subway, but or taxi.

- Possible analysis trees

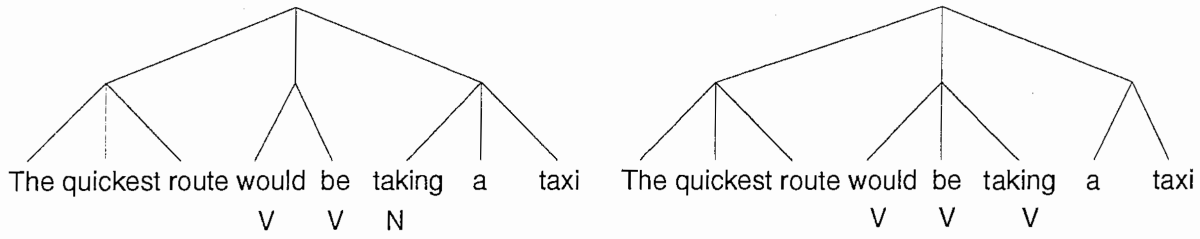


Figure 5.23: Mmc-structure for "The quickest route would be taking a taxi."

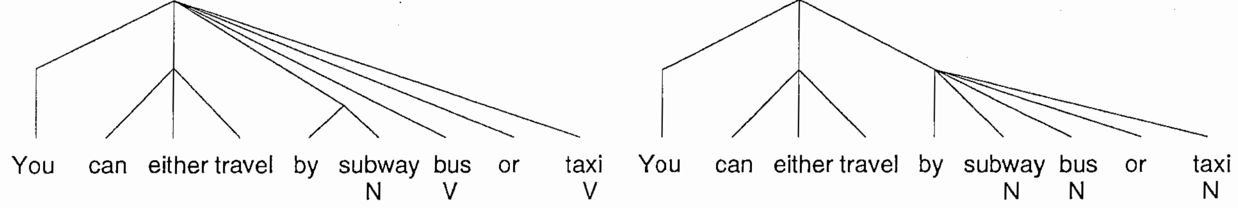


Figure 5.24: Mmc-structure for "You can either travel by subway, but or taxi."

5.3.2. Phrasal verb ambiguity

- Example sentence

– It is difficult to get out of Kyoto station.

– Do you want to go over that again.

- Possible analysis trees

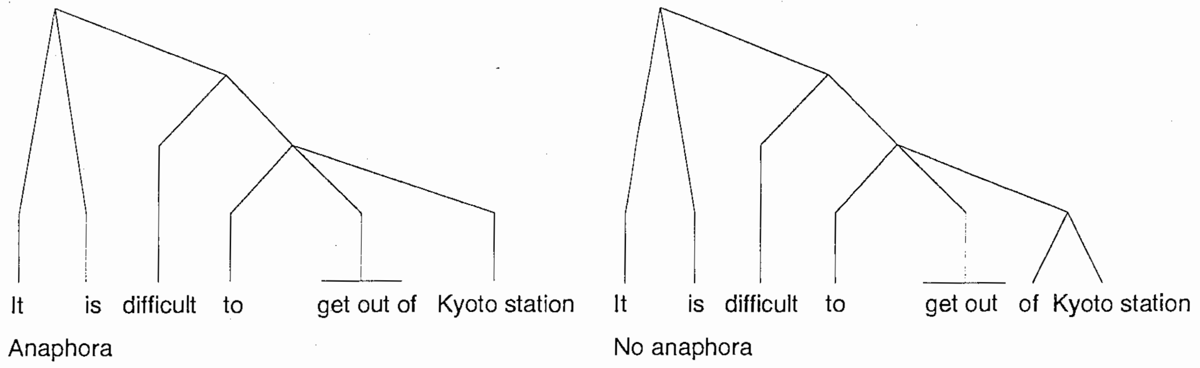


Figure 5.25: Mmc-structure for "It is difficult to get out of Kyoto station."

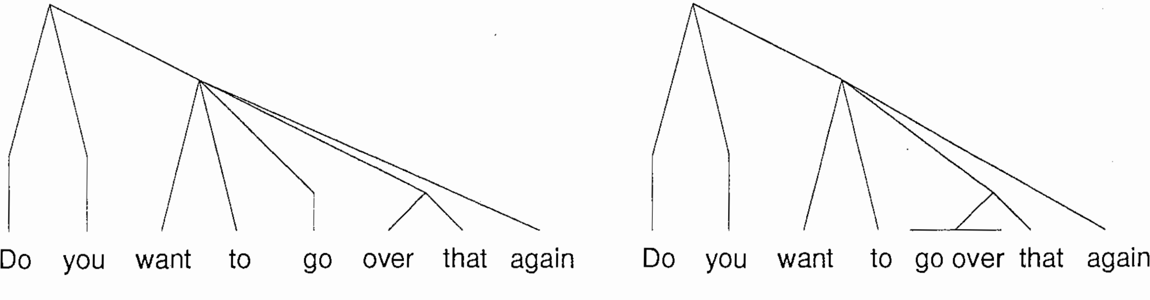


Figure 5.26: Mmc-structure for "Do you want to go over that again."

5.4. Comments

- This description is just a very first draft.
- Work in the field of list-described and recognized ambiguities to be developed.

Chapter 6

Clarification automaton

As shown in the following figure, the language independent part of the disambiguation automaton consists of the language specific ambiguity recognition states.

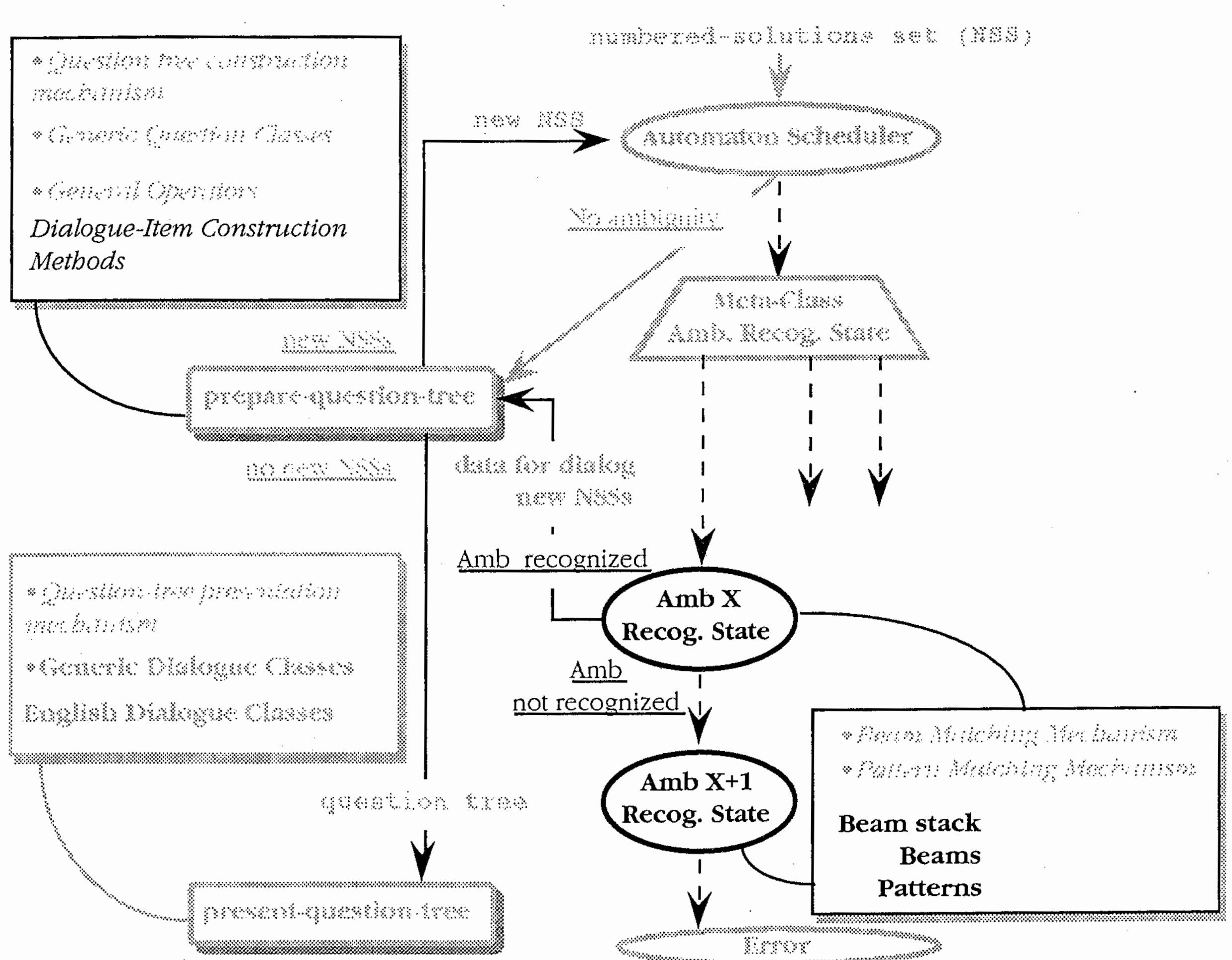


Figure 6.1: The clarification automaton in the global architecture

The disambiguation automaton is made of three kinds of states: an automaton-scheduler, meta-class recognition states, and ambiguity-class recognition states. The ambiguity-class

recognition states are described as a part of the lingware. They will be described here.

There is two kinds of ambiguity-class recognition states: those using the beam matching mechanism for the ambiguities described with beams (cf. chapter 5), and the other ones using a property recognition mechanism (for the ambiguities not described with beams). The work on the second category of states is not yet enough advanced to be described here.

6.1. The beam matching ambiguity recognition states

The beam matching ambiguity class recognition states are described by methods sharing a common skeleton described bellow:

```
(defmethod ambX-recognition-state ((the_language (eql 'english))
                                    the_sentence
                                    the_numbered_analysis_list)
         ;the related beam is attached to the state
  (let* ((the_beam_stack the-stack-describing-ambX)
         ;looking for a beam matching the_numbered_analysis_list
         (the_beam_stack_match (beam-stack-match the_beam_stack
                                                 the_numbered_analysis_list))
         ;was a matching found?
         (matched? (first the_beam_stack_match)))
    (if matched?
      ;if a matching has been found then the searched ambiguity is recognized
            ;instanciation of the ambiguity type that will fixe some
             parameters about the dialogue labelling
      (let ((the_type 'general)
            ;instanciation of the dialogue modality
            (the_modality 'textual)
            ;this list_of_triplets will be used to build the dialogue items
            (the_list_of_triplets (third the_beam_stack_match))
            ;this new_solution_sets will be used to produce the follow up
             questions
            (the_new_solution_sets (fourth the_beam_stack_match)))
        ;continuation of the construction of the question tree
        (prepare-question-tree the_language
                               the_type
                               the_modality
                               the_sentence
                               the_list_of_triplets
                               the_new_solution_sets))
      ;no matching were found, the next state is triggered
      (ambX+1-recognition-state the_language the_sentence
                                the_numbered_analysis_list))))
```

Figure 6.2: Skeleton of an ambiguity recognition state

6.3. The implemented automaton

With the ambiguities defined in the paragraphs 5.2. and 5.3. the following automaton has been defined and implemented.

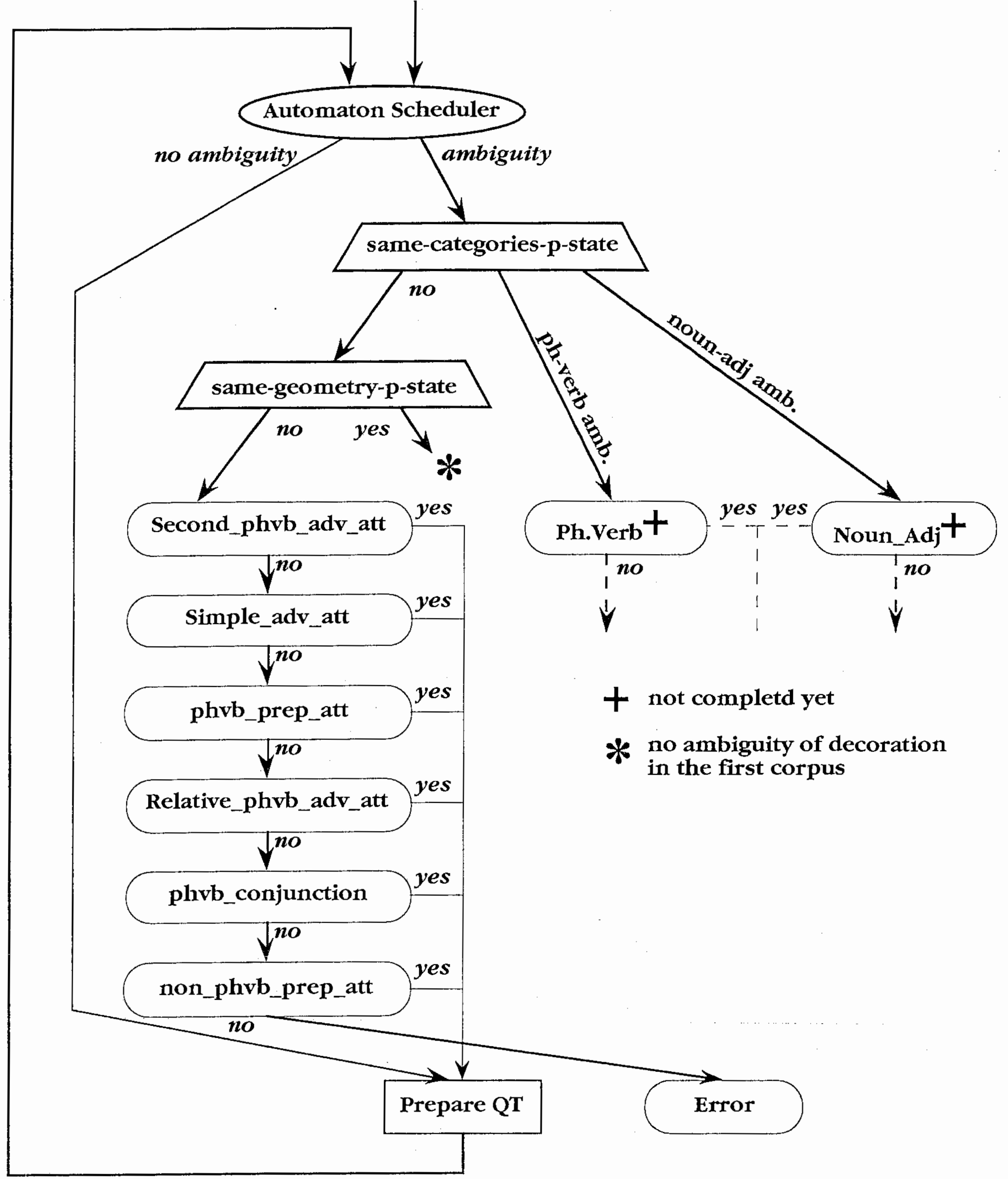


Figure 6.3: The implemented disambiguation automaton

6.3. Comments

- Resolution of the lexical ambiguities: THE problem to be tackled.
- A dynamic automaton via an evolving transition matrix
- Forms for the designer to describe the states of the automaton, the patterns, the dialogue item-production methods, ...

Chapter 7

Dialogues classes

For the language and the ambiguity (inside this language) dependent elements of the disambiguation dialogues to fit imposed constraints (font used, size, style, metalanguage utterances for the presentation of the question), English dialogue classes have been defined.

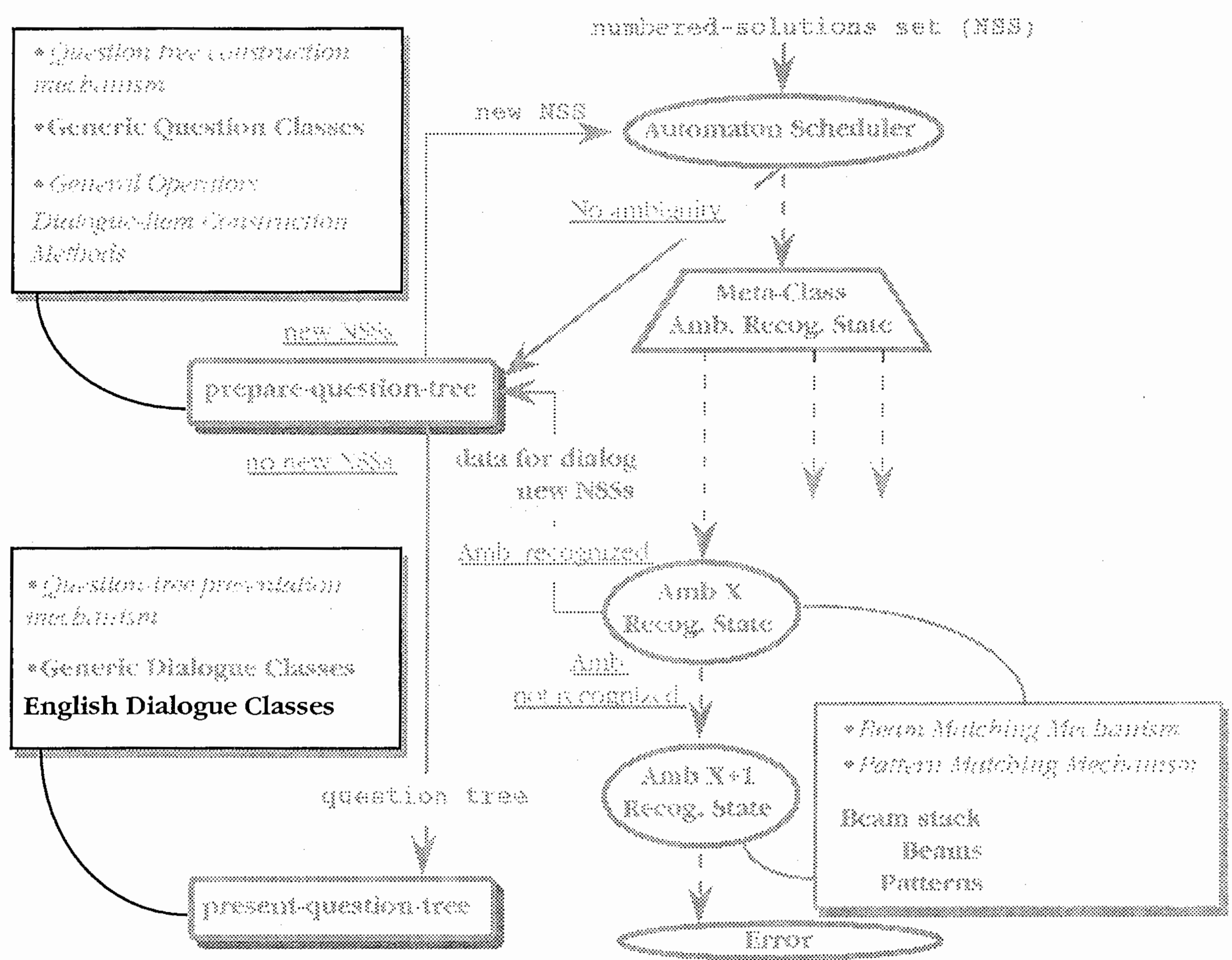


Figure 7.1: The English dialogue classes in the general architecture

7.1. Language dependent constraints

As we are preparing dialogues to be labeled in the language disambiguated, the whole text of the dialogues must be labeled in the disambiguated language. So if the "look and feel" of the dialogue may be language-independent, the "static text" appearing in the dialogue will have to change.

In the designed dialogues, window-titles, invitation-strings and prompt strings (shown bellows) are parts of the "static text" labeled dialogue elements.

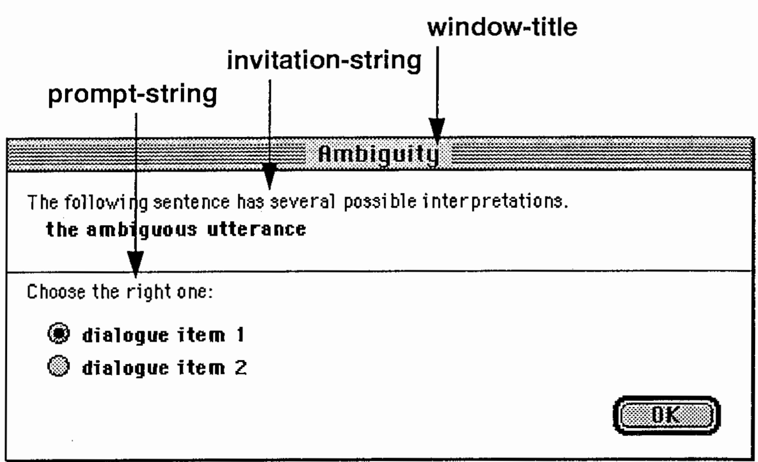


Figure 7.2: Some dialogues' slots

7.2. Dialogue classes

Two specializations of the generic-textual-clarif-dialog-class have been proposed as follows:

– on for the ambiguities of polysemy, and

```
(defclass english-general-textual-dialog-class (generic-textual-clarif-dialog-class)
 ((window-length
   :initform 550)
  (invitation-string
   :initform "The following sentence has several possible interpretations.")
  (invitation-string-font
   :initform '("helvetica" 16))
  (ambiguous-string-font
   :initform '("helvetica" 16 :bold))
  (prompt-string
   :initform "Choose the right one:")
  (prompt-string-font
   :initform '("helvetica" 16))
  (items-font
   :initform '("helvetica" 16 :bold)))
 (:default-initargs
   :window-title "Ambiguity"))
```

Figure 7.3: The lingware class english-general-textual-dialog-class

– the other one for all the other ambiguities.

```
(defclass english-polysemy-textual-dialog-class (generic-textual-clarif-dialog-class)
 ((window-length
   :initform 500)
  (invitation-string
   :initform "The following word has several possible meanings.")
  (invitation-string-font
   :initform '("helvetica" 16))
  (ambiguous-string-font
   :initform '("helvetica" 16 :bold))
  (prompt-string
  :initform "Choose the right one:")
  (prompt-string-font
   :initform '("helvetica" 16 10))
  (items-font
   :initform '("helvetica" 16 :bold)))
 (:default-initargs
  :window-title "Ambiguity"))
```

Figure 7.4: The lingware class english-polysemy-textual-dialog-class

7.3. Comments

- Lingware designer defined dialogue classes, how far can we go in the customization of the presentation? In the current implementation, the designer can change only the static strings, and the font, size and style of what is displayed. Access to a dialogue presentation design environment??
- Forms for the description of the dialogue classes.

Chapter 8

Dialogue item production methods

The dialogue items are actually produced with the dialogue item construction method associated with each pattern.

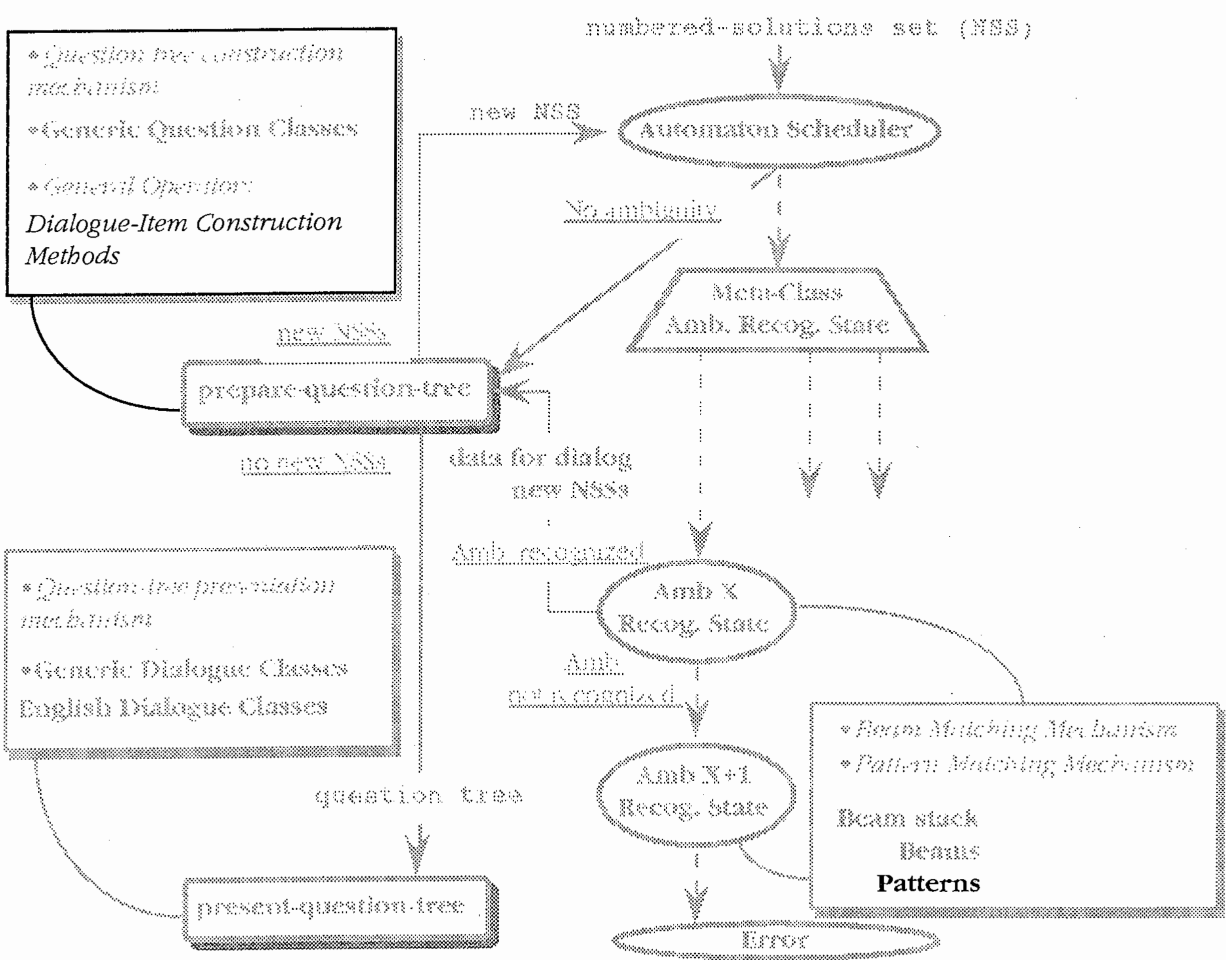


Figure 8.1: The dialogue item production methods in the global architecture

8.1. Principle

Each method produces a string of characters which is an arrangement or a manipulation, with the operators defined in § 2.4, of the binding of some of the variables defined in the pattern the method is associated with. These methods use the lisp-defined function format. The function format is very useful for producing nicely formatted text, producing good-looking messages, and so on. format can generate a string or output to a stream.

The function format is defined as follows[3]:

[Function]

format *destination control-string* &rest *arguments*

format is used to produce formatted output. format outputs the characters of *control-string*, except that a tilde (~) introduces a directive. The typical directive puts the next element of *arguments* into the output, formatted in some special way.

The output is sent to *destination*. If destination is nil, a string is created that contains the output; this string is returned as the value of the call to format.

With the directive ~A[4], an *arg*, any Lisp object, is printed without escape characters (as by princ). In particular, if *arg* is a string, its characters will be output verbatim.

Example

```
(setq y "elephant")
(format nil "Look at the ~A!" y) => "Look at the elephant!"
```

8.2. Methods skeleton

The dialogue item production methods are described with a set of **item-production-method** methods. Such methods have two parameters:

- a pattern-name, which is the specializer of the method,
- a binding, which represents, for the current situation, the value of each of the variables defined in the patterns.

One of these methods is typically defined as follows:

```
(defmethod item-production-method
                        ((pattern-name (eql '*phvbprepatt-t1-1*)) binding)
  (format nil  "~A (~A ~A)."
          (apply #'text (cdr (assoc '?p0 binding)))
          (apply #'text (cdr (assoc '?p1 binding)))
          (apply #'text (cdr (assoc '?p2 binding)))))
```

Figure 8.2: A typical item-production-method method

This method will produce the following string: text(?p0) **(** text(?p1) text(?p2) **).** A complete list of the defined methods is given in Appendix B.

---

[3] The definition is taken from [Steele 1990].

[4] In the descriptions of the directive that follow, the term arg in general refers to the next item of the set of arguments to be processed.

8.3. Comments

- Specializing a method on the pattern-name may not be a good idea. The designer would never do that.
- Provide a means to describe the methods without having to write a lisp program.
- Describe the methods in a language using the operators and a metalanguage (bracketing, point, comma, etc...) directly associated with the patterns.
- A patterns+methods editor. A beam editor. A stack editor.

Conclusion

On the methodology

The methodology we proposed allows to develop customized disambiguation modules that can be easily improved incrementally.

The ability to customize comes from the clear separation of the lingware from the engine. In this framework, different disambiguation modules can be produced for one or several different languages and kinds of input.

The description of the linguistic data can be improved incrementally as the design and the use of a disambiguation module progress.

Certainly, we do not claim that any given module will cover all the ambiguities found in natural language (written or written). On the other hand, we claim that a given module for a given application can incrementally reach a broad coverage for the application it has been designed to be integrated into.

We imagine the following figure.

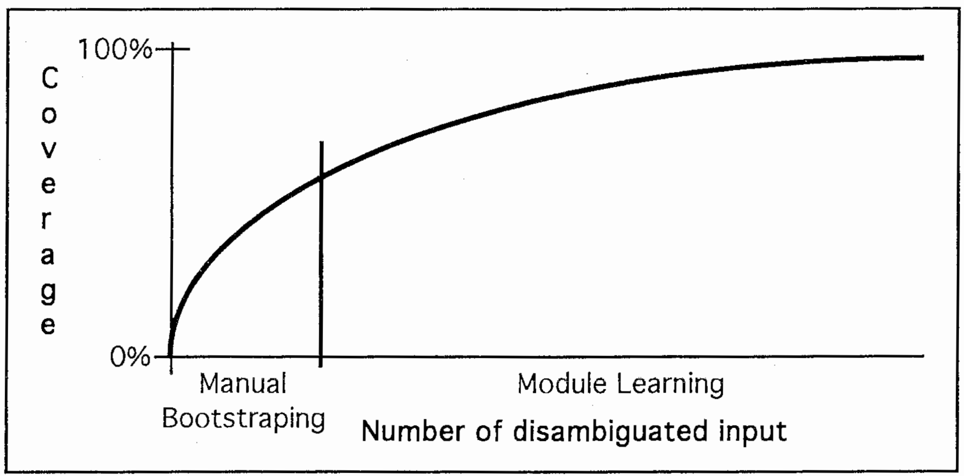


Figure 8: Evolution of the coverage of a given disambiguation module

Global perspectives

In addition to the comments given at the end of each chapter, we can propose broader perspectives and evaluations of this work.

We think that, whenever a system is using a natural language analysis module, the evaluation criterion must not only the task's time-completion, more important is the user satisfaction. That

is why we feel that it is very important to study the design of the clarification sessions, and moreover the design of the questions to be asked. We are aware that this kind of study is energy and time consuming but we can not afford to do without.

Thus, we have proposed to run experiments to study, before other things, the understandability of the proposed disambiguation dialogues. Although we were just able to run only a pilot experiment [Blanchon & Fais 1995] within a too short imposed delay, we have gathered some interesting feedback. We learnt also, and that is very important, how not to design such experiment. We strongly hope that this work on understandability and assessment will have a chance to be carried on.

There is also a need for a kind of real coverage evaluation. For this evaluation to be conduced properly, a large amount of data (analysis) should be provided. There is a possible opportunity to do this using the results produced in the framework of the EBMT system developed at ATR-ITL [Furuse & Iida 1992 ; Furuse & Iida 1994]. Those results that are numerous enough for this study to be meaningful. In this context, we would finally be able to use a huge set of real data produced by an analyzer and not a small set of handmade data.

In the context of the system to be developed a ATR, any data collected through a simulated environment (such as EMMI) or any actual data produced by an analyzer should be carefully analyzed. The analysis should allow to study the kind and the frequency of odd meanings (those not likely to be ranked first by an analyzer). It should allow also to evaluate the consequences of possible mistanslations, and then establish the need or un-need for interactive disambiguation.

Finally, to complement the first studies on French and English, we will be very happy to carry out a similar study for Japanese.

Recommendation

If we were asked to propose a recommendation for the system final system (or demonstrator) to be developed at ATR-ITL, we would say that is it necessary to built an analyzer (speech to concrete analysis) or a set of analyzers (speech to text, and text to concrete analysis) producing multiple ranked structures to leave the door open to interactive disambiguation which may reveal itself indispensable for the targeted task to be achieved correctly at the satisfaction of real potential users.

References

Blanchon H. (1990). *LIDIA-1 : Un prototype de TAO personnelle pour rédacteur unilingue.* Proc. Avignon-90, conférence spécialisée : Le Traitement Automatique des Langues Naturelles et ses Applications. Avignon, France. 28 mai-1 juin 1990, vol. **1/1** : pp. 51-60.

Blanchon H. (1992). *A Solution to the Problem of Interactive Disambiguation.* Proc. Coling-92. Nantes, France. July 23-28, 1992, vol. **4/4** : pp. 1233-1238.

Blanchon H. (1994a). *LIDIA-1 : une première maquette vers la TA interactive "pour tous".* Thèse. Université Joseph Fourier - Grenoble 1. 1994a. 321 p.

Blanchon H. (1994b). *Pattern-based approach to interactive disambiguation: first definition and implementation.* Rap. ATR-Interpreting Telecommunications Research Laboratories. Technical Report. n° TR-IT-0073. Sept. 8, 1994. 91 p.

Blanchon H. (1994c). *Perspectives of DBMT for monolingual authors on the basis of LIDIA-1, an implemented mock-up.* Proc. Coling-94. Kyoto, Japan. August 5-9, 1994, vol. **1/2** : pp. 115-119.

Blanchon H. & Fais L. (1995). *Pilot Experiment on the Understandability of Interactive Disambiguation Dialogues.* Rap. ATR-ITL. Technical repport. n° TR-IT-0130. September, 1995. To be published.

Blanchon H., Fais L. & Guilbaud J.-P. (1995a). *A Corpus of Ambiguous English Sentences with their Multisolution, Multilevel and Concrete Tree Representations.* Rap. ATR-ITL. Technical Report. n° TR-IT-0131. September, 1995. To be published.

Blanchon H., Fais L., Loken-Kim K. H. & Morimoto T. (1995b). *A Pattern-Based Approach for Interactive Clarification of Natural Language Utterances.* in Bulletin of the Information Processing Society of Japan (95-NL-107). vol. **95**(52) : pp. 11-18.

Blanchon H. & Loken-Kim K. H. (1994). *Towards More Robust, Fault-Tolerant and User-Friendly Software Integrating Natural Language Processing Components.* in Bulletin of the Information Processing Society of Japan (94-SLP-4). vol. **94**(109) : pp. 17-24.

Boitet C. (1989). *Motivations, aims and architecture of the Lidia project.* Proc. MT SUUMMIT II. Munich. 16-18 août 1989, vol. **1/1** : pp. 53-57.

Boitet C. (1990). *Towards Personnal MT : general design, dialogue structure, potential role of speech.* Proc. Coling-90. Helsinki. 20-25 Août 1990, vol. **3/3** : pp. 30-35.

Boitet C. & Blanchon H. (1995). *Multilingual Dialogue-Based MT for monolingual authors: the LIDIA project and a first mockup.* in Machine Translation. vol. **9**(2) : pp. 99-132.

Boitet C. & Tomokiyo M. (1995). *Towards ambiguity labelling for the study of interactive disambiguation methods.* Rap. ATR-ITL. Technical Repport. n° TR-IT-0112. April 27, 1995. 22 p.

Caelen J. (1994). *Multimodal Human-Computer Interaction.* in Fundamentals of Speech Synthesis and Speech Recognition. Keller, E. (ed.). John Wiley & Sons. New York. pp. 339-373.

Fais L. (1994). *Effects of communicative mode on spontaneous English speech.* Rap. Institute of Electronics, Information and Communication Engineers. Technical Report. n° NLC94-22. Oct. 94. 6 p.

Furuse O. & Iida H. (1992). *Cooperation Between Transfert and Analysis in Example-Based Framework.* Proc. Coling-92. Nantes, France. July 23-28, 1992, vol. **2/4** : pp. 645-651.

Furuse O. & Iida H. (1994). *Constutuant Boundary Parsing for Example-Based Machine Translation.* Proc. Coling-94. Kyoto, Japan. August 5-9, 1994, vol. **1/2** : pp. 105-111.

Goddeau D., Brill E., Glass J., Pao C., Philips M., Polifroni J., Seneff S. & Zue V. (1994). *GALAXY: a Human-Language Interface to On-Line Travel Information.* Proc. ICSLP 94. Yokohama, Japan. September 18-22, 1994, vol. **2/4** : pp. 707-710.

Haddock N. J. (1992). *Multimodal Database Query.* Proc. Coling-92. Nantes, France. 23-28 juillet 1992, vol. **4/4** : pp. 1274-1278.

Hiyoshi M. & Shimazu H. (1994). *Drawing Pictures with Natural Language and Direct Manipulation.* Proc. Coling-94. Kyoto, Japan. August 5-9, 1994, vol. **2/2** : pp. 722-726.

Kay M., Gawron J. M. & Norvig P. (1994). *Verbmobil: A Translation System for Face-to-Face Dialog. CSLI lecture note no 33.* Center for the Study of Language and Information, Stanford, CA. 235 p.

Keene S. E. (1989). *Object-Oriented Programming in Common Lisp: A Programmer's Guide to CLOS.* Addison-Wesley Publishing Compagny. New York. 266 p.

Loken-Kim K.-H., Yato F., Fais L., Kurihara K., Furukawa R. & Kitagawa Y. (1993a). *マルチモーダル・シミュレータEMMIを用いた道案内データベースのテキスト.* Rap. ATR-ITL. Technical Report. n° TR-IT-0029. December, 1993. 101 p.

Loken-Kim K.-H., Yato F., Kurihara K., Fais L. & Furukawa R. (1993b). *EMMI-ATR environment for multi-modal interactions.* Rap. ATR-ITL. Technical Report. n° TR-IT-0018. Sept 30, 1993. 28 p.

Morimoto T., Suzuki M., Takezawa T., Kikui G., Nagata M. & Tomokio M. (1992). *A Spoken Language Translation System: SL-TRANS2.* Proc. Coling-92. Nantes, France. 23-28 juillet 1992, vol. **3/4** : pp. 1048-1052.

Nishimoto T., Shida N., Kobayashi T. & Shirai K. (1994). *Multimodal Drawing Tool Using Speech, Mouse and Key-Board.* Proc. ICSLP 94. Yokohama, Japan. September 18-22, 1994, vol. **3/4** : pp. 1287-1290.

Norvig P. (1992). *Paradigms of Artificial Intelligence Programming: Case Studies in Common Lisp.* Morgan Kaufmann Publishers. San Mateo, California. 945 p.

Steele G. L. J. (1990). *Common Lisp - The language.* Digital Press. 1027 p.

Zue V., Seneff S., Polifroni J. & Phillips M. (1993). *PEGASUS: a Spoken Dialogue Interface for On-Line Air Travel Planing.* Proc. ISSD-93 — New Directions in Human and Man-Machine Communication. International Conference Center, Waseda University, Tokyo, Japan. November 10-12, 1993, vol. **1/1** : pp. 157-160.

Appendix

Appendix A

Patterns, beams, & stacks defined

```
;;;---------------------------------------------------------------------------
;;;
;;; Title         : English Disambiguation Patterns
;;;
;;;---------------------------------------------------------------------------
;;; Author        : Herve Blanchon
;;; Address       : Advanced Telecommunication Research Labs
;;;               : Interpreting Telephony Research Labs.
;;;               : 2-2 Hikaridai, Seika-cho Soraku-gun
;;;               : Kyoto,Japan 619-02
;;;               : Blanchon@itl.atr.co.jp
;;;---------------------------------------------------------------------------
;;; Filename      : Disamb_English-Patterns.lisp
;;; Version       : 1.0
;;; Abstract      : Here are the disambiguation patterns used for English
;;;               : For on ambiguity the patterns are ordrered on a stack
;;; History       : 01/23/95 Herve Blanchon
;;; Bugs          :
;;; Todo          :
;;;---------------------------------------------------------------------------
```

```
;;;
;;;------------------------------------------------------------
;;;   Patterns for second phvb adverbial attachment
;;;
;;; Author       : Herve Blanchon
;;; Date         : 01/23/95
;;; Abstract     :
;;;------------------------------------------------------------
;;;
```

```
(defvar *2phvbadvatt-1* (make-instance 'pattern
                        :pattern-name '*2phvbadvatt-1*
                        :pattern-value '((?is ?x node-prop-equal-p 'CS 'PHVB)
                                         (?+ ?p0)
                                         ((?is ?y node-prop-equal-p 'CS 'PHVB)
                                          (?+ ?p1)
                                          ((?is ?z node-prop-equal-p 'FS 'OBJ)
                                           (?+ ?p2)
                                           ((?is ?t node-prop-equal-p 'FS 'ATG)
                                            (?+ ?p3)))))
                        :pattern-method #'item-production-method))
```

*Figure A.1: The pattern *2phvbadvatt-1**

```
(defvar *2phvbadvatt-2* (make-instance 'pattern
                        :pattern-name '*2phvbadvatt-2*
                        :pattern-value '((?is ?x node-prop-equal-p 'CS 'PHVB)
                                         (?+ ?p0)
                                         ((?is ?y node-prop-equal-p 'CS 'PHVB)
                                          (?+ ?p1)
                                          ((?is ?z node-prop-equal-p 'FS 'OBJ)
                                           (?+ ?p2))
                                          ((?is ?t node-prop-equal-p 'FS 'CIRC)
                                           (?+ ?p3))))
                        :pattern-method #'item-production-method))
```

*Figure A.2: The pattern *2phvbadvatt-1**

```
(defvar *2phvbadvatt_set_1* (make-instance 'pattern-beam
                            :beam-name '*2phvbadvatt_set_1*
                            :beam-value (list *2phvbadvatt-1* *2phvbadvatt-2*)))
```

*Figure A.3: The beam *2phvbadvatt_set_1**

```
(defvar *2phvbadvatt_beam_stack* (make-instance 'beam-stack
                                 :beam-stack-name '*2phvbadvatt_beam_stack*
                                 :beam-stack-value (list *2phvbadvatt_set_1*)))
```

*Figure A.4: The stack *2phvbadvatt_beam_stack**

```
;;;
;;;------------------------------------------------------------------
;;;     Simple adverbial attachment
;;;
;;; Author       : Herve Blanchon
;;; Date         : 01/23/95
;;; Abstract     :
;;;------------------------------------------------------------------
;;;
```

```
(defvar *spladvatt-1* (make-instance 'pattern
                       :pattern-name '*spladvatt-1*
                       :pattern-value '((?is ?x node-prop-equal-p 'CS 'PHVB)
                                        (?+ ?p0)
                                        ((?is ?y node-prop-equal-p 'K 'GADV)
                                         (?+ ?p1)
                                         ((?is ?z node-prop-in-p 'FS 'CIRC 'GOV)
                                          (?+ ?p2))))
                       :pattern-method #'item-production-method))
```

*Figure A.5: The pattern *2phvbadvatt-1**

```
(defvar *spladvatt-2* (make-instance 'pattern
                       :pattern-name '*spladvatt-2*
                       :pattern-value '((?is ?x node-prop-equal-p 'CS 'PHVB)
                                        (?+ ?p0)
                                        ((?is ?y node-prop-equal-p 'K 'GADV)
                                         (?+ ?p1))
                                        ((?is ?z node-prop-equal-p 'FS 'CIRC)
                                         (?+ ?p2)))
                       :pattern-method #'item-production-method))
```

*Figure A.6: The pattern *2phvbadvatt-1**

```
(defvar *spladvatt_set_1* (make-instance 'pattern-beam
                           :beam-name '*spladvatt_set_1*
                           :beam-value (list *spladvatt-1* *spladvatt-2*)))
```

*Figure A.7: The beam *2phvbadvatt_set_1**

```
(defvar *spladvatt_beam_stack* (make-instance 'beam-stack
                                :beam-stack-name '*spladvatt_beam_stack*
                                :beam-stack-value (list *spladvatt_set_1*)))
```

*Figure A.8: The stack *2phvbadvatt_beam_stack**

```
;;;
;;;-----------------------------------------------------------------
;;;    phvb prepositional attachment
;;; Author        : Herve Blanchon
;;; Date          : 10/19/94
;;; Abstract      :
;;;-----------------------------------------------------------------
;;;
```

```
;;;
;;;-----------------------------------------------------------------
;;;    phvb prepositional attachment type 1
;;; Author        : Herve Blanchon
;;; Date          : 10/19/94
;;; Abstract      :
;;;-----------------------------------------------------------------
;;;
```

```
(defvar *phvbprepatt-t1-1* (make-instance 'pattern
                            :pattern-name '*phvbprepatt-t1-1*
                            :pattern-value '((?is ?x node-prop-equal-p 'CS 'PHVB)
                                             (?+ ?p0)
                                             ((?is ?y node-prop-equal-p 'FS 'OBJ)
                                              (?+ ?p1)
                                              ((?is ?z node-prop-equal-p 'FS 'ATG)
                                               (?+ ?p2))))
                            :pattern-method #'item-production-method))
```

*Figure A.9: The pattern *2phvbadvatt-1**

```
(defvar *phvbprepatt-t1-2* (make-instance 'pattern
                            :pattern-name '*phvbprepatt-t1-2*
                            :pattern-value '((?is ?x node-prop-equal-p 'CS 'PHVB)
                                             (?+ ?p0)
                                             ((?is ?y node-prop-equal-p 'FS 'OBJ)
                                              (?+ ?p1))
                                             ((?is ?z node-prop-equal-p 'FS 'CIRC)
                                              (?+ ?p2)))
                            :pattern-method #'item-production-method))
```

*Figure A.10: The pattern *2phvbadvatt-1**

```
(defvar *phvbprepatt_set_1* (make-instance 'pattern-beam
                              :beam-name '*phvbprepatt_set_1*
                              :beam-value (list *phvbprepatt-t1-1* *phvbprepatt-t1-2*)))
```

*Figure A.11: The beam *2phvbadvatt_set_1**

```
;;;
;;;------------------------------------------------------------
;;;    phvb prepositional attachment type 2
;;; Author       : Herve Blanchon
;;; Date         : 10/19/94
;;; Abstract     :
;;;------------------------------------------------------------
;;;
```

```
(defvar *phvbprepatt-t2-1* (make-instance 'pattern
                            :pattern-name '*phvbprepatt-t2-1*
                            :pattern-value '((?is ?x node-prop-equal-p 'CS 'PHVB)
                                             (?+ ?p0)
                                             ((?is ?y node-prop-equal-p 'FS 'CIRC)
                                              (?+ ?p1)
                                              ((?is ?z node-prop-equal-p 'FS 'ATG)
                                               (?+ ?p2))))
                            :pattern-method #'item-production-method))
```

Figure A.12: The pattern *2phvbadvatt-1*

```
(defvar *phvbprepatt-t2-2* (make-instance 'pattern
                            :pattern-name '*phvbprepatt-t2-2*
                            :pattern-value '((?is ?x node-prop-equal-p 'CS 'PHVB)
                                             (?+ ?p0)
                                             ((?is ?y node-prop-equal-p 'FS 'CIRC)
                                              (?+ ?p1))
                                             ((?is ?z node-prop-equal-p 'FS 'CIRC)
                                              (?+ ?p2)))
                            :pattern-method #'item-production-method))
```

Figure A.13: The pattern *2phvbadvatt-1*

```
(defvar *phvbprepatt_set_2* (make-instance 'pattern-beam
                             :beam-name '*phvbprepatt_set_2*
                             :beam-value (list *phvbprepatt-t2-1* *phvbprepatt-t2-2*)))
```

Figure A.14: The beam *2phvbadvatt_set_1*

```
(defvar *phvbprepatt_beam_stack* (make-instance 'beam-stack
                          :beam-stack-name '*phvbprepatt_beam_stack*
                          :beam-stack-value (list *phvbprepatt_set_1* *phvbprepatt_set_2*)))
```

Figure A.15: The stack *2phvbadvatt_beam_stack*

```
;;;
;;;------------------------------------------------------------------
;;;    relative phvb adverbial attachment
;;;
;;; Author       : Herve Blanchon
;;; Date         : 01/23/95
;;; Abstract     :
;;;------------------------------------------------------------------
;;;
```

```
;;;
;;;------------------------------------------------------------------
;;;    relative phvb adverbial attachment type 1
;;;
;;; Author       : Herve Blanchon
;;; Date         : 01/23/95
;;; Abstract     :
;;;------------------------------------------------------------------
;;;
```

```
(defvar *relphvbadvatt-t1-1* (make-instance 'pattern
                              :pattern-name '*relphvbadvatt-t1-1*
                              :pattern-value '((?is ?x node-prop-equal-p 'CS 'PHVB)
                                               (?+ ?p0)
                                               ((?is ?y node-prop-equal-p 'CS 'PHREL)
                                                (?+ ?p1)
                                                ((?is ?z node-prop-equal-p 'FS 'OBJ)
                                                 (?+ ?p2)
                                                 ((?is ?t node-prop-equal-p 'CS 'GADJ)
                                                  (?+ ?p3)))))
                              :pattern-method #'item-production-method))
```

*Figure A.16: The pattern *2phvbadvatt-1**

```
(defvar *relphvbadvatt-t1-2* (make-instance 'pattern
                              :pattern-name '*relphvbadvatt-t1-2*
                              :pattern-value '((?is ?x node-prop-equal-p 'CS 'PHVB)
                                               (?+ ?p0)
                                               ((?is ?y node-prop-equal-p 'CS 'PHREL)
                                                (?+ ?p1)
                                                ((?is ?z node-prop-equal-p 'FS 'OBJ)
                                                 (?+ ?p2))
                                                ((?is ?t node-prop-equal-p 'CS 'GADV)
                                                 (?+ ?p3))))
                              :pattern-method #'item-production-method))
```

*Figure A.17: The pattern *2phvbadvatt-1**

```
(defvar *relphvbadvatt_set_1* (make-instance 'pattern-beam
                         :beam-name '*relphvbadvatt_set_1*
                         :beam-value (list *relphvbadvatt-t1-1* *relphvbadvatt-t1-2*)))
```

*Figure A.18: The beam *2phvbadvatt_set_1**

```
;;;
;;;------------------------------------------------------------
;;;    relative phvb adverbial attachment type 2
;;; Author        : Herve Blanchon
;;; Date          : 10/19/94
;;; Abstract      :
;;;------------------------------------------------------------
;;;
```

```
(defvar *relphvbadvatt-t2-1* (make-instance 'pattern
                             :pattern-name '*relphvbadvatt-t2-1*
                             :pattern-value '((?is ?x node-prop-equal-p 'CS 'PHVB)
                                              (?+ ?p0)
                                              ((?is ?y node-prop-equal-p 'CS 'PHREL)
                                               (?+ ?p1)
                                               ((?is ?z node-prop-equal-p 'FS 'CIRC)
                                                (?+ ?p2))))
                             :pattern-method #'item-production-method))
```

*Figure A.19: The pattern *2phvbadvatt-1**

```
(defvar *relphvbadvatt-t2-2* (make-instance 'pattern
                             :pattern-name '*relphvbadvatt-t2-2*
                             :pattern-value '((?is ?x node-prop-equal-p 'CS 'PHVB)
                                              (?+ ?p0)
                                              ((?is ?y node-prop-equal-p 'CS 'PHREL)
                                               (?+ ?p1))
                                              ((?is ?z node-prop-equal-p 'FS 'CIRC)
                                               (?+ ?p2)))
                             :pattern-method #'item-production-method))
```

*Figure A.20: The pattern *2phvbadvatt-1**

```
(defvar *relphvbadvatt_set_2* (make-instance 'pattern-beam
                              :beam-name '*relphvbadvatt_set_2*
                              :beam-value (list *relphvbadvatt-t2-1* *relphvbadvatt-t2-2*)))
```

*Figure A.21: The beam *2phvbadvatt_set_1**

```
(defvar *relphvbadvatt_beam_stack* (make-instance 'beam-stack
                        :beam-stack-name '*relphvbadvatt_beam_stack*
                        :beam-stack-value (list *relphvbadvatt_set_1* *relphvbadvatt_set_2*)))
```

*Figure A.22: The stack *2phvbadvatt_beam_stack**

```
;;;
;;;----------------------------------------------------------------
;;;   phvb conjunction attachment
;;;
;;; Author       : Herve Blanchon
;;; Date         : 01/23/95
;;; Abstract     :
;;;----------------------------------------------------------------
;;;
```

```
(defvar *phvbconjatt-1* (make-instance 'pattern
                        :pattern-name '*phvbconjatt-1*
                        :pattern-value '((?is ?x node-prop-equal-p 'CS 'PHVB)
                                         (?+ ?p0)
                                         ((?is ?y node-prop-equal-p 'CS 'PHVB)
                                          (?+ ?pl)
                                          ((?is ?z node-prop-equal-p 'CS 'PHVB)
                                           (?+ ?p2)))
                                         ((?is ?t node-prop-equal-p 'CS 'PHVB)
                                          (?+ ?p3)))
                        :pattern-method #'item-production-method))
```

*Figure A.23: The pattern *2phvbadvatt-1**

```
(defvar *phvbconjatt-2* (make-instance 'pattern
                        :pattern-name '*phvbconjatt-2*
                        :pattern-value '((?is ?x node-prop-equal-p 'CS 'PHVB)
                                         (?+ ?p0)
                                         ((?is ?y node-prop-equal-p 'CS 'PHVB)
                                          (?+ ?pl)
                                          ((?is ?z node-prop-equal-p 'CS 'PHVB)
                                           (?+ ?p2))
                                          ((?is ?t node-prop-equal-p 'CS 'PHVB)
                                           (?+ ?p3))))
                        :pattern-method #'item-production-method))
```

*Figure A.24: The pattern *2phvbadvatt-1**

```
(defvar *phvbconjatt_set_1* (make-instance 'pattern-beam
                            :beam-name '*phvbconjatt_set_1*
                            :beam-value (list *phvbconjatt-1* *phvbconjatt-2*)))
```

*Figure A.25: The beam *2phvbadvatt_set_1**

```
(defvar *phvbconjatt_beam_stack* (make-instance 'beam-stack
                                 :beam-stack-name '*phvbconjatt_beam_stack*
                                 :beam-stack-value (list *phvbconjatt_set_1*)))
```

*Figure A.26: The stack *2phvbadvatt_beam_stack**

```
;;;
;;;-------------------------------------------------------------------
;;;     non phvb prepositional attachment
;;;
;;; Author        : Herve Blanchon
;;; Date          : 01/23/95
;;; Abstract      :
;;;-------------------------------------------------------------------
;;;
```

```
;;;
;;;-------------------------------------------------------------------
;;;     non phvb prepositional attachment type 1
;;;
;;; Author        : Herve Blanchon
;;; Date          : 01/23/95
;;; Abstract      :
;;;-------------------------------------------------------------------
;;;
```

```
(defvar *nphvbprepatt-t1-1* (make-instance 'pattern
                              :pattern-name '*nphvbprepatt-t1-1*
                              :pattern-value '((?is ?x node-prop-equal-p 'CS 'GP)
                                                (?+ ?p0)
                                                (?+ ?p1)
                                                (?+ ?p2)
                                                (?+ ?p3)
                                                (?+ ?p4))
                              :pattern-method #'item-production-method))
```

*Figure A.27: The pattern *2phvbadvatt-1**

```
(defvar *nphvbprepatt-t1-2* (make-instance 'pattern
                              :pattern-name '*nphvbprepatt-t1-2*
                              :pattern-value '((?is ?x node-prop-equal-p 'CS 'GP)
                                                (?+ ?p0)
                                                (?+ ?p1)
                                                ((?is ?z node-prop-equal-p 'FS 'ATG)
                                                 (?+ ?p2)
                                                 (?+ ?p3))
                                                (?+ ?p4))
                              :pattern-method #'item-production-method))
```

*Figure A.28: The pattern *2phvbadvatt-1**

```
(defvar *nphvbprepatt_set_1* (make-instance 'pattern-beam
                               :beam-name '*nphvbprepatt_set_1*
                               :beam-value (list *nphvbprepatt-t1-1* *nphvbprepatt-t1-2*)))
```

*Figure A.29: The beam *2phvbadvatt_set_1**

```
;;;
;;;------------------------------------------------------------------
;;;     non phvb prepositional attachment type 2
;;;
;;; Author        : Herve Blanchon
;;; Date          : 01/23/95
;;; Abstract      :
;;;------------------------------------------------------------------
;;;
```

```
(defvar *nphvbprepatt-t2-1* (make-instance 'pattern
                              :pattern-name '*nphvbprepatt-t2-1*
                              :pattern-value '((?is ?x node-prop-equal-p 'CS 'DGN)
                                               (?+ ?p0)
                                               ((?is ?y node-prop-equal-p 'CS 'GP)
                                                (?+ ?p1)
                                                ((?is ?z node-prop-equal-p 'CS 'GP)
                                                 (?+ ?p2))))
                              :pattern-method #'item-production-method))
```

*Figure A.30: The pattern *2phvbadvatt-1**

```
(defvar *nphvbprepatt-t2-2* (make-instance 'pattern
                              :pattern-name '*nphvbprepatt-t2-2*
                              :pattern-value '((?is ?x node-prop-equal-p 'CS 'DGN)
                                               (?+ ?p0)
                                               ((?is ?y node-prop-equal-p 'CS 'GP)
                                                (?+ ?p1))
                                               ((?is ?z node-prop-equal-p 'CS 'GP)
                                                (?+ ?p2)))
                              :pattern-method #'item-production-method))
```

*Figure A.31: The pattern *2phvbadvatt-1**

```
(defvar *nphvbprepatt_set_2* (make-instance 'pattern-beam
                               :beam-name '*nphvbprepatt_set_2*
                               :beam-value (list *nphvbprepatt-t2-1* *nphvbprepatt-t2-2*)))
```

*Figure A.32: The beam *2phvbadvatt_set_1**

```
(defvar *nphvbprepatt_beam_stack* (make-instance 'beam-stack
                         :beam-stack-name '*nphvbprepatt_beam_stack*
                         :beam-stack-value (list *nphvbprepatt_set_1* *nphvbprepatt_set_2*)))
```

*Figure A.33: The stack *2phvbadvatt_beam_stack**

Appendix B

Methods

```
;;;----------------------------------------------------------------------------
;;;
;;; Title        : English Disambiguation Methods
;;;
;;;----------------------------------------------------------------------------
;;; Author       : Herve Blanchon
;;; Address      : Advanced Telecommunication Research Labs
;;;              : Interpreting Telephony Research Labs.
;;;              : 2-2 Hikaridai, Seika-cho Soraku-gun
;;;              : Kyoto,Japan 619-02
;;;              : Blanchon@itl.atr.co.jp
;;;----------------------------------------------------------------------------
;;; Filename     : Disamb_English-Methods.lisp
;;; Version      : 1.0
;;; Abstract     : English disambiguation methods associated with the
;;;              : English Disambiguation patterns
;;; History      : 10/18/94 Herve Blanchon
;;; Bugs         :
;;; Todo         :
;;;----------------------------------------------------------------------------
```

```
;;;
;;;-------------------------------------------------------------
;;; ITEM-PRODUCTION-METHOD        [Method]
;;; Specializers (eql '2phvbadvatt-1) T
;;;
;;; Author        : Herve Blanchon
;;; Date          : 1/23/95
;;; Abstract      :
;;;-------------------------------------------------------------
;;;
```

```
(defmethod item-production-method ((pattern-name (eql '*2phvbadvatt-1*)) binding)
  (format nil "~A ~A (~A ~A)."
        (string-trim '(#\space) (apply #'texte (cdr (assoc '?p0 binding))))
        (string-trim '(#\space) (apply #'texte (cdr (assoc '?p1 binding))))
        (string-trim '(#\space) (apply #'texte (cdr (assoc '?p2 binding))))
        (string-trim '(#\space) (apply #'texte (cdr (assoc '?p3 binding))))))
```

*Figure B.1: Item-production-method ((pattern-name (eql '*2phvbadvatt-1*)) binding)*

```
;;;
;;;-------------------------------------------------------------
;;; ITEM-PRODUCTION-METHOD        [Method]
;;; Specializers (eql '2phvbadvatt-2) T
;;;
;;; Author        : Herve Blanchon
;;; Date          : 1/23/95
;;; Abstract      :
;;;-------------------------------------------------------------
;;;
```

```
(defmethod item-production-method ((pattern-name (eql '*2phvbadvatt-2*)) binding)
  (format nil "~A, ~A ~A ~A."
        (string-trim '(#\space) (apply #'texte (cdr (assoc '?p3 binding))))
        (string-trim '(#\space) (apply #'texte (cdr (assoc '?p0 binding))))
        (string-trim '(#\space) (apply #'texte (cdr (assoc '?p1 binding))))
        (string-trim '(#\space) (apply #'texte (cdr (assoc '?p2 binding))))))
```

*Figure B.2: Item-production-method ((pattern-name (eql '*2phvbadvatt-2*)) binding)*

```
;;;
;;;-----------------------------------------------------------------
;;; ITEM-PRODUCTION-METHOD        [Method]
;;; Specializers (eql 'spladvatt-1) T
;;;
;;; Author        : Herve Blanchon
;;; Date          : 1/23/95
;;; Abstract      :
;;;-----------------------------------------------------------------
;;;
```

```
(defmethod item-production-method ((pattern-name (eql '*spladvatt-1*)) binding)
  (format nil "~A (~A)."
          (string-trim '(#\space) (apply #'texte (cdr (assoc '?p0 binding))))
          (string-trim '(#\space) (apply #'texte (cdr (assoc '?p2 binding))))))
```

*Figure B.3: Item-production-method ((pattern-name (eql '*spladvatt-1*)) binding)*

```
;;;
;;;-----------------------------------------------------------------
;;; ITEM-PRODUCTION-METHOD        [Method]
;;; Specializers (eql 'spladvatt-2) T
;;;
;;; Author        : Herve Blanchon
;;; Date          : 1/23/95
;;; Abstract      :
;;;-----------------------------------------------------------------
;;;
```

```
(defmethod item-production-method ((pattern-name (eql '*spladvatt-2*)) binding)
  (format nil "~A, ~A ~A."
          (string-trim '(#\space) (apply #'texte (cdr (assoc '?p2 binding))))
          (string-trim '(#\space) (apply #'texte (cdr (assoc '?p0 binding))))
          (string-trim '(#\space) (apply #'texte (cdr (assoc '?p1 binding))))))
```

*Figure B.4: Item-production-method ((pattern-name (eql '*spladvatt-2*)) binding)*

```
;;;
;;;---------------------------------------------------------------
;;; ITEM-PRODUCTION-METHOD       [Method]
;;; Specializers (eql 'phvbprepatt-t1-1) T
;;;
;;; Author       : Herve Blanchon
;;; Date         : 1/23/95
;;; Abstract     :
;;;---------------------------------------------------------------
;;;
```

```
(defmethod item-production-method ((pattern-name (eql '*phvbprepatt-t1-1*)) binding)
  (format nil "~A (~A ~A)."
          (string-trim '(#\space) (apply #'texte (cdr (assoc '?p0 binding))))
          (string-trim '(#\space) (apply #'texte (cdr (assoc '?p1 binding))))
          (string-trim '(#\space) (apply #'texte (cdr (assoc '?p2 binding))))))
```

*Figure B.5: Item-production-method ((pattern-name (eql '*phvbprepatt-t1-1*)) binding)*

```
;;;
;;;---------------------------------------------------------------
;;; ITEM-PRODUCTION-METHOD       [Method]
;;; Specializers (eql 'phvbprepatt-t1-2) T
;;;
;;; Author       : Herve Blanchon
;;; Date         : 1/23/95
;;; Abstract     :
;;;---------------------------------------------------------------
;;;
```

```
(defmethod item-production-method ((pattern-name (eql '*phvbprepatt-t1-2*)) binding)
  (format nil "~A, ~A ~A."
          (string-trim '(#\space) (apply #'texte (cdr (assoc '?p2 binding))))
          (string-trim '(#\space) (apply #'texte (cdr (assoc '?p0 binding))))
          (string-trim '(#\space) (apply #'texte (cdr (assoc '?p1 binding))))))
```

*Figure B.6: Item-production-method ((pattern-name (eql '*phvbprepatt-t1-2*)) binding)*

```
;;;
;;;------------------------------------------------------------------
;;; ITEM-PRODUCTION-METHOD        [Method]
;;; Specializers (eql 'phvbprepatt-t2-1) T
;;;
;;; Author       : Herve Blanchon
;;; Date         : 1/23/95
;;; Abstract     :
;;;------------------------------------------------------------------
;;;
```

```
(defmethod item-production-method ((pattern-name (eql '*phvbprepatt-t2-1*)) binding)
  (format nil "~A (~A ~A)."
          (string-trim '(#\space) (apply #'texte (cdr (assoc '?p0 binding))))
          (string-trim '(#\space) (apply #'texte (cdr (assoc '?p1 binding))))
          (string-trim '(#\space) (apply #'texte (cdr (assoc '?p2 binding))))))
```

*Figure B.7: Item-production-method ((pattern-name (eql '*phvbprepatt-t2-1*)) binding)*

```
;;;
;;;------------------------------------------------------------------
;;; ITEM-PRODUCTION-METHOD        [Method]
;;; Specializers (eql 'phvbprepatt-t2-2) T
;;;
;;; Author       : Herve Blanchon
;;; Date         : 1/23/95
;;; Abstract     :
;;;------------------------------------------------------------------
;;;
```

```
(defmethod item-production-method ((pattern-name (eql '*phvbprepatt-t2-2*)) binding)
  (format nil "~A, ~A ~A."
          (string-trim '(#\space) (apply #'texte (cdr (assoc '?p2 binding))))
          (string-trim '(#\space) (apply #'texte (cdr (assoc '?p0 binding))))
          (string-trim '(#\space) (apply #'texte (cdr (assoc '?p1 binding))))))
```

*Figure B.8: Item-production-method ((pattern-name (eql '*phvbprepatt-t2-2*)) binding)*

```
;;;
;;;----------------------------------------------------------------
;;; ITEM-PRODUCTION-METHOD       [Method]
;;; Specializers (eql 'relphvbadvatt-t1-1) T
;;;
;;; Author       : Herve Blanchon
;;; Date         : 1/23/95
;;; Abstract     :
;;;----------------------------------------------------------------
;;;
```

```
(defmethod item-production-method ((pattern-name (eql '*relphvbadvatt-t1-1*)) binding)
  (format nil "~A, ~A, ~A ~A."
         (string-trim '(#\space) (apply #'texte (cdr (assoc '?p0 binding))))
         (string-trim '(#\space) (apply #'texte (cdr (assoc '?p3 binding))))
         (string-trim '(#\space) (apply #'texte (cdr (assoc '?p1 binding))))
         (string-trim '(#\space) (apply #'texte (cdr (assoc '?p2 binding))))))
```

*Figure B.9: Item-production-method ((pattern-name (eql '*relphvbadvatt-t1-1*)) binding)*

```
;;;
;;;----------------------------------------------------------------
;;; ITEM-PRODUCTION-METHOD       [Method]
;;; Specializers (eql 'relphvbadvatt-t1-2) T
;;;
;;; Author       : Herve Blanchon
;;; Date         : 1/23/95
;;; Abstract     :
;;;----------------------------------------------------------------
;;;
```

```
(defmethod item-production-method ((pattern-name (eql '*relphvbadvatt-t1-2*)) binding)
  (format nil "~A ~A (~A ~A)."
         (string-trim '(#\space) (apply #'texte (cdr (assoc '?p0 binding))))
         (string-trim '(#\space) (apply #'texte (cdr (assoc '?p1 binding))))
         (string-trim '(#\space) (apply #'texte (cdr (assoc '?p2 binding))))
         (string-trim '(#\space) (apply #'texte (cdr (assoc '?p3 binding))))))
```

*Figure B.10: Item-production-method ((pattern-name (eql '*relphvbadvatt-t1-2*)) binding)*

```
;;;
;;;-----------------------------------------------------------------
;;; ITEM-PRODUCTION-METHOD       [Method]
;;; Specializers (eql 'relphvbadvatt-t2-1) T
;;;
;;; Author        : Herve Blanchon
;;; Date          : 1/23/95
;;; Abstract      :
;;;-----------------------------------------------------------------
;;;
```

```
(defmethod item-production-method ((pattern-name (eql '*relphvbadvatt-t2-1*)) binding)
  (format nil "~A (~A ~A)."
          (string-trim '(#\space) (apply #'texte (cdr (assoc '?p0 binding))))
          (string-trim '(#\space) (apply #'texte (cdr (assoc '?p1 binding))))
          (string-trim '(#\space) (apply #'texte (cdr (assoc '?p2 binding))))))
```

*Figure B.11: Item-production-method ((pattern-name (eql '*relphvbadvatt-t2-1*)) binding)*

```
;;;
;;;-----------------------------------------------------------------
;;; ITEM-PRODUCTION-METHOD       [Method]
;;; Specializers (eql 'relphvbadvatt-t2-2) T
;;;
;;; Author        : Herve Blanchon
;;; Date          : 1/23/95
;;; Abstract      :
;;;-----------------------------------------------------------------
;;;
```

```
(defmethod item-production-method ((pattern-name (eql '*relphvbadvatt-t2-2*)) binding)
  (format nil "~A, ~A ~A."
          (string-trim '(#\space) (apply #'texte (cdr (assoc '?p2 binding))))
          (string-trim '(#\space) (apply #'texte (cdr (assoc '?p0 binding))))
          (string-trim '(#\space) (apply #'texte (cdr (assoc '?p1 binding))))))
```

*Figure B.12: Item-production-method ((pattern-name (eql '*relphvbadvatt-t2-2*)) binding)*

```
;;;
;;;--------------------------------------------------------------
;;; ITEM-PRODUCTION-METHOD         [Method]
;;; Specializers (eql 'phvbconjatt-1) T
;;;
;;; Author          : Herve Blanchon
;;; Date            : 1/23/95
;;; Abstract        :
;;;--------------------------------------------------------------
;;;
```

```
(defmethod item-production-method ((pattern-name (eql '*phvbconjatt-1*)) binding)
  (format nil "~A ~A ~A. ~A."
          (string-trim '(#\space) (apply #'texte (cdr (assoc '?p0 binding))))
          (string-trim '(#\space) (apply #'texte (cdr (assoc '?p1 binding))))
          (string-trim '(#\space) (apply #'texte (cdr (assoc '?p2 binding))))
          (string-trim '(#\space) (apply #'moins-coordonnant (cdr (assoc '?p3 binding))))))
```

*Figure B.13: Item-production-method ((pattern-name (eql '*phvbconjatt-1*)) binding)*

```
;;;
;;;--------------------------------------------------------------
;;; ITEM-PRODUCTION-METHOD         [Method]
;;; Specializers (eql 'phvbconjatt-2) T
;;;
;;; Author          : Herve Blanchon
;;; Date            : 1/23/95
;;; Abstract        :
;;;--------------------------------------------------------------
;;;
```

```
(defmethod item-production-method ((pattern-name (eql '*phvbconjatt-2*)) binding)
  (format nil "~A ~A ~A."
          (string-trim '(#\space) (apply #'texte (cdr (assoc '?p0 binding))))
          (string-trim '(#\space) (apply #'texte (cdr (assoc '?p1 binding))))
          (string-trim '(#\space) (apply #'moins-coordonnant (cdr (assoc '?p3 binding))))))
```

*Figure B.14: Item-production-method ((pattern-name (eql '*phvbconjatt-2*)) binding)*

```
;;;
;;;---------------------------------------------------------------
;;; ITEM-PRODUCTION-METHOD      [Method]
;;; Specializers (eql 'nphvbprepatt-t1-1) T
;;;
;;; Author      : Herve Blanchon
;;; Date        : 1/23/95
;;; Abstract    :
;;;---------------------------------------------------------------
;;;
```

```
(defmethod item-production-method ((pattern-name (eql '*nphvbprepatt-t1-1*)) binding)
  (format nil "~A (~A ~A) for ~A"
          (string-trim '(#\space) (apply #'texte (cdr (assoc '?p1 binding))))
          (string-trim '(#\space) (apply #'texte (cdr (assoc '?p2 binding))))
          (string-trim '(#\space) (apply #'texte (cdr (assoc '?p4 binding))))
          (string-trim '(#\space) (apply #'texte (cdr (assoc '?p3 binding))))))
```

*Figure B.15: Item-production-method ((pattern-name (eql '*nphvbprepatt-t1-1*)) binding)*

```
;;;
;;;---------------------------------------------------------------
;;; ITEM-PRODUCTION-METHOD      [Method]
;;; Specializers (eql 'nphvbprepatt-t1-2) T
;;;
;;; Author      : Herve Blanchon
;;; Date        : 1/23/95
;;; Abstract    :
;;;---------------------------------------------------------------
;;;
```

```
(defmethod item-production-method ((pattern-name (eql '*nphvbprepatt-t1-2*)) binding)
  (format nil "~A ~A for (~A ~A)"
          (string-trim '(#\space) (apply #'texte (cdr (assoc '?p1 binding))))
          (string-trim '(#\space) (apply #'texte (cdr (assoc '?p4 binding))))
          (string-trim '(#\space) (apply #'texte (cdr (assoc '?p2 binding))))
          (string-trim '(#\space) (apply #'texte (cdr (assoc '?p3 binding))))))
```

*Figure B.16: Item-production-method ((pattern-name (eql '*nphvbprepatt-t1-2*)) binding)*

```
;;;
;;;--------------------------------------------------------------
;;; ITEM-PRODUCTION-METHOD       [Method]
;;; Specializers (eql 'nphvbprepatt-t2-1) T
;;;
;;; Author       : Herve Blanchon
;;; Date         : 1/23/95
;;; Abstract     :
;;;--------------------------------------------------------------
;;;
```

```
(defmethod item-production-method ((pattern-name (eql '*nphvbprepatt-t2-1*)) binding)
  (format nil "~A (~A ~A)"
          (string-trim '(#\space) (apply #'texte (cdr (assoc '?p0 binding))))
          (string-trim '(#\space) (apply #'texte (cdr (assoc '?p1 binding))))
          (string-trim '(#\space) (apply #'texte (cdr (assoc '?p2 binding))))))
```

*Figure B.17: Item-production-method ((pattern-name (eql '*nphvbprepatt-t2-1*)) binding)*

```
;;;
;;;--------------------------------------------------------------
;;; ITEM-PRODUCTION-METHOD       [Method]
;;; Specializers (eql 'nphvbprepatt-t2-2) T
;;;
;;; Author       : Herve Blanchon
;;; Date         : 1/23/95
;;; Abstract     :
;;;--------------------------------------------------------------
;;;
```

```
(defmethod item-production-method ((pattern-name (eql '*nphvbprepatt-t2-2*)) binding)
  (format nil "~A, ~A ~A"
          (string-trim '(#\space) (apply #'texte (cdr (assoc '?p2 binding))))
          (string-trim '(#\space) (apply #'texte (cdr (assoc '?p0 binding))))
          (string-trim '(#\space) (apply #'texte (cdr (assoc '?p1 binding))))))
```

*Figure B.18: Item-production-method ((pattern-name (eql '*nphvbprepatt-t2-2*)) binding)*

Appendix C

Produced dialogues

In this appendix, we have listed all the dialogues currently produced by the English disambiguation module.

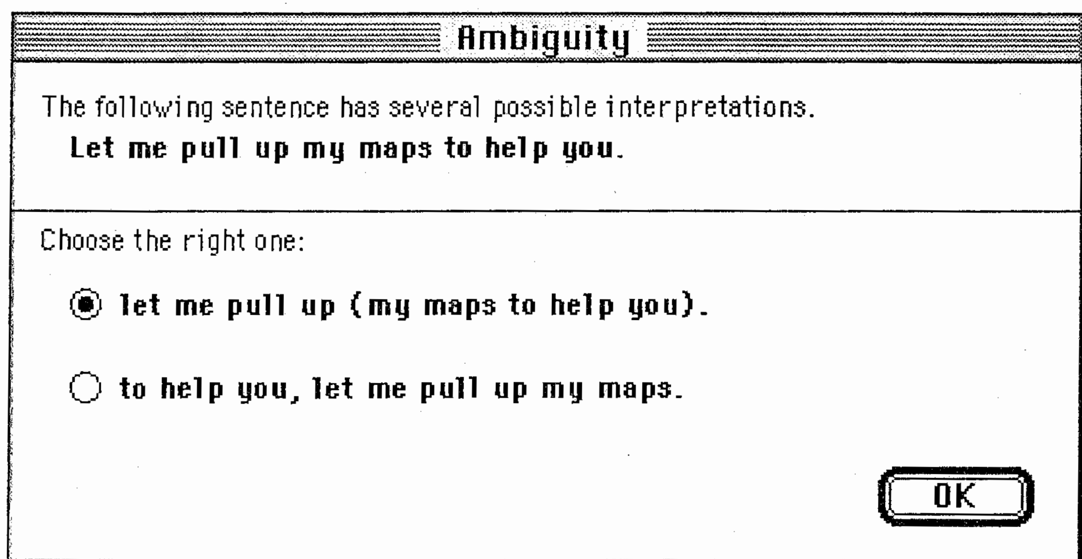


Figure C.1 : Dialogue for "Let me pull up my maps to help you."

Ambiguity

The following sentence has several possible interpretations.
You can pay for it right on the bus.

Choose the right one:

◉ you can pay for it (on the bus).

○ on the bus, you can pay for it right.

OK

Figure C.2 : Dialogue for "You can pay for it right on the bus."

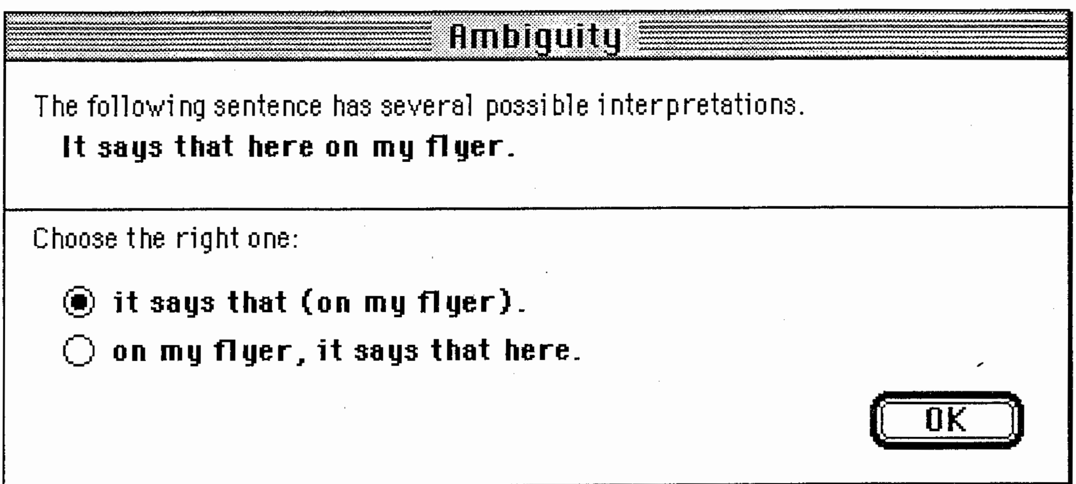


Figure C.3 : Dialogue for "It says that here on my flyer."

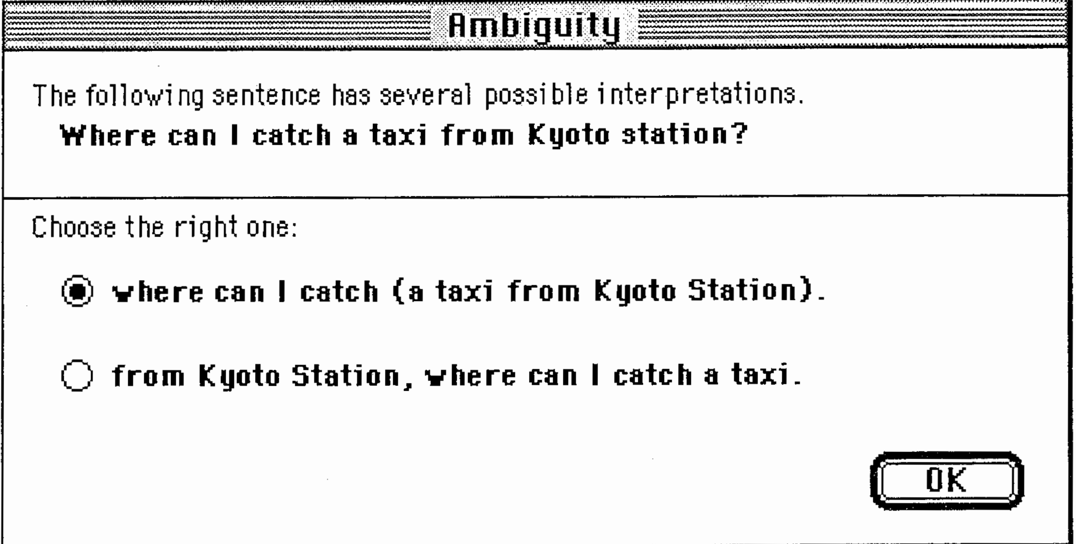


Figure C.4 : Dialogue for "Where can I catch a taxi form Kyoto station."

Ambiguity

The following sentence has several possible interpretations.

Go across the street to the north of the station.

Choose the right one:

◉ **go (across the street to the north).**

○ **to the north, go across the street.**

OK

Figure C.5 : Dialogue for "Go across the stree to the North of the station."

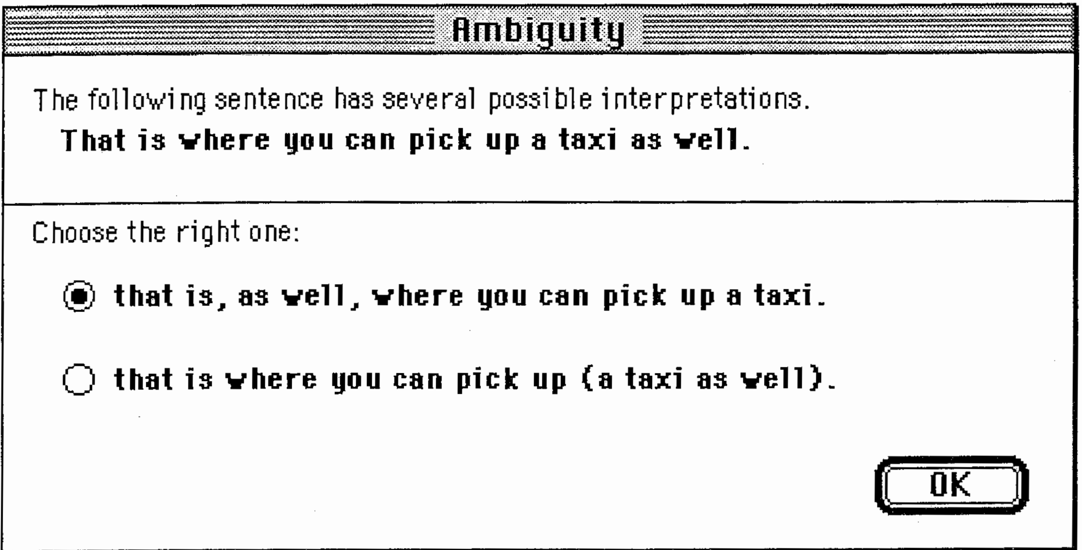


Figure C.6 : Dialogue for "This is where you can pick up a taxi as well."

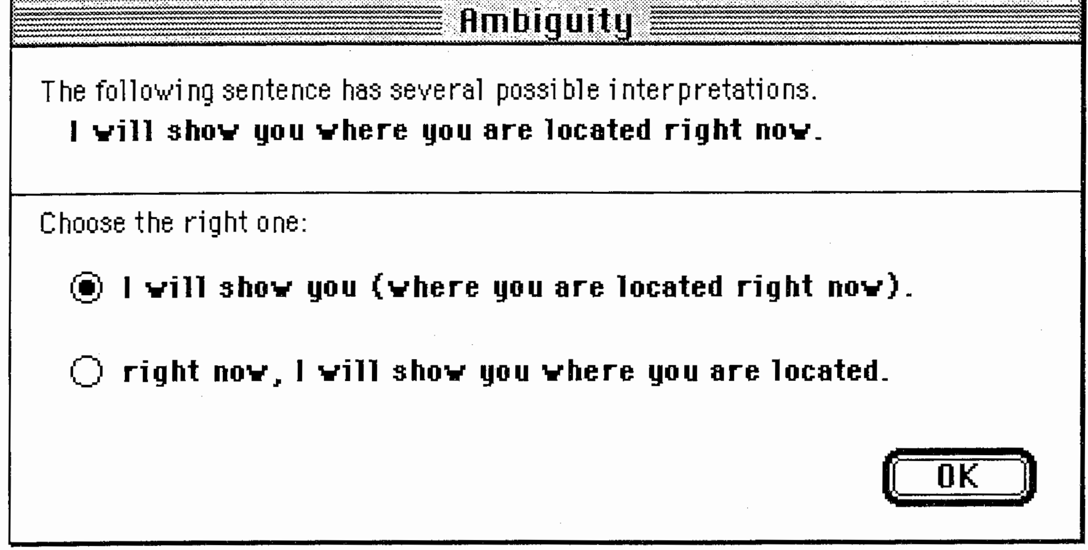


Figure C.7 : Dialogue for "I wil show you where you are located right now."

Ambiguity

The following sentence has several possible interpretations.

You can tell him that you are going to the international conference center and it should be a twenty minutes ride

Choose the right one:

- (•) **you can tell him that you are going to the international conference center. it should be a twenty minutes ride.**
- ( ) **you can tell him that it should be a twenty minutes ride.**

OK

Figure C.8 : Dialogue for "You can tell hin that you are going to the international conference center and it should be a twenty minutes ride."

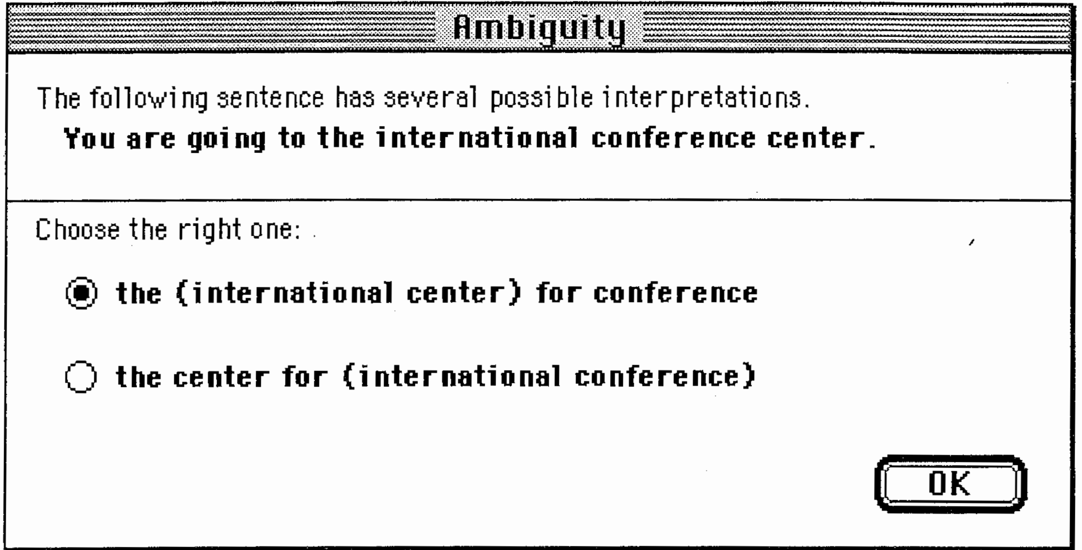


Figure C.9 : Dialogue for "You are going to the international conference center."

Ambiguity

The following sentence has several possible interpretations.

I want the symposium on interpreting telecommunication at the international conference center.

Choose the right one:

◉ **the symposium (on interpreting telecommunications at the international conference center)**

○ **at the international conference center, the symposium on interpreting telecommunications**

OK

Figure C.10 : Dialogue for "I want the symposium on interpreting telecommunications at the international conference center."

Appendix D

Organization of the software

The English disambiguation module is defined thanks to defunit, a kind of defsystem utility.

A unit represents a logical regrouping of lisp source files. They are similar to Common Lisp's concept of modules. The main difference is that by explicitly specifying the files that make up units, operations other than loading can be globally applied to them.

Units are named objects. They are named by strings where case is not treated as significant. Every command in this module that wants a unit as an argument, will accept either strings or symbols. When symbols are used, their print name is taken and uppercased to make it case independent.

The standard way to define units, is via unit definition files. When a command is passed a unit name say x, it first checks if a unit named x is already loaded in memory. If so, it uses that definition, otherwise it searches all the files in `*unit-registry*` for a file with name x. If none is found, an error is signaled, otherwise the file is loaded. The file is supposed to define unit x, if it does not, an error is also signaled.

The actual definition follows.

```
(defunit "English_Disambiguation_Unit"
  (:depends-on :geta-stuff :geta-grapher)
  (:source-pathname "working-folder:ATR's Clarification Process;")
  (:binary-pathname "cc:binaries;projects;ATR's Clarification Process;")
  (:components "tree-grapher"     ;drawing of the selected tree
```

;The engine

;definition of the classes pattern, pattern-beam & beam-stack

"Pattern&others_Class"

;defintion of the class generic-textual-dialog-class, of the presentation methods, and other dialogue-related methodes

"Dialog_Class"

;definition of the classes clarification-question-class & empty-question

"Question_Class"

;definition of the operators

"Operators"

;some functions for the manipulation of the tree structure

"Ling_Struct"

;definition of the pattern matcher

"Pattern-Matcher"

;definition of the predicates to be used in the pattern and other services

"Pattern-Matcher_Utils"

;definition of the beam matching process (filling of the matrix)

"Beam_Match"

;definition of the binding reduction process

"Matching_Reduce"

;definition of the ambiguity meta-class recognition states

"Automaton_Services"

;the question tree construction module

"Question-Tree_Construction"

;the question tree presentation module

"Question-Tree_Presentation"

;The English lingware

;defintion of the disambiguation methods

"English-Methods"

;defintion of the English patterns, beams & stacks

"English-Patterns"

;defintion of the English dialogue classes

"English_Dialog_Classes"

;defintion of the English disambiguation automaton

"English-Automaton"

;Corpus of the mmc-structures to be used in the current impelementation

"English_Analysis"

;definition of the demonstration functions

"Entry_Point"))