CLARE: A Contextual Reasoning and Cooperative Response Framework for the Core Language Engine

Final Report

Hiyan Alshawi, David Carter, Richard Crouch,
Steve Pulman, Manny Rayner and Arnold Smith
with identified additional contributions

December 1992

Prepared for CLARE Natural Language Processing Club
DTI Information Technology Division

IEATP Project IED4/1/1165
SRI Project 8468

Cambridge Computer Science Research Centre
23 Millers Yard, Mill Lane, Cambridge CB2 1RQ, UK
Tel +44-223-324146, Fax +44-223-460402

E-mail: sriclare@cam.sri.com

Executive Summary

This document is the final report of CLARE, a project involving BP Research, British Aerospace, British Telecom, Cambridge University, SRI Cambridge and the UK Defence Research Agency. The project received a grant from the UK Department of Trade and Industry. The report mainly describes the research, design and implementation work carried out in building the CLARE system at SRI, with some discussion of experimentation with the software by the other partners. Low-level interfacing issues and a guide to using the system are covered in a manual provided with the final release of the software. The project also involved a study by Cambridge University Computer Laboratory on evaluating natural language processing systems. A digest of the report for this study appears at the end of the present report.

CLARE was designed as a natural language processing system with facilities for reasoning and understanding in context and for generating cooperative responses. The work plan for the project required both further development of the Core Language Engine (CLE) natural language processor and the design and implementation of new components for reasoning and response generation. All the milestones set in the project plan were achieved, the final system including the following capabilities:

- Wider coverage of English syntax and semantics than the original CLE system. This is quantified in the report.
- Lattice based lexical, phrasal and syntactic analysis. Partial syntactic and semantic analysis of sentences outside coverage.
- Gradual refinement of meaning in context according to a theory of "monotonic interpretation" employing an improved Quasi Logical Form representation.
- Ranking alternatives produced by sentence processing with the use of an extensible set of preference metrics, including semantic collocations.
- Generation of informative paraphrases that indicate the system's interpretation of English inputs.
- Use of an abductive reasoning mechanism to support the processes of interpretation and of interfacing to an application.
- Declarative specification of database domain models, including the specification of equivalence axioms used by the reasoner to translate between linguistic and database representations.
- Generation of natural language statements from database assertions, and questions from incomplete assertions.

For all of these capabilities, the CLARE system has advanced the state of the art either through the use of novel techniques developed on the project, or by extending the coverage or scale of known techniques. On the negative side, we had to specialize our development in the interfacing part of the project to structured database system interfaces. The reason for this is that general, or unstructured, knowledge-based systems currently appear too varied to allow the degree of integration between the language and back-end systems achieved in CLARE with database systems. Nevertheless, the language components remain application-independent and provide interfaces for the development of new types of application.

CLARE was used during the project to build a number of database interface systems: the example Project Resource Management application at SRI, a similar database interface at BT Research Laboratories, and a personnel database application at BP. The language processing components of CLARE are currently being used in experimental work on spoken language systems and text processing: an interface to a road map routing system at DRA Malvern, an English-Swedish speech translation project at SRI, and research on processing specification texts at Cambridge University.

Contents

1 Components of CLARE **9**
1.1 CLARE System Architecture . . . 9
1.2 Processing Phases . . . 10
1.3 Transcript of CLARE PRM Application . . . 13

2 Interfacing by Abductive Equivalential Translation **22**
2.1 Introduction . . . 22
2.2 Ontological issues . . . 25
2.3 Formalizing NL interfacing functionalities . . . 26
2.3.1 Preliminaries . . . 27
2.3.2 Basic form of linguistic domain theories . . . 27
2.3.3 Yes/No questions . . . 30
2.3.4 Commands and WH-questions . . . 31
2.3.5 Meta-questions . . . 32
2.3.6 Generating statements and questions . . . 32
2.4 Abductive Equivalential Translation . . . 33
2.4.1 Translation schemas . . . 33
2.4.2 A simple example . . . 37
2.4.3 Existential quantification . . . 39
2.5 Simplification . . . 40
2.5.1 Non-inferential simplification . . . 41
2.5.2 Inferential simplification . . . 43
2.5.3 Simplification for processing assertions . . . 44
2.6 The Closed World Assumption . . . 46
2.7 Translating to SQL . . . 47

3 Linguistic Domain Theories **52**
3.1 Introduction . . . 52
3.2 LDTs for relational databases . . . 54
3.2.1 General axioms . . . 54
3.2.2 Domain-specific axioms . . . 55
3.2.3 Database-specific axioms . . . 56
3.3 Declaring database and conceptual relations . . . 58

3.4 Declaring functional relationships 59
3.5 Equivalence rules 60
3.5.1 Finding predicates associated with conceptual predicates . 62
3.5.2 Common noun describing a conceptual slot 63
3.5.3 Preposition describing relationship between slots 63
3.5.4 Verb describing relationship between slots 64
3.5.5 Relational nouns 65
3.5.6 Primitive predicates 66
3.5.7 Expressions that map onto code values 67
3.5.8 "Display" predicates 68
3.5.9 Mapping on to combinations of conceptual predicates . . . 69
3.5.10 Translating conceptual into database predicates 69
3.6 Assumptions 73
3.7 Horn Clause axioms 75
3.8 AET with a domain theory: a full example 76

4 Evolution of the QLF Formalism 83
4.1 The QLF Language 83
4.1.1 A Single Level of Logical Form 83
4.1.2 Syntax of QLF 85
4.2 Example QLFs 88

5 Monotonic Semantic Interpretation 96
5.1 Introduction 96
5.2 Syntax of Monotonic QLF 97
5.3 Example QLF Resolutions 98
5.4 Semantics for QLF 102
5.4.1 Simplified Semantics 102
5.4.2 QLF Semantics 104
5.5 Notes on the Semantics 106
5.6 Related Approaches 108
5.7 Implementation Benefits 110

6 The Domain and Context Models in Interpretation 112
6.1 The Domain and Context Models 112
6.1.1 Reverse Compilation of Linguistic Domain Theories 113
6.2 Reference Resolution and Abductive (Equivalential) Translation . 115
6.3 Domain Specific Resolution 118
6.3.1 Anaphoric and Definite Noun Phrases 118
6.3.2 Compound Nouns and Vague Relations 121
6.4 Functional Information and Scoping 122
6.5 The Context Model in Evaluating Resolved QLFs 124
6.5.1 Index–Domain Entity Associations 124

6.5.2 Full term Descriptions . . . 126

7 Resolution of Ellipsis 128

7.1 Analysis of Ellipsis . . . 128
7.2 Ellipsis Resolution . . . 129
7.2.1 Examples of Substitutional QLF Ellipsis . . . 129
7.2.2 Ellipsis and Higher-Order Unification . . . 141
7.3 Discussion . . . 144

8 Generation 146

8.1 Linguistic Generation (Synthesis) . . . 146
8.1.1 The SHD Generation Algorithm . . . 147
8.1.2 Applying Rules for Generation . . . 148
8.1.3 Rule and Lexicon Precompilation . . . 148
8.1.4 Control and Search . . . 149
8.2 Generation for Paraphrasing Resolutions . . . 150
8.2.1 Principles of Resolution Paraphrase . . . 150
8.2.2 Paraphrasing terms . . . 152
8.2.3 Paraphrasing forms . . . 153
8.3 Generation of Facts and Questions . . . 155
8.3.1 Generation from Assertions . . . 155
8.3.2 Learning Case-based Description Rules . . . 156
8.3.3 Description of Database Entities . . . 158
8.3.4 Generating Questions from Incomplete Assertions . . . 162

9 The Reasoning Engine 165

9.1 Logical structure of the reasoning engine . . . 165
9.1.1 Abductive proofs . . . 166
9.1.2 Proving implications . . . 167
9.2 Search strategies for inference . . . 168
9.2.1 Avoiding infinitely recursive branches . . . 170
9.2.2 Avoiding redundant search . . . 171
9.2.3 Dynamic re-ordering of conjuncts . . . 171
9.3 Finding finite proof procedures . . . 172

10 The Preference Mechanism 175

10.1 Preference and the CLE Architecture . . . 175
10.2 User Interaction for QLF Selection . . . 177
10.3 Preference Measures . . . 179
10.4 Scaling Factors . . . 180
10.5 QLF Preferences . . . 181
10.5.1 Structural Preferences . . . 182
10.5.2 Semantic Preferences . . . 182

10.5.3 Semantic Head Collocations 183
10.6 Resolved QLF Preferences 184
10.7 Scoping Preferences 185

11 Lexical and Syntactic Analysis and Failure Recovery 186
11.1 Introduction 186
11.2 Lattices for Lexical Processing 187
11.3 CLARE's Lexical Processing Stages 189
11.4 Clustering 190
11.5 Tokenization 191
11.6 Segmentation 192
11.6.1 Special Forms 192
11.7 Lexical Failure Recovery 193
11.8 Idiom and Abbreviation Recognition 196
11.9 Phrase Parsing 198
11.10 Examples of Lattice Processing 199
11.11 An Evaluation 201
11.12 Responding to Parse Failures 204
11.13 Responding to Sortal Restriction Failures 206
11.14 Summary and Conclusions 209

12 Coverage Extensions and Evaluations 211
12.1 Additions to Coverage 211
12.1.1 Ellipsis 211
12.1.2 Comparatives 212
12.1.3 NP postmodification 213
12.1.4 "Big PPs" 214
12.1.5 Adverbial phrases 214
12.1.6 Idioms 215
12.1.7 Dates, times and numbers 216
12.2 Grammar Documentation Facilities 217
12.3 Conditions of Coverage Evaluation 218
12.3.1 Limiting the vocabulary 219
12.4 Development over Time 222
12.4.1 Proportions of QLFs produced 223
12.4.2 Proportions of good QLFs 228
12.4.3 Development over Time: Conclusions 229
12.5 Dependence on text type 230
12.6 System components and sentence length 232
12.7 Tagging for parsing efficiency: a preliminary experiment 234
12.7.1 Introduction 234
12.7.2 Tagging 236
12.7.3 Experiment One 236

12.7.4 Experiment Two 239
12.7.5 Conclusions 243

13 Applications Development Reports 244
13.1 DRA Malvern 244
13.2 BP Research 248
13.2.1 The context for natural language processing 248
13.2.2 BP's experience with the CLARE Project 250
13.3 BT Labs 257
13.3.1 Sept - Nov 1990 Geoffrey Sampson's evaluation of the July 1989 version of the CLE 257
13.3.2 Summer 1990 - April 1991 Interfacing CLARE to BT's SPUD database 257
13.3.3 April-May 1992: The Use of CLARE in Intelligent Query . 258
13.4 British Aerospace Involvement with the CLARE Project 260
13.5 SRI and Swedish partners: A speech-to-speech translation system 262

14 NLP Evaluation Study – Report Digest 266

Chapter 1

Components of CLARE

1.1 CLARE System Architecture

This chapter serves as a short introduction to the architecture of the CLARE system. Readers unfamiliar with earlier NATTIE and CLARE project reports are likely to find it particularly useful.

CLARE is a natural language processing system with built-in facilities for domain modeling and reasoning. The linguistic processing components, which together are referred to as the Core Language Engine (CLE), are application independent, while the domain modeling and reasoning capabilities are aimed at building interactive natural language interfaces to knowledge based systems. For example, the CLE has been used to build an experimental machine translation system for English-Swedish car hire dialogues (Alshawi et al. 1991), and CLARE has been used to build an interface to a Project Resource Management (PRM) database. Much of the work on reasoning is discussed in this report in the context of the PRM application. The approach to language-application interfacing depends on translating logic expressions using a method we call "abductive equivalential translation" (Chapter 2).

CLARE can be viewed as consisting of five subsystems which in turn consist of processing and data components. The subsystems perform the following functions:

- Language analysis
- Language generation
- Lexical acquisition
- Contextual interpretation
- Domain modeling and reasoning.

The language analysis and lexical acquisition components are similar in operation to those used in the version of the CLE resulting from the NATTIE project, except that coverage has been considerably increased under CLARE. (Chapter 12 describes experiments carried out to assess linguistic coverage.) Contextual interpretation has been largely redesigned under CLARE as described in Chapter 6. Language generation and domain modeling and reasoning were developed as new subsystems under CLARE.

There are two main representations used in CLARE. The first, Quasi Logical Form or QLF, is a logic-like language motivated by the needs of language processing. It extends first order logic by the use of generalized quantifiers, higher order operators, and constructs for expressing underspecification of contextually dependent aspects of interpretation. Language analysis produces QLFs which are then refined by contextual interpretation. QLFs are also the input to the algorithm for linguistic generation. The second representation, Target Reasoning Language (or TRL), is a logic aimed at tractable reasoning; it is much simpler than QLF and close to "standard" first order logic. Contextually resolved logical forms are translated into TRL, and the domain model is also expressed in TRL. Figure 1.1 shows, in schematic form, the relationship between CLARE subsystems.

1.2 Processing Phases

The language analysis subsystem makes use of a general purpose unification grammar for English syntax and semantics, and a core lexicon with around 2000 common English word senses. The lexical acquisition tool allows application developers to augment the core lexicon with application specific vocabulary. (An interface is also provided to allow a large external lexicon to be consulted during linguistic processing.) Linguistic processing employs a staged approach with the following phases:

1. Segmentation
2. Morphological analysis
3. Syntactic analysis
4. Semantic analysis
5. Sorts and preference ordering
6. User interaction (optional).

The current approach to segmentation (i.e. low-level lexical processing) is presented in Chapter 11. Linguistic analysis produces a set of contextually underspecified QLFs ordered by a preference metric (the preference mechanism is described in Chapter 10).

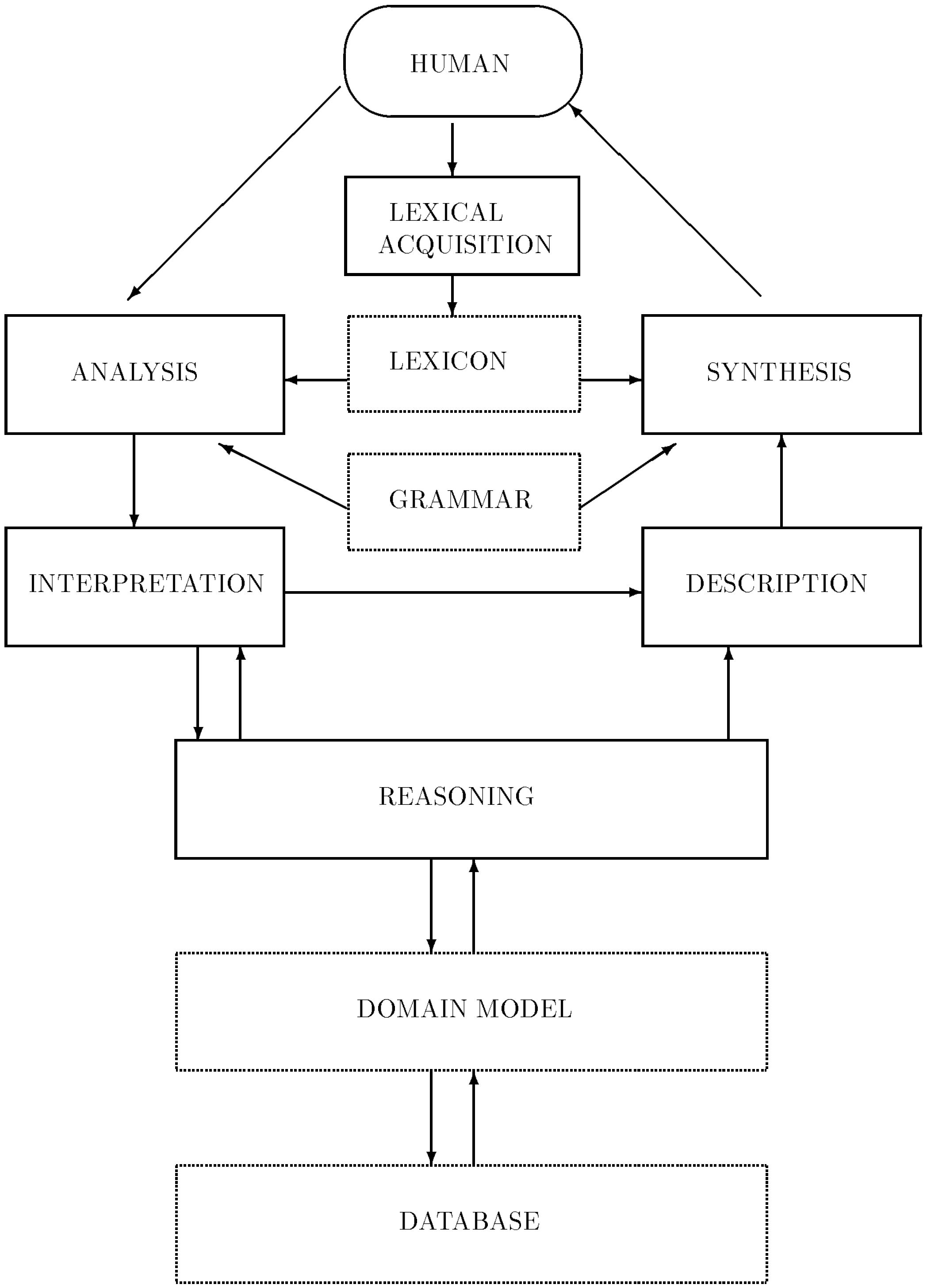


Figure 1.1: Information flow between CLARE data and processing components (data components in dashed boxes)

Contextual interpretation takes underspecified QLFs, and increases their information content by the processes of reference resolution (resolving noun phrase referents, vague relations, ellipsis, and so on) and quantifier scoping. Further preference metrics are applied, and natural language generation allows user verification of some of the choices made during interpretation. The phases of contextual interpretation are as follows:

1. Reference resolution
2. Sorts and preference ordering
3. Paraphrasing substantive resolutions
4. User interaction (optional)
5. Quantifier scoping
6. Preference ordering.

We call the model for interpretation used in CLARE "monotonic semantic interpretation" (Chapter 5) because it acts by further instantiation of underspecified representations without destructive manipulations or loss of information. Interpretation, particularly reference resolution, often makes use of the CLARE domain model and reasoning components (Chapter 6).

In parallel to the division of natural language understanding into analysis and interpretation, we also divide natural language generation into *synthesis* (i.e. linguistic generation) and *description*. Both are covered in Chapter 8. Synthesis takes place from QLF representations by "reverse" application of the unification-based grammar rules and lexical entries. Description is the process of producing a QLF from which synthesis can take place. In CLARE, description is used for two purposes: generation of paraphrases showing the results of interpretation and generating from database assertions including the generation of questions from incomplete assertions. Generation from assertions can take place only if a domain model for interfacing to a database has been defined.

For building a knowledge-based application with CLARE, a domain model (Chapter 3) needs to be specified. We have developed a methodology for declarative specification of a domain model for database interface applications. This methodology is described in detail in the CLARE software manual. For such applications, much of the domain model takes the form of logical equivalences in TRL which act as translation rules between predicates corresponding to word senses and those corresponding to database relations. Processing after contextual interpretation comprises:

1. Translation of resolved QLFs to TRL
2. Translation of TRL to abstract database form

3. Further translation to concrete query or command (e.g. SQL)

4. Execution of query or assertion command.

The translation of queries or assertions with linguistic predicates to expressions with database predicates (Chapter 2) makes use of a reasoner described in Chapter 9. The reasoning process is abductive: it can be used to derive a set of assumptions, given which the database expression is equivalent to the original linguistic logical form. Consequently, assumptions implicit in the database can be coded as conditions on the translation rules and fed back to the user when they are invoked.

1.3 Transcript of CLARE PRM Application

Before describing the technical details of CLARE in the main chapters of this report, we present a transcript of running the system with the Project Resource Management (PRM) application. This should give an informal impression of how the system operates and motivate some of the technical problems addressed later. Although the transcript session is very application-specific, it should be born in mind that CLARE, and especially its language processing components, have been used in several other test applications.

In the PRM application, CLARE is acting as an interface to an ORACLE database containing a dozen or so relations about projects and payments. The structure of the database has its roots in an actual system used at SRI Cambridge, but for confidentiality reasons the original database records have been replaced by fictitious ones. The transcript is edited for brevity, though another version of it that preserves more of the original output layout is given in the CLARE software manual. System and user output is shown in `typeface` font, comments in *italic* font.

```
user>> Show me the projects.
```

A Simple query. After semantic analysis, a bracketing of the sentence is produced, plus a list of word senses. Preference measures normally select the appropriate analysis first, but the user has the option to accept or reject.

After reference resolution, a paraphrase of the resolved QLF is produced if the resolutions are non-trivial. Here it is shown that 'the projects' has been understood as meaning 'all projects'.

Beneath the paraphrase is a list of noun phrases generated from the resolved sentence, and a list of new relations. The noun phrase list often repeats what is already in the paraphrase. But sometimes, generation of the paraphrase fails, and generation of its component noun phrases can provide valuable feedback to the user. The new relations (in this case empty) show what new word senses have been introduced by resolution.

After scoping and conversion to SQL, the answer is printed out, with a key below. In this case, the key says that the first item in every pair is the project, and the second item its project number.

```
   Complete sentence with bracketing:

     "{show {me} {the projects}}."

   Word senses (unordered):

     show: cause to see (rather than "reveal")
     project: activity

Confirm this analysis? (y/n/c/p/?): y

system>>   "show me all projects!"

Confirm this interpretation? (y/n): y

Answer:
        [Archie, 2989]
        [AstroSUD, 1254]
        [Bland Conversation Sprinkler, 8744]
        [CASE Workbench, 8743]
        [CLAM-BAKE, 8468]
        [CandyFloss, 8306]
        [Life Study, 7518]
        [RFO-1990, 1152]
        [WHIZ, 1392]

Key for answer fields:
        [A, B]
   A: project
   B: account number, number, project number

user>> When did CLAM-BAKE start?

system>>   "when did CLAM-BAKE (the project) start?"

Answer: 20/11/1989

user>> When did it end?
```

```
system>>   "when did CLAM-BAKE (the project) end?"

Answer: 19/11/1992

user>> How many CLAM-BAKE payments were made in 1990?
```

This sentence illustrates the resolution of compound nouns. Given the domain model, a plausible interpretation of 'CLAM-BAKE payments' is 'payments on CLAM-BAKE'. This contrasts with an expression like 'Marble Systems payments', which means 'payments to Marble Systems'.

The list of the assumptions states that only payments made by CRI are counted in the answer. If anyone else (not covered by the database) made payments on CLAM-BAKE, they would not be counted.

```
system>>   "how many payments on CLAM-BAKE (the project) were made
            in 1990 (the year)?"

Assumptions used in query translation:

  all_transactions_referred_to_are_from_CRI

Answer: 45

user>> And last year?
```

Resolution of temporal noun phrases, and ellipsis resolution: The paraphrase illustrate how this elliptical question is understood in full.

The warning after the answer is printed because the database is only complete for CRI payments up until 1/4/1991, as stated in the list of assumptions. Since this period ends half way through last year, there may be some payments made last year that are not included in the database, and so not counted in the answer.

```
system>>   "how many CLAM-BAKE (the project) payments were made
            in 1991 (the year)?"

Assumptions used in query translation:

  transactions_referred_to_made_between_1989/8/17_and_1991/4/1
  all_transactions_referred_to_are_from_CRI

Answer: 10

(Warning: the response is based on possibly incomplete information).
```

```
user>> And 1989?
```

Throughout most of this transcript we have chosen the intended interpretations of sentences; here, we diverge for a moment from this principle and show some of the other less plausible resolutions that this elliptical question can be given. One reason for the large number of resolutions (more are proposed than shown here), is that '1989' can be resolved to a year, a project number or just a number. Generation makes the type of resolution explicit.

```
system>>   "how many CLAM-BAKE (the project) payments were made
            in 1989 (the year)?"

Confirm this interpretation? (y/n): n

system>>   "how many 1989 (the year) payments were made
            in 1991 (the year)?"

Confirm this interpretation? (y/n): n

system>>   "how many CLAM-BAKE (the project) payments were made
            in account 1989?"

Confirm this interpretation? (y/n): n

user>> Can you show me the employees who worked on CLAM-BAKE.
```

A meta-question: In order to establish that the system can show the employees, it must determine that it can translate the embedded command ('show me the employees...') into an SQL query. If it can translate, then it can show the employees (whether or not there are any to be shown is another matter). In a spirit of co-operation, as well as giving a literal answer to the question, it also sends of the query constructed in establishing the answer.

```
system>>   "can you show me all employees that worked on
            CLAM-BAKE (the project)?"

Assumptions used in query translation:

  people_who_work_on_projects_do_so_from_their_start_to_their_finish
  all_employees_referred_to_are_at_CRI

Answer: Yes
```

```
    for instance
        Zorba Greek
        Sammy Davis Junior
        Peter Piper
        Maid Marion
        Gordon Bennett
        Geoff Steiner
        Darth Vader


user>> Which of them have cars.
```

Illustrates paraphrasing of more complex anaphora resolution and of bare plurals: 'Them' is resolved to refer to the employees mentioned in the previous question. In this case, it would have been possible generate a paraphrase "Which of the employees that ...', but this is not always syntactically possible and so for robustness a parenthetical appositive is generated instead.

Similarly, the paraphrase 'have some cars' in place of the bare plural 'cars' may seem clumsy. Bare plurals like 'cars' may resolve either to 'all cars' or 'some cars', and in this case we clearly do not wish to ask which employees have all cars. But for a query like 'Show current projects', one would presumably wish all current projects to be shown.

Note also that in translation an assumption is made that the query is about company cars. Since the database only knows about company cars, it could not translate the query without making this assumption. A warning is issued to the effect that this assumption may differ from that made by the user.

```
system>>   "which of them (the employees that worked on
            CLAM-BAKE (the project)) have some cars?"

Assumptions used in query translation:

  all_cars_referred_to_are_company_cars

Answer:
        Peter Piper
        Zorba Greek

Warning: a possibly incorrect assumption was made.

user>> Do you know what the largest payment on CLAM-BAKE was?
```

Another meta-question, and also illustrates superlatives like 'the first', 'the largest', etc.

```
Answer: Yes

Do you want me to tell you? (y/n): y

Answer:
        [100661, 31/10/1990, Marble Systems Ltd, pounds-3395]

Key for answer fields:
        [A,B,C,D]
   A: number, transaction number
   B: date, time
   C: payee
   D: amount, money

user>> How much was it for?

system>>   "how much money was transaction 100661 for?"

  money_is_measured_in_pounds_sterling

Answer: 3395

user>> How many cheques over #1000 did Marble Systems receive?
Answer: 1

user>> Over #200?

system>>   "how many cheques over 200 pounds sterling did
            Marble Systems Ltd (the payee) receive?"
Answer: 6

user>> Show them.
```

CLARE does not always get it right first time. This example shows how paraphrasing is necessary for interactive disambiguation.

The desired resolution would be more effectively paraphrased as 'Show them (the cheques over 200 that Marble Systems Ltd received)'. This paraphrase was in fact attempted from the resolved QLF, but unfortunately failed to produce an English string in this case. Therefore, a simpler paraphrase was attempted.

```
Initial Resolution.
system>>   "show Marble Systems Ltd (the payee)!"
Confirm this interpretation? (y/n): n
```

```
system>>   "show them (the cheques over 200 pounds sterling)!"
Answer:
        [100222, 6/10/1989,  Marble Systems Ltd, pounds-270]
        [100280, 1/12/1989,  Marble Systems Ltd, pounds-360]
        [100338, 9/1/1990,   Marble Systems Ltd, pounds-270]
        [100465, 14/5/1990,  Marble Systems Ltd, pounds-270]
        [100651, 11/10/1990, Marble Systems Ltd, pounds-270]
        [100661, 31/10/1990, Marble Systems Ltd, pounds-3395]

Key for answer fields:
        [A,B,C,D]
   A: number, transaction number
   B: date, time
   C: payee
   D: amount, money

(Warning: the response is based on possibly incomplete information).

user>> How much have we paid Marble Systems?
```

Aggregation: the query is interpreted as as requesting the total amount paid.

```
system>>   "how much money has Cornerstone Research Institute
            paid Marble Systems Ltd (the payee)?"

Answer: 4295

user>> Tell me about CASE Workbench.
```

Generation from the database. English sentences are generated direct from ORACLE database records. Generation does not proceed by slotting names into predefined templates, but emerges naturally from the more general definition of the domain. Checks are made to ensure that (a) all possible information from each record is printed out, and (b) that no sentence merely repeats information that has already been given.

```
------------------------------------------------
Trying to describe a database record:
system>>   "CASE Workbench (the project)'s end date is
          31/1/1992 (the day)."
system>>   "CASE Workbench (the project)'s number is 8743."
system>>   "CASE Workbench (the project)'s start date is
          1/2/1990 (the day)."
(The record was fully described)
```

```
-------------------------------------------------
Trying to describe a database record:
system>>   "Geoff Steiner (the employee) is a person."
system>>   "CASE Workbench (the project) has Geoff Steiner (the employee)."
(The record was fully described)

user>> FOO is a project.
```

Making assertions: As well as querying the database, one can also add new information. Moreover, this information does not have to be added in such a way that database records are completed in one fell swoop. As the assertion below shows, three of the for fields in the `CRI_PROJECT` *(*`C,D,E`*) have not been instantiated yet.*

```
Assertion:

CRI_PROJECT(FOO,_,_,_),

user>> .ask
(try to complete partial database records by asking questions [clare])
```

CLARE includes a facility for asking the questions needed to complete database records. As with generation from database records, no special templates need to be predefined. The form of the question is determined automatically from the general domain definition.

Answers to the questions are treated as ordinary English sentences, which may be elliptical. These go through the normal process of analysis and translation without being treated in any special way.

CLARE keeps asking questions until it has completed all the incomplete database records, whereupon it returns to the normal CLARE loop, allowing the user to ask the questions instead of the system.

```
system>> what is FOO (the project)'s account?

user>> 1234.

system>>    "account 1234 is FOO (the project)'s account."

Confirm this interpretation? (y/n): y
Assertion:

  CRI_PROJECT(FOO,1234,_,_)

system>> what is FOO (the project)'s start date?
```

```
user>> It started on 1/1/91.

system>>   "it (the start date of FOO (the project))
            started on 1/1/1991 (the day)."
```

Answer is a full sentence rather than an elliptical noun phrase. Unfortunately the first resolution of the pronoun 'it' is not the one required, and the second reading is accepted.

```
Confirm this interpretation? (y/n): n

system>>   "FOO (the project) started on 1/1/1991 (the day)."

Confirm this interpretation? (y/n): y

Assertion:
  CRI_PROJECT(FOO,1234,1-JAN-91,_)

system>> what is FOO (the project)'s end date?

user>> Tomorrow.

system>>   "27/11/1992 (the day) is FOO (the project)'s end date."

Assertion:
CRI_PROJECT(FOO,1234,1-JAN-91,27-NOV-92)
```

New record is now complete.

Chapter 2

Interfacing by Abductive Equivalential Translation

2.1 Introduction

In this and the next chapter, we will describe the modules in CLARE that allow it to be used as a natural-language interface to a knowledge base, and in particular to a relational database. Before going any further, we will establish more precisely what we mean by "natural-language interfacing". Normally, the problem is thought of as that of producing answers to natural-language questions posed to a knowledge base (cf. Perrault and Grosz, 1988). Here, we will adopt a more general view; we will regard it as the problem of formalizing the relationship between 1) the contents of a database, and 2) the way in which people use language to talk about it. Asking questions about the contents of databases is far from being the only way in which language and database can be related; thus apart from question-answering, we wish the model to support reasoning that will allow at least the following.

- Responding to natural-language commands.
- Accepting statements describing new records for the database.
- Reasoning about whether the database contains the information necessary to answer an NL question.
- Indicating whether answers to WH-questions are complete or not, and distinguishing between "No" and "Don't know" answers to Y-N questions.
- Describing existing records in the database using NL.
- When records are incomplete, formulating NL questions to elucidate the information needed to complete them.

We want the information needed to support these functionalities to be expressed declaratively, using some sort of predicate logic; the result will be a *linguistic domain theory* for the database, a theory of how language and database are related. Languages are normally based on vague, common-sense concepts, databases on precise, formal ones. Relating these two disparate systems raises the usual problem of common-sense reasoning: interpretation of vague NL concepts in terms of precise database concepts is often only possible if additional assumptions, not implied by the explicit linguistic content, are made. These assumptions are in general defeasible. In some cases, it may be desirable to relay them back to human users. If the scheme is to be practically useful, we also want to be able to specify a methodology for constructing linguistic domain theories. For the usual software engineering reasons, they should also, as far as possible, be modular and re-usable. These, in brief, are our top-level goals. We will now make them more specific, and relate them to the concrete architecture of the Core Language Engine.

The CLE, being a general natural language system (rather than one tailored to a specific application), constructs representations of utterances that essentially mirror their linguistic content. Normally, every content word in the utterance will correspond to an occurrence of a word-sense predicate in that utterance's TRL representation. Thus for example in the CLARE Project Resource Management domain the sentence

(S1) List all payments made to BT during 1990.

gets a TRL representation which contains predicates corresponding directly to *list*, *payment*, *make* and *during*. In order to carry out the command, however, CLARE needs to construct an SQL `SELECT` statement which searches for `TRANS` tuples where the `payee` field is filled by `bt`, and the `cheque_date` field by a date constrained to be between 1st January and 31st December, 1990. The differing nature of the two representations can lead to several possible kinds of difficulties, depending on how the "linguistic" and "database" representations are connected. There are three in particular that we will devote most of our attention to in what follows:

1. A query can be *conceptually* outside the database's domain. For example, if "payments" in (S1) is replaced by "phone-calls", the interface should be able to indicate to the user that it is unable to relate the query to the information contained in the database.

2. A query can be *contingently* outside the database's domain. Thus if "1990" is replaced by "1985", it may be possible to derive a query; however, if the database only contains records going back to 1989, the result will be an empty list. Presenting this to the user without explanation is seriously misleading.

3. A query may need additional implicit assumptions to be translatable into database form. Asking (S1) in the context of our example Project Resource Management domain, it is implicitly understood that all payments referred to have been made by SRI. If the user receives no feedback describing the assumptions that have been made to perform the translation, it is again possible for misunderstandings to arise.

With the problems just mentioned born in mind, we will consider previous methodologies used to attack the NL interfacing problem. The currently most popular approach is perhaps to attempt to effect the connection between LF and database query by encoding the database as a set of unit clauses, and then building an interpreter for the logical forms which captures the relations between linguistic and database predicates as "rules" or "meaning postulates" written in Horn-clause form (cf. e.g. McCord 1987, Pereira and Shieber 85). Anyone who has experimented with this scheme will, however, know that it tends to suffer from all three of the types of problem listed above. This is hardly surprising, when one considers that Horn-clauses are "if" rules; they give conditions for the LF's being true, but (as pointed out in Konolige 1981), they lack the "only if" half that says when they are false. It is of course possible to invoke the Closed World Assumption (CWA); in this interpretation, finite failure is regarded as equivalent to negation. Unfortunately, experience also shows that it is extremely difficult to write meaning postulates for non-trivial domains that are valid under this strict interpretation.

For these reasons, Scha (1983) argues that approaches which express the connection between LF and database query in terms of first-order logic formulas are unpromising. Instead, previous approaches to query derivation which attempt to justify equivalence between queries and semantic representations have been limited (at least in implemented systems) to employing restricted forms of inference. Examples are the type inference used in PHLIQA (Bronnenberg et al 1980) and Stallard's 'recursive terminological simplification' (Stallard 1986).

In what follows, we will show how a more general deductive approach can be taken. This depends on coding the relationship between LF and database forms not as Horn-clauses but as "definitional equivalences", conditional if-and-only-if rules of a particular form. Our approach retains computational tractability by limiting the way in which the equivalences can take part in deductions, roughly speaking by only using them to perform directed "translation" of predicates. However we still permit nontrivial goal-directed domain reasoning in justifying query derivation, allowing, for example, the translation of an LF conjunct to be influenced by any other LF conjuncts, in contrast to the basically local translation in PHLIQA. This approach deals with the first two points above without recourse to the CWA and simultaneously allows a clean integration of the "abductive" reasoning needed to take care of point 3. As we shall see, it also makes it possible to use substantially the same framework to achieve interfacing functionalities other than question-answering. The main technical problems to be solved are caused by the fact that the left-hand sides of the equivalences are generally not atomic.

The rest of this chapter is structured as follows. In the next section, we give an overview of the interface system, concentrating on the denotational and functional aspects. We describe roughly what information will be found in a linguistic domain theory (LDT), and how the interface functionalities listed at the beginning of the section can be formalized as tasks of the form " given a formula P, find a formula P' of a specified type, such that P and P' are equivalent given the LDT and some permitted assumptions." We will refer to tasks of this kind as "performing abductive equivalential translation (AET) on P". In section 2.4, we describe AET as a computational process. Section 2.5 describes the *simplifier*, a module that plays an important supporting role in the translation process; section 2.6 considers the connection between AET and the Closed World Assumption. In the following chapter, we describe in detail the construction of LDTs for relational database interfacing.

2.2 Ontological issues

We start by defining the basic ontology underlying the linguistic domain theory; the ideas are closely related to those proposed in (Konolige 1981). We assume as usual that there is a set of objects, **O**, and a set of relations obtaining between them, **R**. **O** will contain all the things that the predicates in logical forms range over — the things, like suppliers, parts, projects, payments, deliverables, etc. that natural language refers to. This includes events. Similarly, **R** will contain all the relations obtaining between elements of **O** that we will wish to refer to.

There will be no particular need in what follows to be much more specific about exactly what can and cannot belong to **O** and **R**, though this is obviously not a trivial matter. We will however be interested in picking out certain distinguished subsets of these two sets which have direct relevance to database interfacing. We take the position that databases are also objects in the world, consisting of finite collections of rows of marks; this is not a common viewpoint at the moment, but we think it corresponds well to the naive view of "what a database is". We consequently assume the existence of a subset $\mathbf{O_{db}}$ of **O** consisting of *database objects*, which will be numbers, strings, date representations, and other things that can be found filling fields in database records. Similarly, we will assume the existence of a subset $\mathbf{R_{db}}$ of **R**, which will be the *database relations*. Database relations are defined by the database's internal structure in terms of tuples: a database relation D holds of a list of arguments Arg_i iff there is a tuple of type D in the database whose fields are filled by the Arg_i. Since little can be done with database objects without recourse to elementary arithmetic and an ability to display results, we assume two more distinguished subsets of

R. **Arith** will be the relevant arithmetic relations (like "being numerically less than") that can obtain between members of O_{db}. **Exec** will be a primitive set of relations which the interface can cause to hold by performing actions. A typical relation in **Exec** would be the one that holds between a database-object, and a time and place at which the interface displayed it.

In what follows, the distinction between database objects, the non-database objects they name, and the terms in the linguistic domain theory that refer to both of them will be central. Thus for example we distinguish between

- a transaction (a non-database object, an event in the exterior world)
- the term referring to the transaction in the linguistic domain theory
- the database object (a number) that is the transaction's ID
- the term in the theory referring to the database object.

Although these distinctions may not immediately seem necessary, they are in fact motivated by typical properties of real databases. Database objects are often codes or non-unique names, that cannot simply be treated as names in a first-order theory; for example, the same number, occurring in different fields in a relation, can be related to distinct real-world entities, or (if used as a code value), to a property of an entity.

The linguistic domain theory relates database objects with real-world objects, allowing us to draw conclusions about the properties of the real-world objects by examining the database records. Less obviously, it can also allow us to draw conclusions about properties of real-world objects from the *lack* of records of a particular type in the database. This amounts to a principled generalization of the "closed world assumption", and is described further in section 2.6. The linguistic domain theory can also partially model the interaction between system and user, by defining a relation between predicates holding in the "utterance situation" (the real-world situation in which the user is interacting with the interface), and the executable relations. For example there are real-world predicates which correspond to verbs like "show" and "list", which are commonly used in commands (e.g. "Show me all payments over £500"); these are related by the linguistic domain theory to the primitive executable relation of displaying characters on the screen. Treating the utterance situation uniformly as part of the domain adds both conceptual elegance and a real increase in the interface's scope; so it is for instance possible to deal with queries like "Which of the payments that you showed me in answer 12 were made this year?" in a principled way.

2.3 Formalizing NL interfacing functionalities

2.3.1 Preliminaries

At a pre-theoretical level, our characterization of the NL interfacing functionalities is actually a very simple one: it is only the technical details that will prove complex. The key notion is that of "translation". Given an interfacing task, described in one vocabulary, we wish to *translate* it into an equivalent task, described in another vocabulary. Thus for example the "question-answering" task is of the form "produce an answer to the question whose logical form is P"; we wish to translate this into an equivalent task of the form "execute the database query P', and print the results in a readable way". Here, the original task is expressed using predicates corresponding to word-senses; the problem is to translate it into a task expressed using predicates corresponding to database relations, arithmetical operations and primitive executable relations. What we want to do now is to express these ideas in formal terms, so that we can then realize them as an inference problem. In particular, we want to introduce linguistic domain theories so that translation can be regarded as logical equivalence with respect to a theory. We will start by sketching the appearance of a linguistic domain theory, and defining the formal concept of *effective translation*. Throughout most of the chapter, we will regard it as sufficient to allow the target representation (the result of performing the translation) to be a predicate calculus expression that treats database relations as predicates; in section 2.7, we briefly review the module that performs the final conversion into executable SQL queries.

2.3.2 Basic form of linguistic domain theories

Since we are primarily interested in establishing equivalences (rather than implications), it makes sense to write the LDT as far as possible using formulas which themselves are equivalences. For various reasons, it turns out that it is convenient to allow these equivalences to be conditional; to limit computational complexity, we will only allow their left- and right-hand sides to be existentially quantified conjunctions of atomic formulas. Thus in summary, the greater part of an LDT will be composed of formulas of the general appearance

$$\forall \vec{x}.(Left \equiv Right) \leftarrow Conds$$

where $Left$, $Right$ and $Conds$ are existentially quantified conjunctions. Linguistic domain theories typically also contain Horn-clause axioms and declarations of functional relationships between arguments of predicates (these are described in detail in section 2.5.2). It is permitted during inference to assume goals abductively, at a cost depending both on the goal and the context in which the assumption is made. We do not attempt to define a formal semantics for the notion of abductively justified inference: we merely assume that costs are assigned according to some scheme that generally makes low-cost proofs intuitively more appealing than high-cost ones. Since abductive assumptions are in any event made explicit to the user of the system, we view the costs essentially as heuristics to control the order in which the space of possible proofs is searched. The range of goals that may be abductively assumed is controlled by declarations of the form

```
assumable(Goal,Cost,Justification,Type,Cond)
```

where `Goal` and `Cond` are atomic formulas, `Cost` is a non-negative integer, `Justification` is a unique tag that names the assumption-type, and `Type` is the assumption-type (this is defined below). The intended semantics are that `Goal` may be assumed at cost `Cost` in an environment where `Cond` holds. The "justification" can be used to identify the assumption to the user. The inference engine (which is basically a backwards-chaining Horn-clause interpreter) contains no general treatment of negation, but in order to deal with cases where `Goal` is explicitly contradicted by its context it is possible to define rules of the form

```
neg(Goal) <- Body
```

where `Goal` is an atomic formula and `Body` is an arbitrary formula. A rule of this type has the content "the negation of `Goal` is implied by `Body`". It is only invoked to attempt to identify inconsistent uses of assumptions. The framework thus allows in effect a limited use of normal defaults.

Experimentation with CLARE seems to indicate that one can profitably divide acceptable assumptions into at least three distinct types; we illustrate using the example PRM domain, which covers a project and payment database for SRI Cambridge. In the PRM domain, it is reasonable to assume (lacking evidence to the contrary) that "project" is equivalent with "project at SRI Cambridge". The content of assumptions of this kind is that the speaker means something more specific than what was actually said. We consequently refer to them as "specializations". In contrast, it is also reasonable to assume that "payment" means "SRI payment made during the period covered by database records". In this case, however, it seems intuitively less clear that the speaker intended to use the word in the more restricted sense, and it is more appropriate to assume that the database is limited in a way which the speaker may not be fully aware of. We will call assumptions of this kind "limitations". Finally, it may sometimes be appropriate to make assumptions that can actually be incorrect: for example, we assume that completion dates for future deliverables have been correctly estimated. We call these assumptions "approximations". Distinguishing between different kinds of assumption will become important later on, when we consider issues regarding the completeness of answers to questions.

Effective translation

We now introduce the key formal definition. We write Γ to symbolize a linguistic domain theory, and let F_{source} be an arbitrary formula; then we define F_{target} to be an *effective translation* of F_{source} iff there exists a set of abductive assumptions A such that

(2.1) $\Gamma \cup A \Rightarrow (F_{source} \equiv F_{target})$

and

- Each assumption in A is acceptable in the context in which it is made.
- There is a finite proof procedure for determining the truth of F_{target}.

The question of whether or not a finite proof procedure exists for a formula, given a theory, is of course undecidable in general; CLARE only attempts to solve a simple subcase of this problem.

The first criterion that must be met is that all predicates in F_{target} must be *potentially finite*: that is to say, they should have the property that for at least some combinations of instantiation of their arguments there are only finitely many instantiations of the remaining arguments that make them true, and that these instantiations can be found within bounded time. Thus the minimum requirement is that if all the arguments are ground it is possible to determine the truth of the relation within bounded time. There are in practice three kinds of potentially finite predicates in a domain model: database predicates, arithmetic relation predicates, and primitive executable predicates. We examine each of these in turn:

Database predicates These are predicates directly corresponding to the database relations $\mathbf{R_{db}}$; a database predicate holds of its arguments iff the database relation has a row with those arguments. The finiteness of database predicates follows from the finiteness of the corresponding database relations. Database predicates are consequently always finite, irrespective of their instantiation.

Arithmetic relation predicates These correspond the arithmetic relations in **Arith**, such as addition, subtraction, and inequality. In general, arithmetic relation predicates are only finite if all or all but one of their arguments are instantiated. For example, if X is instantiated to a given number, there is an infinite set of values for Y such that $X < Y$ holds. If both X and Y are fixed, however, the truth or falsity of $X < Y$ can in practice be determined in bounded time.

Primitive executable predicates In order to be able to reason about CLARE's ability to perform actions like displaying of objects, there is a set of predicates which correspond to the primitive executable relations **Exec**. We assume that these predicates are finite for *instantiated* actions.

Since the finiteness of some predicates depends on their instantiation, the existence of a finite proof procedure generally depends on being able to find a evaluation order which ensures that conditionally finite predicates are sufficiently instantiated by the time they are queried; CLARE can search for suitable evaluation orders by permuting the order of evaluation of conjunctions. For example, if $TRANS/3$ is a database predicate then the strategy "find an X such that $X > 100$, then find values of Y such that $TRANS(john, X, Y)$" is not a finite strategy; however, reversing the order to make the strategy "find X and Y such that $TRANS(john, X, Y)$, then determine whether $X > 100$ holds" is finite. The search is carried out using a simple abstract interpretation method, which uses meta-information about finiteness of predicates with respect to different instantiation patterns to mimic the behaviour of the real execution module. This part of the system is described further in section 9.3.

2.3.3 Yes/No questions

We are now in a position to describe formally the various interfacing functionalities. The simplest one to begin with is that of answering a Y-N question. The formal statement of the problem is as follows. We are given a formula, F_{ling}, which is the logical form of a Y-N question to be answered. We wish to find a formula F_{eval} and a set of permissible assumptions A, such that F_{eval} is an effective translation of F_{ling} modulo A in the sense defined in section 2.3.2. There are several possibilities.

- No such F_{eval} can be found: the answer is **Don't know**.
- A is empty and F_{eval} can be proved: the answer is **Yes**.
- A is empty and F_{eval} cannot be proved: the answer is **No**.
- A is non-empty and F_{eval} can be proved: the answer is **Yes, conditional on the validity of the assumptions**.
- A is non-empty and F_{eval} cannot be proved: the answer is **No, conditional on the validity of the assumptions**.

If assumptions have been made, the user is told what they were. An example of each case, taken from the PRM domain, follows.

- The question is *Does Peter have a dog?*. There is no effective translation of the logical representation of this question, so the answer is **Don't know**.
- The question is *Does Peter have a car?*. Peter is an employee, and there is an effective translation to a query that accesses the `SRI_EMPLOYEE` relation. The translated query can be proved, so the answer is **Yes**.

- The question is *Does Gordon have a car?*. This is the same as the previous case, except that Gordon does not have a car according to the database, and the effective translation cannot thus be proved. The answer is **No**.
- The question is *Has Peter booked less than 200 hours to CLARE?* Peter is an employee, and CLARE is a project, so there is an effective translation to a query that accesses the `BOOKING_TO_PROJECT` relation under the assumption that the time referred to was booked during the period for which records were kept. The database reveals that only 165 hours were booked during this period, so the answer is **Yes, conditional on the validity of the assumptions**.
- The question is *Has Peter booked less than 100 hours to CLARE?* The assumption used when translating is the same as in the last example, and the answer is **No, conditional on the validity of the assumptions**.

2.3.4 Commands and WH-questions

The functionality of responding to a natural language command can be captured by a fairly simple extension of the previous definition. We will assume that the logical form representation of a command is a formula which is true if the command will be carried out at some future time. Once again, we let F_{ling} be the logical form of the command, and the task is to find an effective translation F_{eval} modulo a set of assumptions A. F_{eval} will usually be a formula that contains primitive evaluable predicates.

Answering WH-questions can be treated as a special case of responding to commands, if we assume that the logical form of a WH-question is of the type $\lambda x.Q(x)$, where $Q(x)$ holds iff x is an object satisfying the question. (cf. Rayner and Janson 1987). Thus for example the logical form for *Which payments in June were over £1000?* will be roughly

```
X^and(payment1(X),
      and(in1(X,june1),
          and(over1(X,1000))))
```

The asking of a WH-question can be regarded as a command to display all the objects that would be answers to it. So if $\lambda x.Q_{ling}(x)$ is the logical form of the question, $displayed(x,t)$ is a predicate that holds if x is displayed at time t, and $in\_future(t)$ is a predicate that holds of future times, then the question can be regarded as equivalent with a command whose logical form is

$$\forall x.Q(x) \rightarrow \exists t.displayed(x,t) \wedge in\_future(t)$$

The different cases that can arise in responding to a command or WH-question are fairly similar to those for responding to Y-N questions. We summarize, this time without examples:

- No such F_{eval} can be found: the answer is **Don't know how to respond**.
- A is empty, or only contains "specialization" assumptions, and F_{eval} can be proved: the response is to perform all the necessary actions (e.g. displaying objects) and inform the user that the response is complete.
- A is empty, or only contains "specialization" assumptions, and F_{eval} cannot be proved: the response is to inform the user that it is impossible to carry out the command. (This cannot occur with questions).
- A contains "limitation" or "approximation" assumptions, and F_{eval} can be proved: the response is to perform all the necessary actions and inform the user that the completeness and/or accuracy of the answer depends on the assumptions.
- A contains "limitation" or "approximation" assumptions, and F_{eval} cannot be proved: the response is to inform the user that it is impossible to carry out the command if the assumptions are correct.

2.3.5 Meta-questions

Both Y-N and WH questions can occur embedded in meta-questions of the form "Do you know Q?". (E.g. "Do you know how many people work on CLARE?", "Do you know whether there were any payments on 1/11/91?"). If F_{ling} is the logical form of Q, it follows from the preceding discussion that a sufficient criterion for an affirmative answer is that there is an effective translation F_{eval} modulo a set of assumptions A, where A contains only "specialization" assumptions. There are also several ways in which inability to meet this criterion can be meaningful to a user. If translation can only be carried out by assuming a "limitation" or "approximation" assumption, or by making an assumption whose negation is implied by its context, then the offending assumption normally constitutes a good explanation of "why the system doesn't know".

Thus for example, if database records only extend back to 17/8/88 it may only be possible to translate the predicate corresponding to the word "payment" into an expression involving the transaction relation if it is assumed that the payment was made after 17/8/88. In this case the meta-question "Do you know whether there were any payments on 5/5/86?" will receive the answer **No, because this violates the assumption that all payments referred to were made after 17/8/88**.

2.3.6 Generating statements and questions

Finally, we consider the functionalities of generating statements and questions. If S_{db} is a database formula that we wish to realize in language, the problem is to find a set of NL statements S_i $(i = 1...n)$ with logical forms L_i, and an acceptable set of assumptions A such that

$$\Gamma \cup A \Rightarrow (\bigwedge L_i \equiv S_{db})$$

This is fairly obvious; more interestingly, a very similar treatment works for questions. For example, if Q_{db} is a formula composed of database predicates, and containing a free variable x, then $\lambda x.Q_{db}$ can reasonably be thought of as an abstract representation of the WH-question "What x satisfies Q_{db}?". Assuming as before that the logical form for a WH-question is a lambda-abstraction, the problem of finding a linguistic realization of $\lambda x.Q_{db}$ can be formalized as that of finding a single natural-language question Q with logical form $\lambda x.Q_{ling}(x)$, and an acceptable set of assumptions A such that

$$\Gamma \cup A \Rightarrow \forall x.(Q_{ling}(x) \equiv Q_{db}(x))$$

The issues involved are discussed at greater length in chapter 8.3.

2.4 Abductive Equivalential Translation

2.4.1 Translation schemas

We will now consider the actual computational mechanisms used to effect the task of carrying out abductive translation. Recall that the main body of the declarative knowledge used is coded as a set of conditional equivalences, formulas of the type[1]

(2.2) $Conds \rightarrow (\exists \vec{x}.P_1 \wedge P_2 \wedge P_3...) \equiv P'$

In this and the next chapter, these rules are written in a notation illustrated by the following example,

```
exists([Event],
  and(work_on1(Event,Person,Project),
      project1(Project))) <->
DB_PROJECT_MEMBER(Project,Person)
```

in which `work_on1` and `project1` are linguistic predicates and `DB_PROJECT_MEMBER` is a database relation (we will adhere to the convention of capitalizing names of database relations).

The attractive aspect of this type of equivalence stems from the fact that it can be given a sensible interpretation in terms of the procedural notion of

[1]Quantification over the $\vec{x}$ on the left-hand side will often in practice be vacuous. In this and other formulas, we assume implicit universal quantification over free variables.

"translation". We will neglect for the moment the existential quantification, i.e. we will assume that (2.2) has the simplified form

(2.3) $Conds \rightarrow (P_1 \wedge P_2 \wedge P_3 ...) \equiv P'$

(We will return to considering the general form (2.2) in section 2.4.3). The intuitive idea is that is that (2.3) can be read as "P_1 can be expanded to P' if it occurs in an environment where $P_2 \wedge P_3 ... \wedge Conds$ can be inferred". The "environment" is provided by the conjuncts occurring together with P_1 in the original logical form, other meaning postulates, and the contents of the database. This provides a framework in which arbitrary domain inference can play a direct role in justifying the validity of the translation of an LF into a particular database query.

We hasten to give our claim some formal backing; our strategy will be to present it first in a fairly specific form, then in a more general one. Suppose then that P is a ground goal that unifies with P_1 with most general unifier θ, i.e. $\theta(P_1) = P$. Suppose also that P occurs in a context where it is conjoined with another formula R, such that R implies $\theta(P_2 \wedge P_3 ... \wedge Conds)$. Then we claim that

(2.4) $R \wedge P \equiv R \wedge \theta(P')$

The proof is simple. In the $\Leftarrow$ direction, we have that R implies $\theta(Conds)$ by hypothesis, and that $\theta(P' \wedge Conds)$ implies $\theta(P_1), which by hypothesis is equal to P$, from 2.3. Hence $R \wedge \theta(P')$ implies P and thus $R \wedge P$.

In the $\Rightarrow$ direction we have by hypothesis that $P = \theta(P_1)$ and R implies $\theta(P_2 \wedge P_3 ... \wedge Conds)$. Hence $R \wedge P$ implies $\theta(P_1 \wedge P_2 \wedge P_3 ... \wedge Conds)$ which by (2.3) implies $\theta(P')$. Hence $R \wedge P$ implies $R \wedge \theta(P')$. We can summarize (2.4) as the inference rule

$$\begin{array}{lll} & & (Conds \rightarrow (P_1 \wedge P_2 \wedge P_3 ...) \equiv P') \wedge \\ & & R \rightarrow \theta(P_2 \wedge P_3 ... \wedge Conds) \\ (2.5) & \Rightarrow & R \wedge \theta(P_1) \equiv R \wedge \theta(P') \end{array}$$

The next step is to generalize (2.5) by making explicit the concept of the "conjunctive context". We do this by splitting it into two inference rules, (2.6) and (2.7), as follows:

$$\begin{array}{lll} & & (Conds \rightarrow (P_1 \wedge P_2 \wedge P_3 ...) \equiv P') \wedge \\ & & Context \rightarrow \theta(P_2 \wedge P_3 ... \wedge Conds) \\ (2.6) & \Rightarrow & Context \rightarrow (\theta(P_1) \equiv \theta(P')) \end{array}$$

$$\begin{array}{lll} & & Context \wedge Q \rightarrow (P \equiv P') \\ (2.7) & \Rightarrow & Context \rightarrow (P \wedge Q \equiv P' \wedge Q) \end{array}$$

The proofs of (2.6) and (2.7) follow from that of (2.5). The two rules can be used recursively together to translate formulas composed using an arbitrary number of occurrences of the conjunction operator. In both rules, the formulas before the $\Rightarrow$ are the premises, and the formula after the conclusion. (2.6) is the base case: it gives sufficient conditions for using (2.3) to translate P_1 to P'. The other formula, (2.7), is the recursive case; it expresses translation of a conjunction in terms of translation of one of its conjuncts, adding the other conjunct to the conjunctive context as it does so. The advantage of splitting (2.5) into (2.6) and (2.7) is that it is then possible to add rules for other logical operators, each of which passes around the conjunctive context; we will call rules of this kind *translation-schemas*. We also show the translation-schemas for quantified expressions and implications. (2.8) and (2.9) express translation of a quantified form in terms of translation of its body. In each of them, θ substitutes a unique constant for the $\vec{x}$.

$$\begin{array}{lll} & & Context \rightarrow (\theta(P) \equiv \theta(P')) \\ (2.8) & \Rightarrow & Context \rightarrow (\exists\vec{x}.P \equiv \exists\vec{x}.P') \end{array}$$

$$\begin{array}{lll} & & Context \rightarrow (\theta(P) \equiv \theta(P')) \\ (2.9) & \Rightarrow & Context \rightarrow (\forall\vec{x}.P \equiv \forall\vec{x}.P') \end{array}$$

(2.10) and (2.11) express translation of an implication in terms of translations of its left- and right-hand sides. Note that the left-hand side is added to the context when translating the right-side side in (2.11), but not *vice versa*. The proofs of (2.8)–(2.11) are straight-forward.

$$\begin{array}{lll} & & Context \rightarrow (P \equiv P') \\ (2.10) & \Rightarrow & Context \rightarrow (P \rightarrow Q \equiv P' \rightarrow Q) \end{array}$$

$$\begin{array}{lll} & & Context \wedge P \rightarrow (Q \equiv Q') \\ (2.11) & \Rightarrow & Context \rightarrow (P \rightarrow Q \equiv P \rightarrow Q') \end{array}$$

The central use of the equivalences is thus as truth-preserving conditional rewriting rules, which licence translation of one of the conjuncts on the left-hand side into the right-hand side, in suitable conjunctive contexts. There is a second use of the equivalences as normal Horn-clauses, which as we soon shall see is also essential to the translation process. An equivalence of the form

$$\forall\vec{x}.(P_1 \wedge P_2 \wedge \ldots \equiv \exists\vec{y}.Q_1 \wedge Q_2 \wedge \ldots) \leftarrow Conds$$

implies the validity, for any k, of all Horn-clauses either of the form

$$\forall\vec{x}.\forall\vec{y}.(P_k \leftarrow Q_1 \wedge Q_2 \wedge \ldots \wedge Conds)$$

or

$$\forall \vec{x}.(\theta(Q_k) \leftarrow P_1 \wedge P_2 \wedge \ldots \wedge Conds)$$

where θ replaces the $\vec{y}$ with Skolem functions of the $\vec{x}$. We will refer to these, respectively, as *normal* and *backward* Horn-clause readings of the equivalence. For example, the rule

```
and(man1(X),employee1(X)) <->
exists([HasCar],employee(X,m,HasCar))
```

produces two normal Horn-clause readings,

```
man1(X) <- employee(X,m,HasCar).

employee1(X) <- employee(X,m,HasCar).
```

and one backward Horn-clause reading,

```
employee(X,m,sk1(X)) <- man1(X),employee1(X).
```

where `sk1` is a Skolem function. In the implementation, each `equiv` rule is compiled in three different ways, to yield the normal Horn-clause, backward Horn-clause and equivalential readings. The Horn-clause readings are used to support proofs from the conjunctive context.

We can now define the basic translation process, still neglecting for the time being the problems associated with abductive proof and existential quantification on the left-hand side of equivalences. Our strategy is to use (2.6) and the translation-schemas as the kernel of a system that allows translation of logical forms, using the equivalences as expandable complex definitions.

The actual process of translation of a complex formula F is a series of single translation steps, each of which consists of the translation of an atomic constituent of F. A translation step contains the following sub-steps:

Recurse: descend through F using the translation-schemas, until an atomic subformula A is reached. During this process, a conjunctive context E has been accumulated in which conditions will be proved, and some bound variables will have been replaced by unique constants.

Translate: find a rule $(H \wedge R \equiv B) \leftarrow C$ such that H unifies with A with m.g.u. θ. If it is then possible to prove $\theta(R \wedge C)$ in E, replace A with $\theta(B)$. When carrying out a **translate** step, we will sometimes refer to H as the *head* of the rules used, and $R \wedge C$) as its *conditions*.

Simplify: if possible, apply simplifications to the resulting formula.

2.4.2 A simple example

An example follows to illustrate how the process works. In the interests of expositional clarity we use a simplified version of the actual CLARE domain rules. We start with the sentence (S2),

(S2) Do any women work on CLARE?

which receives the LF

```
exists([Person,Event],
       and(woman1(Person),
           work_on1(Event,Person,clare)))                (LF1)
```

(LF1) has to be mapped to a query which accesses two database relations, SRI_EMPLOYEE(Empl,Sex,HasCar) and SRI_PROJECT_MEMBER(Empl,Project) (Sex can be w or m, and HasCar can be y or n). The desired result is thus (LF2):

```
exists([Empl,HasCar],
       and(SRI_EMPLOYEE(Empl,w,HasCar),
           SRI_PROJECT_MEMBER(clare,Empl)))              (LF2)
```

The most clearly non-trivial part is justifying the conversion between the linguistic relation woman1(X) and the database relation SRI_EMPLOYEE(X,w,_). Even in the limited PRM domain, it is incorrect to state that "woman" is equivalent to "employee classed as being of female sex"; there are for example several women who are listed in the PAYEE relation as having been the recipients of payments. It is more correct to say that a tuple of type SRI_EMPLOYEE(X,w,_) is equivalent to the conjunction of two pieces of information: firstly that X is a woman, and secondly that she is an employee. This can be captured in the rule

```
and(woman1(Person),
    employee1(Person)) <->
exists([HasCar],
  SRI_EMPLOYEE(Person,w,HasCar))                         (EQ1)
```

In the left-to-right direction, the rule can be read as "woman1(X) translates to SRI_EMPLOYEE(X,w,_), in contexts where it is possible to prove employee1(X)." For the rule to be of use in the present example, we must therefore provide a justification for employee1(X)'s holding in the context of the query. The simplest way to ensure that this is so is to provide a Horn-clause meaning postulate,

```
employee1(X) <-
  SRI_PROJECT_MEMBER(Project,X).                        (HC1)
```

which encodes the fact that project members are employees.

Similarly, we will need an equivalence rule to convert between work_on1 and SRI_PROJECT_MEMBER. Here the fact we want to state is that project-members are precisely people who work on projects, which we write as follows:

```
exists([Event],
  and(work_on1(Event,Person,Project),
      project1(Project))) <->
SRI_PROJECT_MEMBER(Project,Person)                        (EQ2)
```

We will also make indirect use of the rule that states that projects are objects that can be found in the first field of an `SRI_PROJECT` tuple,

```
project1(Proj) <->
exists([ProjNum,Start,End],
  SRI_PROJECT(Proj,ProjNum,Start,End))                    (EQ3)
```

since this will allow us to infer (by looking in the database) that the predicate `project1` holds of `clare`

Two translation steps now produce the desired transformation; in each, the schemas (2.8) and (2.7) are used in turn to reduce to the base case of expanding an atom. Remember that schema (2.8) replaces variables with unique constants; when displaying the results of such a transformation, we will consistently write `X*` to symbolize the new constant associated with the variable `X`.

The first atom to be expanded is `woman1(C*)`, and the corresponding conjunctive context is `work_on1(E*, C*,clare)`. `woman1(C*)` unifies with the first conjunct on the left-hand side of the `(EQ1)`, making its conditions `employee1(C*)`. Using the Horn-clause meaning postulate `(HC1)`, this can be reduced to `SRI_PROJECT_MEMBER(Project,C*)`. Note that `C*` in this formula is a constant, while `Project` is a variable. This new goal can be reduced again, by applying the rule `(EQ2)` as a backwards Horn-clause, to

```
and(work_on1(Event,C*,Project),
    project1(Project))),
```

The first conjunct can be proved from the assumptions, instantiating `Project` to `clare`; the second conjunct can then be derived from the normal Horn-clause reading of rule `(EQ3)`, together with the fact that `clare` is listed as a project in the database. This completes the reasoning that justifies expanding `woman1(C)` in the context of this query, to

```
exists([HasCar],
  and(SRI_EMPLOYEE(C,w,HasCar)))
```

The second translation step is similar. The atom to be expanded here is `work_on1(E*, C*,clare)`, and the conjunctive context is `woman1(C*)`. Now the rule `(EQ2)` can be used; its conditions after unification with the first conjunct in the LHS are `project1(clare)`, the validity of which follows from another application of `(EQ3)`. So `work_on1(E,C,clare)` can be expanded to `SRI_PROJECT_MEMBER(clare,C)`, giving the desired result.

2.4.3 Existential quantification

We finally consider cases where conditional equivalences contain existential quantifiers on their left-hand sides. It will be sufficient to restrict ourselves to unconditional equivalences where the LHS is an existentially quantified atomic formula, that is equivalences of the form

$$(2.12)(\exists\vec{x}.P_1) \equiv P'$$

since an expression of the form

$$(2.13) Conds \rightarrow (\exists\vec{x}.P_1 \wedge P_2 \wedge P_3 \ldots) \equiv P'$$

can be rewritten, introducing the new predicate P_{123}, as the two equivalences

$$(2.14)\forall\vec{x}.(Conds \rightarrow (P_1 \wedge P_2 \wedge P_3 \ldots) \equiv P_{123}(\vec{x}))$$

(of type 2.3), and

$$(2.15)(\exists\vec{x}.P_{123}) \equiv P'$$

(of type 2.12).

Equivalence of type 2.12 become important in connection with the so-called "Doctor on Board" problem (Perrault and Grosz, 1988), which in our domain can be illustrated by a query like (S3),

(S3) Does Mary have a car?

This receives the LF

```
exists([Event,Car],
       and(car1(Car),have1(Event,mary,Car))))          (LF3)
```

for which the intended database query is

```
exists([Sex],
       SRI_EMPLOYEE(mary,Sex,y))                       (LF4)
```

if Mary is listed as an employee; the relevant equivalence from the LDT could be

```
exists([Event,Car],
       and(employee1(Empl),
           and(car1(Car),
               have1(Event,Empl,Car))) <->
exists([Sex]
       SRI_EMPLOYEE(Empl,Sex,y))
```

which we rewrite as

```
and(employee1(Empl),
    and(car1(Car),
        have1(Event,Empl,Car))) <->
employee_has_car(Event,Empl,Car)                          (EQ4)

exists([Event,Car],
       employee_has_car(Event,Empl,Car)) <->
exists([Sex],
       SRI_EMPLOYEE(Empl,Sex,y))                          (EQ5)
```

(EQ4) and (EQ5) illustrate the coding trick described above; (EQ4) is of type 2.3, (EQ5) is of type 2.12, and the role of predicate P_{123} is played by employee_has_car.

We now describe briefly how (EQ4) and (EQ5) can be used to translate (LF3). If we assume that employee1(mary) can somehow be proved, we can apply (EQ4) to (LF3) to translate the sub-expression car1(Car), using the schemas 2.8, 2.7 and 2.6; the details are similar to those of the derivation in section 2.4.2, and the result is (LF5).

```
exists([Event,Car],
       and(employee_has_car(Event,mary,Car),
           have1(Event,mary,Car)))                         (LF5)
```

The second conjunct of (LF5) can be simplified away (see section 2.5.2); since one of the normal Horn-clause readings of (EQ4) is

```
have1(Event,Empl,Car) <- employee_has_car(Event,Empl,Car)
```

we have that have1(Event,mary,Car) is implied by its conjunctive context employee_has_car(Event, mary, Car) and can thus be removed. This reduces (LF5) to (LF6)

```
exists([Event,Car],
       employee_has_car(Event,mary,Car))                   (LF6)
```

after which (EQ5) can be applied directly, yielding (LF4) as required.

2.5 Simplification

AET, like many systems based on the idea of rewriting, tends to suffer from the problem that expressions can rapidly grow in size as they pass through successive stages of translation. The standard solution to the problem is to include a *simplifier*, a module which takes the output from a translation stage and attempts to reduce it to a more compact form. This section will describe CLARE's simplifier. Although the simplifier's primary use is in conjunction with the translator, it has turned out to provide a functionality that several other parts of the system utilise. In section 2.5.3 we describe how the simplifier is used in the processing of assertions; here, the problem is to take the conjunction of the TRL representations of several assertions, and combine them into a compact form.

The simplifier consists of two main pieces. The first, described in section 2.5.1 is a collection of standard logical rewriting techniques, none of which use inference, that have the common function of reducing expressions to a canonical form. Simplification can also be carried out by using the inference-based equivalential translation methods described earlier in this chapter, in which inference justifies the replacement of sub-expressions by other forms the equivalence of which is implied by the context. This second type of simplification is described in section 2.5.2.

2.5.1 Non-inferential simplification

The non-inferential simplification module consists of the set of rewriting methods listed below. Most of them are fairly obvious, though there are a few tricky points. We present the methods in the order in which they are applied.

Moving existential quantifiers outwards

This method essentially consists of two kinds of rewriting rule. The first simplifies an expression built out of conjunction and existential quantification operators into an equivalent form with a single existential operator on the outside binding all the variables; thus for example

```
and(exists([X],
      p(X)),
    exists([Y],
      q(Y)))
```

will get rewritten to

```
exists([X,Y],
  and(p(X),Q(Y)))
```

The other rewriting rule is applied when an existentially quantified form occurs on the LHS of an implication; in this case, the existentially quantified variables are moved upwards to become universally quantified with wide scope over the implication. The justification for this manoeuvre is the equivalence

$$((\exists x.P) \rightarrow Q)) \equiv \forall x.(P \rightarrow Q)$$

which is easily proved. The result of recursively applying the two rules is to give existential quantifiers as wide scope as possible, if necessary turning them into universals on the way.

Simplification of equalities

The next simplification step removes equalities from the expression when this is possible. The basic equivalences that licence this type of simplification are the following:

$$(\exists x.\exists y.P \wedge (x = y)) \equiv (\exists x.P[y/x])$$

$$(\forall x.\exists y.P \wedge (x = y)) \equiv (\forall x.P[y/x])$$

and

$$(\exists y.P \wedge (y = a)) \equiv P[y/a]$$

where a is a constant. The methods are implemented by recursively descending through the expression to find equality sub-forms, and in suitable cases unifying together their two arguments, replacing the equality with an occurrence of the identically true predicate. After this, a second pass removes the occurrences of $true$ and the vacuous and repeated quantifications introduced.

If the expression contains an equality whose arguments consist of non-unifiable constants, its value can be known to be identically false (assuming non-identity of reference of distinct constants). In this case, the simplifier replaces the equality with the form

```
mismatch(Arg1,Arg2)
```

Subsequent behaviour depends on the nature of the expression being manipulated. If it is a question, the `mismatch` form is later replaced with an occurrence of the identically false predicate, which will generally lead to a "No" answer being produced. If the expression however represents an assertion, the mismatch normally represents a presupposition failure. In this case, the interface handler will inform the user that the expression was translated to an identically false form due to the occurrence of the mismatch, and attempt to print the incompatible arguments in an informative way.

Simplification of ground sub-expressions

The final type of simplification involves ground sub-expressions, that is to say sub-expressions containing no variables. Since these can be assigned a truth-value irrespective of their context, the expression can be simplified by computing it and replacing by an occurrence of either $true$ or $false$. At present, the method is restricted in two ways: ground sub-expressions are only replaced if their predicates are declared to be executable relations (cf. section 2.3.2); also, only $true$ ground expressions are substituted.

2.5.2 Inferential simplification

We will now consider methods that use equivalential translation to carry out simplification; the notion of *conjunctive context*, defined in section 2.4.1, will play a key role. We can immediately describe one way to achieve this end. Suppose that F is a formula containing an occurrence of the subformula F', and suppose further that F' is implied by its conjunctive context. It follows that replacing F' with *true* in F will result in a formula equivalent with F. This gives a simple method for removing duplicated conjuncts and the like.

A similar, but more refined, form of simplification can also be performed, which exploits functional relationships between arguments in predicates. We start by re-examining our example (S1) from the beginning of the chapter, reproduced here for convenience.

(S1) List all payments made to BT during 1990.

In section 3.8, we will see that the logical form originally derived from (S1) contains after a few translation steps three separate instances of the `transaction` relation, one from each of the original linguistic predicates `payment1`, `make2` and `during1`. Assuming that the arguments of the `transaction` relation are

```
transaction(Payer,Transaction,Date,Payee,Amount)
```

`payment1(Payment)` expands, roughly speaking, to

```
transaction(_,Payment,_,_,_)
```

`make2(Event,Agent,Payment,Payee)` expands to

```
transaction(Agent,Payment/Event,_,Payee,_)
```

and `during1(Payment,Interval)` expands to

```
and(transaction(_,Payment,Date,_,_),
    time_during(Date,Interval))
```

The database query will conjoin all three instances together. It is clearly preferable, if possible, to merge them instead, yielding a composite predication

```
transaction(Agent,Payment/Event,Date,Payee,_)
```

The information that licences this step as a valid simplification is that `transaction` is a function from its second argument (the payment) to the remaining ones (the agent, the date, the payee and the amount); in other words, a given payment is made by a unique agent, on a unique date, to a unique payee, for a unique amount. The system allows the information to be entered as a "function" meaning postulate in the form

```
function(transaction(Payer,Transaction,Date,Payee,Amount),
         [Transaction] -> [Payer,Date,Payee,Amount])
```

which is treated as a concise notation for the conditional equivalence postulate

```
(transaction(Payer,Transaction,Date,Payee,Amount) <->
 and(Payer = Payer1,
     and(Date = Date1,
         and(Payee = Payee1,
             Amount = Amount1)))))  <-
 transaction(Payer1,Transaction,Date1,Payee1,Amount1)
```

It is thus possible to view "merging" simplification of this kind as just another instance of equivalential translation. In the current version of the system, the transformation process operates in a cycle, alternating normal translation followed by simplification using the same basic interpreter; simplification consists of functional "merging" followed by reduction of equalities where this is applicable.

2.5.3 Simplification for processing assertions

The simplification process also plays an important role in the processing of assertions. Consider, for example, what would happen to the pair of sentences (S6) - (S7) without simplification:

(S6) Clara is an employee who has a car.

(S7) Clara is a woman.

(S6) translates into the database form

```
exists([A,B],
       SRI_EMPLOYEE(clara,A,y))
```

(The second field in `SRI_EMPLOYEE` indicates sex, and the third whether or not the employee has a company car). This can then be put into Horn-clause form as

```
SRI_EMPLOYEE(clara,sk1,y)
```

and asserted into the Prolog database. Since Clara is now known to be an employee, (S7) will produce the unit clause

```
SRI_EMPLOYEE(clara,w,sk2)
```

The two clauses produced would contain all the information entered, but they could not be entered into a relational database as they stand; a normal database has no interpretation for the Skolem constants `sk1` and `sk2`. However, it is possible to use function information to merge them into a single record. The trick is to arrange things so that the system can when necessary recover the existentially quantified form from the Skolemized one; all assertions which contain Skolem constants are kept together in a "local cache". Simplification of assertions then proceeds according to the following sequence of steps:

1. Retrieve all assertions from the local cache.
2. Construct a formula A, which is their logical conjunction.
3. Let A_0 be A, and let $\{sk_1 \ldots sk_n\}$ be the Skolem constants in A. For $i = 1 \ldots n$, let x_i be a new variable, and let A_i be the formula $\exists x_i.A_{i-1}[sk_i/x_i]$, i.e. the result of replacing sk_i with x_i and quantifying existentially over it.
4. Perform normal function merging on A_n, and call the result A'.
5. Convert A' into Horn-clause form, and replace the result in the local cache.

In the example above, this works as follows. After (S6) and (S7) have been processed, the local cache contains the clauses

```
SRI_EMPLOYEE(clara,sk1,y)

SRI_EMPLOYEE(clara,w,sk2)
```

$A = A_0$ is then the formula

```
and(SRI_EMPLOYEE(clara,sk1,y)
    SRI_EMPLOYEE(clara,w,sk2))
```

and A_2 is

```
exists([X1,X2]
  and(SRI_EMPLOYEE(clara,X1,y)
      SRI_EMPLOYEE(clara,w,X2))
```

Since `SRI_EMPLOYEE` is declared functional on its first argument, the second conjunct is reduced to two equalities, giving the formula

```
exists([X1,X2]
  and(SRI_EMPLOYEE(clara,X1,y)
      and(X1 = w,
          y = X2))
```

which finally simplifies to A',

```
SRI_EMPLOYEE(clara,w,y)
```

a record without Skolem constants, which can be added to a normal relational database.

2.6 The Closed World Assumption

This section will briefly consider the question of how the Closed World Assumption fits into the picture we have constructed; to make the discussion concrete, we will concentrate on the case of Y-N questions. We start with a logical form, F_{ling}, for which we assume that we can find an effective translation F_{eval}, and assume further that it turns out that there is no proof of F_{eval}. Inability to prove F_{eval} will imply that it is false, since we have assumed the existence of a finite proof procedure; this in turn implies (if the abductive assumptions are granted) that F_{ling} is false, since the two are equivalent. With regard to the CWA, the interesting thing to consider is the part played by the database predicates that occur in F_{eval}. Knowledge of the database predicates is complete, since they have "trivial" semantics, only referring to the existence of database tuples and not directly to the rest of the world; they are closed by construction and the Closed World Assumption may safely be used on them. However, the predicates in F_{ling} are in general not closed. It is interesting to examine in more detail how the connection between the "closed" predicates in F_{eval} and the "open" predicates in F_{ling} is defined.

In practice, the key step in proving the equivalence between F_{ling} and F_{eval} is usually of the following form. There are two predicates $P(x, y)$ and $P_{rec}(n_x, n_y)$, whose intended semantics are, respectively, "x and y stand in relationship P" and "It is recorded in the database that objects named n_x and n_y stand in relationship P". Thus P_{rec} is closed, but P isn't. The two predicates are related by a domain axiom of the form

$$\begin{aligned} C(y) \rightarrow (P(x, y) \equiv \exists n_x, n_y . name_1(x, n_x) \\ \wedge name_2(y, n_y) \\ \wedge P_{rec}(n_x, n_y)) \end{aligned} \tag{2.16}$$

Here, each $name_i(x, n_x)$ $(i = 1, 2)$ is a predicate that relates an object x and the identifier n_x assigned to it according to naming convention i. (As pointed out above in section 2.2, it is important to take account of the fact that the same identifier can name different objects in different contexts). $C(y)$ defines a sufficient condition for $P(x, y)$ to have been recorded; for example, y might be a date and $C(y)$ could say that y is within the period for which records have been kept.

Now if $P(x,y)$ occurs in F_{ling} conjoined with some expression $D(y)$, where $D(y)$ can be proved to imply $C(y)$, then (2.16) and (2.7) can be invoked to justify replacing $P(x,y)$ in F_{ling} with $\exists n_x, n_y.name_1(x,n_x) \wedge name_2(y,n_y) \wedge P_{rec}(n_x,n_y)$ while maintaining equivalence. In this way, equivalential translation allows the CWA in effect to be made conditional on the context of use. Equivalence of type (2.16) are discussed further in section 3.5.10.

2.7 Translating to SQL

This section briefly describes the module responsible for synthesis of actual SQL queries. The conversion module takes as input a TRL form P, assumed to represent a question; it outputs a form P', such that P and P' are equivalent and as much as possible of P has been replaced by calls to the SQL interface. The interface is mediated through the predicate

```
trl_select(Vars,SQLQuery)
```

where `SQLQuery` is a term representing an SQL `SELECT` statement, and `Vars` is a list whose length is equal to the number of selected variables in `SQLQuery`. The intended semantics are that

```
trl_select(Vars,SQLQuery)
```

holds iff `Vars` are a row selected by `SQLQuery`. `SQLQuery` is a logical term representing the abstract SQL syntax; conversion to concrete SQL syntax is handled by the low-level SQL interface, and is straightforward. We now describe how the conversion process is carried out; we will illustrate with the extended example from section 3.8

(S1) List all payments made to BT during 1990.

This receives the final translated TRL representation (slightly simplified in the interests of readability)

```
forall([TransId,Date,DBDate,AMNum],
       impl(and(TRANS(TransId,DBDate,bt,AMNum)
                and(db_date_convert(Date,DBDate),
                    and(t_precedes(date([1990,1,1]),Date),
                        t_precedes(Date,date([1990,12,31])))))
            x([Id,DisplayT],
              and(execute(DisplayEv,
                          display([TransId,DBDate,bt,AMNum]),
                          clare,
                          DisplayT),
                  t_precedes(<Now>,DisplayT)))))
```

which can be glossed as

> "Find all `TRANS` tuples with `trn_id` field `TransId`, `cheque_date` field `DBDate`, `payee` field `bt` and `amount` field `Amt`, such that `DBDate` represents a date after 1/1/90 and before 31/12/90; and for each one display the list `[TransId, DBDate, bt, Amt]`."

Taking the conclusion first, the translation to SQL form produced is

```
forall([TransId,DBDate,AMNum],
       impl(trl_select([TransId,DBDate,AMNum],
                        SELECT([t_1.trn_id,t_1.amount,t_1.cheque_date],
                               FROM([alias(TRANS,t_1)]),
                               WHERE([t_1.payee=bt,
                                      sql_date_=<(1-JAN-90,
                                                  t_1.cheque_date)),
                                      sql_date_=<(t_1.cheque_date,
                                                  31-DEC-90))]))),
            x([Id,DisplayT],
              and(execute(DisplayEv,
                          display([TransId,DBDate,bt,AMNum]),
                          clare,
                          DisplayT),
                  t_precedes(<Now>,DisplayT)))))
```

Here, the abstract SQL syntax is a hopefully transparent representation of the query whose concrete syntax will be

```
"SELECT DISTINCT t_1.trn_id , t_1.cheque_date , t_1.amount
 FROM TRANS t_1
 WHERE t_1.payee = 'bt'
 AND '1-JAN-90' <= t_1.cheque_date
 AND t_1.cheque_date <= '31-DEC-90"
```

We now explain in more detail how this result can be produced. Processing traces the following series of top-level steps:

1. Translate representation-independent evaluable predicates such as `t_precedes` into database access language dependent primitives using AET.

2. Insert database column names into database predicates.

3. Make column names unique by associating a "relation alias" with each occurrence of a database predicate.

4. Group together conjunctions of goals suitable for turning into `SELECT` statements. Goals suitable for inclusion are goals representing database lookups and goals that map onto conditions in `WHEN` clauses. For each such conjunction, do the following:
   (a) Extract the variables that are to be selected.
   (b) Replace variables by column descriptions.
   (c) Extract the variables that are constrained to have fixed values, and add these as suitable equalities to the `WHEN` clause.
   (d) If more than one relation instance is referenced, add equalities to the `WHEN` clause to represent the join.
5. Simplify the result.

We use the example to illustrate. The first step is to use AET to translate the occurrences of `t_precedes`. The relevant equivalence is

```
t_precedes(Date1,Date2) <->
exists([DBDate1,DBDate2],
  and(db_date_convert(Date1,DBDate1),
  and(db_date_convert(Date2,DBDate2),
      sql_date_=<(DBDate1,DBDate2))))
```

and applied twice; after some simplification (exploiting the fact that `db_date_convert` is a function from each of its arguments to the other one, see section 2.5.2), the result is

```
forall([TransId,Date,DBDate,AMNum],
       impl(and(TRANS(TransId,DBDate,bt,AMNum)
                and(db_date_convert(Date,DBDate),
                    and(sql_date_=<(1-JAN-90,DBDate)
                        sql_date_=<(DBDate,31-DEC-90))))),
            x([Id,DisplayT],
              and(execute(DisplayEv,
                          display([TransId,DBDate,bt,Amt]),
                          clare,
                          DisplayT),
                  t_precedes(<Now>,DisplayT)))))
```

Note that `1-JAN-90` and `31-DEC-90` are atomic SQL date representations. The next two steps are fairly trivial in nature, and involve substituting SQL column names in the `TRANS` relation and creating a unique relation alias `t_1` for it. (Since there is only one relation here, this step is not actually necessary, but we show it for completeness). The result is

```
forall([TransId,Date,DBDate,AMNum],
       impl(and(tuple(t_1,TRANS([trn_id=TransId,
                                 cheque_date=DBDate,
                                 payee=bt,
                                 amount=AMNum])),
                and(db_date_convert(Date,DBDate),
                    and(sql_date_=<(1-JAN-90,DBDate)
                        sql_date_=<(DBDate,31-DEC-90))))),
            x([Id,DisplayT],
              and(execute(DisplayEv,
                          display([TransId,DBDate,bt,Amt]),
                          clare,
                          DisplayT),
                   t_precedes(<Now>,DisplayT)))))
```

Conjunctions are now when possible turned into `trl_select` goals; the only suitable conjunction is

```
and(tuple(t_1,TRANS([trn_id=TransId,
                     cheque_date=DBDate,
                     payee=bt,
                     amount=AMNum])),
     and(sql_date_=<(1-JAN-90,DBDate)
         sql_date_=<(DBDate,31-DEC-90)))
```

Here, the database look-up goals are the singleton

```
{tuple(t_1,TRANS([trn_id=TransId,
                  cheque_date=DBDate,
                  payee=bt,
                  amount=AMNum]))}
```

and the initial `WHEN` clause goals are

```
{sql_date_=<(1-JAN-90,DBDate)
 sql_date_=<(DBDate,31-DEC-90))}
```

Consulting the database look-up goals, the variables to be selected are `TransId`, `DBDate` and `AMNum`, so they end up being the list of variables in the `trl_select` goal's first argument; then replacing the variables inside the `SELECT` with column descriptions, we replace `TransId` with `t_1.trn_id`, `DBDate` with `t_1.cheque_date` and `AMNum` with `t_1.amount`. The `WHEN` clause goals now become

```
{sql_date_=<(1-JAN-90,t_1.cheque_date)
 sql_date_=<(t_1.cheque_date,31-DEC-90))}
```

There is a single column, `payee`, constrained to have a fixed value, `bt`, so it adds another goal

```
t_1.payee=bt
```

to the `WHEN` clause. The final result is the `trl_select` goal

```
trl_select([TransId,DBDate,AMNum],
           SELECT([t_1.trn_id,t_1.amount,t_1.cheque_date],
                  FROM([alias(TRANS,t_1)]),
                  WHERE([t_1.payee=bt,
                         sql_date_=<(1-JAN-90,
                                     t_1.cheque_date)),
                         sql_date_=<(t_1.cheque_date,
                                     31-DEC-90))])))))
```

Replacing the conjunction with the `trl_select` goal, we reach the final form,

```
forall([TransId,DBDate,AMNum],
       impl(trl_select([TransId,DBDate,AMNum],
                        SELECT([t_1.trn_id,t_1.amount,t_1.cheque_date],
                               FROM([alias(TRANS,t_1)]),
                               WHERE([t_1.payee=bt,
                                      sql_date_=<(1-JAN-90,
                                                  t_1.cheque_date)),
                                      sql_date_=<(t_1.cheque_date,
                                                  31-DEC-90))]))),
            x([Id,DisplayT],
              and(execute(DisplayEv,
                          display([TransId,DBDate,bt,AMNum]),
                          clare,
                          DisplayT),
                  t_precedes(<Now>,DisplayT)))))
```

Chapter 3

Linguistic Domain Theories

3.1 Introduction

From the examples presented in the previous chapter, the reader will already have gained some impression of what a linguistic domain theory looks like. In this chapter, we will consider the subject in more detail. We describe the various types of information that compose an LDT, and a methodology that can be used to construct LDTs for interfacing to relational databases.

There are four main kinds of information in an LDT. The greater part of the theory is normally composed of conditional equivalences, which have already been discussed at length in chapter 2. Apart from these, there are Horn-clause formulas, declarations of functional relationships between arguments of predicates, and declarations of what assumptions are permissible. We will first review each of these briefly; later on in the chapter, we explain exactly how they fit into the methodology for constructing an LDT.

Horn-clauses Certain relationships in the LDT are inherently ones of implication rather than equivalence; in these cases, Horn-clauses, rather than equivalences, are appropriate. For example, it can be necessary to capture the fact that everyone who books time to a project is of necessity an employee. This cannot easily be made into an equivalence, since being an employee does not directly imply that one books time to any particular project. A suitable Horn-clause will be something of the approximate form

```
employee_Worker(Person) <-
  booking_to_project(Person,Project,NumberOfHours,Date)
```

Functional relationships Many relations have the property of being functional on some subset of their arguments; that is to say, selecting a particular given value for that subset determines a unique value for the remaining arguments. In practice, the most important case is that of *primary key* fields in database predicates. We shall later see that there are a number of ways in which information about functional relationships can prove very useful. Functional relationships are defined by declarations of the form

```
function(<Template>,<FunctionalArgs> -> <RemainingArgs>)
```

Most commonly, `<FunctionalArgs>` is a list consisting of a single argument. For example, if the first argument of the `TRANS` relation is an identifier that functions as a primary key, this can be captured by a declaration like

```
function(TRANS(TransId,ChequeNum,Date,Payee,Account,Amount),
         [TransId] -> [ChequeNum,Date,Payee,Account,Amount])
```

Assumption declarations Assumption declarations are used to control the abductive proof mechanism: it is only permissible to make an abductive assumption when this is explicitly licenced by an assumption declaration. An assumption declaration has the general form

```
assumable(<Goal>,<Cost>,<Justification>,<Type>,<Conditions>)
```

The intended semantics are that `<Goal>` may be assumed at cost `<Cost>` in an environment where `<Conditions>` hold. `<Type>` can be either `specialization`, `limitation` or `approximation`; this is explain further in sections 2.3.2 and 3.6. `<Justification>` is a tag which can be used to identify instances of use of this declaration. For example, if one wants to declare that an individual which occurs in the first, "payer" argument place of a `transaction` relation can be assumed without cost to be SRI, a suitable declaration would be

```
assumable(Payer = sri,
          0,
          all_transactions_referred_to_are_from_SRI,
          specialization,
          transaction(Payer,_,_,_,_,_,_))
```

The rest of the chapter will explain in detail how these various kinds of information can be used to build up an LDT; we will focus on the case of relational databases, but much of the methodology will carry over to other kinds of knowledge-base applications. We start in section 3.2 by giving an overview of what an LDT for a relational database will look like. In the following sections, we go into more detail about specific modelling problems.

3.2 LDTs for relational databases

This section gives an overview of what an LDT for a relational database looks like, based on our experience with the example LDT for the PRM domain. The PRM application, which is based on a real SRI projects and payments database, contains about 280 equivalences, 60 Horn clauses, 45 declarations of functional relationships, and 25 assumption declarations. The axioms are stratified into three distinct groups, listed in order of increasing specificity: this reflects a strategy which translates the original logical form predicates into predicates which are increasingly closer to the database. We refer to the groups as *general*, *domain-specific* and *database-specific*, and describe each in turn.

3.2.1 General axioms

About 175 of the axioms fall into the first group, which we refer to collectively as "the general LDT". About 45 of them relate the words used in the utterance situation ("show", "question", "answer" etc.) to the primitive executable relations. Thus for example a typical equivalence of this type would be the one that translates the ditransitive verb "show" ("Show me the payments") into an `action` of type `display`:

```
show2(Event,Agent,X,Audience) <->
exists([Time],
  action(Event,display(X),Agent,Audience,Time))
```

Of the remaining "general" axioms, about 100 translate basic word senses, like temporal prepositions and size expressions, into a small set of "primitive predicates" that associate events and objects with characteristic attributes such as time of occurrence and magnitude. Thus for example the temporal preposition "during" is translated by the following equivalence:

```
during1(E1,E2) <->
exists([T1,T2,Gran1,Gran2],
  and(associated_time(E1,T1,Gran1),
  and(associated_time(E2,T2,Gran2),
      time_during([T1,Gran1],[T2,Gran2]))))
```

This expresses the fact that an event `E1` can be regarded as being "during" another event `E2` if the time associated with the first event is contained in the time associated with the second one. The arguments `Gran1` and `Gran2` express the conceptual level of granularity of the events `E1` and `E2`; at present they can be either `days` (appropriate for database events) or `seconds` (appropriate for utterance situation events). Note that the definition of the `associated_time` predicate is provided by domain-specific axioms; the equivalence performs the domain-independent function of translating the linguistic predicate `during1` into a more convenient form expressed in terms of the general predicates `associated_time` and `time_during`.

The final set of about 30 general rules deal with translations of predicates like `time_during`, which relate times; the complications are caused by the fact that "times" can be either points or intervals, and can also be expressed at different levels of granularity. The temporal axioms translate all the temporal predicates into a uniform representation in terms of a single predicates, `time_point_precedes` that compares time-points with regard to temporal precedence.

3.2.2 Domain-specific axioms

The axioms of the second group, which for the PRM domain are slightly less numerous than the first, are domain-specific but not database-specific. They translate predicates representing senses of domain words ("project", "transaction", "deliverable" etc.) and semantic primitive predicates like `associated_time` into a set of target predicates which we refer to as the "conceptual" predicates: the intention is that the conceptual predicates should be roughly the ones that would occur in a conceptual database schema for the domain. This means in practice that each distinct type of object and event will have its own conceptual predicate, whose arguments correspond to the various entities associated with the object. "Conceptual predicate" has of course much in common with standard AI notions like "Minsky frame". The PRM domain has about ten conceptual predicates, corresponding to concepts like "project", "transaction" and "deliverable"; a conceptual predicate typically has between three and seven arguments. For example, the conceptual predicate for "transaction" has arguments corresponding to the transaction itself, the payer, the payee, the cheque used, the amount, the account on which the payment was made and the date.

Translation of the "primitive predicates" is normally dependent on the nature of their arguments, and the corresponding equivalences are thus either conditional or have complex left-hand sides. For example, the time associated with a payment is the transaction date, but that associated with a project is the interval spanned by the start- and end-dates. The equivalences that express this are

```
and(associated_time(Trans,Date,Granularity),
    transaction1(Trans)) <->
exists([Payer,Cheque,Payee,Acc,Amt],
   and(transaction(Payer,Trans,Cheque,Date,
                   Payee,Acc,Amt),
       Granularity = days))

and(associated_time(Project,Interval,Gran),
    project1(Project)) <->
```

```
exists([StartDate,EndDate],
    and(project(Org,Project,Account,
                StartDate,EndDate),
        Interval = interval(StartDate,EndDate))
```

3.2.3 Database-specific axioms

The third group of axioms relate the "conceptual predicates" to the associated concrete database predicates. They capture specific coding peculiarities which relate database-objects and their associated real-world counterparts, which as explained above in section 2.6 includes contingent limitations on the availability of recorded data. The database-specific group is small, consisting of about 25 axioms; these consist of one equivalence linking each conceptual predicate to its concrete counterpart, together with some axioms that define the "contingent limitations" when necessary. For example, the axioms for the `book_time_to_project` relation (slightly simplified) are

```
(exists([Ev],
   book_time_to_project(Ev,Emp,Amt,D,Acc) <->
 exists([EmpId,T,TNum,DateNum,AC],
   and(SRI_TIMESHEET(EmpId,Amt,DateNum,AccId),
       and(employee_id(Emp,EmpId),
           and(date_id(D,DateNum),
               account_id(Acc,AccId)))))) <-
  timesheet_data_available(D,Acc)

timesheet_data_available(D,<CLAREAccount>) <-
  and(time_point_precedes(nonstrict,(<CLAREStartDate>,D),
      time_point_precedes(nonstrict,(D,<CLARELastRecordedDate>))

timesheet_data_available(D,<SLTAccount>) <-
  and(time_point_precedes(nonstrict,(<SLTStartDate>,D),
      time_point_precedes(nonstrict,(D,<SLTLastRecordedDate>))
...
```

where `<CLAREAccount>` is a ground term representing the CLARE account, and so on. Here `employee_id`, `date_id` and `account_id` are "naming predicates" which relate objects and their identifiers in the PRM database. `timesheet_data_available` is a predicate which defines for each account the period for which timesheet records are complete.

Most of the permissible assumptions that can be made are related to the database-specific axioms, and are of one of two types. The first type, a "specialization" (see section 2.3.2), permits the assumption that arguments like "employer" in the `project` conceptual relation or "payer" in the `transaction` relation are filled by `SRI` unless explicitly filled by something else. We illustrate with the `project` relation: the equivalence that translates it into the associated database predicate is schematically of the form

```
(project(Org,Proj,Acc,StartDate,EndDate) <->
 ... <Right-hand side> ...) <-
  Org = organization(SRI)
```

and there is an assumption declaration (see section 2.3.2)

```
assumable(Org = organization(SRI),
          0,
          all_projects_referred_to_are_at_SRI,
          specialization,
          project(Org,_,_,_,_))
```

The content of the declaration is that it is permissible to assume that a term X is equal to `organization(SRI)` if it occurs in a conjunctive context where X is the first argument in an occurrence of the `project` relation. The atom `all_projects_referred_to_are_at_SRI`, the "justification", is the tag used to identify the assumption to the user.

The second type of assumption, a "limitation", is related to temporal incompleteness of database records: here, the declaration is to the effect that it is permissible to assume that data is within the period for which records have been kept unless the conjunctive context implies the contrary. We illustrate with the `timesheet_data_available` predicate above. There is one assumption declaration for each account. The declaration for the CLARE account is for example

```
assumable(
  timesheet_data_available(D,<CLAREAcc>)
  15,
  time_was_booked_after_<CLAREStartDate>_and_\\
                before_<CLARELastRecordedDate>,
  limitation,
  book_time_to_project(_,_,_,D,<CLAREAcc>))
```

Use of the declaration in cases where it is contradicted by its context is prohibited by "negated goal" rules (see section 2.3.2)

```
neg(timesheet_data_available(D,<CLAREAcc>)) <-
  time_point_precedes(strict,D,<CLAREStartDate>)

neg(timesheet_data_available(D,<CLAREAcc>)) <-
  time_point_precedes(strict,<CLARELastRecordedDate>,D)
```

1. For each database relation, construct an associated "conceptual relation". Declare each database and conceptual relation as such, and declare the SQL column names associated with each database relation. Declare any fields in conceptual relations which can only be filled by one of a finite set of code values, together with a list of those values. (Section 3.3).

2. For all of the conceptual and database relations in which some fields are functions of other fields, write appropriate `function` declarations. (Section 3.4).

3. For each word sense that needs to be related to the database, write the necessary equivalence (`equiv`) meaning postulates. (Section 3.5.) These equivalences, which can be thought of as translation rules, are the heart of the interfacing process. It will usually be necessary to add a number of Horn-clauses to supplement the `equiv` rules. (Section 3.7).

4. Declare any assumptions used in the equivalences with `assumable` declarations. (Section 3.6)

Figure 3.1: Steps for constructing a domain model for a database interface

The remaining sections in this chapter describe in more detail the methodology we have developed for constructing LDTs for interfacing to relational databases. Much of the material is taken directly from the corresponding chapter in the CLARE software manual, though we have attempted where possible to raise the level of abstraction by removing references to essentially implementation-level constructs.

The basic procedure for construction of an LDT for a relational database is shown in Figure 3.1, which breaks it down into a sequence of steps. More detailed descriptions of the work involved in each step are given under the sections indicated in the figure.

3.3 Declaring database and conceptual relations

The first step is to identify the tables used in the actual SQL database. Each table will correspond to two logical predicates: the *database relation*, R_{db}, and the associated *conceptual relation*, R_{conc}. R_{db} is a predicate which exactly corresponds to the database table. The arguments of R_{db} are *TRL database objects*, which is to say TRL terms standing in direct correspondence to the actual contents of the database tuple fields. (See also section 2.2). Since the database relations and database objects are normally inconvenient to work with directly, the domain modelling procedure requires the definition of an associated *conceptual predicate* R_{conc} for each database predicate R_{db}; the connection between language and database will always go through the conceptual predicates. There are several important differences between R_{db} and R_{conc}. (The precise relationship between them is discussed later, in section 3.5, especially 3.5.10). The only one that we will consider for the moment is however that R_{conc} will normally have more arguments than R_{db}. There are two reasons why it may be necessary to add arguments to R_{db}.

Firstly, it may be the case that the conceptual relation corresponding to R_{db} has one or more "implicit" arguments, which can be referred to in language, but which would always have the same value in the database. For example, the `TRANS` relation has an implicit "payer" argument, corresponding to the agent who performs the transaction. Since the database only contains payments made by SRI, there is no reason to include this as a database column; however, language permits phrases like "payments from SRI", implying that the "payer" exists at least at the level of conceptual modelling.

The second reason for including extra arguments in R_{conc} is related to the standard database concept of "primary key". Most database relations will have a primary key field: that is to say, for any given constant, there is guaranteed to be at most one record where the primary key has that value. In the case of the `TRANS` relation, the `transaction_id` field is the natural choice as the primary key. However, some relations will lack a primary key. For example, the `SRI_PROJECT_-MEMBER` relation has two fields, `employee_name` and `proj_name`, neither of which can be primary; a project normally has more than one member, and most people work on more than one project. For a number of reasons, it is usually desirable for conceptual relations to have a primary key argument; if there is none in the database relation R_{db}, it is normal to add one R_{conc}. Thus the conceptual relation `works_on_project` (corresponding to `SRI_PROJECT_MEMBER`) has an artificial third argument to give it a primary key. This can be thought of as the "event" of the person's working on the project.

3.4 Declaring functional relationships

As described in the preceding section, database relationships usually tend to be functions on at least one of their arguments (those that correspond to primary keys); conceptual relations, should always have this property. The translation mechanism is able to make use of information about functional relationships in several ways (see e.g. 2.5.2). Functional relations are defined by declarations of the form

```
function(<Template>,<From> -> <To>)
```

This is to be read as stating that in <Template>, an assignation of values to the variables making up the list <From> uniquely determines the values of the variables in the list <To>.

Function declarations should be supplied both for database and conceptual predicates: for example, the appropriate declarations for TRANS and transaction are

```
function(TRANS(Event,C,D,Payee,AC,AM),
         ([Event] -> [C,D,Payee,AC,AM]))
function(TRANS(Event,C,D,Payee,AC,AM),
         ([C] -> [Event,D,Payee,AC,AM]))

function(transaction(Agent,Event,C,D,Payee,AC,AM),
         ([Event] -> [Agent,C,D,Payee,AC,AM]))
function(transaction(Agent,Event,C,D,Payee,AC,AM),
         ([C] -> [Agent,Event,D,Payee,AC,AM]))
```

Either the transaction ID or the cheque number could be used as a primary key, and hence there are two functional relationships.

3.5 Equivalence rules

The major part of the information needed to describe the domain model is expressed using equivalence rules. These come in two forms: conditional equivalences and unconditional equivalences. The abstract syntax we use for unconditional equivalences is

```
<LHS> <-> <RHS>
```

For conditional equivalences, we write

```
(<LHS> <-> <RHS>) <- Conds
```

Here <LHS>, <LHS> and Conds are TRL expressions, which may not use logical operators other than conjunction (and) and existential quantification (exists). The declarative semantics of equivalence rules are extensively discussed in chapter 2.

Writing equivalences is the non-trivial part of constructing the domain model. In the following subsections, we will describe in more detail how this is done. The main steps are summarized in figure 3.2.

Illustrative examples will be taken from the PRM application. The conceptual predicates we will use, and their corresponding database relations, are the following:

- Transactions

1. For each conceptual predicate R_{conc}, collect together the word-senses and other predicates that are directly associated with R_{conc} (Section 3.5.1).

2. For each conceptual predicate R_{conc}, write `equiv` rules that relate the predicates associated with R_{conc} to the conceptual predicate. We have subdivided this step into several sections (3.5.2 – 3.5.7) for different types of predicate.

3. Collect predicates that map on to expressions which include several database predicates, and write `equiv` rules to map them onto the corresponding conceptual predicates (3.5.9).

4. For each conceptual predicate R_{conc}, write `equiv` rules that relate R_{conc} to its associated database predicate. (Section 3.5.10). This often involves conditions which should be assumed rather than proved.

5. Write `assumable` declarations for the "assumable" predicates introduced in the last step. (Section 3.6).

Figure 3.2: Steps for writing equivalences

```
transaction(Payer,TransId,ChequeId,Date,Payee,Acc,Amt)
```

```
TRANS(TransId,ChequeId,Date,Payee,Acc,Amt)
```

- Projects

```
project(Organization,ProjectName,ProjectNum,StartDate,EndDate)
```

```
SRI_PROJECT(ProjectName,ProjectNum,StartDate,EndDate)
```

- Payees (classified as men, women, companies or universities)

```
payee(Type,Payee)
```

```
PAYEE(Type,Payee)
```

- Employees

```
employee(Employer,Employee,Sex,HasCar),
```

```
SRI_EMPLOYEE(Employee,Sex,HasCar),
```

- Bookings of time to projects

```
booking_to_project(BookingEvent,Person,Time,Date,Project)

BOOKING_TO_PROJECT(Person,Time,Date,Project)
```

We will assume that the LDT is built on top of the general LDT, described in the section 3.2.1. The general LDT supplies in particular axioms that translate linguistic predicates having to do with *time* and *quantity* into a more uniform representation. This is explained in more detail in sections 3.2.1 and 3.5.6.

3.5.1 Finding predicates associated with conceptual predicates

The first step is to collect, for each conceptual predicate R_{conc}, the predicates occurring in untranslated TRL that are directly associated with it. It is difficult to give an exhaustive list of the ways in which a linguistic predicate can be associated with R_{conc}; the following are the most common cases.

1. Nouns and adjectives describing objects that fill a single argument slot in R_{conc}. For example, the word-sense predicates `transaction_Event/1` and `payment_PayingEvent/1` describe the object filling the `TransId` argument slot in `transaction`, while `cheque_Document/1` describes objects filling the `ChequeId` one. (See also section 3.5.2). The appropriateness of these predicates can often be conditional on the value of another slot being filled by a suitable code; for example, the predicate `man_MalePerson/1` maps onto the `Employee` field of `employee` if the `Sex` field is filled by an `m`. This is explained further in section 3.5.7.

2. Prepositions describing the relationship between pairs of argument slots in R_{conc}, or between an argument slot in R_{conc} and some other type of object. For example, `for` relates the `ChequeId` and `Amount` slots in `transaction` (one can say "cheque **for** over 200 pounds"), while `to` relates the `TransId` and `Payee` slots ("payment **to** Maxwells"). See also section 3.5.3. Note that many preposition senses, in particular temporal ones, are translated by rules in the general LDT.

3. Verbs that describe the relationship between an event and one or more argument slots in R_{conc}. (See also section 3.5.4). Normally, the sense entry for a verb will relate an event, the object denoted by the verb's subject, and zero or more other arguments. For example, the predicate corresponding to the ditransitive sense of "pay" ("SRI paid Maxwells 100 pounds") is `pay_GiveTo/3`. Here, the second argument maps into the `Payer` slot, the third argument (the object) maps onto the `Amount` slot in `transaction`, and the fourth argument maps to the `Payee` slot. The first argument (the

paying event), is regarded as being equal to the transaction. It is in general a natural idea to map events onto primary keys, since an event will often uniquely determine the objects participating in it.

4. Relational nouns that describe the relationship between pairs of argument slots in R_{conc}. These are similar to prepositions. For example, the relational predicate deriving from the compound noun *project number* is `Project_Number_Account_Number_For_ProjectOf`, and this relates the `ProjectName` and `ProjectNum` slots in the `project` relation. See also section 3.5.5.

5. Temporal relations between slots. These arise when there is an argument slot in R_{conc} that associates a time with the object filling another argument slot. The relation in question can be `associated_time`, i.e. time of occurrence (e.g. the relation between the `TransId` and `Date` slots in `transaction` relation); `associated_start_time`, i.e. time of starting, (e.g. the relation between the `Project` and `StartDate` slots in `project`); or `associated_end_time`, i.e. time of ending. See section 3.5.6.

6. The `associated_size` relation; this arises when one slot in R_{conc} describes the size or magnitude of another. For example, the `Amount` slot in `trans` is the size of the `TransId` expressed as a quantity of `pounds_sterling`. See section 3.5.6.

3.5.2 Common noun describing a conceptual slot

In this and the following sections, we will describe how to write the equivalences that translate the linguistic and primitive predicates into conceptual predicates. For convenience of reference, we will divide up the material by different types of linguistic predicate. We begin with the simplest case: a common noun which describes a conceptual slot. (We will use the term "conceptual slot" as a handy abbreviation for "argument of a conceptual predicate"). In the following example, the rule says that an object which fills the third argument place in the conceptual `transaction` relation is a `cheque_Document`:

```
cheque_Document(Cheque) <->
exists([Payer,Trans,Date,Payee,Acc,Amt],
  transaction(Payer,Trans,Cheque,Date,Payee,Acc,Amt))
```

Note that the variable `Cheque` is implicitly universally quantified, but the remaining variables are existentially quantified on the RHS of the equivalence.

3.5.3 Preposition describing relationship between slots

Equivalences that translate preposition senses are also straight-forward. The following are the equivalences that cover "cheque for amount" ("cheques for more than £200") and "cheque to payee" ("cheque to Maxwells").

```
and(for(Cheque,Amt),
    cheque_Document(Cheque)) <->
exists([Payer,Trans,Date,Payee,Acc,Amt],
  transaction(Payer,Trans,Cheque,Date,Payee,Acc,Amt))

and(to(Cheque,Payee),
    cheque_Document(Cheque)) <->
exists([Payer,Trans,Date,Payee,Acc,Amt],
  transaction(Payer,Trans,Cheque,Date,Payee,Acc,Amt))
```

It can be helpful to think of the first atom in the LHS in a rule of this kind (the "head") as a pattern to be matched; the remaining atoms on the LHS are conditions of applicability, while the RHS is the result of applying the rule. Thus in the first rule, the pattern to be matched is `for(Cheque, Amount)`, and the rule is applicable if the context implies that `Cheque` satisfies the predicate `cheque_Document`.

It may sometimes be necessary to use more than one condition, as for example in the following rule, which would be used for an expression like "payments on account 8468". Here, one must specify both that the first argument is a cheque, and that the second argument is an account, thus:

```
and(on(Cheque,Acc),
and(cheque_Document(Cheque),
    account_ChargeNumber(Acc))) <->
exists([Payer,Trans,Date,Payee,Acc,Amt],
  transaction(Payer,Trans,Cheque,Date,Payee,Acc,Amt))
```

3.5.4 Verb describing relationship between slots

Rules for translating verb senses are similar to those for translating preposition senses. The following, for example, is the rule that covers "pay money to", as in "SRI paid £120 to Maxwells".

```
and(pay_GiveTo(Trans,Payer,Amt,Payee),
    money_AbstractCredit(Amt)) <->
exists([Payer,Date,Acc],
  transaction(Payer,Trans,Cheque,Date,Payee,Acc,Amt))
```

Note that the "paying event" is regarded as being the same as the transaction, and that the agent of the paying event becomes the `Payer` in the conceptual `transaction` relation.

Support verb constructions like "make a payment" can be treated in the same way. The next example shows the relevant equivalence:

```
and(make_Commit(MakeEvent,Payer,Trans),
    payment_PayingEvent(Trans)) <->
exists([Cheque,Date,Payee,Acc,Amt],
  and(transaction(Payer,Trans,Cheque,Date,Payee,Acc,Amt)
      MakeEvent = Trans))
```

Here the "making" event is once again regarded as being the same as the payment. In cases like this, where two variables from the LHS (`MakeEvent` and `Trans`) are being mapped onto a single event on the RHS, it is important to write the rule in the way shown, with an equality on the right. It is easy to show that the following variant

```
and(make_Commit(Trans,Payer,Trans),
    payment_PayingEvent(Trans)) <->
exists([Cheque,Date,Payee,Acc,Amt],
  transaction(Payer,Trans,Cheque,Date,Payee,Acc,Amt))
```

is not equivalent. The easiest way to convince oneself of the truth of this statement (which may seem counter-intuitive to readers used to Horn-clauses) is to consider the pair of formulas

```
p(X,Y) <-> and(q(X),X = Y)
```

and

```
p(X,X) <-> q(X)
```

The first of them means that `p(a,b)` implies `q(a)`, but not the second.

3.5.5 Relational nouns

Rules for translating relational noun senses are of a form similar to those for prepositions and transitive verbs. For example, the rule for the relational sense of the compound noun *project number* (e.g. "CLAM-BAKE's project number") is

```
project_number_AccountNoForProjectOf(Number,Project),
exists([Org,Start,End,Account],
  and(project(Org,Project,Account,Start,End),
      numbered_object(Account,account_ChargeNumber,Number)))
```

Note the use of the predicate `numbered_object` on the RHS. This expresses the fact that `Account` is the numbered object of type `account_ChargeNumber` with number `Number`, i.e. the object referred to by the English expression "account `Number`". See also section 3.5.10.

3.5.6 Primitive predicates

If a relation has a field that can be interpreted as associating a date or time with the tuple, it is possible to use temporal expressions, involving words like "when", "during", "after", "start" and "end". Equivalences in the general LDT (see section 3.2.1) should translate the linguistic predicates deriving from expressions like these into forms which use one of the "primitive predicates" `associated_time`, `associated_start_time` and `associated_end_time`. Each of these is a three-place predicate, whose arguments are, in order

- the object (which will often be an event)
- the associated date or time
- the "granularity" of the date or time, which at present must be either `days` or `seconds`

Similarly, some relations may have a field that can be interpreted as denoting the "size" or "magnitude" of one of the objects, and can be used to assign a meaning to expressions that compare objects with reference to their extent; expression of this form include comparatives ("larger", "smaller") superlatives ("largest", "smallest"), and some prepositions like "over" ("over £250"). All of these are translated by equivalences in the general LDT into a form expressed in terms of the primitive predicate `associated_size`.

Domain-dependent equivalences must then be provided to translate the primitive predicates in the contexts provided by the domain word-senses: the RHS of these equivalences will usually contain conceptual predicates. The following rule, for example, translates the `associated_time` predicate in a context where its first argument is a transaction; it states that the fourth field of a `transaction` relation is the time (at the level of granularity of days) associated with the transaction.

```
and(associated_time(Trans,Date,Granularity),
    transaction_Event(Trans)) <->
exists([Payer,Cheque,Payee,Acc,Amt],
  and(transaction(Payer,Trans,Cheque,Date,Payee,Acc,Amt),
      Granularity = days))
```

Similarly, some relations, representing events that occur over a period of time, may contain fields denoting the start and end of this period. The following pair of rules state that the start and end dates of a project are encoded in the fourth and fifth fields of the `project` relationship respectively.

```
and(associated_start_time(Project,Start_date,Granularity),
    project_Activity(Project)) <->
exists([Org,ProjNum,End_date],
  and(project(Org,Project,ProjNum,Start_date,End_date),
```

```
      Granularity = days))

and(associated_end_time(Project,End_date,Granularity),
    project_Activity(Project)) <->
exists([Org,ProjNum,Start_date],
  and(project(Org,Project,ProjNum,Start_date,End_date),
      Granularity = days))
```

The `associated_size` primitive predicate takes two arguments: the first is the object in question, and the second is its "size", which should be a number. Finding a suitable number will often involve including an equality on the RHS, which extracts the number from a term representing an amount. In the following example, which defines the "size" of a cheque, the extraction is performed using the `amount_object` predicate, defined in the general LDT.

```
and(associated_size(Cheque,N),
    cheque_Document(Cheque)) <->
exists([Payer,Trans,Date,Payee,Acc,Amt],
  and(transaction(Payer,Trans,Cheque,Date,Payee,Acc,Amt),
      amount_object(AM,sterling_unit,N)))
```

3.5.7 Expressions that map onto code values

The filler of a field in a database relationship is often a code, and an English expression will map onto a value, or set of values, for this code. Thus for example it might easily be the case that a word like "man" or "woman" ultimately maps onto a field being filled by an `m` or a `w`. Two examples follow. The first states that "man" when applied to an object known to be a payee, maps onto the first field of the `payee` relation being filled by an `m`. The second states that an overhead payment is a transaction where the "account" field is filled by a numbered object of type "account" whose number satisfies the predicate `overhead_code`.

```
and(man_MalePerson(Payee),
    payee_Recipient(Payee)) <->
payee(m,Payee)

overhead_payment_PaymentMadeOnOverhead(Trans) <->
exists([Payer,Cheque,Date,Payee,Acc,Amt,Num],
  and(transaction(Payer,Trans,Cheque,Date,Payee,Acc,Amt),
  and(numbered_object(Account,account_ChargeNumber,Num),
      overhead_code(Num))))
```

A further definition is then needed to specify the range of permitted values for overhead codes: here, they are defined to be exactly those numbers which are members of the set {`775, 675`}:

```
overhead_code(Num) <-> one_of(Num,[775,675])
```

An important special case is that of the code which indicates existence or non-existence of an object related to the filler of some other field in the relation. (See also section 2.4.3). In the PRM database, an example is provided by the `employee` relation, whose third argument is set to `y` if the employee in question has a company car. The following two rules (one for "have" and one for "car") encode this fact.

```
and(employee1(Empl),
and(car1(Car),
    have1(Event,Empl,Car))) <->
employee_has_car(Event,Empl,Car)

exists([Event,Car],
  employee_has_car(Event,Empl,Car)) <->
exists([Sex]
  SRI_EMPLOYEE(Empl,Sex,y))
```

3.5.8 "Display" predicates

The general LDT (see section 3.2.1) contains equivalences that translate words like "show" and "list" into a uniform representation using the predicates `executable_-action/6` and `display_format/2`. The second of these is a predicate which determines the appearance of the term printed to represent a database object: the intention is that `display_format(Object,Format)` should hold if `Format` is either a conceptual object, or a list containing one or more conceptual objects that can be used together as a representation of `Object`. For example, in the PRM domain it is sensible to print out a transaction as a list consisting of its transaction ID, the date of the payment, the payee, and the amount. This is captured in the following equivalence,

```
and(display_format(Trans,Format),
    transaction_Event(Trans)),
exists([Id,Payer,C,Date,Payee,AC,Amount],
  and(transaction(Payer,Trans,C,Date,Payee,AC,Amount),
  and(numbered_object(Trans,transaction_Event,Id),
      Format = [Id,Date,Payee,Amount])))
```

Normally, the domain implementor should provide one `display_format` definition for each type of object. Many objects will be "named objects" of some type (see section 3.5.10), whose names serve as adequate descriptions. This is for example the case for sponsors in the PRM domain. If this is so, an equivalence of the following kind is sufficient:

```
(display_format(Sponsor,Format) <->
 named_object(Sponsor,sponsor_ContributorOfFunding,Format)) <-
  sponsor_ContributorOfFunding(Sponsor).
```

3.5.9 Mapping on to combinations of conceptual predicates

Sometimes, a linguistic predicate will relate slots taken from more than one conceptual predicate. `Equiv` rules of this type are a little more complex than those which map onto a single conceptual predicate, which is why we have left this case for last; there are a couple of coding tricks that can be used, however, to simplify the structure of the rules.

We illustrate with the example of the preposition *on*, used to relate a payment to a project (e.g. "Which payments on CLAM-BAKE were over £1000?"). The rule must in effect join the `TRANS` and `SRI_PROJECT` relations through the `Account` field. The simplest way to do so is to write the following equivalence:

```
and(on(Trans,Account),
and(payment_PayingEvent(Trans),
    project_Activity(Project))) <->
exists([Org,Acc,StartDate,EndDate,
        Payer,Cheque,Date,Payee,Amt],
  and(project(Org,Project,Acc,StartDate,EndDate),
      transaction(Payer,Trans,Cheque,Date,Payee,Acc,Amt)))
```

The rule as presented, is, however, uneconomical, since it fails to exploit the general fact that "on a project" means generally "on that project's account"; what we have here is actually a simple case of the linguistic phenomenon of *metonymy*. The practical consequence is that a second equivalence will have to be added to deal with the expression "cheque on project". A better solution is to write a conditional equivalence which makes explicit the general relation between "on project" and "on account", thus:

```
(on(X,Project) <-> on(X,Account)) <-
  project(Org,Project,Account,Start,End).
```

3.5.10 Translating conceptual into database predicates

The final step when writing the equivalences is to add the rules which translate between the conceptual and database predicates. This is intimately connected with the assumption mechanism, and it will be helpful to review the ideas involved.

There are three main reasons for distinguishing between "conceptual predicates" and "data-base predicates". The first has already been discussed in section 3.3, and arises from the "extra arguments" in the conceptual predicate, like the `Organization` argument in `project`, or the `Payer` argument in `transaction`. The fact we wish to express here is that these arguments must have a single, fixed value, in order for the predicate to be capable of being translated. In the two examples quoted, the appropriate value is `SRI`. If SRI is mentioned explicitly in the query, (because some expression like "project at SRI" or "SRI payment" was used), the slots in question will already be filled by the appropriate value. More frequently, though, the value of the extra arguments will be left unspecified, and the value of the filler should simply be assumed to be SRI. The intended behaviour, then, is that an argument like `Organization` or `Payer` should be assumed to be equal to the appropriate value unless this is explicitly contradicted by the information in the original query.

The second, related, point is the need to be able to deal with cases where information in the database is systematically incomplete (see also section 2.6); the most important case is that of a relation which lists records kept over a specified period. For example, the `TRANS` relation only holds information about transactions carried out over the 18-month period starting in August 1989. Questions about payments carried out in 1988 or earlier can thus not be answered, and the desired behaviour is for query translation to fail with an appropriate warning. If the query makes no mention of time (e.g. "List all payments over £200 made on the WHIZ project"), then it is reasonable to assume that the user meant only to refer to payments within the period covered, as long as a message is printed to inform her that the assumption is being made. If the query explicitly refers to a time-span within the one for which records exist, translation is unconditionally valid.

The third point is the necessity of making a clear distinction between the types of object occurring respectively as arguments to conceptual and database predicates. (See also section 2.2). The intention is that the arguments to the database predicates should denote the *actual contents of the database records*, that is to say numbers, dates and strings. The arguments to the conceptual predicates, on the other hand, are intended to denote the *real-world objects* that the database records refer to. It may at first glance seem that making a distinction between these two types of object is a needless form of hair-splitting, but it is in fact primarily motivated by practical considerations. If the distinction is not made, problems can arise: in particular, it may easily be the case that the same database object, used in different database fields, can refer to different objects. For example, in the PRM domain there are some numbers which are both cheque and transaction IDs; however, the fact that a cheque ID happens to be identical with a transaction ID does *not* imply that the cheque and transaction are identical, or even that they are related. Because the problem with non-unique identifiers is so common and important, CLARE provides a special mechanism for dealing with it. Terms of the form

```
SF(c(x^[<PredicateName>,x],ID))
```

are treated by the system as referring to "the object of type <PredicateName>, with identifier ID". (An alternate notation, which we sometimes use as an abbreviation, is <PredicateName>#ID). Thus for example SF(c(x^[transaction_Event,x],123)) is the object that would be referred to by the English noun-phrase *transaction 123*. There is a predicate in the general LDT, named_object/3, which can be used to refer to terms of this type; thus for example

```
named_object(Trans,transaction_Event,123)
```

is equivalent to

```
Trans = SF(c(x^[transaction_Event,x],123))
```

All three types of functionality can be achieved in a uniform way, by suitably defining the respective truth-conditions of the conceptual and database predicates. The key idea can be summarized as follows:

- the conceptual predicate holds of the conceptual objects if the event in question has happened
- the database predicate holds of the corresponding database objects if the event is recorded in the database.

Since the types of argument to the two predicates are in general distinct, the equivalences used to define the relationship between the conceptual and database predicates will include a number of occurrences of *conversion predicates*, which relate database and conceptual objects. There are at present four such predicates in the general LDT: named_object/3 (defined above), sql_numbered_object/3, sql_number_convert/2, and sql_date_convert/2. The last three are defined as follows:

sql_numbered_object(Obj,Type,N) Obj is a named object of type Type whose identifier is a number, and N is the database representation of that number. This is the predicate used to relate a number field in a database record with the object that number refers to, in cases where the number is being used as an identifier.

sql_number_convert(Num,DBNum) Converts between numbers and database number objects (in the current implementation, Prolog atoms whose print-names are strings representing numbers).

sql_date_convert(Date,DBDate) Converts between dates and database date objects (in the current implementation, Prolog atoms whose print-names are strings representing dates in SQL format).

Looking at a concrete example should make these ideas clearer. The database relation TRANS/6 holds if an appropriate SRI transaction is recorded in the database. The corresponding conceptual predicate, transaction/7, holds if an appropriate transaction (not necessarily made by SRI) occurred. The predicates agree if two conditions are fulfilled: firstly, that the transaction was made by SRI, and secondly, that it was made within the recorded period. This can be expressed (removing a few details irrelevant to the present discussion) using the following equiv rule:

```
(and(transaction(Payer,Trans,Cheque,Date,Payee,Acc,Amt),
     transaction_data_available(Date)),
 exists([Id,CNum,AC,AM,AMNum,DBDate],
   and('TRANS'(Id,CNum,DBDate,Payee,AC,AM),
   and(sql_numbered_object(Trans,transaction_Event,Id),
   and(sql_numbered_object(Cheque,cheque_Document,CNum),
   and(sql_date_convert(Date,DBDate),
   and(sql_numbered_object(Account,account_ChargeNumber,AC),
   and(amount_object(Amount,sterling_unit,AMNum),
       sql_number_convert(AMNum,AM))))))))) <-
  Payer = sri.
```

This rule contains four kinds of predicate. Firstly, there are the conceptual predicate (transaction(...)), and the database predicate (TRANS(...)) that it is being mapped into. Secondly, there are the "conversion predicates" (sql_numbered_object etc.); as can be seen, they are used to link the arguments to the conceptual predicate (Trans, Cheque, Date etc.) with the corresponding arguments to the database predicate (Id, CNum, DBDate etc.) Finally, there are the predicates =/2 and transaction_data_available/1, which are used to express the contingent relationship between transaction and TRANS: namely, that the Payer has to be SRI, and that the transaction has to have been made during the specified period.

Here transaction_data_available/1 is a predicate supplied by the user, which holds if its argument is a date between 1/8/1989 and 31/3/1991. This is expressed by the Horn-clause

```
transaction_data_available(Date) <-
  time__precedes(nonstrict,date([1989,8,1]),Date),
  time_precedes(nonstrict,Date,date([1991,3,31]))
```

It is also necessary to add assumable declarations for both =/2 and transaction_data_available/1. This is explained in the next section.

3.6 Assumptions

The concept of *assumptions* has already been referred to several times; this section, will consider the subject in more detail.

It will frequently turn out that the equivalences, and in particular the equivalences which translate conceptual predicates into database predicates, are not strict, but rather depend on tacit assumptions related to the nature of the domain. For example, in the PRM domain it is normally reasonable to assume that the cheques, payments, projects and so on referred to in queries are all related in direct way to SRI, and that queries about all records refer to the time-periods for which they are available, unless this is explicitly contradicted by the query.

The implementor indicates which assumptions are permissible by including declarations of the form

```
assumable(<Goal>,<Cost>,<Justification>,<Type>,<Condition>)
```

This states that the goal `<Goal>` may be assumed, rather than being proved, at a cost `<Cost>` which should be a non-negative number, for the reason `<Justification>`, if the `<Condition>` can be inferred from the context in which the goal is being proved. `<Type>` should be one of the atoms `specialization`, `limitation` or `approximation`. The choice of assumption-type is related to the production of possible warning-messages if that assumption is used: the implementor should decide whether it is most appropriate to

- produce no warning message (`specialization`)
- warn that the answer may depend on incomplete knowledge (`limitation`)
- warn that the answer may depend on an incorrect assumption (`approximation`).

The total cost of a proof is defined to be the number of Horn-clauses used, plus the sum of the costs of all assumed rules; the search through the proof space is carried out using an iterated-deepening algorithm. By making the cost of an assumption sufficiently large, it is thus possible to ensure that proofs which do not use it are preferred to ones that do.

To give a better idea of how to write `assumable` declarations, we will continue the example started in the previous section. Recall that we have the equivalence

```
(and(transaction(Payer,Trans,Cheque,Date,Payee,Acc,Amt),
     transaction_data_available(Date)),
 exists([Id,CNum,AC,AM,AMNum,DBDate],
   and('TRANS'(Id,CNum,DBDate,Payee,AC,AM),
   and(sql_numbered_object(Trans,transaction_Event,Id),
   and(sql_numbered_object(Cheque,cheque_Document,CNum),
   and(sql_date_convert(Date,DBDate),
```

```
   and(sql_numbered_object(Account,account_ChargeNumber,AC),
   and(amount_object(Amount,sterling_unit,AMNum),
       sql_number_convert(AMNum,AM)))))))) <-
  Payer = sri.
```

linking the conceptual relation `transaction` and the database relation `TRANS`. We want `Payer = SRI` and `transaction_data_available(Date)` to be assumed in this context, as long as this is not explicitly inconsistent with the query. The relevant `assumable` statements are

```
assumable(
      Payer = SRI,
      0,
      all_transactions_referred_to_are_from_SRI,
      specialization,
      transaction(Payer,_,_,_,_,_,_))

assumable(
      transaction_data_available(Date),
      15,
      transactions_referred_to_made_between_1988/8/17_\\
             and_1991/4/1,
      limitation,
      transaction(_,_,_,Date,_,_,_))
```

In both cases, the condition for the validity of the rule is to be inferred from the context, in particular from the LHS of the equivalence. The zero cost on the first rule reflects the intuition that it is normally permissible to assume without further reasoning that payments are made by SRI. However, the second rule costs 15 units to apply, since it is quite frequently possible to infer from other conjuncts that the payment actually is within the known range. Since it is preferable to prove conditions rather than assume them, a charge is made for the assumption. The "justifications" are in both rules the atoms that will be used as tags when printing out the lists of assumptions made during translation.

The system saves each assumption made during the proof together with its context of use. After completing the translation process, an attempt is made to prove the *negation* of each assumption from its associated conditions. If this can be done, query processing is terminated, and the user is informed that the use of the relevant assumption is inappropriate in the given context. The prover does not provide full support for negation; it is however sufficient to provide Horn-clause meaning postulates of the form

```
neg(<Goal>) <- <Conditions>
```

which will allow normal Horn-clause proofs of goals of the form `neg(G)`. Since this makes `neg` a one-place predicate, it is important to realize that the prover has no way of relating proofs of `neg(G)` to proofs of `G`. Lack of this capability does not seem, however, to be a serious problem for the simple proofs carried out in this context.

The rules to define the negation of `=/2` and `time_precedes/3` are supplied in the general LDT; those for other assumed predicates may be added by the user. For example, the appropriate definitions for `transaction_data_available/1` are

```
neg(transaction_data_available(D)) <-
  neg(time_precedes(nonstrict,date([1989,8,17]),D))
negation_of(transaction_data_available(D)) <-
  neg(time_precedes(nonstrict,D,date([1991,3,31])))
```

3.7 Horn Clause axioms

Typically a number of non-equivalential meaning postulates are also required for the domain. These are represented as (interpreted) Horn clauses. We have already seen some examples of Horn-clause axioms in the preceding section, used to define conditions under which conceptual and database predicates correspond. There are also some more straight-forward cases where predicates may not be definable by means of equivalences, and where a Horn-clause definition may consequently be appropriate. The most common example results from the predicate `personal/1`, which holds of entities that can be referred to by the interrogative pronoun "who". In a normal context like *Who works on CLAM-BAKE?* or *Who booked time to project 8744 this month?*, no information is contributed by the occurrence of `personal`; it is implicit that entities who can work on projects, or book time to projects, can be referred to as "who". The appropriate way to declare this is by including the Horn-clauses

```
personal(Person) <-
  works_on_project(Event,Person,Project).

personal(Person) <-
  booking_to_project(Event,Person,Time,Date,Account).
```

The translation process will be able to use these to deduce that the occurrences of `personal` are implied by their contexts, and can safely be thrown away. In general, one such Horn-clause will be needed for every argument position in a conceptual predicate holding an object that `personal` can hold of.

3.8 AET with a domain theory: a full example

In this final section, we present a detailed example showing how CLARE uses the PRM linguistic domain theory to find an effective translation of the TRL representation of sentence (S1) (repeated here for convenience).

(S1) List all payments made to BT during 1990.

We will take a few liberties with the exact representation of the domain axioms in the interests of increased readability. In particular we have suppressed the complications caused by the treatment of granularity, and removed some arguments from predicates when they were irrelevant to the example. To make the formulas more compact, we also introduce two small notational abbreviations. We write `x` instead of `exists` to represent existential quantification, and we use the notation

```
<Predicate>#<Id>
```

to mean "the object of type `<Predicate>` with identifier `<Id>`". We also shorten the names of the linguistic predicates in an obvious way; thus for example we write `payment1` instead of `payment_PayingEvent`.

The original TRL representation of (S1) is

```
forall([Trans],
       impl(x([Agnt,Ev],
              and(payment1(Trans),
                  and(make2(Ev,Agnt,Trans,payee1#bt),
                      during1(Ev,interval(date([1990,1,1]),
                                          date([1990,12,31])))))),
            x([ShowEv],
              and(show1(ShowEv,clare,Trans),
                  before1(<Now>,ShowEv)))))
```

(We write `<Now>` to stand for the TRL representation of the present moment). The formula can be glossed as "For all `Trans` such that `Trans` is a payment made by some `Agent` during the interval from 1/1/90 to 31/12/90, it is the case that CLARE will show `Trans` at some time after the present time".

The desired effective translation will be expressed in terms of three evaluable predicates. The database information will be accessed by the predicate `TRANS(TransId,DBDate,PayeeName,Amount)`, which represents a database of SRI payments. Dates will be manipulated by the arithmetic predicate `t_precedes`, which holds if its arguments are two representations of calendar dates such that the first is temporally before the second. Execution of actions will be represented by the predicate

```
execute(Event,Action,Agent,Time)
```

The actions we are interested in here are displaying actions carried out by CLARE: these will be represented by letting `Agent` be equal to `clare` and `Action` be of the form `display(FormatList)` where `FormatList` is a list of objects to be displayed.

The first step is to use equivalences which translate the linguistic predicates `payment1`, `make2`, `during1`, `show1` and `before1` into conceptual predicates. We examine these in turn.

```
payment1(Trans*)
```

is translated to

```
x([Payer,Date,Payee,Amt],
  transaction(Payer,Trans*,Date,Payee,Amt))
```

using the domain-specific equivalence

```
transaction1(Trans) <->
x([Payer,Date,Payee,Amt],
  transaction(Payer,Trans,Date,Payee,Amt))).
```

Then

```
make2(Ev*,Agnt*,Trans*,payee1#bt)
```

is translated into

```
x([Date1,Amt1],
  and(transaction(Agnt*,Ev*,Date1,payee1#bt,Amt1),
      Ev*=Trans*))
```

using the rule

```
and(make2(Event,Payer,Payment,Payee),
    transaction1(Payment)) <->
x([Date,Amt],
  and(transaction(Payer,Event,Date,Payee,Amt),
      (Event = Payment))
```

The next rule to be applied is the one that translates `during1`. The intended semantics of `during1(E1,E2)` are "`E1` and `E2` are events, and the time associated with `E1` is inside that associated with `E2`". The relevant equivalences are

```
during1(E1,E2) <->
x([T1,T2],
  and(associated_time(E1,T1),
  and(associated_time(E2,T2),
      time_during(T1,T2))))
```

composed with the equivalence

```
and(associated_time(Trans,Date),
    transaction1(Trans)) <->
x([Payer,Payee,Amt],
  transaction(Payer,Trans,Date,Payee,Amt))
```

which says that the `associated_time` of a `transaction` is its third argument, and with a third equivalence from the general LDT which says that the time associated with an explicit interval is the interval itself. Applying all three of these in turn,

```
during1(Ev*,interval(date([1990,1,1]),date([1990,12,31])))
```

is translated into

```
x([Payer2,Date2,Payee2,Amt2],
  and(transaction(Payer2,Ev*,Date2,Payee2,Amt2),
      time_during(Date2,interval(date([1990,1,1]),date([1990,12,31])))))
```

After this,

```
show1(Agnt*,clare,Trans*)
```

is translated by following two equivalences. The first,

```
show1(E,Agent,X) <->
x([DisplayT,Format],
  and(execute(E,display(Format),Agent,DisplayT),
      display_format(X,Format)))
```

which is in the general LDT, can be glossed "A showing of `X` by `Agent` is the execution of a displaying of `Format`, where `Format` is the display-format of `X`". The second,

```
and(display_format(Trans,Format),
    transaction1(Trans)) <->
x([TransId,PayeeId,Payer,Date,Payee,Amt],
  and(transaction(Payer,Trans,Date,Payee,Amt),
  and(Trans = transaction#TransId,
  and(Payee = payee#PayeeId,
      Format = [TransId,Date,PayeeId,Amt]))))
```

(see section 3.5.8) is a context-sensitive domain-dependent equivalence which defines the display-format for a transaction to be a four-element list consisting of the transaction ID, the transaction date, the payee's name, and the amount. The result of applying both of these is

```
x([Id,Date3,Payee3,Amt3,DisplayT],
  and(execute(Agnt*,display([Id,Date3,Payee3,Amt3]),clare,DisplayT3),
      and(transaction(Payer3,Trans*,Date3,Payee3,Amt3),
          Trans*=transaction1#Id)))))
```

Finally,

```
before1(<Now>,ShowEv*)
```

is translated into

```
x([Format1,DisplayAgent,DisplayT]
  and(execute(ShowEv*,Format1,DisplayAgent,DisplayT),
      time_before(strict,<Now>,DisplayT)))
```

using rules similar to those used above to translate `during1`. When all the rules mentioned so far have been applied, the translated form is

```
forall([Payer,Trans,Date,Payee,Amt,Payer1,Date1,Amt1,Payer2,
        Date2,Payee2,Amt2],
       impl(and(transaction(Payer,Trans,Date,Payee,Amt),
                and(transaction(Payer1,Trans,Date1,payee1#bt,Amt1)
                    and(transaction(Payer2,Trans,Date2,Payee2,Amt2)
                        time_during(Date2,
                                    interval(date([1990,1,1]),
                                             date([1990,12,31])))))),
            x([Id,Date3,Payee3,Amt3,Format1,DisplayAgent,DisplayT3,
               DisplayT],
              and(execute(DisplayEv,
                          display([Id,Date3,Payee3,Amt3]),
                          clare,
                          DisplayT1),
                  and(execute(DisplayEv,Format1,DisplayAgent,DisplayT),
                      and(time_before(<Now>,
                                      DisplayT),
                          and(transaction(Payer3,Trans,Date3,
                                          Payee3,Amt3),
                              Trans=transaction1#Id)))))))
```

Exploiting the fact that `transaction` is functional on its second argument, and `execute` on its first, this reduces after simplification involving functional merging (see section 2.5.2) to

```
forall([Payer,Trans,Date,Payee,Amt],
       impl(and(transaction(Payer,Trans,Date,payee1#bt,Amt),
                time_during(Date,
```

```
                              interval(date([1990,1,1]),
                                       date([1990,12,31])))),
              x([Id,DisplayT],
                and(Trans=transaction1#Id,
                    and(execute(DisplayEv,
                                display([Id,DisplayT,payee1#bt,Amt]),
                                clare,
                                DisplayT),
                        time_before(<Now>,
                                    DisplayT))))))
```

Applying rules from the general LDT to translate `time_before` and `time_during` into `t_precedes`, we get

```
forall([Payer,Trans,Date,Payee,Amt],
       impl(and(transaction(Payer,Trans,Date,payee1#bt,Amt),
                and(t_precedes(date([1990,1,1]),Date),
                    t_precedes(Date,date([1990,12,31])))),
            x([Id,DisplayT],
              and(execute(DisplayEv,
                          display([Id,Date,bt,Amt]),
                          clare,
                          DisplayT),
                  and(Trans=transaction1#Id,
                      t_precedes(<Now>,DisplayT))))))
```

Now we apply the equivalence that translates the conceptual predicate `transaction` into the database predicate `TRANS`,

```
(and(transaction(Payer,Trans,Date,Payee,Amount),
     transaction_data_available(Date)) <->
 x([TransId,AMNum,PayeeName,DBDate],
   and(TRANS(TransId,DBDate,PayeeName,AMNum),
       and(Trans = transaction1#TransId,
           and(Payee = payee1#PayeeName,
               and(db_date_convert(Date,DBDate),
                   Amount = amount(sterling,AMNum))))))) <-
 Payer = sri
```

When the equivalence is applied, the conjunctive context is the set

```
{transaction(Payer*,Trans*,Date*,Payee*,Amount*),
 t_precedes(date(1990,1,1),Date*),
 t_precedes(Date*,date(1990,12,31)}
```

There are two conditions that must be proved from the context for the rule to be applicable,

```
Payer* = sri
```

and

```
transaction_data_available(Date*)
```

The first of these cannot be proved, but the assumption declaration

```
assumable(Payer = sri,
          0,
          payments_referred_to_are_from_SRI,
          specialization,
          transaction(Payer,Trans,Date,Payee,Amount))
```

("X can be assumed to be SRI if it occurs in the Payer argument of a transaction relation") makes it permissible to assume it. The other condition must be inferred. The relevant axioms are now two Horn-clauses: one defines the period for which transaction records exist, viz.

```
transaction_data_available(Date) <-
   and(t_precedes(date(1,8,89),Date2),
       t_precedes(Date,date(1,4,91)))
```

and the other encodes the fact that `t_precedes` is transitive,

```
t_precedes(Date1,Date3) <-
   and(t_precedes(Date1,Date2),
       t_precedes(Date2,Date3))
```

By chaining backwards through these to the context, it is then possible to prove that `transaction_data_available(Date*)` holds, and translate to the final form

```
forall([TransId,Date,DBDate,AMNum],
       impl(and(TRANS(TransId,DBDate,bt,AMNum)
                and(db_date_convert(Date,DBDate),
                    and(t_precedes(date([1990,1,1]),Date),
                        t_precedes(Date,date([1990,12,31])))))
            x([Id,DisplayT],
              and(execute(DisplayEv,
                          display([TransId,Date,bt,Amt]),
                          clare,
                          DisplayT),
                  t_precedes(<Now>,DisplayT)))))
```

The final formula admits a finite proof procedure, (cf sections 2.3.2 and 9.3), since its truth can be ascertained by an algorithm that can be summarized as

1. Search for all tuples from the finite `TRANS` relation.

2. Check each to find the ones where the `Payee` field is equal to `bt` and the `Date` field is suitably constrained by the arithmetic predicates `db_date_convert` and `t_precedes`

3. Assume that appropriate lists will in each such case be displayed by the interface's causing the corresponding instantiated instance of the `execute` predicate to hold in the future.

The system has also managed to prove, under an abductive assumption whose justification is `payments_referred_to_are_from_SRI`, that the final formula is equivalent to the original query. Granted this assumption, the user can be informed that the response is complete (cf section 2.3.4).

Chapter 4

Evolution of the QLF Formalism

This chapter describes the Quasi Logical Form formalism as currently used in CLARE, highlighting the differences between the current version of the QLF formalism and QLF as it was at the end of the CLE project (Alshawi 1992, Alshawi *et al* 1989).

The QLF formalism presented here provides one way, amongst many, of implementing a monotonic approach to semantic interpretation. However, various details of the QLF formalism arise in response to specific features of the semantics of English, or because they have proved convenient in the CLARE system. These details do not form part of the core of monotonic interpretation, and the reader is directed to Chapter 5 for a more theoretical, implementation independent account of monotonic semantic interpretation. In this chapter we will concentrate on specific features rather than the general theory, and hope to provide the reader with sufficient background to enable them to understand particular semantic analyses produced by the CLARE system.

4.1 The QLF Language

4.1.1 A Single Level of Logical Form

In the CLE-3 system, three distinct semantic representation languages were used: (i) Quasi Logical Form (QLF) to represent the results of the syntax driven semantic interpretation rules, (ii) Resolved Quasi Logical Form (RQLF) to represent the results of raising quantifiers, replacing anaphoric terms by their referents and so forth, and (iii) Logical Form (LF) representing the truth conditional parts extracted from RQLFs. Converting one representation into another usually involved destructive manipulation of the logical forms and/or loss of information contained in the earlier logical forms. Moreover, the QLF and RQLF languages did not have a proper semantics; it was only through converting them to LFs that they acquired a derivative semantics.

In CLARE (CLE-6) only one semantic representation language is used — QLF. QLF is a development of the CLE-3 QLF language[1], but differs from it in a number of important respects.

First, expressions in QLF may be underinstantiated. Underinstantiation is represented by means of meta-variables contained within a QLF expression. Reference resolution or scoping instantiates the QLF by means of instantiating some or all of its meta-variables to further QLF (sub-)expressions. Meta-variable instantiation is non-destructive, and is implemented as straightforward Prolog unification. Thus destructively raising quantifiers out of their surface positions or replacing anaphoric terms by their referents plays no role in scoping or resolution. Instead, a scoping constraint meta-variable may be instantiated to show where a particular term is scoped, or the referent meta-variable of an anaphoric term may be instantiated to an expression denoting the referent.

Second, the QLF language has a directly defined semantics. This applies even when QLF expressions contain uninstantiated meta-variables. Semantically, instantiating meta-variables has the effect of narrowing the class of models in which a QLF formula counts as true (see Chapter 5). As a simple illustration, the initial QLF for a sentence like *He slept* would simply say that some male entity, salient in context, slept. The unresolved QLF could thus be true if, say, John Smith slept, or if George Brown slept, or if Tom Jones slept. Resolving the pronoun might result in a QLF that said that some male entity, salient in context and identical to John Smith, slept. This narrows the class of models in which the QLF could count as true. The resolution just adds the extra information that the object denoted by the term is identical to John Smith.

The QLF instantiation brought about by reference resolution and scoping is both syntactically and semantically monotonic. Syntactically because it just instantiates variables, and semantically because it narrows the class of possible models. Since QLFs, whether completely uninstantiated, partially instantiated, or fully instantiated, all have truth conditions associated, there is no need to postulate either a separate level of RQLF or a separate level of LF. Both these levels of representation have disappeared from the CLARE system.

However, an LF-like representation can be obtained, if necessary, as part of the process of evaluating QLFs. For example, in the PRM application (Chapters 2, 3) expressions in a Target Reasoning Language (TRL) are extracted from QLFs after they have been scoped and resolved. One can look on TRL as a form of meta-language in which the semantic definitions for QLF are expressed[2] It is generally more convenient to reason using a TRL representation of a QLF's truth conditions

[1] To avoid confusion, we will henceforth refer to the CLE-3 QLF language as QLF-3, leaving QLF to refer to the current version of the language.

[2] In giving the semantics for a language like the propositional calculus, one finds definitions like "$S\phi \vee \psi$ is true iff ϕ is true or ψ is true" where the object language connective $\vee$ is defined in terms of the meta-language connective "or". The TRL connectives and quantifiers behave as meta-language versions of QLF constructs.

rather than the QLF itself, but from a purely linguistic point of view TRL is an eliminable level of representation.

Finally, there are a number of more minor differences between QLF-3 and QLF. For example, the QLF-3 constructs `qterm` and `a_term` have been reduced to a single `term` construct, and `a_form` constructs are now `forms`. Constructs like `term_coord` have been eliminated thanks to a more satisfactory treatment of NP conjunction, as have 'union terms'. Tense operators like `past` or `pres` have also been dispensed with. Other differences will become apparent below.

4.1.2 Syntax of QLF

The QLF language has the following BNF syntax:

$$\begin{array}{lll}
\langle \textit{qlf_formula} \rangle & \rightarrow & \texttt{form(}\langle \textit{word_string} \rangle\texttt{,}\ \langle \textit{category} \rangle\texttt{,}\ \langle \textit{form_index} \rangle\texttt{,}\ \langle \textit{qlf_restriction} \rangle\texttt{,} \\
 & & \quad \langle \textit{form_referent} \rangle\texttt{)} \ \mid \\
 & & \texttt{[}\langle \textit{functor} \rangle\texttt{,}\langle \textit{argument}_1 \rangle\texttt{,}\ldots\langle \textit{argument}_n \rangle\texttt{]} \ \mid \\
 & & \texttt{sc(}\langle \textit{scope_constraint} \rangle\texttt{,}\langle \textit{qlf_formula} \rangle\texttt{)}
\end{array}$$

$$\begin{array}{lll}
\langle \textit{qlf_term} \rangle & \rightarrow & \langle \textit{term_index} \rangle \ \mid \\
 & & \texttt{term(}\langle \textit{word_string} \rangle\texttt{,}\ \langle \textit{category} \rangle\texttt{,}\ \langle \textit{term_index} \rangle\texttt{,}\ \langle \textit{qlf_restriction} \rangle\texttt{,} \\
 & & \quad \langle \textit{quantifier_referent} \rangle\texttt{,}\ \langle \textit{term_referent} \rangle\texttt{)} \ \mid \\
\multicolumn{3}{l}{\langle \textit{constant} \rangle \mid \langle \textit{variable} \rangle \mid \langle \textit{term_index} \rangle}
\end{array}$$

$$\langle \textit{qlf\_abstract} \rangle \rightarrow \langle \textit{variable} \rangle\texttt{\^{}}\langle \textit{qlf\_formula} \rangle \mid \langle \textit{variable} \rangle\texttt{\^{}}\langle \textit{qlf\_abstract} \rangle$$

$$\langle \textit{qlf\_restriction} \rangle \rightarrow \langle \textit{variable} \rangle\texttt{\^{}}\langle \textit{qlf\_formula} \rangle$$

$$\langle \textit{functor} \rangle \rightarrow \langle \textit{atom} \rangle$$

$$\langle \textit{argument} \rangle \rightarrow \langle \textit{qlf\_formula} \rangle$$

$$\langle \textit{argument} \rangle \rightarrow \langle \textit{qlf\_term} \rangle$$

$$\langle \textit{argument} \rangle \rightarrow \langle \textit{qlf\_abstract} \rangle$$

$$\langle \textit{argument} \rangle \rightarrow \texttt{[apply,}\langle \textit{qlf\_abstract} \rangle\texttt{,}\langle \textit{argument} \rangle\texttt{]}$$

$$\langle \textit{constant} \rangle \rightarrow \langle \textit{atom} \rangle$$

$$\langle \textit{constant} \rangle \rightarrow \texttt{'N'(}\langle \textit{number} \rangle\texttt{)}$$

$$\langle \textit{constant} \rangle \rightarrow \texttt{'SF'(c(}\langle \textit{qlf\_restriction} \rangle\texttt{,}\langle \textit{atom} \rangle\texttt{))}$$

$$\langle \textit{variable} \rangle \rightarrow \langle \textit{prolog\_variable} \rangle$$

$$\langle \textit{term\_index} \rangle \rightarrow \langle \textit{variable} \rangle \mid \langle \textit{atom} \rangle$$

$$\langle \textit{form\_index} \rangle \rightarrow \langle \textit{variable} \rangle \mid \langle \textit{atom} \rangle$$

$$\langle \textit{quantifier} \rangle \rightarrow \langle \textit{distrib\_quant} \rangle \mid \langle \textit{set\_quant} \rangle$$

⟨*quantifier*⟩ → `wh` | ⟨*variable*⟩`^count(`⟨*variable*⟩`)` |
⟨*variable*⟩`^sum(`⟨*variable*⟩`)` |
`amount(`⟨*variable*⟩`^`⟨*variable*⟩`^`⟨*qlf_formula*⟩`,` ⟨*unit*⟩`)`

⟨*distrib_quant*⟩ → `forall` | `exists` | `most` | ...

⟨*distrib_quant*⟩ → ⟨*variable*⟩`^`⟨*variable*⟩`^`⟨*lf_formula*⟩

⟨*set_quant*⟩ → `set(`⟨*distrib_quant*⟩`)`

⟨*unit*⟩ → ⟨*qlf_restriction*⟩

⟨*word_string*⟩ → `l([`⟨*word*⟩`*])`

⟨*word*⟩ → ⟨*atom*⟩

⟨*form_referent*⟩ → ⟨*meta_variable*⟩ | ⟨*qlf_formula*⟩ |
`app(`⟨*qlf_abstract*⟩`,`⟨*form_index*⟩

⟨*quantifier_referent*⟩ → ⟨*meta_variable*⟩ | ⟨*quantifier*⟩

⟨*term_referent*⟩ → ⟨*meta_variable*⟩ |
`strict(`⟨*term_index*⟩`)` |
`sloppy(`⟨*term_index*⟩`,`⟨*qlf_restriction*⟩`)` |
`ent(`⟨*constant*⟩`)` |
`qnt(`⟨*term_index*⟩`)`

⟨*scope_constraint*⟩ → `[`⟨*term_index*⟩`*]` | ⟨*meta_variable*⟩

⟨*meta_variable*⟩ → ⟨*prolog_variable*⟩

Below, an informal description of the meaning of various QLF constructions is given. See also Chapter 5. Note also that in Chapter 5 a variant of the QLF formalism is used. This differs primarily in (a) using round bracket notation for predicate argument formulas, i.e.

```
Pred(Arg1,..,Argn)
```

instead of `[Pred,Arg1,...,Argn]`, and (b) omitting the `word_string` argument to `term`s and `form`s.

`terms` and `forms`

The main components of the QLF language are `form`s and `term`s. The first four arguments to `form` and `term` expressions are: (1) A list of words representing the surface string that the `form` or `term` represents; (2) A category, which can be any Prolog `functor(arguments...)` expression; (3) An index uniquely identifying the `form` or `term`; (4) A restriction, which is a one-place abstract. For `term`s, the restriction is a one-place predicate, corresponding perhaps to the meaning of a common noun. For `forms`, the restriction is typically a one-place higher-order predicate.

The remaining arguments to the `forms` and `terms` are normally left as uninstantiated meta-variables after the initial stages of semantic analysis. The meta-variables can then be instantiated by reference resolution.

`form` Resolutions

`forms` have just one unresolved / meta-variable argument. This can be instantiated in one of two ways. It can either be instantiated to a formula or to an application. An application (`app(`⟨*abstract*⟩`,`⟨*index*⟩`)`) is a way of stating that the restriction of the `form` should be applied to the ⟨*abstract*⟩ (the ⟨*index*⟩ is always the index of the `form`). When resolved to a formula, the formula represents the result of such an application. The formula (or result of the application) constitutes the final meaning of the `form`.

`term` Resolutions

`terms` have two unresolved arguments. The first is resolved to the quantifier associated with the `term`. The second (the 'term referent') provides a further contextual restriction on the range of this quantification (the contextual restriction being combined with the `term`'s restriction).

Quantifiers Quantifiers are generalised determiners, i.e. two place predicates on set cardinalities. The cardinality may either be represented explicitly, or in abbreviated form (e.g. `exists` or `forall`). There is a difference between ordinary (distributive) quantifiers and set quantifiers. A distributive quantifier quantifies over individual objects satisfying the restriction predicate, whereas the set quantifier quantifies over sets of such objects.

There also a few 'non-standard' quantifiers. The WH-quantifiers (`wh, count, sum`, corresponding roughly to *which, how many* and *how much*) are in fact abstraction operators (so that WH-questions are semantically treated as lambda-abstracts and not quantified formulae). The `amount` quantifier is used for expressions like *three pints of milk*, where the unit of measure for the cardinality predicate needs to be specified.

The treatment of quantifiers / generalised determiners is broadly the same as that described in Alshawi 1992.

Contextual Restrictions / Term Referents The `term`'s contextual restriction comes in a number of varieties. The simplest is a `qnt(`⟨*index*⟩`)` restriction. This just says that the term acts as an ordinary quantifier with no restriction on its range of quantification beyond that specified by the `term`'s restriction.

The ⟨*index*⟩ in these case is always the `term`'s index. For terms (like proper names) referring to specific entities, an `ent(`⟨*constant*⟩`)` resolvent is used, where ⟨*constant*⟩ is a constant term referring to some object. This means that the `term` is treated as an (existential) quantifier over objects identical to that denoted by the constant. For anaphoric terms (like pronouns and some definite descriptions), a `strict(`⟨*index*⟩`)` resolution is appropriate. Here, the index is that of the antecedent `term`. If the anaphoric `term` is within the scope of the antecedents quantification, this index becomes bound by the antecedent. Otherwise a predicate uniquely describing the antecedent is obtained from context and used as a further contextual restriction. For things like 'one-anaphora', a `sloppy(`⟨*index*⟩`,`⟨*predicate*⟩`)` resolution is needed. Here, the index is that of the antecedent term, and the description is obtained from the antecedent restriction.

The `qnt` style resolution could be extended to include a general contextual restriction on quantification not deriving from any particular antecedent term. This would necessitate including another argument place for a property expression. This has not been implemented, however.

Scoping Constraints

Scoping constraints are (possibly empty) ordered lists of term indices. Each `term` is 'quantified in' at the point where its index occurs. Scoping constraints are normally only instantiated by reference resolution. Chapter 5 provides a fuller account of scoping in monotonic interpretation.

Sorted Constants

Sorted constants of the form `'SF'(c(`⟨*sort*⟩`,`⟨*object*⟩`))` are a way of explicitly associating a sortal predicate with some individual denoting constant. They are usually introduced when `term`s are resolved to refer to `ent`ities. The `'N'` wrapper also identifies constants referring to numbers.

Categories

No syntax for category expressions has been specified. Usually, they are atoms or Prolog `functor(arguments...)` expressions, where the arguments are give the values of certain features. Sometimes, as in the case of some `term` categories, one of these values can be a QLF expression. This most often occurs when a complex comparative determiner is built up (e.g. for *more cows than Mary has sheep*).

4.2 Example QLFs

We now illustrate the way that the QLF formalism is used to handle various semantic phenomena in English. This is done by giving the QLFs assigned to various sentences. It will usually be convenient first to show the uninstantiated QLF initially produced by syntax driven interpretation, and then an instantiated QLF as produced by reference resolution.

Proper Names and Tense

A simple sentence like *John slept* illustrates the basic treatment of proper names and tense. The unresolved QLF for the sentence is

```
[dcl,
 form(l([John,slept]),
      verb(past,no,no,no,y),
      A,
      B^[B,[sleep, A,
            term(l([John]),proper_name(tpc),C,
                 D^[name_of,D,'John'],_,_)]],
      _)]
```

At the top level, this a formula with the mood operator `dcl` applying to a verb `form`. The verb `form` has the category `verb(past,no,no,no,y)` which says that the `form` corresponds to a verb phrase, that it is in the past tense, is non-perfective, non-progressive, contains no modal verb, and is in the active voice. The `form` restriction essentially says that some (contextually salient) property needs to be applied to the formula `[sleep,A,term(...)]` in order to resolve the `form`. The resolution meta-variable is uninstantiated. Unlike the QLF-3 representation of this sentence, there are no tense operators, and no terms referring to events. The latter, though, are introduced by reference resolution.

The `term` corresponding to the subject noun phrase occurs within the restriction of the verb `form`. Its category specifies that the term corresponds to a proper name, and the `tpc` feature indicates that the NP occupies subject / topic position. The `term`'s restriction is simply the property of having the name 'John'. Both resolution meta-variables are uninstantiated.

Resolving the QLF above might instantiate the meta-variables as follows (QLF shown abbreviated)

```
[dcl,
 form(l([John,slept]), .., .., P^[P,[sleep,...]],
      sc([E],[sleep,
              term(.., .., E,
                   X^[and,[event,X],[precedes,X,now]],
                   exists,
                   qnt(E)),
              term(l([John]), .., C, ..,
                   exists,
                   ent('SF'(c(Y^[person,Y],john_brown))))]
         ))]
```

The verb form is resolved to a formula rather than an application. Since the resolution includes much of the structure of the form's restriction, and hence the bulk of the QLF, we show only this.

The resolution of the verb form essentially involves abstracting over the form index and replacing it with an event term. This results in a quantified formula saying that there was an event preceding the current time, and that it was an event of John sleeping. The event term is given scope over the sleep predication.

It should be pointed out that the use of quantification over events and times is not an inherent part of the QLF formalism. We could instead have used tense operators, or some mixture of quantification over time and tense operators. Nor are we necessarily committed to the view that quantification over events and times is the past way of handling temporal reference in a language like English: it just happens to be the way it is done here.

The second argument to the resolved verbal predication, the term referring to John, is identical to the term as it occurs in the form restriction. The term is resolved to be an existential quantification over objects identical to the sorted entity john_brown, where john_brown is a person. Since this term can be given any scope without affecting the truth conditions of the QLF, its scope is left uninstantiated.

NP Modification and PPs

Modified NPs are dealt with by conjoining extra properties to the restriction of the NP's term. A resolved term corresponding to *A man in London* is

```
term(l([a,man,in,London]),
     q(_,a,sing), M,
     X^[and,[man,X],
            form(l([in,London]),
                 prep(in), F,
                 P^[P,X,
                      term(l([London]),
                           proper_name(_),L,
                           Y^[name_of,Y,'London'],
                           exists, ent('SF'(c(Z^[town,Z],london))))],
                 app(R^S^[in_spatial,R,S],F))],
      exist,qnt(M))
```

The term's category, q(_,a,sing) shows that it is a quantificational term, with determiner *a* and in the singular. The underscore is for a subject/topic feature, and is uninstantiated since the term is not included in a wider sentence. The term is resolved to be a straight-forward existential quantification.

The term's restriction is a conjunction of the unmodified noun property (X^[man,X]) and a prepositional phrase property (X^form(...)). The category of the PP `form` states which preposition is used, but does not attempt to distinguish between different temporal, spatial or other uses. The `form` resolution indicates the specific sense in which the preposition is to be understood. Applying the `form` restriction to the resolvent property, `R^S^[in_spatial,R,S]`, gives

```
[in_spatial, X,
            term(l([London]),
                 proper_name(_),L,
                 Y^[name_of,Y,'London'],
                 exists, ent('SF'(c(Z^[town,Z],london))))]
```

By treating prepositions as having vague senses that must be further resolved to spatial, temporal or other properties, one can avoid a proliferation of sense entries for prepositions in the lexicon.

VP Modification

Prepositional phrases can modifiy verb phrases as well as noun phrases. Here the QLF representation is slightly different. The PPs, as well as other modifiers to the VP, are included in a list in the verb `form`'s restriction preceding the verbal predication itself. Thus, a sentence like *John slept well on Tuesday* would receive the following QLF

```
[dcl,
 form(l([John,slept,well,on,Tuesday]),
      verb(past,no,no,no,y),
      A,
      B^
      [B,
       form(l([on,Tuesday]),
            prep(on), C,
            D^[D, v(A),
                  term(l([Tuesday]),
                      time(day), E,
                      F^[day_name,F,tuesday_DayName],
                      _,_)],
            _),
       [good,v(A)],
       [sleep, A,
               term(l([John]),proper_name(tpc),R,
                    S^[name_of,S,John],_,_)]],
      _)]
```

The adverb *well* is interpreted as being derived from the adjective *good*, and gives rise to a predication on the verb `form`'s index. The verb `form`'s index also provides the first argument in the prepoisitional phrase restriction. When the verb `form` is resolved, occurrences of the `form`'s index will be replaced by the corresponding event variable, and modifiying predications will be conjoined with the main verbal predication. The net result is a Davidsonian style treatment of adverbial modification (Davidson 1967), where modifiers are predicates of an event.

'Big' PPs When several prepositional phrases modify a verb phrase, as in *John slept in London on Tuesday*, the prepositional phrases are parsed as forming a single 'big' PP acting as just one modifier to the verb phrase. The QLF reflects the 'big' PP structure:

```
[dcl,
 form(l([John,slept,in,London,on,Tuesday]),
      verb(past,no,no,no,y),A,
      B^
      [B,
       form(l([in,London,on,Tuesday]),
            conj(pp,implicit_and),_,
            C^
            [C,
             form(l([in,London]),prep(in),_,
                  D^
                  [D,v(A),
                   term(l([London]),proper_name(_),_,
                        E^[name_of,E,London],_,_)],
                  _),
             form(l([on,Tuesday]),prep(on),_,
                  F^
                  [F,v(A),
                   term(l([Tuesday]),time(day),_,
                        G^[day_name,G,tuesday_DayName],_,_)],
                  _)],
            _),
       [sleep,A,
        term(l([John]),proper_name(tpc),_,
             H^[name_of,H,John],_,_)]],
      _)]
```

The big PP is in fact treated as an implicit conjunction of the component prepositional phrases. After resolution it gives rise to the same conjunction of predications on the event variable as would arise if the PPs has been treated as individual modifiers.

Coordination

Conjunctions are treated as lists of the conjoined QLF expressions, forming the restriction of a conjunction `form`. For example *John ate, drank and slept* gets the

QLF:

```
[dcl,
 form(l([John,ate,,,drank,and,slept]),
      conj(vp,and),_,
      A^
      [A,
       form(l([slept]),verb(past,no,no,no,y),B,
            C^
            [C,[sleep,B,
              term(l([John]),proper_name(tpc),D,E^[name_of,E,John],
                   _,_)]],
            _),
       form(l([drank]),verb(past,no,no,no,y),F,
            G^[G,[drink,F,v(D)]],_),
       form(l([ate]),verb(past,no,no,no,y),H,
            I^[I,[eat,H,v(D)]],_)],
      _)]
```

The category `conj(vp,and)` states what type of constituent is coordinated and the conjunction used. Resolving the `form` results in a logical conjunction being built up from the resolved elements of the conjunction list.

NP Conjunction Coordination of most other constituents is handled analagously to that of VPs. However, noun phrase conjunction is handled slightly differently, since one needs to produces a conjoined `term` rather than a conjoined `form`. A sentence like *Every man, every woman and every child slept* gets the following QLF:

```
[dcl,
 form(l([every,man,,,every,woman,and,every,child,slept]),
      verb(past,no,no,no,y),A,
      B^
      [B,
       [sleep_BeNaturallyUnconscious,A,
        term(l([every,man,,,every,woman,and,every,child]),
             conjdet(tpc,and,np,C,_),C,
             D^
             form(_,conj(np,and),_,
                  E^
                  [E,D,
                   term(l([every,child]),q(tpc,every,sing),_,
                        F^[child_YoungPerson,F],_,_),
                   term(l([every,woman]),q(tpc,every,sing),_,
                        G^[woman_FemalePerson,G],_,_),
                   term(l([every,man]),q(tpc,every,sing),_,
```

```
                        H^[man_MalePerson,H],_,_)],
                 _),
            _,_)]],
      _)]
```

Here, a conjunction `form` serves as the restriction to a special conjunction `term`. In resolving the conjunction, one first of all constructs a group denoting term, such that every child is included within the group, every woman is included within the group and (or) every man is included within the group. One then quantifies over individual members of this group (distributive conjunction) or over the group as a whole (collective conjunction).

The (distributive) resolution of the conjoined NP above would be something like

```
term(..,conjdet(...),C
     D^form(...,
            [partof,D,
              term(_,group,G,
                   X^[and,sc([C1],[partof,X,
                                   term(l([every,child]),..,C1,..)]),
                      [and,sc([W1],[partof,X,
                                    term(l([every,woman]),..,W1,..)]),
                           sc([M1],[partof,X,
                                    term(l([every,man]),..,M1,..)])]],
                   exists,qnt(G))]),
     forall,qnt(C))
```

The group term `G` is always given wide scope relative to the conjunction term `C`.

This treatment of NP coordination avoids use of earlier `term_coord` constructions and union terms. It uses only propositional conjunction and disjunction, and may be used to combine all forms of noun phrase to give either a distributive or collective coordination.

Possessives and Partitives

A possessive noun phrase, like *John's mother*, gets the following QLF

```
term(l([John,'s,mother]),q(_,poss_some,_),_,
     B^[and,[mother_FemaleParent,B],
            form(_,poss,_,
                 C^
                 [and,[mother_FemaleParent,B],
                  [C,B,
                   term(l([John]),proper_name(_),_,
                        D^[name_of,D,John],_,_)]],
                 _)],
           _,_)
```

The embedded `poss form` is resolved to a particular relation holding between people and their mothers. For different possessives, different relations might be appropriate. In the PRM domain for example a possessive noun phrase like *the project's payments* would be understood as referring to payments on the project.

Partitive noun phrases, like *each of the men* are treated as follows

```
term(l([each,of,the,men]),
     ell(q(_,each,sing),l([])),A,
     B^[and,[entity,B],
        form(l([of,the,men]),
            genit(sub),_,
            C^[and,[entity,B],
               [C,B,
                  term(l([the,men]),
                       ref(def,the,plur,l([])),_,
                       D^[man_MalePerson,D],_,_)]],
             _)],
      _,qnt(A))
```

Syntactically, partitives are treated as prepositional phrases (*of the men*) modifying bare determiners acting as pronouns (*each, some, all* etc). However, rather than resolving the bare determiner NP in the normal way as an anaphor, the initial semantic analysis forces it to be quantificational by instantiating the term referent. The `genit(sub) form` is resolved to give the property of being an element that is `partof` the denotation of the (group denoting) NP argument (i.e. being one of the men).

Partitives differ from possessive NPs like *the mother of John* in that (i) the genitive relation is not fixed as `sub`, (ii) there is no elliptical determiner NP, and (iii) the referent is not instantiated by the semantics:

```
term(l([the,mother,of,John]),ref(def,the,sing,l([])),_,
     B^[and,[mother_FemaleParent,B],
            form(l([of,John]),genit(_),_,
                  C^
                  [and,[mother_FemaleParent,B],
                   [C,B,
                    term(l([John]),proper_name(_),_,
                         D^[name_of,D,John],_,_)]],
                  _)],
      _,_)
```

(In fact, leaving the sub-class of the `genit` category uninstantiated is not quite correct, and the grammar has since been altered to ensure that the category is properly instantiated).

Chapter 5

Monotonic Semantic Interpretation

5.1 Introduction

The monotonicity property of unification based grammar formalisms is perhaps the most important factor in their widespread use for grammatical description and parsing. Monotonicity guarantees that the grammatical analysis of a sentence can proceed incrementally by combining information from rules and lexical entries in a nondestructive way. By contrast, aspects of semantic interpretation, such as reference and quantifier scope resolution, are often realised by non-monotonic operations involving loss of information and destructive manipulation of semantic representations. A 'two-level' approach to semantic interpretation tends to result (Bronneberg *et al.* 1980), where an initial, underspecified representation is transformed into a separate, specified, representation.

This chapter describes a model for semantic interpretation that is fully monotonic in both linguistic and contextual aspects of interpretation, and which employs just one level of semantic representation — Quasi Logical Form (QLF). Contextual resolution of underspecified QLF expressions involves the instantiation of QLF meta-variables. The semantics for the QLF formalism makes the denotation of a QLF formula a partial function to truth-values, with resolution leading to a monotonic extension of the denotation function. We believe that there are several advantages to the approach taken, including:

- Order independence of resolution operations
- Production of partial interpretations
- Simpler interactions between phenomena
- Reversibility for synthesis/generation

The QLF formalism is a development of (Alshawi 1990). As before, underspecified QLFs are produced on the basis of a unification grammar. Previously, QLF resolution was only partially monotonic; full monotonicity required changes to the original QLF formalism and the resolution and scoping processes, as described in Chapter 4.

The chapter is organized as follows. Section 5.2 provides the syntax of the QLF language and Section 5.3 gives some illustrative examples of monotonic QLF resolution. Sections 5.5 and 5.4 present the semantics of the QLF formalism. Section 5.6 discusses the relationship between monotonic interpretation, Pereira's categorial semantics (Pereira 1990), and context change approaches to semantics. Section 5.7 mentions some benefits of using QLF-like representations in implementing natural language systems.

5.2 Syntax of Monotonic QLF

We give here a fuller syntactic description of the QLF constructs for terms and formulas[1].

A QLF term must be one of the following

- a term variable: `X`, `Y`, ...
- a term index: `+i`, `+j`, ...
- a constant term: `7`, `mary1`, ...
- an expressions of the form:
  `term(Idx,Cat,Restr,Quant,Reft)`

The term index, `Idx`, uniquely identifies the `term` expression. `Cat` is a list of feature-value equations, for example `<type=pro,num=sing,...>`. `Restr` is a first-order, one-place predicate. For a resolved `term`, `Quant` will be a generalized quantifier (a cardinality predicate holding of two properties) and `Reft`, the `term`'s 'referent', will be a constant or term index. For an 'unresolved' `term`, `Quant` and `Reft` may be meta-variables (`_x`,`_y`,...). (QLF terms may also be functional applications, though we will ignore these here).

A QLF formula must be one of the following

- the application of a predicate to arguments:
  `Predicate(Argument1,...,Argumentn)`
- an expression of the form:
  `form(Category,Restriction,Resolution)`

[1]The notation we use in implementations is slightly different but equivalent to that presented here.

- a formula with scoping constraints:
  `Scope:Formula`

`Predicate` is a first or higher-order predicate, including the usual logical operators `and`, `not`, etc. An argument may be a term, a formula or a lambda abstract. Lambda abstracts take the form `Var^Body` where `Body` is a formula or an abstract and `Var` is a variable ranging over individuals or relations. `Restriction` is a higher-order predicate. `Resolution` is a formula (the 'referent' of the `form` expression), or is a meta-variable if the `form` expression is unresolved. `Scope` is either a meta-variable when scoping information is underspecified or a (possibly empty) list of term indices e.g. `[+i,+j]` if term `+i` outscopes `+j`. The terms identified by the indices must occur within `Formula`.

The degree to which a QLF is unresolved corresponds approximately to the extent to which meta-variables (appearing above as `Quant`, `Reft`, `Scope`, and `Resolution`) are instantiated to the appropriate kind of object level expressions (though see Section 5.4 for an explicit characterization of unresolved QLFs and partial interpretations.)

5.3 Example QLF Resolutions

Resolution of QLFs through the instantiation of meta-variables has been applied to a wide range of phenomena. These include pronouns, definite descriptions, implicit or vague relations, ellipsis and temporal relations (see Alshawi 1990 for an account of some kinds of reference resolution in an earlier QLF formalism). For concreteness, we present a few illustrative examples of monotonic QLF resolution[2]. We do not attempt to describe the mechanism by which the resolutions are chosen.

It will become evident that the notation is closer to (the syntactic structure of) natural language than is the case for traditional logical formalisms. For example, `term`s usually correspond to noun phrases, with information about whether e.g. they are pronominal, quantified or proper names included in the `term`'s category. This makes the QLF representation easier to read than it might seem at first, once its initial unfamiliarity is overcome.

Quantification: *Every boy met a tall girl* illustrates the representation of quantification. The basic QLF analysis might be (ignoring tense):

```
_s:meet(term(+b,<type=q,lex=every>,boy,_q,_x),
        term(+g,<type=q,lex=a>,
             Y^and(girl(Y),tall(Y)),_r,_y)).
```

[2] Although the QLF framework can support a variety of alternative semantic analyses for specific phenomena, to provide concrete illustrations one or other analysis needs to be chosen. In the following examples, it should be possible to separate particular analyses from the general points we wish to make about monotonic interpretation.

A resolved structure could be obtained by instantiating the quantifier meta-variables `_q` and `_r` to `forall` and `exists`[3], and the scoping meta-variable `_s` to `[+b,+g]` for the '∀∃' reading:

```
[+b,+g]:
 meet(term(+b,<type=q,lex=every>,
           boy,forall,+b),
      term(+g,<type=q,lex=a>,
           Y^and(girl(Y),tall(Y)),exists,+g)).
```

In a restriction-body notation for generalized quantifiers, the truth conditional content of this resolved expression corresponds to

```
forall(B,boy(B),
         exists(G,and(girl(G),tall(G)),
                  meet(B,G))).
```

Anaphora: *Every boy claims he met her* illustrates the treatment of anaphora (in a context where Mary is assumed to be salient)[4]

Unresolved:

```
_s1:claim(
     term(+b,<type=q,lex=every>,boy,_q1,_x),
     _s2:meet(term(+h1,<type=pro,lex=he>,
                   male,_q2,_y),
              term(+h2,<type=pro,lex=her>,
                   female,_q3,_z))).
```

Resolved:

```
[+b]:claim(
      term(+b,<type=q,lex=every>,
           boy,forall,+b),
      [+h1]:meet(term(+h1,<type=pro,lex=he>,
                      male,exists,+b),
                 term(+h2,<type=pro,lex=her>,
                      female,exists,mary))).
```

The pronominal term for *her* is resolved so that it existentially quantifies over female objects identical to `mary`. The 'bound variable' pronoun *he* has a referent

[3] The benefits of being able to resolve determiners to quantifiers are discussed in Alshawi 1990. For example, determiners like *some* (plural) could be resolved to collective or distributive quantifiers, *three* could be interpreted as meaning either 'exactly three' or 'at least three', and if need be, bare plurals like *dogs* could be variously interpreted as meaning 'some dogs', 'all dogs' or 'most dogs'.

[4] Here we simplify the issues arising out of the semantics of intensional, sentential complement verbs like *claim*.

coindexed with its antecedent, +b. The scope of +h2 is left unspecified, since exactly the same truth conditions arise if it is given wide or narrow scope with respect to *every boy* or *he*.

Vague Relations: An unresolved QLF expression representing the noun phrase *a woman on a bus* might be a term containing a form that arises from the the prepositional phrase modification:

```
term(+w,<lex=a,..>,
     X^and(woman(X),
           form(<type=prep,lex=on>,
                R^R(X,term(+b,<lex=a,..>,
                            bus,_q2,_b)),
                _f)),
     _q1,_w).
```

Informally, the form is resolved by applying its restriction, R^R(...) to an appropriate salient predicate, and instantiating the form's meta-variable, _f, with the result. In this case, the appropriate predicate might be inside, so that _f is instantiated to

```
inside(X,term(+b,<lex=a,..>,bus,_q2,_b)).
```

Tense: One way of treating tense is by means of a temporal relation form in the restriction of an event term. For *John slept* we might have:

```
_s:sleep(term(+e,<type=event>,
              E^form(<type=trel,tense=past>,
                     R^and(event(E),R(E)),
                     _t),
              _q1,_e),
         term(+j,<type=name>,
              J^name(J,'John'),_q2,_j)).
```

Since the tense on the temporal relation category is past, the resolution says that the event occurred before a particular speech time, t7:

```
[+e]:
 sleep(
  term(+e,<type=event>,
       E^form(<type=trel,tense=past>,
              R^and(event(E),R(E)),
              and(event(E),precede(E,t7))),
       exists,+e),
  term(+j,<type=name>,
       J^name(J,'John'),exists,john1)).
```

The resolution `and(event(E),precede(E,t7))` is the result of applying the `form`'s restriction `R^and(event(E),R(E))` to a contextually derived predicate, in this case `E1^precede(E1,t7)`.

QLF is not committed to an event based treatment of tense. An alternative that has also been implemented is to treat the verbal predication `sleep(...)` as a temporal `form`, whose category specifies tense and aspect information.

Ellipsis: A more complex example, involving ellipsis and quantification, is provided by

Each boy claimed he was clever, and so did John.

A partially resolved QLF, but one in which the ellipsis is still unresolved, might be as follows (ignoring tense and event variables):

```
and(
claim(term(+b,<lex=every>,
           boy,exists,+b),
      clever(term(+h,<lex=he>,
                  male,exists,+b))),
form(<type=vpellipsis>,
     P^P(term(+j,<type=name>,J^name(J,'John'),
              exists,john)),
     _e)).
```

This is a conjunction of the QLF for the antecedent clause (*Each boy claimed he was clever* under a bound pronoun reading) with a `form` expression for the verb phrase ellipsis. Solutions for instantiating the meta-variable `_e` for the ellipsis are the result of applying a property `P1`, derived from the antecedent clause, to the `term` with index `+j`. The sentence has two readings: a sloppy reading where John claims that he is clever, and a strict one where John claims that each of the boys is clever. The choice between a strict or sloppy reading depends on how the term *he* is reinterpreted in the ellipsis resolution. Intuitively, strict identity involves referring to the same object as before, whereas sloppy identity involves referring to a relevantly similar object.

In QLF, a strict reading results from re-interpreting the ellipsis pronoun as co-indexed with the original, i.e. taking `P1` to be:

```
X^claim(X,clever(+h)).
```

Constraints on legitimate scoping (Section 5.4) force `+b` and `+h` to take wide scope over both the antecedent and ellipsis. The sloppy reading results from re-indexing the ellipsis pronoun so that it has the same restriction and category as the original, but is resolved to `+j` and has a new index `+h1`. This corresponds to taking `P1` to be:

```
X^claim(X,clever(term(+h1,<lex=he>,
                      male,exists,+j))).
```

More generally, in Chapter 7 we explore the claim that solutions to verb phrase ellipsis have the general form:

```
P1 = X1^..Xi^S:{s1/X1,..,si/Xi,..,sn/tn}.
```

That is, `P1` is formed out of an antecedent clause QLF `S` by abstracting over the 'parallel elements' `s1..si`, perhaps with some additional substitutions for terms `si+1..sn` in `S` (`E:{a/b}` is the expression `E` with `b` substituted for `a`. The substitutions are applied in a slightly non-standard way — see Chapter 7 and section 5.4 below). This seems to be sufficient to cover the range of examples treated by Dalrymple, Shieber and Pereira (1991), but that is a specific linguistic claim about verb phrase ellipsis in English and not central to the present chapter. Ellipsis is further discussed in Chapter 7.

5.4 Semantics for QLF

In this section we outline the semantics of the QLF language in a way that is as close as possible to classical approaches that provide the semantics in terms of a function from models to truth values. The main difference is that denotation functions will be partial functions for some unresolved QLF formulas, reflecting the intuition that these are 'partial interpretations'. The denotation of a QLF expression will be extended monotonically as it is further resolved, a fully resolved formula receiving a total function as its denotation. The semantics is not intended to describe the resolution process.

Before giving evaluation rules for the QLF language, we first present a simplified version of the semantics for fully instantiated QLF expressions. This is for expository purposes only; the full QLF semantics does not depend on the simplified version.

5.4.1 Simplified Semantics

We will use the notation $[[\texttt{E}]]m$ for the truth value of an expression `E` with respect to a model m (but will leave m implicit). m includes an interpretation function I for mapping constants and predicates into domain individuals and relations. Also left implicit is a function assigning values to variables, which is required for the evaluation of lambda abstracts as characteristic functions.

Constructs in the 'standard' predicate logic subset of QLF receive their semantics with the usual evaluation rules, for example:

- $[[\texttt{P(a1,...,an)}]] = 1$ iff $\mathrm{I}(\texttt{a1}) \ldots \mathrm{I}(\texttt{an})$ are in the relation $\mathrm{I}(\texttt{P})$, and 0 otherwise.

- [[and(F1,F2)]] = 1 iff [[F1]]=1 and [[F2]]=1, and 0 otherwise.

The evaluation rule for a formula F with a scoping variable instantiated to [I,J,...] and containing a term T=term(I,C,R,Q,A) is as follows:

- [[[I,J,...]:F]]=1 iff [[Q(R',F')]]=1, and 0 otherwise, where
  R' is X^(and(R(X),X=A))[I/X], and
  F' is X^([J,...]:and(F,X=A))[T/X, I/X]

This evaluation rule states that a formula with a scoping constraint list may be evaluated by 'discharging' the term for the first index on the list with respect to a formula with a reduced scoping constraint. The rule discharges the term by abstracting over occurrences of the term and its index, and applying the generalized quantifier Q to the term's restriction and the abstract derived from the formula. In Section 5.4 we will say more about the ramifications of adopting this type of quantifier evaluation rule. Note that this rule is also applicable to resolved terms such as pronouns for which Q has been resolved to exists and T is a constant or a scoped variable.

The denotation assigned to a resolved formula form(C,R,F') in which the resolution variable has been instantiated to a formula F' is simply:

- [[form(C,R,F')]]=1 iff [[F']]=1, and 0 otherwise.

Note on Substitutions

The substitution notation,

E[I/X,T/X]

in the semantic meta-language and

E:{I/X,T/X}

in the object language, needs to be read in a slightly non-standard way.

In the usual notation, the substitutions are applied in left to right order to the expression E. That is, first replace all occurrences of I bt X, and then replace all occurrences of T by X. This may lead to problems if I is a subexpression of T (e.g. the index of a term), since after the first set of substitutions is applied, the second is no longer applicable. In the conventional notation, the order of substitutions is important.

In monotonic interpretation, the substitutions are to read as directives on semantic valuation. Thus I/X means 'occurences of I are to be evaluated as though they were occurrences of X'. Since semantic evaluation applies recursively over the structure of the QLF, it is the order in which I and T occur in the QLF expression that is important, and not the order in which they occur in the substitution list.

5.4.2 QLF Semantics

As mentioned earlier, the denotation of a formula `F` in the QLF language will be a possibly partial function ([[...]]) from models to truth values. Again we use the notation [[`F`]]m for the truth value of a formula `F` with respect to a model m (explicit reference to a variable assignment function is again suppressed). For interpretation to be monotonic, we want [[`G`]] to be an extension of [[`F`]] whenever `G` is a more resolved version of `F`, and in particular for [[`G`]] to be total if `G` is fully resolved.

We will define [[...]] for QLFs in terms of a relation W between formulas, models and truth values. Evaluation rules will be given for W(`F`,m,v), but since more than one rule may apply (or a rule may apply in more than one way), W will in general be a relation. The relationship between [[...]] and W for a formula `F` is as follows:

- [[`F`]]m=1 iff W(`F`,m,1) but not W(`F`,m,0);
- [[`F`]]m=0 iff W(`F`,m,0) but not W(`F`,m,1);
- [[`F`]]m undefined iff W(`F`,m,1) and W(`F`,m,0).

Henceforth we will leave the model argument m implicit. The evaluation rules for W will generally take the form

W(`F`,v) if W(`F'`,v)

where `F'` contains one fewer unresolved expression than `F` (so that it is possible for the process of rule application to terminate). The use of *if* rather than *iff* in these rules means that it is possible for rules producing more than one value v to apply and hence for [[`F`]] to be partial.

The model provides an interpretation function I mapping constants and predicates to individual and relations. We will also need to assume a relation S(`C`,`H`) (for 'salient') between QLF categories `C` and QLF expressions `H` standing for individuals, quantifiers, or predicates, but the precise nature of the salience relation and the way it changes during a discourse are not important for the evaluation rules for QLF given here. The intuitive motivation for S is that the category in an unresolved QLF expression restricts the set of possible referents for that expression. S is discussed further in Section 5.4. We are now in position to present the evaluation rules, which we number Q1, Q2, etc.

For standard connectives we have the obvious evaluation rules, for example,

Q1 W(`and(F,G)`,1) if W(`F`,1) and W(`G`,1).

Q2 W(`and(F,G)`,0) if W(`F`,0) or W(`G`,0).

Q3 W(`not(F)`,1) if W(`F`,0).

Q4 W(`not(F)`,0) if W(`F`,1).

Two rules applicable to a formula `F` containing a term with uninstantiated referent and quantifier meta-variables:

Q5 W(`F`,v) if W(`F[_q/exists,_r/A]`,v) and W(`R(A)`,1),
where:
`F` is a formula containing the term
`T=term(I,C,R,_q,_r)`, and
`A` is a term such that S(`C`,`A`).

Q6 W(`F`,v) if W(`F[_q/Q, _r/I]`,v),
where:
`F` is a formula containing the term
`T=term(I,C,R,_q,_r)`, and
`Q` is a quantifier such that S(`C`,`Q`).

(The substitutions for the meta-variables `_r` and `_q` are to be read as part of the evaluation rule.)

A rule applicable to a formula `F` in which a (possibly unscoped) quantified term occurs:

Q7 W(`F`,v) if W(`Q(R',F')`,v),
where:
`F` is a formula containing the term
`T=term(I,C,R,Q,A)`,
`R'` is `X^(and(R(X),X=A))[I/X]`, and
`F'` is `X^(and(F,X=A))[T/X, I/X]`.

A rule applicable to a formula with an instantiated scoping constraint

Q8 W(`[I,J,...]:F`,v) if W(`Q(R',F')`,v),
where:
`F` is a formula containing the term
`T=term(I,C,R,Q,A)`,
`R'` is `X^(and(R(X),X=A))[I/X]`, and
`F'` is `X^([J,...]:and(F,X=A))[T/X, I/X]`.

We also need a trivial rule for a formula with an uninstantiated scoping constraint so that it reduces to application of other rules:

Q9 W(`_s:F`,v) if W(`F`,v).

Two rules are applicable to `form` expressions, corresponding to the cases of an uninstantiated or instantiated resolution meta-variable:

Q10 W(`F`,v) if W(`F[_r/R(P)]`,v)
where:
`F` is a formula `form(C,R,_r)`
`P` is a predicate such that S(`C`,`P`).

Q11 $\mathcal{W}$(`form(C,R,F')`,v) if $\mathcal{W}$(`F'`,v)
where `F'` is a QLF formula.

In a more complete description of the semantics we would also have to state that the evaluation rules provided give the only way of determining membership of the relation $\mathcal{W}$.

5.5 Notes on the Semantics

Monotonicity: We are using monotonicity in two senses which (by design) turn out to be consistent. The first is a syntactic notion for QLF representations (instantiation rather than destructive manipulation), while the second is semantic:

1. `F1` is a more resolved version of `F2` if `F1` can be obtained by instantiating zero or more meta-variables in `F2`.

2. `F1` is a less partial interpretation than `F2` if [[`F1`]] is an extension of [[`F2`]].

The claim of monotonicity for QLF is that for formulas `F1` and `F2`, if `F1` is a more resolved version of `F2` then `F1` is a less partial interpretation than `F2`.

Scoping Constraints: The quantification rules, (Q7) and (Q8), (i) select a term from a formula, (ii) discharge all occurrences of the term and its index in the formula and the term's restriction, replacing them by a variable, and (iii) apply the term's quantifier to the discharged restriction and formula. The difference between (Q7) and (Q8) is simply that the latter also discharges the head of the scoping list, in this case by removing it rather than by replacing it. (Keep in mind that the discharge and replacement operations take place at the level of the evaluation rules for QLF; they are not applied to the QLF expressions representing natural language meanings themselves).

As with Lewin's scoping algorithm, (Lewin 1990), there are no constraints built explicitly into the QLF semantics on where a quantification rule for a term may be applied, or indeed on the number of times it may be applied. However, several constraints arise out of (a) the *absence* of any semantic rules for evaluating isolated `terms`, term indices or scope lists, and (b) the requirement that a `term` be selected from a formula so that its quantifier is known.

The emergent conditions on legitimate scoping are

1. No term may be quantified-in more than once: The first application of the quantifier rule discharges the term. Subsequent applications of the rule lower down in the evaluation would fail to select an undischarged term.

2. When a term's index occurs in a scope list, the quantifier rule for the term must be applied at that point: It must be applied to discharge the head of the scope list, and by (1) above cannot additionally be applied anywhere else.

3. All occurrences of a term's index must occur within the scope of the application of the term's quantifier rule: The quantification rule will only discharge indices within the formula to which it is applied. Any occurrences of the index outside the formula will be undischarged, and hence unevaluable.

4. If a term R occurs within the restriction of a term H, and R is to be given wide scope over the restriction, then R must also be given wide scope over H: Otherwise, suppose H is given wide scope over R. Term H will first be discharged, replacing the term, and with it its restriction, in the formula to which the rule is applied. Then the quantification rule for R needs to be applied to the discharged formula, but the formula will not contain an occurrence of the term R, making the rule inapplicable.

The last two constraints have often been attributed to restrictions on free variables and vacuous quantification. The attribution is problematic since open formulas and vacuously quantified formulas are both logically well defined, and without suspect appeal to the syntax of the logical formalism they cannot be ruled out as linguistically ill-formed. By contrast, QLF makes these violations semantically unevaluable.

Unscoped Terms: When a term's index is not mentioned in any scope list, the term may be quantified in at any point within the formula. For anaphoric terms whose referent has been resolved to some individual constant, it does matter where the quantification rule is applied; since the term existentially quantifies over things identical to a single object, the scope of the quantification is immaterial. It is thus convenient to leave anaphoric terms like this unscoped in QLF. Although this makes the QLF look (syntactically) as though it is not fully resolved, semantically it is. For other unscoped terms, alternative applications of the quantifier rule may well lead to distinct truth conditions, and in these cases the QLF is genuinely unresolved.

Context Dependence: Fully resolved QLFs are context-independent in the same sense that holds for closed formulas in traditional predicate logic (i.e. if the interpretation of the constant symbols in the language is fixed). Unresolved QLFs behave more like open formulas, and there is an analogy between assignments to unbound variables in predicate logic and possible resolutions of meta-variables admitted by the salience relation S. S(`C`,`H`) should be thought of as providing QLF expressions whose denotations are possible referents for unresolved expressions with category `C`. (It would have been possible to define S as a direct relation between categories and referents, but this complicates the statement of its role in resolution and in the semantic definitions.) We used S above in the definition of QLF semantics, but it is also central to NL processing: being able to compute S can clearly play an important role in the process of reference resolution during NL interpretation and in the process of building descriptions during NL synthesis. (The computational analogue of S was implemented as a collection of 'resolution rules' in Alshawi 1990.)

An important question is what to allow as possible expressions in the range of S. One observation is that as the range is widened, more NL resolution phenomena are covered. A rough summary is as follows:

- constants: intersentential pronouns
- predicate constants: compound nouns, prepositions
- quantifiers: vague determiners
- indices: bound variables, intrasentential pronouns
- predicates built from NP restrictions: one-anaphora
- predicates built from previous QLFs: intersentential ellipsis
- predicates built from current QLF: intrasentential ellipsis

5.6 Related Approaches

Viewed from a slightly different perspective, monotonic interpretation has a number of points of contact with Pereira's categorial semantics (Pereira 1990). Put briefly, in categorial semantics, semantic evaluation is represented as deduction in a functional calculus that derives the meanings of sentences from the meanings of their parts. Considerable emphasis is placed on the nature of these semantic derivations, as well as on the final results of the derivations (the 'logical forms' of sentences).

One significant advantage of this approach is that constraints on legitimate scoping emerge naturally from a consideration of permissible derivations of sentence meaning, rather than arising artificially from syntactic constraints imposed on logical forms. Derivations involving quantified terms first introduce an assumption that allows one to derive a simple term from a quantified term. This assumption is later discharged by the application of a quantifier. Conditions on the appropriate introduction and discharge of assumptions in natural deduction systems impose restrictions on the way that quantifiers may legitimately be applied. For example, a quantifier assumption may not be discharged if it depends on further assumptions that have not themselves been discharged. This prevents the occurrence of free variables in logical form, but without appeal to the syntax of logical form.

The discharge of terms and term indices when evaluating QLF closely parallels the discharge of quantifier assumptions in categorial semantics. Indeed, the terms and the indices are precisely the assumptions introduced by quantified expressions, and which need to be discharged. Furthermore, the different orders in which quantifier assumptions may be discharged in categorial derivation correspond to the choices that the quantifier rules permit for discharging quantified terms.

Where monotonic interpretation and categorial semantics part company is on the degree of explicitness with which semantic derivations are represented. In categorial semantics, derivation is a background process that builds up logical forms, but is not explicitly represented in the semantic formalism. By contrast, the annotation of QLFs with scope lists provides an extra level of information about how the derivations proceed. In particular, they indicate which evaluation rules should be applied where.

QLF thus provides a (usually partial) specification of a semantic derivation, showing (a) what the initial 'premises' are (roughly, lexical meanings, although these too may only be partially specified), and (b) the rules by which the 'premises' are combined. QLF resolution amounts to further instantiating this specification. This view of QLF can be contrasted with Logical Form as it is normally understood, which represents the results of carrying out a semantic derivation.

The difference between specifying a derivation and carrying it out is what makes resolution order independent in monotonic interpretation. Making a resolution to QLF only specifies when and how an expression should be evaluated during semantic derivation; it does not carry out that part of the derivation. Where no distinction is drawn between making a resolution and carrying out the corresponding step of the derivation, the order of resolution can be important. Thus, for Dalrymple, Shieber and Pereira (1991), where this distinction is not drawn, the precise interleaving of scope and ellipsis resolution determines the interpretation of the sentence. In QLF, resolutions dictate the order in which various steps of the derivation are carried out, but the resolution order does not reflect the derivation order.

Distinguishing between specifying and performing a derivation also means that a monotonic treatment of ellipsis resolution does not need to resort to higher-order unification. Dalrymple, Shieber and Pereira use higher-order unification to 'unpick' the composition of constituent meanings obtained in the semantic derivation from the ellipsis antecedent. Some of these meanings are then put back together to produce a predicate that can be applied to the ellipsis arguments. Since monotonic resolution does not carry out the final composition of meanings, but merely sets out conditions on how it is to take place, there is no need to unpick composed meanings and put them back together again.

It is worth pointing out that monotonic interpretation is compatible with approaches to meaning as a transition between contexts or information states, and where the order in which transitions are made is significant (e.g. Veltman 1991). In such a framework, monotonic interpretation would amount to making decisions about which transitions to take when, but would not involve putting those decisions into action. The monotonicity in monotonic interpretation thus refers to the way in which alternative derivations of sentence meanings may be chosen, but not to the semantic effects of those sentence meanings.

5.7 Implementation Benefits

A description of the language processing mechanisms to which we have applied the monotonic semantics model is beyond the scope of this chapter. However, we believe that the QLF representation presented here brings significant advantages to implementing mechanisms for reference resolution, scoping, preference and generation.

Reference and Scoping: The order independence of resolution operations allows for a variety of control structures in implementing a resolution mechanism. We find it convenient to make a bottom up pass through QLFs making reference resolutions, followed by a stage of scoping resolution, and to iterate over this should any of the resolutions introduce further unresolved expressions.

The salience relation S can be implemented as procedures that search for properties, objects or indices in context. Scoping proceeds simply by the non-deterministic instantiation of scoping constraints, subject to the restrictions imposed on evaluable QLFs (Section 5.4), plus techniques for ignoring logically equivalent scopings, as for example described by Moran (1988).

Preference and Disambiguation: A resolved QLF preserves all the information in the original unresolved QLF, and also records the correspondence between resolved and unresolved expressions. This makes it possible to define preference metrics that can be used for ranking alternative interpretations independently of the search strategies used to derive them. For example, in the case of scoping, these metrics can combine information about how far a quantifier was 'raised' with information about the surface form of its determiner. Preference ranking over alternative resolutions facilitates automatic disambiguation of input. Interactive disambiguation can make use of generation from resolved QLFs for confirmation by a user.

Generation: There is a strong connection between monotonicity and reversibility in language processing systems. Monotonicity of unification means that algorithms such as head-driven generation (Shieber *et al* 1990) can be applied to grammars developed for analysis. We use a variant of this algorithm for generating from QLFs, and the monotonicity of semantic interpretation means that the grammar used for generating from unresolved QLFs (the normal 'output' of the grammar) can also be used for generation from resolved QLFs.

In parallel to the distinction between grammatical analysis (of NL into unresolved QLFs) and interpretation, we make the distinction between grammatical synthesis (of NL from QLFs) and *description*. Description is the process of deriving a QLF from which synthesis proceeds by taking a fact (e.g. a database assertion) as input. However, one of the principles of QLF-based description is that while interpretation instantiates referent fields in underspecified QLFs, description involves instantiating category and restriction fields for QLFs in which referent fields are already instantiated. The preference metrics applied to rank alternative interpretations can be applied equally well to ranking resolved QLFs produced by a nondeterministic description process, so there is a sense in which the preference mechanism can also be made reversible. This matter is further discussed in Chapter 8.

Chapter 6

The Domain and Context Models in Interpretation

In the language understanding model embodied in CLARE, semantic processing is divided into *analysis* and *interpretation*. The first corresponds to the semantic contribution of the grammar, and yields contextually underspecified QLF (Quasi Logical Form) representations. Semantic interpretation further specifies these QLF representations through the processes of scoping and reference resolution. This interpretation process can take into account linguistic and nonlinguistic aspects of context, including the domain model described in the previous chapters of this report. In this chapter we describe the role played by the domain model in interpretation.

This chapter is organized as follows. In section 6.1 we briefly describe the context model, which is the part of the domain model that records information about the utterances made by the user and the CLARE system. Section 6.2 presents an abstract characterisation of reference resolution in terms of abductive equivalential translation, using the domain model as the background theory. Sections 6.3 and 6.4 then describe a number of resolution methods and preferences that require inference using the domain model. Finally, section 6.5 explains how the domain model is sometimes also used when resolved QLFs are evaluated in terms of TRL expressions.

6.1 The Domain and Context Models

The context against which any natural language utterance is interpreted includes both non-linguistic facts about the world (e.g. that two entities stand in a certain relation to one another) and linguistic facts about the world (e.g. that the previous utterance in the discourse was made at a certain time and had a particular surface form). For convenience we can divide the overall context into (1) the context model more narrowly defined, which records facts about the utterances in the discourse, and (2) the domain model, which records other facts about the world.

Domain models have been described in some detail in earlier chapters. Here we will only note that it is not strictly accurate to characterise the domain model as capturing all and only the non-linguistic aspects of context. From at least one perspective, the definitional equivalences contained in a linguistic domain theory can be seen as a way of relating linguistic predicates / word senses to the underlying model. However, it will do us no harm here if we talk of the domain model as though it dealt only with non-linguistic facts.

Context Model A variety of facts about utterances are recorded in the context model. Each utterance is indexed by a unique identifier, and the information recorded includes

- The surface form of the utterance (a word string)
- The (resolved) QLF assigned to the utterance
- The time at which the utterance was made
- Who made the utterance

The QLF formalism also lends itself to recording further information about the constituents of an utterance. Individual noun phrases within the utterance correspond to uniquely indexed `term`s within the utterance's QLF, and so in the context model extra information is associated with `term` indices. Of importance below (section 6.5) is the way that `term` indices can be associated with internal domain model names (skolem constants) for entities newly introduced in the domain model.

The context model is used in three main ways. (1) To provide information necessary for certain kinds of reference resolution. For instance, records of QLFs for previous utterances are used to extract possible antecedents for ellipses (Chapter 7). Similarly, a list of salient `term`s or entities is maintained, which is exploited in the resolution of anaphora. (2) To answer certain kinds of meta-question, such as *How many questions have you been asked? How many have you answered? Who asked them?*. (3) To assist in the evaluation of (resolved) QLFs in terms of TRL expressions.

6.1.1 Reverse Compilation of Linguistic Domain Theories

The Linguistic Domain Theory (LDT, Chapter 3) is the part of the domain model that links linguistic predicates to domain predicates. For converting resolved QLFs to TRL domain queries and assertions, this link is exploited in the linguistic to domain direction. Since the linkage is expressed by means of equivalences, it is also possible to use it in the opposite direction, to go from domain predicates to linguistic predicates. The ability to use the linkage in the reverse direction is useful both for certain aspects of reference resolution, and in the generation of English sentences from domain facts (section 8.3).

In practice, linkages in the domain to linguistic predicate direction are obtained by compiling out the LDT to obtain the following:

TRL Lemmas

These are implications of the form

```
Conditions -> (TRLPredications -> QLFPredication)
```

Subject to the `Conditions` holding, this says that whenever the TRL predications hold, the QLF predication holds. As an example, we might have

```
true ->
 and(project(C,B,D,E,F),transaction(G,H,A,I,J,D,K))
 -> on(A,B)
```

which says that whenever `A` and `B` occur in the `project` and `transaction` (conceptual) domain relations as shown, then linguistically `A` may be said to be *on* `B` (where `A` is a cheque and `B` is a project).

As an example of a lemma with non-trivial conditions, we might have

```
 and(SF(amount(I,sterling_unit))=A,
 and(SF(amount(J,sterling_unit))=H,
 and(safe_<(K,0),
     measure_diff(I,J,K))))
->
 transaction(C,D,B,E,F,G,H) -> under(A,B)
```

This says that (a cheque) `B` may be described as being *under* an amount `A`, provided that `A` is an amount of money, and that the amount the cheque is for, `H`, is less than `A`.

Compilation The TRL Lemmas form the backbone of the reverse compilation of the linguistic domain theory. Other compiled implications and relations are derived in part from TRL lemmas plus other LDT relations. Compilation of the TRL Lemmas is described in Chapter 9.

Sort Declarations

For each argument to a domain relation, it is useful to identify a set of linguistic predications that can be applied to that argument. This information is already implicit in the TRL lemmas, since there will be lemmas like

```
true -> transaction(C,D,B,E,F,G,H) -> cheque_Document(B)
```

Lemmas like these can be regarded as sort declarations.

Scope Functional Constraints

The LDT contains a number of functional declarations, e.g. in the PRM domain it is declared that there is a functional relation from projects to their start dates. This functional information is expressed in terms of relations between arguments of domain predicates. However, it is also useful to have the same information expressed in terms of linguistic predicates.

Linguistic functional information may be obtained by (1) finding lemmas that connect functionally related domain arguments, (2) determining the linguistic sorts on those arguments, and (3) creating a function declaration that state that arguments to the relevant linguistic predication are functionally related provided that the appropriate sortal predicate apply.

As a concrete example, suppose we have

```
function(project(A,B,C,D),([A]-->[C])

true -> project(A,B,C,D) -> start_date_of(A,C)

true -> project(A,B,C,D) -> project_Activity(A)

true -> project(A,B,C,D) -> date_Time(C)
```

We can then create a new, purely linguistic, functional declaration as follows:

```
and(project_Activity(A),date_Time(C))
->
 function(start_date_of(A,C),([A]-->[C]))
```

6.2 Reference Resolution and Abductive (Equivalential) Translation

The category of a QLF `term` or `form` expresses a relation between context, the `term` or `form`'s restriction, and the `term` or `form`'s resolvent(s). We can express this more formally in terms of abductive equivalential translation, as described in Chapter 2.4.

The basic idea behind using the translation schema for interpretation is that interpretation is a process of reducing the underspecification in a formula `Q` to produce a (possibly fully specified) formula `Q'`. In other words, we want `Q'` to be semantically subsumed by `Q` such that by suitably restricting the set of models under consideration (through knowledge of the context, meaning postulates and possibly additional assumptions made during interpretation), `Q'` can then be made equivalent to `Q` with respect to this restricted set of models.

This leads to the following characterization of interpretation in terms of the translation schema: Given `Q`, a possibly underspecified formula corresponding to a speaker utterance, and a context `C`, determine `Q'` and `A` such that

$$C \cup A \Rightarrow (Q \leftrightarrow Q')$$

where `Q'` is syntactically subsumed by `Q` and `A` is a set of assumptions consistent with `C`. For a plausible interpretation we wish to find fully resolved solutions `Q'` which "minimize the cost" of the assumptions `A`.

As an example of this general schema in operation, consider the resolution of a sentence like *The prototype disintegrated*:

```
context(i1,C) /\ most_salient(i1,C,X^prototype(X),zed9) ... =>
    disintegrate(term(i1,<det=the>,X^prototype(X),_,_))
    <->
    disintegrate(term(i1,<det=the>,X^prototype(X),exists,zed9)).
```

Here, `C` is the context that is current for the utterance of the `term i1`. Relative to this context, we are supposing the `zed9` is the most salient prototype. This supposition may either (a) already follow from the context `C`, or (b) be an extra assumption that we make about the context. The latter case shows our treatment permits a 'relational' account of meaning — interpreting an utterance may serve to fix the context as much as it fixes the meaning of the utterance. Thus, if it is not already the case that `zed9` is the most salient prototype in `C`, at the cost of making an assumption we can resolve the `term i1` to refer to `zed9` and force it to be the most salient prototype (the cost of the assumption will obviously depend on how salient `zed9` already is in context `C`).

For reference resolution solely in the interpretive direction (i.e. where we are not interested in driving resolution backwards for the sake of synthesis and generation), it is convenient to reduce the equivalential schema to Horn clauses. The Horn clause relating specifically to the resolution of the definite `term` above would be something like

```
context(I,C) /\ det(I,C,the) /\ restriction(I,C,R)
             /\ most_salient(I,C,R,Rft)
   -> referent(I,C,Rft)
```

Here, the functions `det`, `restriction` and `referent` serve to identify the different components of the `term` indexed by `I`, and `context` and `most_salient` are as before. Note that the Horn clause formulation is entailed by the equivalence, but not vice versa.

The predicate most_salient is in fact a *resolution method*, suitable for the resolution of definite noun phrases. In practice, the resolution method will be implemented as a Prolog procedure, which may well call upon the services of the domain model inference engine. Also, Horn clause axioms of the form shown above are in fact represented in a short-hand form as reference rule declarations. These declarations, of the form

```
ref_rule(term(I,Cat,Restriction,Quant,Referent),
         Method).
```

associate resolution methods, like most_salient with terms (or forms) of various categories. Several different resolution methods may be associated in this way with a single category. When this is so, the relation the category expresses between the term or form's restriction and the context is the union of the relations expressed by the different resolution methods.

With the general translation schema for term referents in place, we can also slightly reformulate the QLF evaluation rules for unresolved terms (Chapter 5). The revised version of the valuation rule is as follows (the valuation relation V has the same arguments as W but also takes an assignment g):

> *V*(F(...term(I,R,exists,_)...),m,g,v) if
> g(_)=A and
> *V*(F(...term(I,R,exists,A)...),m,g,v) and
> *V*(referent(I,A),m,g,1).

This valuation rule together with the definite description postulate given above allows the solution presented earlier for the interpretation of *The prototype disintegrated.* This approach has the advantage of making notions like salience part of the linguistic theory of reference rather than of the valuation rules for the logic.

One can also apply abductive translation to the resolution of quantifier scope. Consider a sentence like *A nut secures every bolt*, and the following translation schema:

```
context(U,C) /\
function(C,X^Y^and(nut(X),and(bolt(Y),secures(X,Y))))... |=
      _:secures(term(i2,<det=a>,X^nut(X),_),
                term(i3,<det=every>,Y^bolt(Y),_,_))
       <->
[i3,i2]:secures(term(i2,<det=a>,X^nut(X),exists,i2),
                term(i3,<det=every>,Y^bolt(Y),forall,i3)).
```

Here, U identifies the utterance (U will normally be the index of a form);

```
function(X^Y^p(X,Y))
```

is to be read as meaning

```
forall([X,Y1,Y2],
   impl(and(p(X,Y1),p(X,Y2)),
       Y1=Y2))
```

The function assertion captures the domain fact that a nut cannot secure more than one bolt. This, together with a cardinality assumption saying there is more than one bolt (left implicit here, and implied by the use of the determiner *every*), forces the universal to take wide scope. The reverse scoping would violate the functional constraint by having at least one nut that secures all the bolts.

Functional constraints are in practice also applied in an implicational rather than equivalential form. Moreover, they are generally applied in the contrapositive direction in order to rule out scoping that violate functional constraints rather than promote scopings that obey them. The details of this are given in section 6.4 below.

6.3 Domain Specific Resolution

Deciding whether a particular resolution method relates a QLF index to a referent via the context requires a certain amount of inference. In some cases the amount of inference required may be relatively small; for example, most ellipsis resolution (see Chapter 7) requires only identifying an antecedent QLF and deciding on a range of substitutions to be made to it. In other cases, however, more extensive inference using the domain model is required. In this section we discuss four cases where more extended inference is required: anaphora resolution, compound noun and prepositional phrase resolution, scoping preferences, and general domain preferences.

6.3.1 Anaphoric and Definite Noun Phrases

To resolve an anaphoric or definite noun phrase, one needs to find an entity (or set of entities) that are salient in the current context, and which satisfy some minimal description obtained from the `term`'s restriction. Entities may be salient in context in the following ways: (a) they have recently been referred to by some `term` in the discourse, (b) they are functionally related to some previously mentioned entity (unimplemented), (c) they are the only objects in the domain having a certain property.

Pronouns Pronouns like *it, he* or *she* look for recently mentioned entities which satisfy the properties `impersonal`, `masculine` and `feminine` respectively. Which entities have recently been mentioned is inferrable from the context model simply by looking through the most recently uttered `terms` as recorded there. A limited amount of inference using the domain model is also required to determine whether the requisite properties apply, in a way now described.

At the QLF level, pronoun resolutions are expressed by means of co-indexing QLF `terms`. For a discourse like "*John walked. He talked.*" we might therefore have the following pair of QLFs (simplified)

```
walk(term(i1,proper_name,X^name_of(X,'John'),exist,ent(john_smith)))

talk(term(i2,he,Y^masculine(Y),exists,strict(i1)))
```

For the resolution of the second QLF to be acceptable, we must establish that the entity referred to by the `term` `i1` satisfies the property `Y^masculine(Y)`. Given that `i1` is a directly referential `term`, there are two alternate queries that may be addressed to the inference engine:

```
masculine(john_smith)

forall([X],impl(name_of(X,'John'),masculine(X)))
```

(note that the queries addressed to the inference engine are first converted from QLF to TRL).

In a discourse like "*A man walked. He talked.*" the first query above cannot be formulated, since *a man* will not be resolved to refer to a specific named entity. However, as discussed below, the indefinite antecedent will lead to the introduction of some (possibly) new entity into the domain model, having its own internal name. Direct use of internal names are not permitted in QLF, in part because internal names keep changing. However the domain model does maintain a list of assertions of the form

```
ctx_idx_assoc(Idx,IntName)
```

which associates `term` indices (`Idx`) with internal names for objects (usually skolem constants or functions). So in this case, there are again two queries that may be tried:

```
ctx_idx_assoc(i1,X) /\ masculine(X)

forall([X],impl(man(X),masculine(X)))
```

(In fact the first query generalizes the more specific query `masculine(john_smith)` above, since in that case, the domain model will imply `ctx_idx_assoc(i1,john_smith)`).

Pronouns may also be used in a non-referential way, e.g. *John gets a lot of headaches, and I get them too.* Here, the pronoun *them* does not refer to the headaches that John gets but just to headaches in general. In cases such as these, the pronoun is resolved in such a way that it quantifies over objects satisfying the same restriction property as the antecedent (or a more general description entailed by the antecedent's restriction), e.g.

```
term(i4,them,X^entity(X),exist,sloppy(i3,Y^headache(Y)))
```

Strictly speaking, two queries should be addressed to the inference engine:

```
and(restriction(i3,X^P),
    forall([Y], impl(P(Y), headache(Y))))

forall([Y], impl(headache(Y),entity(Y)))
```

In practice, only the second (trivial) query would be used, since the resolution property is normally obtained by removing conjuncts from the antecedent restriction, syntactically ensuring the first entailment. As it is, this way of resolving pronouns is not currently implemented within CLARE.

Definite Noun Phrases One way of resolving definite noun phrases is very similar to that for pronouns: select a salient antecedent `term`, and ensure that any entities it refers to are compatible with the restriction of the definite `term`.

When the antecedent refers to a single named entity, as before one can simply apply the definite restriction to the entity and check against the domain model that the resultant predication is true. Thus, for *John walked. The man talked.*, we would need to establish that

```
man(john_smith)
```

But when the antecedent does not refer to a single named entity, the compatibility check needs to be more elaborate.

In essence, we need to check that a full description of the antecedent is not incompatible with the restriction of the definite `term`. By full description, we mean not just the description given by the restriction of the antecedent `term`, but a more specific description of the `term` as it occurs in its sentence. To illustrate why a full description is needed, consider *Six people work on CLARE. The people who do not work on CLARE work on BCI.* Obviously, we would not wish to resolve the definite NP so that it refers to (some of?) the six people who work on CLARE. Yet if we just look at the antecedent's restriction (i.e. people) there is nothing to prevent such a resolution. The full description of the antecedent is 'people that work on CLARE', which is clearly incompatible with the description 'people who do not work on CLARE'.

For a fully correct treatment of definites, two queries would be required (here `A` stands for the full antecedent description, and `R` for the restriction of the definite term

```
not(forall([X], impl(A(X), not(R(X)))))

forall([X], impl(R(X), A(X)))
```

The first query ensures consistency, and the second may be used to favour resolutions where the definite term restriction is as specific or is more specific than the antecedent description. The preference query is required in cases such as *One man walked and another man ran. The man that walked whistled* — whistling is not inconsistent with running.

In practical terms, however, these queries often turn out to be excessively complex to answer, especially the consistency checking. Instead, much simpler queries are formulated. The head predicates (corresponding to the head nouns) for the definite and antecedent are retrieved, and it is checked that the antecedent head entails the definite head. This is error prone when the definite is modified, e.g. by a relative clause, but in such cases there is a general preference in favour of attributive rather than anaphoric resolutions.

Singular definites may also be resolved referentially if there is a single object in the domain satisfying the definite restriction. For example *the start date of CLAM-BAKE* may be resolved to a particular date, even if that date has not previously been mentioned. In cases such as these the definite `term`'s restriction is used to formulate a query

```
extension_of(R,Ext)
```

that determines the extension `Ext` of the `term` restriction `R` in the domain model. If there is just a single object in the extension, this may serve as the referent.

Functional resolutions of definites are not currently implemented. To do so would require using functional declarations in the LDT to link previously mentioned entities to functional dependents satisfying the restriction of the definite `term`.

In formulating queries for the resolution of definites, it is necessary to ensure that the `term` restriction is fully resolved and scoped, and then translated into TRL. (Though partially resolved QLFs do have well defined truth conditions, no proof theory for them has yet been worked out).

6.3.2 Compound Nouns and Vague Relations

In the PRM domain, one would expect the following different resolutions for two syntactically and semantically similar compound nouns

company payments ⇒ payments to companies
project payments ⇒ payments on projects

The reason for the difference is that the domain model determines that (a) the relation between companies and payments is one where the payments are made to the companies and (b) the relation between projects and payments is one where the payments are made on a project account.

Information encoded in the domain (in non-linguistic form) is thus used to resolve compound nominals like the ones above. In brief, the procedure is as

follows.
(1) The head property of the compound noun (i.e. payments in the case above) is used to retrieve, via sortal TRL lemmas, the kind of domain relations that payments can occur in.
(2) Chaining declarations, which specify how various domain relations intersect, are then used to select further domain relations that may be connected to the original.
(3) Sortal TRL lemmas are then used again to determine which arguments on the connected relations could be described by the other property mentioned in the compound nominal (i.e. projects or companies).
(4) At this point, one will have identified one or more two place domain relations, e.g.

```
A^B^and(transaction(A,_,_,_,C,_,_),project(B,C,_,_))
```

The TRL lemmas can once more be consulted to determine what linguistic predicates (if any) correspond to these properties.

Vague preposition senses may be resolved in exactly the same way. The only significant difference is that in some cases the second argument to the preposition may be a referential `term` (e.g. for a proper name). Here, the domain model has to be consulted directly to see which relations that named entity may occur in, rather than using sortal TRL lemmas.

6.4 Functional Information and Scoping

The domain model contains declarations concerning functional relations between different objects. For example, a project might uniquely define a start and end date. This information can be used to rule out linguistically possible quantifier scopings that are impossible in the domain. A query like *List every start date of a project* has two linguistically possible scopings: (a) for one project, list all of its start dates, and (b) list all the dates on which some project started. Given the functional relation between projects and their start dates, only the latter reading is plausible.

In the functional relation above, projects are the *domain* of the function, and start and end dates the *range*. We need to distinguish between *iterative* quantifiers (universals, distributive plurals) and non-iterative quantifiers (existentials, plural collectives). We also need to be able to say whether one term originates from the restriction of another, i.e. whether they are in a *head–restriction* relation. We can then specify functional constraints on scoping between two terms as follows.

1. Domain and range terms not in head–restriction relation

    (a) Iterated range term within scope of domain term disallowed

(b) Iterated range term with scope over iterated domain term disallowed

2. Domain term in restriction of range term

   (a) Iterated plural range term within scope of domain term disallowed. (Universals allowable, since they can be used to refer to single object, though this is dispreferred. Numerical quantifiers etc. disallowed).
   (b) Iterated range term with scope over iterated domain term disallowed

3. Range term in restriction of domain term

   (a) Iterated range term within scope of domain term disallowed
   (b) Iterated range term with scope over iterated domain term *is* allowed.

Examples of scopings affected by these restrictions are

A project started on every date disallowed if *a project* takes wide scope. (Violates 1a)

Every start date of a project dispreferred if *a project* takes wide scope. (Dispreferred by 2a)

Three start dates of a project disallowed if *a project* takes wide scope. (Violates 2a)

A project starting on every date disallowed if *a project* takes wide scope. (Violates 3a)

Every project started on every date disallowed on any scoping. (Violates 1a or 1b)

Every start date of every project disallowed if *every start date* has wide scope. (Violates 2b).

Every project starting on every date disallowed if *every project* takes wide scope (violates 3a), but OK if *every date* has wide scope (3b).

Every project started on a date is OK under any scoping, though giving *a date* wide scope is dispreferred on other grounds.

A start date of every project is OK under any scoping, though giving *a date* wide scope is dispreferred on other grounds.

Every project starting on a date is OK under any scoping, though giving *a date* wide scope is dispreferred on other grounds.

The constraints on functional relations are applied during preference ranking of scoped forms (Chapter 10). In order to establish whether a functional restriction applies to a given predication, a query to this inference engine is constructed as follows:

(1) all one-place linguistic predications applying to the predication's arguments in the QLF are conjoined with the predication itself.
(2) A query is produced to see if this conjunction implies the conditions on the compiled scope function declaration for the predication.

6.5 The Context Model in Evaluating Resolved QLFs

As well as providing information necessary for the resolution of certain QLF expressions or determining the preference rankings for alternate resolutions, the context model also plays an important (though limited) role when it comes to evaluating QLFs through conversion to TRL expressions. This applies particularly to the way that anaphorically resolved `term`s are evaluated, and in this connection two pieces of contextual information recorded about (antecedent) `term`s are of importance — the `term` index's domain model association, and the full description associated with the index.

6.5.1 Index–Domain Entity Associations

The role of index associations is best understood by considering the effect that assertions involving indefinite noun phrases have on the domain model. An assertion like

A woman works on FOO.

will, after resolution and conversion to TRL, lead to the addition of skolemised clauses to the domain model, e.g.

```
CRI_PROJECT_MEMBER(sk(89),FOO).
CRI_EMPLOYEE(sk(89),w).
```

The skolem constant, `sk(89)`, represents an internal name that the domain model assigns to the as yet unidentified woman.

The assignment of an internal name to the referent of the indefinite noun phrase occurs only after the resolution of the QLF for assertion. At the QLF level, all we have is some `term` with a uniquely identifying index. It is important to associate the QLF `term` index with the internal domain model name.

The necessity for this association can be seen if we consider a follow-on assertion like

She also works on CLAM-BAKE.

At the QLF level, the pronoun will be resolved so that it is co-indexed with the antecedent `term`:

```
term(<pro=she..>,idx77,X^[female,X],exists,strict(idx73))
```

where `idx73` is the index of the `term` corresponding to *a woman*. The QLF resolution makes no reference to the internal names used in the domain model. (This is of vital importance, since the technique de-skolemising and re-skolemising incomplete assertions means that the internal names constantly change; see Chapter 2). However, we need to make sure that when the QLF is converted to TRL we have the following (still incomplete) clauses:

```
CRI_PROJECT_MEMBER(sk(89),CLAM-BAKE).
CRI_PROJECT_MEMBER(sk(89),FOO).
CRI_EMPLOYEE(sk(89),w).
```

so that it is the same woman who works on CLAM-BAKE and on FOO.

To achieve this effect, we proceed as follows. During the conversion of the first assertion to TRL, we will first arrive at an untranslated TRL expression (i.e. before equivalential translation) of the form

```
exists([X],and(and(woman(X),works_on(X,FOO)),
               ctx_idx_assoc(idx73,X))).
```

That is, the QLF `term` interpretation rules ensure that an extra conjunct is added when interpreting all `term`s associating the `term`'s index with the bound variable produced in discharging occurrences of the `term`. After the untranslated TRL undergoes equivalential translation and skolemisation, we will in fact have three clauses that are added to the domain model:

```
ctx_idx_assoc(idx73,sk(89)).
CRI_PROJECT_MEMBER(sk(89),FOO).
CRI_EMPLOYEE(sk(89),w).
```

The first of these has the effect of including information in the domain model about certain aspects of the utterance.

In discharging the resolved pronominal `term` to a TRL quantification, the index association for `idx73` is used as follows. The `term` is strictly co-indexed with `idx73`, which in turn is associated with the skolem constant `sk(89)`. Therefore, in discharging the `term` an extra conjunct is added:

```
exists([X], ...and(and(female(X),eq(X,sk(89))),
                   ctx_idx_assoc(idx77,X))
            ....)
```

After translating, simplif ying identities and re-skolemising all the currently incomplete assertions, this will lead to the required set of skolemised clauses being produced.

Should we make a third assertion such as

Her name is Clara Luser

the way that incomplete assertions are de-skolemised and then re-skolemised would ensure that we end up with the following domain model clauses:

```
ctx_idx_assoc(idx73,Clara Luser).
ctx_idx_assoc(idx77,Clara Luser).
CRI_PROJECT_MEMBER(Clara Luser,FOO).
CRI_PROJECT_MEMBER(Clara Luser,CLAM-BAKE).
CRI_EMPLOYEE(Clara Luser,w).
```

At this point, the internal name for the woman is replaced by a (unique) external name.

In principle, this treatment can be extended to cases where indices are associated with skolem functions, or implicitly universally quantified TRL variables (re-skolemisation of the results would ensure these variables are properly 'bound'). However, this runs into a number of technical difficulties with regard to interpreting collective versus distributive anaphor resolutions, and an alternative approach, described below, is adopted.

6.5.2 Full `term` Descriptions

To understand what is meant by restricted and full descriptions of a `term`, it is easiest to present an example. Consider the noun phrase *a dog that bit him* in the following sentence

Every man kicked a dog that bit him

A restricted description of the entities referred to by this noun phrase is

dogs that bit a man

A full description would be

dogs that bit a man and were kicked by him

The full description can be used to interpret the resolution of the pronoun *they* in

They all ran away

At the QLF level, the pronoun *they* is simply resolved to be co-indexed with the antecedent `term`, e.g. for *a dog that bit him*. Supposing that the antecedent `term` gave rise to a skolem constant associated with the `term` index, then this resolution could be interpreted in the way described in the last section. That is, we refer directly to those entities referred to by the antecedent `term`.

However, in this case the antecedent is associated with a skolem function, and we employ a different method of interpretation. This is simply to impose the full antecedent `term` description as an additional restriction on the range pronominal `term`'s quantification. That is, the sentence is interpreted as meaning something like: the dogs that bit and were kicked by a man all ran away. Here we rely on a sufficiently detailed description to pick up exactly those entities referred to by the antecedent `term`.

Unfortunately, we cannot always guarantee that the description will be sufficiently constraining to pick up exactly the same entities. This is especially pressing with the case of indefinite / existential terms. For example, in a sentence like *A man kicked a dog*, the full description for *a dog* would be 'dogs that were kicked by a man'. Even if we anchor the temporal reference of the sentence down to a particular time, it is quite possible that there is more than one dog that was kicked by a man at that time. But the sentence itself refers to just one man and one dog. Fortunately, it is exactly in cases such as these that the association of indices with skolem constants can be used to interpret subsequent anaphoric references, avoiding the problem of insufficiently constrained descriptions.

When descriptions are used it is because the antecedent `term` either is equal to or is within the scope of some `term` that behaves like a universal quantifier. In such cases, it is much more likely that the `term` description will be sufficiently constrained. So, for *a dog* in *Every man kicked a dog*, the description 'dogs that were kicked by a man' is more likely to be sufficiently constrained, provided that the temporal reference of the sentence and description is anchored down, and provided that any further contextual restrictions on *every man* are also captured in the description. Also, descriptive interpretations are less liable to lead to inaccuracy in evaluating questions than they are for assertions.

A full description for a term `term(i,..X^R..)` in a formula `S(..i..)` is

```
X^and(R'(X),S'(..X..))
```

Here, `R'` is the `term`'s restriction plus any other contextual restrictions imposed by the `term`'s resolution. `S'` is the original QLF with the `term` and all occurrences of its index abstracted over. Furthermore, all universally quantified `terms` with wide scope over `i` are exchanged for existentials. Full descriptions for `terms` correspond closely to Webber's (1983) discourse descriptions.

Chapter 7

Resolution of Ellipsis

7.1 Analysis of Ellipsis

Elliptical input, in the form of fragmentary input and certain coordination constructions, is common in natural language discourse, especially typed interactive discourse of the kind a user of CLARE would engage in, e.g.

> *John, Which house?, Inside, On the table, Difficult to do,*
> *John doesn't, He might not want to.*

Syntactically it is not essential to treat these fragments as instances of S constituents, although for our purposes it is convenient to do so.

Many elliptical fragments can be analysed by allowing an elliptical S to consist of one or more phrases (NP, PP, AdjP, AdvP) or their corresponding lexical categories. Most other commonly occurring patterns can be catered for by allowing verbs that subcategorize for a nonfinite VP (modals, auxiliary *do*, *to*) to appear without one, and by adding a special lexical entry for a main verb *do* that allows it to constitute a complete VP. In our grammar, the latter two moves allow all of the following to be analyzed:

> *Will John?, John won't, He may do, He may not want to, Is he going to? etc.*

At the QLF level, elliptical fragments are analysed as `forms`, where the `form` category gives information about the kind of ellided constituent. For example, for an elliptical NP like "John" we would get the following QLF (square bracket notation used):

John:

```
form(l([John]), ell(np), +e,
     P^[P, term(l([John]), proper_name, +j
                X^[name_of,X,'John'],
```

```
                        _,_)],
        _)
```

To resolve the elliptical `form`, one needs to find a property salient in context, and apply the `form`'s restriction (`P^[P,term(..+j..)]`) to the salient property. Either the property itself or the result of its application is used to instantiate the `form`'s resolvent meta-variable.

For ellipsis resolution thus construed, the hard part lies in determining what counts as a contextually salient property suitable for resolving the ellipsis.

7.2 Ellipsis Resolution

The nature of ellipsis in English is that its resolution is strongly dependent on the local linguistic context, as provided by the current sentence, the preceding sentence, or the preceding exchange in a turn-taking dialogue. The basic claim we wish to defend in this chapter is as follows: The properties suitable for ellipsis resolution can be obtained by the substitution of `terms`, `forms` and their indices in the QLFs of the sentences contained in the local linguistic context.

In some ways this claim is too strong. For example, someone can shout 'Stampede!', and without any prior linguistic context this can be interpreted as meaning 'Look out everybody, there is a cattle stampede'. However, we will ignore here cases of ellipsis with non-linguistic antecedents. More importantly, the claim may be too strong in that there are some cases of ellipsis resolution that go beyond simple substitution of `terms`, `forms` and their indices in QLFs.

However, as we will see below, a substitutional treatment covers a wide range of cases, and with a greater degree of simplicity than many alternative approaches. In particular, while bearing a close connection to approaches using higher-order unification (Dalrymple, Shieber and Pereira 1991), a substitutional treatment avoids the computational complexity of using greater than second-order matching to account for certain interactions between ellipsis and quantifier scope.

In this section, we will first present a number of examples illustrating a substitutional treatment of ellipsis. We will then discuss the use of higher-order unification by Dalrymple, Shieber and Pereira (1991) to cover the same kinds of example, and then comment on the connections between the two approaches.

7.2.1 Examples of Substitutional QLF Ellipsis

A Simple Example

To present the basic ideas behind a substitutional approach to ellipsis resolution in QLF, let us consider the following simple example

John slept. So did Mary.

The resolved QLF for the first sentence and the unresolved QLF for the second are given below, in a heavily abbreviated and simplified form. The reason for this simplification is that there are a number of complications to do with mood and tense that we do not yet wish to mention.

Antecedent:

```
form(<type=verb,...>, +s,
     P^[P,[sleep, term(+j,...,john)]],
     [sleep, term(+j,...,john)])
```

Ellipsis:

```
form(<type=vp_ellipsis,...>, +e,
     R^[R,term(+m,...,mary)],
     _)
```

Since the ellipsis has the category of VP ellipsis, it is resolved by finding a verb `form` from a previous QLF, and making the following interpretive substitutions.

First, the subject `term` of the antecedent QLF is interpreted as though it were the subject of the ellipsis. This means that wherever the subject `term` or its index occurs in the antecedent, it should be re-interpreted as though it were the ellipsis subject `term` or its index. Second, the index of the antecedent `form` also needs to be replaced since the ellipsis amounts to a re-interpretation of the original `form`. Since the re-interpreted antecedent is intended to be the resolution of the elliptical `form`, it makes sense to substitute the antecedent's index by the ellipsis' index.

We can therefore represent the resolved elliptical QLF as follows:

```
form(<type=vp_ellipsis,...>, +e, R^[R,term(+m,...,mary)],
     form(<type=verb,...>, +s,
          P^[P,...],
          [sleep, term(+j,...,john)]):
     {+j/+m,
      term(+j,...,john)/term(+m,...,mary),
      +s/+e})
```

The resolvent field is the antecedent verb `form` followed by a list of substitutions, `{+j/+m,...}`.

Substitutions It should be emphasised that these are 'interpretive' substitutions, rather than syntactic substitutions to be made in place on the antecedent `form`. The substitution `term(+j,...,john)/term(+m,...,mary)` says that during the course of re-interpreting the antecedent `form`, whenever you come across an interpretive rule that makes reference to `term(+j,...,john)`, you should recast it as making reference to `term(+m,...,mary)`, and similarly for the indices `+j`, `+m`, `+s` and `+e`.

More specifically, in the course of interpreting the original antecedent `form` we might have employed the following rule for an (unscoped) `term`:

> $\mathcal{W}$(F,v) if $\mathcal{W}$(Q(R',F'),v),
> where:
> F is a formula containing the term
> T=term(+j,C,R,Q,john),
> R' is X^(and(R(X),X=john))[X/+j], and
> F' is X^(and(F,X=john))[X/T, X/+j].

Under the extra interpretive substitutions imposed by the ellipsis resolution, we now wish to treat the term and its index as though they were `term(+m,...,mary)` and `+m` respectively. To do so, we abstract over the original `term` and index as before, but also make sure we abstract over the new `term` and index, and also use the restriction of the new `term`:

> $\mathcal{W}$(F,v) if $\mathcal{W}$(Q2(R2',F'),v),
> where:
> F is a formula containing the term
> T1=term(+j,C1,R1,Q1,john),
> T2= term(+m,C2,R2,Q2,mary),
> R2' is X^(and(R2(X),X=mary))[X/+j, X/+m], and
> F' is X^(and(F,X=mary))[X/T1, X/T2,X/+j,X/+m].

(Note that for each substitution in the semantic interpretation rule, the ellipsis substitutions leads to an addition of a corresponding new one).

Provided that the ellipsis antecedent is fully resolved, it is possible to treat the ellipsis substitutions as though they were ordinary syntactic substitutions. But the order of application of the substitutions does not depend on the order in which they are listed. Instead, they depend on the order in which the substituees occur in (the recursive, top-down descent through) the antecedent QLF. This is unlike the normal notation for substitutions, where {`+j/+m,term(+j,...,john)/term(+m,...,mary)`} would mean 'first replace `+j` by `+m`, and then replace `term(+j,...,john)` by `term(+m,...,mary)`'.

If we were to spell these substitutions out syntactically, the resolved ellipsis would be equivalent to:

```
form(<type=vp_ellipsis,...>, +e,
     R^[R,term(+m,...,mary)],
     form(<type=verb,...>, +e,
          P^[P,[sleep,term(+m,...,mary)]],
          [sleep, term(+m,...,mary)]))
```

i.e. 'Mary slept'. In practice, it is often easier to see what the results of a particular ellipsis resolution are if the substitutions are treated as syntactic substitutions in this way.

Strict and Sloppy Identity

There are a number of distinct readings for a sentence like

John read a book he owned, and so did Bill.

(a) Bill read a book he (Bill) owned.
(b) Bill read a book he (John) owned.
(c) Bill read the book that John owned and read.

These differences depend on whether the expressions *he* and *a book he owned* are reinterpreted in the ellipsis as (i) strictly referring to the same objects as in the antecedent, or (ii) sloppily to the same kind of object.

A strict substitution for a `term` involves replacing occurrences of that `term` by its index. A sloppy substitution involves re-indexing the `term`. This can be demonstrated as follows:

- Partially Resolved QLF:

```
[and,
  [read,
       term(+j,...,john),
       term(+b,B^[and,[book,B],
                      [own,term(+h,...,+j),B]],
            ...)),
  form(<type=vpellipsis,...>,
       R^[R,term(+w,...,bill)],
       _E)]
```

- Ellipsis antecedent / source (`S`):

```
S =
[read,term(+j,...,john),
      term(+b,B^[and,[book,B],
                 [own,term(+h,...,+j),B]],
           ...)]
```

(a) Sloppy 'he', sloppy 'book': `_E` =

```
S:{term(+w,...)/term(+j,...), +w/+j,
   +h1/+h,
   +b1/+b}
```

Equivalent QLF:

```
[and,
 [read,term(+j,...,john),
       term(+b,B^[and,[book,B],
                  [own,term(+h,..,+j),B]],
            ..)],
 [read,term(+w,...,bill),
       term(+b1,
             B1^[and,[book,B1],
                 [own,term(+h1,..,+w),B1]],
            ..)]].
```

(b) Strict *he*, sloppy *book*:

```
S:{term(+w,...)/term(+j,...), +w/+j,
   +h/term(+h,...),
   +b1/+b}
```

Equivalent QLF:

```
[and,
 [read,term(+j,...,john),
       term(+b,B^[and,[book,B],
                  [own,term(+h,..,+j),B]],
            ..)],
  [read,term(+w,...,bill),
        term(+b1,B1^[and,[book,B1],
                     [own,+h,B1]],
             ...)]].
```

(c) Strict *book*:

```
S:{term(+w,...)/term(+j,...), +w/+j,
   +b/term(+b,...), ...}
```

Equivalent QLF:

```
[and,
 [read,term(+j,...,john),
      term(+b,B^[and,[book,B],
                 [own,term(+h,..,+j),B]],
          ..)],
  [read,term(+w,...,bill),+b]]
```

Whenever a `term` is re-interpreted strictly it in effect leaves behind a `term` index. For the QLF to be interpretable, it is necessary that the antecedent term is given sufficiently wide scope so that it binds this occurrence of the index. However, this constraint does not need to be explicitly enforced: any QLF that violates it would simply be uninterpretable in the same way that other ill-scoped QLFs are. However, we have omitted explicit scoping constraints in this example.

Primary and Secondary Substitutions The possibility of making a choice between strict and sloppy substitutions only arises with *secondary* noun phrases, i.e. *a book* and *he*. The substitutions open to *primary* noun phrases (*John* and *Bill*) are fixed — replace the antecedent `term` and all occurrences of its index.

Which noun phrases are primary and which are secondary is largely determined by the form of the ellipsis. Thus, in *John read a book he owned, and so did Bill* the fact that *Bill* occurs within the elliptical phrase means that it is primary, and the fact that it is a verb phrase ellipsis means that its parallel primary term must be the subject of the antecedent (the QLF for the antecedent will carry enough information to enable one to recognise its subject). In other cases, there may be a choice about which terms in the antecedent are to be taken as primary. For example, with a noun phrase ellipsis like

> John greeted Mary before Bill

either *John* or *Mary* may be taken as primary (i.e. *John greeted Mary before Bill greeted Mary* or *John greeted Mary before he greeted Bill*).

Scope Parallelism

In a sentence like

> John gave every student a test and so did Bill.

there is scope parallelism between antecedent and ellipsis. That is, if John gave every student a different test, then Bill also gave every student a different test. But if John gave just one test to all the students, then Bill also gave just one test to all the students. Readings where John gave a different test to each student but Bill gave the same test to all the students, or vice versa, are not available.

This parallelism of scope arises naturally through the way that sloppy substitutions are handled. Essentially, scope constraints in the antecedent are re-interpreted as applying to the re-indexed `term`s in the ellipsis resolution. Whatever scoping is imposed on the antecedent is thus automatically 'copied across' to the ellipsis resolution.

Consider one of the possible scopings for the antecedent,

```
S =
 [+s,+t]:
   [give, term(+j,...,john),
          term(+s,..student..),
          term(+t,..test..)]
```

with both the secondary `term`s given sloppy substitutions in the ellipsis resolution:

```
S:{term(+w,...,bill)/term(+j,...,john), +w/+j,
   +s1/+s
   +t1/+t}
```

Spelling these substitutions out, we get

```
[+s1,+t1]:
   [give, term(+w,...,bill),
          term(+s1,..student..),
          term(+t1,..test..)]
```

i.e. with the same scope order as in the antecedent.

It should be borne in mind that the ellipsis resolution is a re-interpretation of the antecedent. The fact that the antecedent and ellipsis have the same scoping configurations is not due to an arbitrary decision to keep them the same. Rather, the scope of the antecedent is an integral part of the antecedent and the way it is re-interpreted.

Strict Identity and Parallelism A strict substitution for a secondary `term` necessitates the antecedent `term` being given scope over both antecedent and ellipsis, on pain of the QLF being uninterpretable. One might wonder what happens if the antecedent `term` is given wide scope, but is given a sloppy substitution.

It turns out that under such circumstances, strict and sloppy substitutions are equivalent. For if the antecedent `term` has wide scope over antecedent and ellipsis, applying its quantifier rule will discharge all occurrences of the `term`, both in antecedent and ellipsis resolution. This means that when it comes to re-interpreting the antecedent as part of the ellipsis resolution, there will be no occurrences of the original index left to re-interpret: they will all have been discharged. Hence it makes no difference to the final interpretation of the ellipsis whether the `term` is treated as strict or sloppy; in either case the ellipsis `term` will be discharged by scoping the antecedent `term`.

As a result, we can say that as far as scope parallelism is concerned (a) either the antecedent `term` is given wide scope, and the ellipsis behaves as though there were a strict identity, or (b) if the `term` has narrow scope, it is given a sloppy identity and scope parallelism ensues.

(No) Type-Raising for Quantifiers

As we will see below, examples like

> Every boy revised his paper, and then John did.

create a number of complications for a higher-order unification treatment of ellipsis. Under a substitutional treatment, this example poses no particular problems, and is covered by the principles already outlined.

Two readings for this sentence are possible, depending on whether a strict or sloppy substitution is made for *his paper*. A sloppy identification leads to the reading where John revises his own paper, and a strict reading where he revises the paper of each boy.

```
S =
 [revise,
       term(+b,<lex=each>,boy,forall,+b),
       term(+p,<type=possdet>,
               X^[and,[paper,X],
                   [poss,term(+h,..,+b),X]],
            exists,+p)]
```

Strict:

```
S:{term(+j,..)/term(+b,...), +j/+b,
   term(+p,...)/+p}



[+b,+p]:
 [and,
  [revise,
       term(+b,...),
       term(+p,...)]
  [revise,
       term(+j,...),
       +p]]
```

Sloppy:

```
S[term(+j,..)/term(+b,...), +j/+b,
  +p1/+p, +h1/+h]
```

```
[and,
 [+b,+p]:
   [revise,
       term(+b,...),
       term(+p,...)]
 [+j,+p1]:
   [revise,
       term(+j,...),
       term(+p1,
            X^[and,[paper,X],
                   [poss,term(+h1,..,+j),X]],
            exists,+p1)]]]
```

A substitutional treatment also lends itself readily to examples such as

> A Canadian flag was hanging in front of each house, and an American one was too.

which were discussed by Hirshbühler (1982).

(a)
```
S =
[+h,+c]:[hang,term(+c,..canadian_flag..),
              term(+h,..house..)]
```

Resolution:

```
S:{term(+a,..)/term(+c,...), +a/+c,
   +h1/+h}
```

That is:

```
[and,
 [+h,+c]:[hang,term(+c,..canadian_flag..),
               term(+h,..house..)]
 [+h1,+a]:[hang,term(+a,..american_flag..),
                term(+h1,..house..)]
```

(b)
```
S =
[+c,+h]:[hang,term(+c,..canadian_flag..),
              term(+h,..house..)]
```

Resolution:

```
S:{term(+a,..)/term(+c,...), +a/+c,
   +h1/+h}
```

That is:

```
[and,
 [+c,+h]:[hang,term(+c,..canadian_flag..),
              term(+h,..house..)]
 [+a,+h1]:[hang,term(+a,..american_flag..),
               term(+h1,..house..)]
```

(c) `S =`

```
[+c]:[hang,term(+c,..canadian_flag..),
           term(+h,..house..)]
```

Resolution:

```
S:{term(+a,..)/term(+c,...), +a/+c,
   +h/term(+h,...)}
```

That is:

```
[+h]:
 [and,
    [+c]:[hang,term(+c,..canadian_flag..),
               term(+h,..house..)]
    [+a]:[hang,term(+a,..american_flag..),
               +h]
```

(Note that cases (a) and (c) are logically equivalent).

De Dicto / De Re

It should be apparent that the treatment of strict and sloppy identity also lends itself to a treatment of de re / de dicto ambiguities:

Bill wants to read a good book and John does too.

- ```
  S = [wants, term(+w,...,bill)
               [read,+w,term(+b,book,..)]]
  ```
- De Re: same book (strict)
  ```

```
[+b]:[and,
       S,
       S:{term(+w,...)/term(+j,...), +w/+j,
          +b/term(+b,...)}]
```

- De Re: different book (sloppy)

```
[and,[+b]:S,
     [+b1]:S:{term(+w,...)/term(+j,...),
              +w/+j,+b1/+b}]
```

- De Dicto: different book

```
[and,
 [wants,term(+w,...,bill)
        [+b]:[read,+w,term(+b,book,..)]],
 [wants,term(+w,...,bill)
        [+b]:[read,+w,term(+b,book,..)]:
        {term(+w,...)/term(+j,...), +w/+j,
         +b1/+b}]]
```

- *De Dicto: same book

```
[and,[wants,term(+w,...,bill)
            [+b]:[read,+w,term(+b,book,..)]],
     [wants,term(+w,...,bill)
            [+b]:[read,+w,term(+b,book,..)]:
            {term(+w,...)/term(+j,...), +w/+j,
             +b/term(+b,...)}]]
```

De re readings are obtained by giving the `term` for *a book* wide scope over the intentional predicate *want*. This can be done in two ways, so that it is either the same book that John and Bill want to read or different books. By contrast, there is only one way of obtaining a de dicto reading, and here there is no commitment to it being the same book that John and Bill want to read. (The strict, de dicto reading is uninterpretable, since it will leave an undischargeable index, `+b`, in the ellipsis)

Antecedent Contained Ellipsis

The following sentence might appear to present problems for our treatment of ellipsis, in that the ellipsis is contained within the (object noun phrase of the) antecedent.

John greeted everyone that Bill did.

However, this merely serves to rule out the possibility of making a sloppy substitution for the offending `term`.

- S=
```
[greet,term(+j,...,john),
       term(+e,X^[and,[person,X],
                    form(<type=vpellipsis>,
                         R^[R,term(+b,...,bill)],
                         _E)])]
```

- Strict identity for *everyone*:
```
_E =
S:{term(+b,...)/term(+j,...), +b/+j,
   +e/term(+e,...)}

[+e]:
 [greet,term(+j,...,john),
        term(+e,X^[and,[person,X],
                    [greet,term(+b,...,bill),+e],
             ..)]
```

- Sloppy identity for *everyone*:
```
_E =
 S:{term(+b,...)/term(+j,...), +b/+j,
    +e1/+e}
[greet,
 term(+b,...,bill),
 term(+e1,
      X^[and,[person,X],
         form(<type=vpellipsis>,
              R^[R,term(+b,...,bill)],
              [greet,term(+b,...,bill),
                     term(+e1,X^[and,[person,X],
                                  form(****)],..)],
                         )])]
```

Here, giving a sloppy reading for "everyone" leads to an uninterpretable QLF — the ellipsis occurs in an undischarged form in its own resolution.

Prepositional Phrase Ellipsis

So far we have concentrated on ellipses where resolution essentially involves replacing one `term` by another. Not all ellipses are like this, of course. For example in,

John won in 1990. And in 1991.

the ellipsis essentially involves replacing one prepositional phrase by another. This type of ellipsis can readily be accommodated within our framework: one substitutes for `form`s and `form` indices in the same way as for `term`s.

Tense and Mood

The current treatment of tense and mood in QLF forces a slight departure from a strict substitutional approach to ellipsis. This affects pairs of sentences such as the following

John slept. So will Mary.

Because tense, aspect and mood information is all contained in the verb `form` category, it will not do simply to re-index the original verb `form` in resolving the ellipsis. This would result in a past tense resolution of the elliptical sentence.

Instead, the verb `form` categories on the antecedent and ellipsis are merged. A new `form` is constructed with the merged category and the body of the antecedent `form` and the ellipsis resolution substitutes this constructed `form` for the original.

7.2.2 Ellipsis and Higher-Order Unification

Dalrymple, Shieber and Pereira (1991) (henceforth referred to as DSP) suggest that the resolution of ellipsis can be cast as a matter of solving a number of equations using higher order unification.

Simple Example As a simple case (which doesn't require the full power of higher order unification) consider

Dan likes golf. So does George.

Dan and *George* are parallel elements in the ellipsis and it antecedent. Resolving the ellipsis is a matter of recovering a property from the antecedent which (i) when applied to the parallel elements in the antecedent yields the antecedent itself, and (ii) when applied to the parallel elements in the elllipsis yield the resolution of the ellipsis.

In simplified form, we can represent the 'logical forms' of the sentences above as

like(dan,golf)
P(george)

where P is (a variable standing for) the property that has to be recovered. Applying P to the parallel elements in the antecedent, we get the following equation

(7.1) $P(dan) = like(dan, golf)$

The solution to this equation is given by

$$P = \lambda x.like(x, golf)$$

When applied to the parallel ellipsis elements, this solution yields

$$\lambda x.like(x, golf)[george] = like(george, golf)$$

In more complex cases, solving equations like (7.1) may involve using higher order unification.

Strict and Sloppy The difference between strict and sloppy readings for

Dan likes his wife. So does George

(strict = George likes Dan's wife, sloppy = George likes George's wife) emerges from different ways of solving the ellipsis equations:

P(dan) = like(dan,wife-of(dan)
strict: $P = \lambda x.like(x, wife - of(dan))$
sloppy: $P = \lambda x.like(x, wife - of(x))$

As DSP note, there are two other solutions to the equation, where the primary or parallel occurrence of *dan* is not abstracted over. They impose an additional constraint on solutions, namely that primary occurrences must always be abstracted over.

Interactions with Quantifier Scope DSP assume a semantic framework where 'quantifier scope is generated through a mechanism of discharging [quantifier] assumptions introduced in the course of a derivation'. To see roughly how this works, consider a sentence like

John greeted every person.

The quantified noun phrase *every person* is given an interpretation consisting of a quantifier assumption and a term meaning:

$$\langle every, x, person(x)\rangle \vdash x$$

which is to be read as saying that the meaning of the noun phrase is x under the assumption to the left of the turnstile. At some point in the derivation of the meaning for the whole sentence, the quantifier assumption will have to be discharged. Where it gets discharged determines the scope of the quantifier. Applying the meaning of the verb phrase to the meaning of the noun phrase (i.e. x) and then the meaning of the subject *John*, yields an interpretation that still rests on an undischarged scope assumption:

$$\langle every, x, person(x)\rangle \vdash greet(j, x)$$

Discharging the assumption yields

$$\vdash every(x, person(x), greet(j, x))$$

which no longer rests on any quantifier assumptions.

Turning to the interactions of scope and ellipsis, note that

John greeted every person when Bill did

has two readings. One is a distributive reading where each person gets simultaneously greeted by John and by Bill. The other is a collective reading where John greets everyone (en masse) and Bill does the same. These differences are accounted for by resolving the ellipsis at different points during the introduction and discharging of quantifier assumptions.

First take the distributive case. Here we solve the ellipsis equations before the quantifier assumption has been discharged. Thus the equation is

$$P(john) = greet(john, x)$$

(the undischarged assumptions do not form part of the equation). Applying the solution to this equation to the sentence as a whole

$$\langle every, x, person(x)\rangle \vdash when(greet(john, x), P(bill))$$

we get

$$\langle every, x, person(x)\rangle \vdash when(greet(john, x), greet(bill, x))$$

Once the quantifier assumption is discharged, we have

$$\vdash every(x, person(x), when(greet(john, x), greet(bill, x)))$$

giving a distributive reading.

For the collective case, the quantifier assumption is first discharged and then the ellipsis equation is solved. The equation is thus

$$P(john) = every(x, person(x), greet(john, x))$$

the solution to which is

$$\lambda y.every(x, person(x), greet(y, x))$$

Applying this to the ellipsis we get

$$\vdash when(every(x, person(x), greet(john, x)), every(x, person(x), greet(bill, x)))$$

i.e. a collective reading.

Higher than Second-Order Matching The interactions between ellipsis and scope sometimes give rise to the need to use more than second order matching in the solution of ellipsis equations. Consider

Every student revised his paper, and then John did

As before, we can resolve the ellipsis before or after discharging the scope assumption (leading to a difference between Bill revising his own paper or every students).

If the quantifier is discharged first, we need to solve an equation that is approximately represented as

$$P(''every\ student'') = every(x, student(x), revise(x, paperof(x)))$$

the problem with this is that in place of *"every student"* we cannot use the conditional noun phrase interpretation

$$\langle every, x, student(x)\rangle \vdash x$$

since the conditional interpretation only makes sense in the course of a derivation. To circumvent this, the noun phrase is type raised by applying it to an arbitrary property S, then discharging the quantifier assumption, and then abstracting over S. This yields an ellipsis equation

$$P(\lambda S.every(x, student(x), S(x)) = every(x, student(x), revise(x, paperof(x)))$$

And this equation requires more than just second order matching to solve.

7.3 Discussion

Pereira (1990) observes, in the context of categorial semantics, that it is productive to look at the way in which the meaning of a sentence is derived by composing the meanings of its constituents, as well as at the end result of the derivation. Ellipsis resolution involves recomposing some of the constituents of the antecedent to form a property that can be applied to the ellipsis arguments. Higher-order unification is required because the ellipsis equations are stated using the end results of the meaning derivations, and not the derivations themselves. Thus, one has to unpick the original derivation before putting some of the constituents back together again.

As we have seen, monotonic semantics in the shape of QLF ellipsis resolution can achieve the same effects, but without recourse to higher-order unification. Nor is the precise interleaving of scoping decisions in attempts to solve ellipsis equation crucial. Both these factors considerably simplify the treatment of ellipsis in QLF. Nevertheless, the underlying model of ellipsis in monotonic semantics and in DSP's account is essentially the same.

The reason that the QLF treatment of ellipsis can get away with being simpler is this: To a limited extent, QLF resolutions record the steps that need to be taken in the course of a semantic derivation, rather than just the end results of the derivation. This is especially so for quantifier scope constraints. The scope constraints simply say when and where during the course of the semantic derivation quantifier assumptions are to be discharged. Having this derivation information more immediately available makes the unpicking and recomposing of derivations more straightforward. QLFs may thus be construed as explicitly recording aspects of semantic interpretation that are left implicit in an approach like Pereira's (1990) categorial semantics, as suggested in Chapter 5.

Chapter 8

Generation

The most important uses of the generator in CLARE are (i) to allow users to confirm the results of reference resolution, and (ii) to generate sentences from database assertions (or questions from incomplete assertions). We had expected that its use for paraphrasing QLF analyses for structural disambiguation would be also be useful. In practice, it turns out to be difficult to generate paraphrases exhibiting a choice of structure without introducing further ambiguity.

Our model for natural language generation involves the processes of *synthesis*, i.e. producing a natural language string from a given QLF, and *description*, i.e. producing a QLF from which synthesis can proceed. In section 8.1 we review the way that synthesis is implemented by means of a version of the "semantic head driven" algorithm. In section 8.2 we discuss how description operates when generating paraphrases to make reference resolutions to QLFs explicit. Finally, in section 8.3 we discuss description of TRL facts, which is used in the generation of statements and questions from the contents of the database.

8.1 Linguistic Generation (Synthesis)

Since the CLE grammar formalism is declarative, it can be used for generation (synthesis) as well as analysis. In particular, given that QLF analyses are built up by unification rules for syntax, semantics, and morphology, it is also possible to devise algorithms for building a tree of constituents by unification by applying these rules in the "reverse" direction. If the semantic rules had been more in the style of traditional Montague semantics, generation from structures that had undergone lambda reductions would have presented search difficulties because the reductions would have to be applied in reverse during the generation process. This turns out to be an important practical advantage of unification-based semantics over the traditional approach.

In the context of a system like CLARE for interaction with a knowledge base, generation is potentially useful to the aim of natural cooperative dialogues in a number of ways. These include the generation of responses: answers, asking the user to confirm consequences of their requests (e.g., changes to the knowledge base), explanations of analysis failure, and so on. Generation is also useful for interactive disambiguation as it often allows user inputs to be paraphrased in a way that makes explicit their interpretation by the system.

The CLE component for generation from QLF employs an algorithm that is based on the "semantic head driven" (SHD) generation algorithm (Shieber et al. 1990). We will briefly describe the original algorithm and then discuss some steps we have taken in the CLE implementation to make generation efficient in the context of a large grammar.

8.1.1 The SHD Generation Algorithm

The SHD algorithm is usually applied to a pair consisting of the semantic structure being generated from (in our case a QLF), together with a category for the phrase type we are attempting to generate. We will refer to such a pair as a *node*, so semantic rules in the CLE grammar formalism are lists of such nodes. At the start of the process, the category may be more or less instantiated, it may simply be the start category for the grammar, or it may be a suitably instantiated sentence or noun phrase category. In the CLE implementation, the output (which is nondeterministic) is a phrase structure tree for some sentence having the input QLF as its semantic analysis.

The SHD algorithm proceeds as follows. The QLF for the node currently being generated is matched against the QLF of the mother of a *nonchain rule* that is, a rule in which the QLF for the mother is different from that of any of its daughters. The choice of rule is constrained further to rules for which there is a sequence (or chain) of categories, $C_1, \ldots, C_n$, where C_1 is the category of the node being generated, C_n is the mother node category of the chosen rule, and each pair (C_i, C_{i+1}) corresponds to the mother and daughter categories of a *chain rule*. A chain rule is one in which some daughter (the semantic head) has the same QLF as the mother node of that rule. To complete the construction of the phrase structure tree, all that is needed is to apply the algorithm recursively to the daughters of the chosen nonchain rule and to the nonhead daughters of each chain rule in the sequence.

The effectiveness of the algorithm comes from the way it quickly leads to a backbone of the target phrase structure tree being proposed according to the input QLF and the semantic head relation for the grammar. For a more detailed presentation of the algorithm and a discussion of its merits, the reader is referred to Shieber et al. 1990.

8.1.2 Applying Rules for Generation

In the CLE implementation, the rules used in generation are those built by unifying the categories in semantic rules with the categories in their syntactic counterparts (syntactic rules with matching rule identifiers) and the categories in word sense derivation rules with those in corresponding morphology rules. It is also possible to run the generator simply with semantic rules and sense derivation rules, and apply the constraints from the syntactic rules in a second stage, but this leads to much increased nondeterminism and a consequent increase in processing time. Syntactic constraints for lexical entries are also unified into sense entries when these are being applied, as daughterless nonchain rules, during generation.

The more compositional the semantic rules of the grammar, the more efficient generation becomes with this algorithm. In the context of the CLE semantic rules, this corresponds to having the mother QLF in a rule built from daughter QLFs rather than from the values of semantic features. As a result, applying the CLE grammar in the generation direction, rather than the analysis direction for which it was originally designed, was more effective after some re-organization of the rules that increased the compositionality of the grammar—this also made the grammar easier to understand.

Different uses to which a generator is employed require different trade-offs between efficiency and the number of paraphrases that the generator needs to produce for a given QLF. To control the set of rules that the generator applies, the rule compilation process is sensitive to declarations specifying which rules are to be included, and the order of their application, in the generation process.

8.1.3 Rule and Lexicon Precompilation

The CLE implementation of the SHD algorithm includes a number of control structure and precompilation optimizations. The first of these is precompilation of the transitive closure of the semantic head relation for the categories of the grammar to allow more efficient application of the chaining step. This precompilation proceeds in a similar fashion to the computation of the left-corner linking relation used by the CLE parser in that pairs in the relation are kept in their most general form with subsumed pairs being deleted.

Further preprocessing takes place by unifying the daughter element of pairs in the semantic-head relation with the mother categories of nonchain rules producing composed rules of the form

```
gen_rule(Index,ChainCategory,MotherNode,DauNodes).
```

These rules are applied at run time by matching the mother QLF and the chain category against the target QLF and category being generated. It is further assumed that these two terms (the QLF and category) will be instantiated when the rule is applied, so a combined index is derived from them to improve the efficiency of rule look-up. This assumption can be enforced by reordering the generation of daughter nodes dynamically, as explained in the next section.

As mentioned above, sense entries are treated by the algorithm as daughterless nonchain rules, so it is possible to include them in the precompilation process. However, this results in a very large set of `gen_rule` entries (several times the number of entries in the lexicon) with corresponding increased storage and access overheads. Our strategy is to only include lexicon entries in the compilation process if they cannot be indexed effectively by a unique (or almost unique) symbol, such as the word sense predicate symbols in the QLFs for most noun, verb, and adjective entries. In effect, the precompilation process decides which words are to be treated as "function" words whose entries are compiled together with grammar rules, and which are treated as "content" words. Applying the former will be constrained during generation by the semantic-head relation, while the latter will only be considered when generating from a QLF (sub)expression if the symbol under which it is indexed is present in that QLF. Since there is a constant number of function words, and a constant number of content words indexed from the symbols in a QLF, the time taken for generation is effectively independent of the size of the lexicon.

Literals appearing in grammar rules (i.e., daughters that are specified as words rather than categories) are compiled directly into the run-time rules, so no access to the lexicon is necessary for them; they are treated as daughters whose generation has been completed at compile time.

8.1.4 Control and Search

The CLE implementation of the SHD algorithm includes an improvement to the basic control structure: generation of nodes with as yet uninstantiated QLFs can be delayed until the QLFs become instantiated. Generating from nodes with instantiated QLFs reduces unguided search and allows the improved indexing mentioned above. There are cases where the delay is essential.

An example of this is generation of subject-auxiliary inversion questions like *Will John sleep?* if these are treated by a rule of the form:

```
sem(s_aux_np_vp,inverted,
    (AuxQLF,s:[type=ynq,...]) -->
    [(AuxQLF,aux:[arglist=[(VpQLF,VpCat)],subjval=Np],
     (Np,np:[...]),
     (VpQLF,VpCat)]).
```

When this rule is used for analysis, the subject noun phrase meaning `Np` is passed to the verb phrase meaning `VpQLF` via the sense entry for the auxiliary. During generation, however, the noun phrase meaning variable `Np` will only become instantiated after the verb phrase node `(VpQLF,VpCat)` has been generated. It is therefore necessary to delay generation of the noun phrase node until this has happened, as otherwise all possible noun phrases compatible with the noun phrase category will be generated before their meanings are matched against the noun phrase meaning in the input QLF.

In this particular case, it is only necessary to delay generation of the subject noun phrase locally, i.e. until the other daughters of the rule introducing it have been generated. More generally though, it may be necessary to delay generation non-locally, so that uninstantiated nodes may be threaded upwards until nodes higher up the tree have been generated.

The search strategy for generation in the CLE implementation is an iterative deepening strategy whereby a series of depth-limited searches are performed to successively higher limits (Korf 1985). The depth of a solution corresponds here roughly to the number of applications of rules or lexical entries during the search, so a shallower solution usually corresponds to a simpler sentence. The strategy thus tends to locate the simple solutions closer to the top of the search space than the depth-first strategy would. This is particularly suitable for generation because in many applications we are only interested in a single, simple rendering of the semantic representation in natural language. This is in contrast to parsing where we are often interested in all possible analyses, the reason for the asymmetry being that the semantic representation language is less ambiguous a representation than natural language. The iterative deepening strategy also means that finite solutions will be found eventually, even if the grammar allows infinite phrase structure derivations for the input semantic representation.

In order to reuse intermediate results from one search path (or from a previous depth iteration), whenever a possible phrase structure for a constituent is generated, this is stored together with the QLF-category node associated with it, provided that it is not subsumed by an existing intermediate result. When a daughter node is being generated it is first matched against previously generated nodes, so that the stored phrase structure can be reused. Again, the fact that we do not want to exhaustively search for all possible solutions means that such a simple caching technique can be effective in reducing the time spent finding initial solutions.

8.2 Generation for Paraphrasing Resolutions

8.2.1 Principles of Resolution Paraphrase

In Chapter 6, reference resolution was characterised in terms of finding solutions to the following kind of translation equivalence,

```
C U A |= (Q <-> Q')
```

(where `C` represents the context, `A` any additional assumptions, `Q` the unresolved QLF, and `Q'` the resolved QLF). Reference resolution involves finding a `Q'` that is syntactically more instantiated than `Q`, but equivalent to it given `C` and additional assumptions `A`. A more concrete example of this schema is

```
context(C) /\ most_salient(i1,C,X^prototype(X),zed9) ... |=
    disintegrate(term(i1,<det=the>,X^prototype(X),_,_))
    <->
    disintegrate(term(i1,<det=the>,X^prototype(X),exists,zed9)).
```

which shows how the sentence *The prototype disintegrated* might be resolved.

In order to paraphrase a resolution, the translational schema can be applied in the reverse direction. That is, given a resolved QLF `Q`, one needs to find an unresolved QLF `Q''` that is equivalent to `Q` in a context `C'` plus assumptions `A'`, where preferably `C'` should be entailed by the more specific context `C` used in reference resolution. That is

```
C' U A' |= (Q' <-> Q'')
```

Here there is no requirement that `Q''` be syntactically more instantiated then `Q'`, or vice versa. As a more concrete example, we might have

```
wider-context(C',C) /\... |=
    disintegrate(term(i1,<det=the>,X^prototype(X),exists,zed9))
    <->
    disintegrate(term(i1,<proper_name>,X^name_of(X,'Zed-9'),_,_)).
```

This shows how *The prototype disintegrated* could be paraphrased as *Zed-9 disintegrated*. In the new context, `C'`, we no longer need assume that `zed9` is the most salient prototype, though it does need to be the case that the name for `zed9` is *Zed-9* in `C'` (and hence also in the original context `C`). The idea behind using a wider, less specific context is that the paraphrase should consequently be more specific and less context-dependent. In general, the context would be widened by weakening conditions in the discourse model (e.g. what is salient / has recently been mentioned) rather than altering the domain model itself.

To paraphrase more complex QLFs one would normally proceed bottom up, replacing QLF sub-expressions by equivalent paraphrases before trying to paraphrase the expression itself. This has the benefit of largely preserving the predicate-argument structure of the original QLF, which might otherwise be lost. The QLF sub-expressions paraphrased will be individual `terms`, `forms` and scope constraints.

For `terms` and `forms`, deriving equivalent paraphrases is in many respects the reverse of making resolutions. Resolution involves instantiating unspecified `term` or `form` resolvents, given specified categories and restrictions; paraphrase involves instantiating categories and restrictions given specified resolvents. Thus, in forming a description for `term i1`, we might start with an underspecified `term`

```
term(i1,_,_,exists,zed9)
```

and instantiate the category and restriction to appropriate values. This, in effect applies reference resolution methods in reverse.

Resolution Paraphrase in Practice It is not currently feasible to derive QLF paraphrases by proving equivalence using resolution methods in the reverse direction, and simultaneously fixing a wider context. As a simpler compromise, a number of what are in effect special purpose reverse resolution methods are employed. These take a fully resolved `term` or `form` and produce either (i) a new (unresolved) `term` or `form` that is equivalent to the original in a wider variety of contexts, or otherwise (ii) a copy of the original. A paraphrase QLF is built up through replacing `term`s and `form`s by their paraphrases. In the next sections, we describe some of the reverse resolution / paraphrase methods in more detail.

8.2.2 Paraphrasing `terms`

The following examples show the effects achieved by paraphrasing `terms`. (Generated sentences are shown in typewriter font.)

- Pronouns:
  Does Peter Piper work on CLAM-BAKE? ...
  Which projects does he work on?
  `>>> Which projects does Peter Piper (the employee) work on?`

  How many payments were there on CLAM-BAKE? ...
  Who were they to?
  `>>> Who were they (the payments on CLAM-BAKE (the project)) to?`

- Ambiguous Names:
  Show all payments for 1990.
  `>>> Show all payments during 1990 (the year).`
  or
  `>>> Show all payments on account 1990.`

- Ambiguous Quantifiers:
  Show payments on CLAM-BAKE
  `>>> Show all payments on CLAM-BAKE (the project).`
  or
  `>>> Show some payments on CLAM-BAKE (the project).`

  Which employees have cars?
  `>>> Which employees have some cars?`
  or
  `>>> Which employees have all cars?`

Below, we explain how these paraphrases are brought about.

Referring `terms`

When `terms` refer to a specific entity, an appositive proper name `term` is created in its place. The restriction for the `term` is formed as follows. The inference engine is used to determine the name of the entity. The term in the domain model referring to the entity will normally be sorted; that is, part of the term will include a property (such as `project_Activity` or `employee`) giving the basic sort of the entity. This is used to form an appositive modification to the `term`'s restriction (in order to generate NPs like *Peter Piper (the employee)*).

This method can also be used to produce `terms` referring to objects from scratch, and is indeed used when generating sentences from domain facts (section 8.3).

Contextually Restricted Quantification

Some `terms` are resolved to act as contextually restricted quantifiers, e.g. pronouns that do not specific entities. Normally, the contextual restriction corresponds to the full description (Chapter 6) of an antecedent `term`. This description is used to form an appositive modification to the original `term` restriction.

Bare Plurals

Some determiners are resolved in non-trivial ways. For example, bare plurals may sometimes correspond either to universal or existential quantifiers. Paraphrase can make the resolution explicit by altering the category of the `term`.

8.2.3 Paraphrasing `forms`

Examples of `form` paraphrases are:

- Compound Nominals:
  Show me the Scandy payments for last year
  `>>> Show me the payments to Scandy for 1990.`
  Show me the CLAM-BAKE payments for last year
  `>>> Show me the payments on CLAM-BAKE for 1990.`

- Vague Prepositions and Possessives:
  Show payments for CLAM-BAKE
  `>>> Show all payments on CLAM-BAKE (the project).`
  Show CLAM-BAKE's payments
  `>>> Show all payments on CLAM-BAKE (the project).`

- Ellipsis:
  How many projects does James Cagney work on?

Peter Piper?
```
>>> How many projects does Peter Piper work on?
```

Vague Relations

Compound nominals, possessives and some prepositions introduce `form`s that are typically resolved to specific prepositional senses. When this is the case, the category of the `form` is altered so that it regenerates as the appropriate preposition.

Ellipsis

Ellipsis resolutions are paraphrased essentially by making the necessary substitutions marked on the ellipsis antecedent, and generating from that.

8.3 Generation of Facts and Questions

8.3.1 Generation from Assertions

The description part of generation from a TRL assertion requires two steps to be carried out: (i) determining an appropriate set of linguistic predications (i.e. applications of word sense predicates) for the assertion, and (ii) producing a set of QLFs which can be used for linguistic synthesis.

Rules for the first of these steps are compiled from the domain model equivalences expressing the relationship between linguistic and database predicates. This compilation process involves application of the equivalences in the "reverse" direction, i.e. starting from database predicates and ending with word senses. (The initial stages of this compilation process are the same as that required by the interpretation process for domain-specific resolution of vague relations — see Chapter 6.)

The second step could in principle proceed by the process of monotonic description sketched out in Section 8.2. However, this process would be highly nondeterministic, both in selecting possible restrictions and possible categories for the QLFs being built up. Our first attempt at a solution to the problem simplifies it in two ways. Firstly, we only consider description from the most common kind of TRL form, namely an existentially quantified conjunction of atomic goals. Secondly, and more crucially, description is not free, but rather constrained to follow one of a set of "template" rules. These are automatically produced by generalizing specific training examples, using a variant of the so-called "case-based learning" method. The learning procedure is described in Section 8.3.2. It produces rules of the approximate form

```
qlf_for_trl(<QLF>) :- <Body>
```

where `<Body>` is a set of TRL goals. (We refer to these as TRL description generation rules, or simply TRL generation rules.) The goals in `<Body>` can be of several different types: they include the component conjuncts in the TRL form that is being generating from, and conditions constraining `terms` in `<QLF>` to refer to TRL constants. Examples of learned TRL description rules are given in the next section.

The set of TRL description rules is not complete; a QLF can be produced only if it is of the same form as one of the training examples. They are not necessarily guaranteed to be sound either, though we have not in practice observed any unsound rules to have been inferred by the process. The status of the rules is thus that of *generation heuristics*, induced by analysis of specific examples. To check the soundness of a rule, it necessary to make sure that the original TRL can be recovered from the proposed QLF.

Loss of completeness is not necessarily important, since there is no need to be able to produce *every* way of expressing something. Indeed, everyday experience suggests that people generally make use of an array of stock phrases when expressing themselves; there is no obvious reason to believe that one has to have a general mechanism for generation at this level. The unsoundness of the method is more significant, since checking correctness is moderately expensive. One way to attack this problem might be to attempt to infer rules using the Explanation-Based Learning method (Hirsh 1987, Rayner 1988), the application of reference resolution methods would be conditions on the inferred rules. In the rest of this chapter we describe in more detail TRL generation as implemented in CLARE.

8.3.2 Learning Case-based Description Rules

The learning method used for acquiring TRL description rules is extremely simple, and is perhaps best introduced by an example. Suppose that the training example is the sentence

John is a dog.

uttered in a context in which "John" can be used to refer to the entity `john1`. This produces the QLF

```
[dcl,
 form(verb(pres,no,no,no,y),
      A,
      B^
      [B,
       [be,
        A,
        term(proper_name(tpc),C,D^[name_of,D,John],E,F),
        G^[eq,G,term(q(H,a,sing),I,J^[dog_Animal,J],K,L)]]],
      M)]
```

which in turn produces a TRL form which after simplification is

```
dog_Animal(john1)
```

The idea is to generalize the training sentence by looking for correspondences between the QLF and the TRL representations, and hypothesizing that they are instances of a general rule. In our example, there are two such correspondences. The predicate symbol `dog_Animal` occurs in both QLF and TRL; also, the QLF contains the term

```
term(proper_name(tpc),C,D^[name_of,D,John],E,F)
```

which the according to the information in the domain model has the term `john1` occurring in the TRL representation, as a referent. Looking up the linguistic properties of `dog_Animal` using the predicate `genpred/2`, we have

```
genpred(dog_Animal,([dog_Animal,_],nbar(_,_)))
```

– that is, dog_Animal is a one-place predicate and a common noun (nbar) sense. Our precise hypothesis with regard to replacing dog_Animal is that this information constitutes a sufficiently detailed description of its properties that any other predicate with a similar entry would be processed in the same way. So we are guessing, for example, that in a context where "Mary" can be used to refer to the entity mary1, the TRL

```
cat_Animal(mary1)
```

could have been produced from the QLF

```
[dcl,
 form(verb(pres,no,no,no,y),
      A,
      B^
      [B,
       [be,
        A,
        term(proper_name(tpc),C,D^[name_of,D,Mary],E,F),
        G^[eq,G,term(q(H,a,sing),I,J^[cat_Animal,J],K,L)]]],
      M)]
```

our justification being that

```
genpred(cat_Animal,([cat_Animal,_],nbar(_,_)))
```

holds and that

```
term(proper_name(tpc),C,D^[name_of,D,Mary],E,F)
```

resolves to mary1. In the general case, our heuristic rule will be to guess that

```
[dcl,
 form(verb(pres,no,no,no,y),
      A,
      B^
      [B,
       [be,
        A,
        <Term>,
        G^[eq,G,term(q(H,a,sing),I,J^[<PredAtom>,J],K,L)]]],
      M)]
```

is a QLF from which it would be possible to derive the TRL form

```
<PredAtom>(<Ref>)
```

under the assumptions that `<PredAtom>` is a atom, `<Term>` is a term, the `genpred` entry for `<PredAtom>` is

```
genpred(<PredAtom>,([<PredAtom>,_],nbar(_,_)))
```

and `<Ref>` is the referent of `<Term>`. This information is what is encoded in the induced rule,

```
trl_gen([dcl,
          form(verb(pres,no,no,no,y),A,
               B^[B,[be,
                     A,
                     Term,
                     C^[eq,C,term(q(_,a,sing),_,D^[Pred,D],_,_)]]],
               _)],
         [Pred,Ref],
         [entity_ref_term(Ref,Term),
          trlgoal([Pred,Ref],genpred(Pred,(E^[Pred,E],nbar(_,_))))])
```

The rule is capable of producing QLFs for sentences of the form

$\langle NP \rangle$ is a/an $\langle Noun \rangle$.

where $\langle NP \rangle$ is any referring expression for which there is a reverse resolution method (see section 8.2.1). This includes NPs like "John", "Mary (the employee)", "1990 (the year)" and "1/1/89 (the day)".

8.3.3 Description of Database Entities

Generation from TRL is currently used in the database entity description (or simply "description") module, which is linked to the semantics of English verbs like "talk about (something)" and "tell (someone) about (something)". Processing of a sentence such as

Talk about Peter Piper.

produces a call to describe the conceptual object

```
SF(c(x^[employee_Worker,x],Peter Piper))
```

The description module is passed the conceptual object X that is to be described, and begins by finding all conceptual goals C_i (see section 3.2) which have the property that at least one argument position is filled by an occurrence of X. This provides an abstract "conceptual-level" summary of "what the database knows about X". The main part of the processing is then carried out by two sub-modules, the *utterance synthesizer* and the *utterance filter*, which are called in alternation. The synthesizer uses the TRL description rules to attempt to find new utterances whose truth-conditions are implies by some C_i. After each such utterance is found, but before it is displayed, the utterance filter is called into play to determine whether or not the new utterance should be shown to the user, and if so whether any more utterances need be searched for. We illustrate the behaviour of the description module by sketching the course of processing for the example given immediately above. The first C_i found is the predication (C1)

```
employee(SF(c(x^[company_Organization,x],CRI)),
         SF(c(x^[employee_Worker,x],Peter Piper)),
         m,
         y)                                                  (C1)
```

which conveys the information that the employee Peter Piper is a employee at the company CRI, is of sex m (for "Man"), and has a car (y as opposed to n).

The utterance synthesizer is called first. When passed a conceptual goal C_i, the synthesizer begins by computing the set of goals logically implied by the union of C_i with the current linguistic domain theory *excluding* the database: the result is then uses as input to the learned TRL description rules. The efficiency of the process is increased by pre-compiling the set of consequences of a generic representative of each class of tuple, and caching the results in the form of "TRL lemmas". The relevant lemmas will be of the form

```
Conditions -> (<C> -> <L>)
```

where <C> is a conceptual goal matching C_i, Conditions is a conjunction of evaluable predicates, and <L> is an atomic goal whose predicate is a linguistic predicate (cf sections 6.1.1 and 9.1.1). Continuing with the example, synthesis from the goal (C1) proceeds as follows. First the TRL lemmas are used to find the set of consequences of (C1). When the concrete values for the relevant instances have been instantiated, the result is a list which includes the goals

```
car_Vehicle(sk12(SF(c(x^[employee_Worker,x],Peter Piper))))
have_3p(sk13(SF(c(x^[employee_Worker,x],Peter Piper))),
        SF(c(x^[employee_Worker,x],Peter Piper)),
        sk12(SF(c(x^[employee_Worker,x],Peter Piper)))),
male_Masculine(SF(c(x^[employee_Worker,x],Peter Piper))),
man_MalePerson(SF(c(x^[employee_Worker,x],Peter Piper))),
name_IdentifyingPhraseOf(Peter Piper,SF(c(x^[employee_Worker,x],
                             Peter Piper))),
sex_GenderOf(Male,SF(c(x^[employee_Worker,x],Peter Piper))),
person_HumanBeing(SF(c(x^[employee_Worker,x],Peter Piper))),
at_Locational(SF(c(x^[employee_Worker,x],Peter Piper)),
              SF(c(x^[company_Organization,x],CRI))),
employee_Worker(SF(c(x^[employee_Worker,x],Peter Piper))),
```

Here, sk12(X) and sk13(X) are Skolem functions representing, respectively, the car employee X has (if such an object exists), and the event of his or her having it. It is now possible to attempt to apply learned description rules, some of which succeed. For example, the first two "consequence" goals licence an application of the rule generalized from

John loves a cat.

with have_3p substituting love_Like, car_Vehicle substituting cat_Animal, and Peter Piper substituting John. Synthesis proceeds from the QLF produced, yielding the sentence

Peter Piper (the employee) has a car.

Before the sentence is displayed, the utterance filter is called. The filter keeps track of the combined propositional content of the sentences so far uttered when describing the current tuple, which is stored as a global variable P_{old}. (Initially, the value of P_{old} is $true$). When the filter is passed a new candidate utterance with propositional content P, it first ascertains whether or not P is implied by P_{old}. If this is the case, it returns the information that the new utterance is redundant and need not be displayed. If not, it displays the new utterance, and updates the combined propositional content to $P_{old} \wedge P$. It then checks again to find out whether $P_{old} \wedge P$ is equivalent to the current tuple. If so, no more utterances need to be generated.

The expression representing the content is first transformed if necessary by replacing Skolem functions with existentially quantified variables and then translated into conceptual form. Thus the propositional content of the first utterance is originally

```
and(car_Vehicle(sk12(SF(c(x^[employee_Worker,x],Peter Piper)))),
    have_3p(sk13(SF(c(x^[employee_Worker,x],Peter Piper))),
            SF(c(x^[employee_Worker,x],Peter Piper)),
            sk12(SF(c(x^[employee_Worker,x],Peter Piper)))))
```

After replacement of Skolem functions by existentially quantified variables, this becomes

```
exists([X,Y],
   and(car_Vehicle(X),
       have_3p(Y,
               SF(c(x^[employee_Worker,x],Peter Piper)),
               X)))                                          (C2)
```

and after translation,

```
exists([X],
   employee(SF(c(x^[company_Organization,x],CRI)),
            SF(c(x^[employee_Worker,x],Peter Piper)),
            X,
            y)                                                    (C3)
```

Translating (C2) into the canonical form (C3) makes it simple to ascertain that the tuple has so far not been completely described, by comparing (C3) and the original tuple (C1). On the next cycle, the compiled consequence goal

```
man_MalePerson(SF(c(x^[employee_Worker,x],Peter Piper)))
```

is used to generate the sentence

Peter Piper (the employee) is a man.

The conjunction of (C3) and the propositional content of the second sentence now however translates to (C1), so the filter can return with the information that the tuple has been completely described.

For tuples whose structure is more complex, the filter leads to a very substantial reduction in the verbosity of the output produced by the description module. For example, when generating from a `project` tuple the output produced is something like the following

```
>>>>>>  CLAM-BAKE (the project)'s end date is 19/11/1992 (the day).
>>>>>>  CLAM-BAKE (the project)'s number is 8468.
>>>>>>  CLAM-BAKE (the project)'s start date is 20/11/1989 (the day).
```

With the utterance filter disabled, the output becomes

```
>>>>>>  CLAM-BAKE (the project)'s end date is 19/11/1992 (the day).
>>>>>>  CLAM-BAKE (the project)'s number is 8468.
>>>>>>  CLAM-BAKE (the project)'s start date is 20/11/1989 (the day).
>>>>>>  CLAM-BAKE (the project)'s account is account 8468.
>>>>>>  CLAM-BAKE (the project)'s project number is 8468.
>>>>>>  CLAM-BAKE (the project)'s account number is 8468.
>>>>>>  19/11/1992 (the day) is an end date.
>>>>>>  8468 is a number.
>>>>>>  20/11/1989 (the day) is a start date.
>>>>>>  account 8468 is an account.
>>>>>>  CLAM-BAKE (the project) is a project.
>>>>>>  8468 is a project number.
>>>>>>  8468 is an account number.
>>>>>>  CLAM-BAKE (the project) began on 20/11/1989 (the day).
>>>>>>  CLAM-BAKE (the project) ended on 19/11/1992 (the day).
>>>>>>  CLAM-BAKE (the project) finished on 19/11/1992 (the day).
```

```
>>>>>>  CLAM-BAKE (the project) started on 20/11/1989 (the day).
>>>>>>  CLAM-BAKE (the project) began.
>>>>>>  CLAM-BAKE (the project) ended.
>>>>>>  CLAM-BAKE (the project) finished.
>>>>>>  CLAM-BAKE (the project) started.
```

As can been seen, the filter's simple uniform mechanism manages to eliminate several different kinds of excess verbosity, some of which are not obviously redundant and require non-trivial inference to detect.

8.3.4 Generating Questions from Incomplete Assertions

It is also possible, using the methods we have just described, to generate questions. The module in CLARE responsible for doing this is invoked by the `.ask` command: if some incomplete tuples have been created by processing assertions (see section 2.5.3), CLARE will respond to `.ask` by entering a loop where it poses questions to the user intended to elucidate the information necessary to fill the unknown argument positions. This can involve generation of both Y-N and WH-questions. We begin by giving an example dialogue: utterances by CLARE and the user are prefixed by `clare>` and `user>` respectively, and comments are in italics.

`user>  ARLA is a project.`

The user inputs a declarative statement. After processing it, CLARE has an incomplete `project` *record.*

`user>  .ask`

The user invokes the question-asking module to fill the unspecified fields in the record.

`clare> what is ARLA (the project)'s account?`

CLARE asks a question to find out the filler of the "account" field in the record, and hands control back to the user to get the answer.

`user>  1234.`

The user answers. The utterance is processed by CLARE in the normal way, which involves syntactic and semantic analysis, followed by ellipsis resolution in the context of the previous question.

`clare> account 1234 is ARLA (the project)'s account.`

CLARE prints out a paraphrase showing the result of performing resolution, to keep the user informed. It then translates the result in the usual way, merging it with the previous one as described in section 2.5.3.

```
clare> what is ARLA (the project)'s start date?
```

CLARE asks another question, this time to find out the project's start date. Processing is as before.

```
user>  1/4/93.
clare> 1/4/1993 (the day) is ARLA (the project)'s start date.
clare> what is ARLA (the project)'s end date?
```

Another question from CLARE; this time, the user replies with a sentence. Processing is still the same as usual.

```
user>  ARLA will end on 1/4/96.
```

When this sentence has been processed, CLARE has successfully filled all the fields in the tuple.

The range of questions currently handled is limited to Y-N questions whose propositional content is an existentially quantified conjunction, and WH-questions whose propositional content is a lambda-bound existentially quantified conjunction. For questions of these types, we can acquire case-based rules by a slight extension of the methods described in section 8.3.2; the only extra point is that it is necessary to store the lambda-bound variable for a WH-question. Thus for example the rule induced from the training sentence

Who loves Mary?

is

```
trl_gen([whq,
         form(verb(pres,no,no,no,y),
              A,
              B^[B,[C,A,term(q(tpc,wh,_),_,D^[personal,D],_,_),E]],_)],
        F^exists([G],
             and([personal,F],
                 [C,G,F,H]))),
        [entity_ref_term(H,E),
         personal(I),
         trlgoal([C,_,I,H],
                 genpred(C,(form(_,verb(_,_,_,_,_),
                                 J,K^[K,[C,J,_,_]],_),
                                 v(_,_))))],
        [I])
```

where the fourth argument `[I]` to `trl_gen` is the list of lambda-bound variables in the conditions; note that the second argument, the propositional content, is already a lambda-bound form. We now describe how rules of this kind can be used.

Let us begin by supposing that we have a conceptual goal C, in which some argument X is so far uninstantiated. There are two obvious strategies to use when trying to find a filler for X. The first is to attempt to frame a WH-question whose propositional content is equivalent to $\lambda X.C$; the second is to substitute some plausible value a for X in C, and then try to find a Y-N question whose content is equivalent to $C[X/a]$. CLARE can use either strategy, choosing according to the nature of the argument position. If it is an argument that has to be filled by one of a known finite set of possible code values, it tries the second strategy; otherwise, it uses the first one.

For both strategies, the major difference compared to generating assertions is that what is required is an utterance whose propositional content is *equivalent* to that of the question, not *implied* by it. We will refer to the initial expression, either $\lambda X.C$ or $C[X/a]$ as the case may be, as the *abstract question*; we can reduce the two cases to one by considering abstract questions as forms bound by a set of lambdas, where the set is empty for Y-N questions and non-empty for WH-questions.

Suppose then that the abstract question is $\lambda\vec{X}.C$ where $\vec{X}$ is a vector of variables. Generation proceeds as follows. We try to find a description rule R, generalized from a question, for which the set of lambda-bound variables is $\vec{Y}$, the conditions are $Conds$, the propositional content is $\lambda\vec{Y}.P_R$, and the following hold:

1. The cardinalities of $\vec{X}$ and $\vec{Y}$ are equal.

2. If $\vec{a}$ is a set of unique constants with the same cardinality as $\vec{X}$ and $\vec{Y}$, then $Conds[\vec{Y}/\vec{a}]$ is implied by $C[\vec{X}/\vec{a}]$.

3. $P_R[\vec{Y}/\vec{a}]$ implies $C[\vec{X}/\vec{a}]$

If these conditions are met, then the propositional content of R is equivalent to that of $\lambda\vec{X}.C$. The conditions are phased the way they are so as to allow use of the techniques presented in section 8.3.3; thus the second condition is satisfied using the compiled TRL lemmas, and the third one with essentially the same mechanism as that used by the "utterance filter" described at the end of the last section.

Chapter 9

The Reasoning Engine

CLARE's inference engine is used by several other components. It supports proofs from conjunctive contexts when performing translation (cf. section 2.4.1), and evaluates translated queries; it is also employed to compile the lemmas used when generating facts and questions (chapter 8.3), and for various other purposes during reference resolution (chapter 6). In each case, the basic functionality is some version of the following. We are provided with a Horn-clause theory Γ, a formula F (which may contain free variables $\vec{\mathbf{x}}$), and a criterion which determines permissible abductive assumptions. The task is to find a substitution θ on $\vec{\mathbf{x}}$ and a set of permissible assumptions A such that

(9.1) $\Gamma \Rightarrow (A \rightarrow \theta(F))$

or terminate with the information that no such θ and A exist. This chapter describes how the inference functionalities are realized in CLARE as a type of Prolog meta-interpreter. Since these have been described many times in the literature (e.g. (Sterling and Shapiro, 1986)), we will concentrate on the unusual features of the CLARE inference engine. The expositional strategy used will be to describe a simplified version of the inference engine, beginning with a the basic meta-interpreter for pure Horn-clause theories shown in figure 9.1 and adding features as we go along to cover new functionalities.

9.1 Logical structure of the reasoning engine

The basic form of the inference engine is that of a slightly extended Prolog meta-interpreter. Superimposed on top of it, there is an iterated-deepening A* search strategy. We will only consider the logical aspects of the inference engine in this section, postponing discussion of the search strategy until section 9.2. There are two important extensions over the standard Prolog meta-interpreter: these are to cover abductive proofs and proofs of universally quantified implications. We take them in turn.

```
prove(and(L,R)) :-
  prove(L),
  prove(R).

prove(G) :-
  horn_clause_in_theory(G <- Body),
  prove(B).

prove(G) :-
  unit_clause_in_theory(G).
```

Figure 9.1: Basic Horn-clause interpreter

9.1.1 Abductive proofs

We will use the term "abductive proof" in this section to mean construction of a proof-tree in which some leaves are undischarged assumptions. CLARE performs abductive proofs for two essentially distinct reasons. The first has been referred to many times already in chapters 2 and 3; when using logic to reason about common-sense concepts, such as those expressed in natural language, it is frequently necessary to assume background knowledge which is not part of the explicit content of the utterance's logical representation. In this case, the abductively assumed conditions are normally shown to the user in order to check their appropriateness. Abductive proofs are also used in order to compile "implicational lemmas", which are utilized by several other components of the system (see chapters 6 and 8). In this mode of operation, the reasoning engine is given a goal F containing free variables $\vec{x}$ and a set of criteria for making abductive assumptions; the assumptions may also contain free variables. If an abductive proof can be found with assumptions A and a substitution θ for the $\vec{x}$, this can be regarded as a strict (non-abductive) proof for the Horn-clause

$$\forall \vec{y}.(A \rightarrow \theta(F))$$

where $\vec{y}$ are the free variables in $\theta(\vec{x})$. We will refer to formulas of this kind as "implicational TRL lemmas", or just "TRL lemmas". As part of the process of compiling a linguistic domain theory, all TRL lemmas of a particular type are generated. At present, the criterion used for limiting abductive assumptions restricts A to being a set of which all but at most two members are *arithmetic relation goals* (cf section 2.3.2). The remaining members of A are either a single atomic predication whose predicate is a conceptual predicate, or a conjunction of two such predications sharing a common variable and licenced by a `db_chain` declaration (see the manual).

```
prove(and(L,R),AssumptionsIn-AssumptionsOut) :-
  prove(L,AssumptionsIn-AssumptionsNext),
  prove(R,AssumptionsNext-AssumptionsOut).

prove(G,AssumptionsIn-AssumptionsOut) :-
  horn_clause_in_theory(G <- Body),
  prove(Body,AssumptionsIn-AssumptionsOut).

prove(G,AssumptionsIn-AssumptionsIn) :-
  unit_clause_in_theory(G).

prove(G,AssumptionsIn-[G|AssumptionsIn]) :-
  abductively_assumable(G).
```

Figure 9.2: Abductive Horn-clause interpreter

Two minor changes are needed to extend a Prolog meta-interpreter to allow abductive proofs: another argument is added to the main predicate, which passes around a difference-list holding the assumptions, and an extra clause is added which allows "proof" of an atomic goal by adding it to the assumption-list, if the abductive assumability criteria permit this. The basic interpreter, modified in the way described, is shown in figure 9.2.

9.1.2 Proving implications

We now consider the task of extending the inference engine to be able to prove formulas of type

```
forall(Vars,impl(LHS,RHS))
```

where the variables in `Vars` occur free in `LHS` and `RHS`. Two well-known strategies exist for attempting to prove such expressions; they can be proved either *intensionally* or *extensionally*. Intensional proofs are carried out by substituting unique constants for the `Vars`, assuming `LHS`, and attempting to prove `RHS`; extensional proofs by listing all values of `Vars` for which a proof of `LHS` can be found, and for each one ascertaining that a proof of `RHS` exists. The CLARE inference engine can use either strategy.

To implement the intensional proof strategy it is necessary to add yet another argument to the `prove` predicate in the interpreter, to hold the set of assumptions derived from the `LHS`; it is tempting to try and include them in the abductive assumption list, but in practice this manoeuvre does not seem to simplify matters. The need to take account of possible abductive assumptions also introduces some complications in the extensional strategy, since it is necessary to collect the assumptions used for each separate proof of `RHS`. The skeleton interpreter, modified to allow proof of universally quantified implications by the intensional and extensional strategies, is presented in figure 9.3. Here the first clause of form `prove(forall...)` is for the intensional strategy; `replace_by_unique_-constants` is assumed to be a predicate which replaces all the variables in its argument by unique ground terms. The following clause is for the extensional strategy. Here `safe_bagof(X,G,Xs)` is like `bagof(X,G,Xs)`, with the difference that it first binds any free variables in G, and succeeds with `Xs=[]` if there is no proof of `G`; `union_n_1(List,Union)` takes a list of lists as its first argument and returns their union as the second one.

9.2 Search strategies for inference

In this section we consider the search strategy used in the inference engine, which as already stated is an iterated-deepening A* search strategy (Korf, 1986, Stickel, 1986); that is, search proceeds by first defining a cost limit D, and then performing a depth-first search where each branch is cut off when the sum of the cost of goals already processed, together with an estimate of the cost of solving pending goals, exceeds D. The cost function used at present charges one unit for each rule application, and conservatively estimates one unit for each pending goal; as explained in section 3.6, it is possible to define extra costs for abductively assumed goals. The motivation for using IDA*, rather than simple depth-first search, is that it avoids getting stuck in infinite recursions; it essentially performs the function of simulating breadth-first search, and is thus theoretically complete. However, experiment quickly shows that infinite recursion still causes severe practical difficulties without further elaboration of the method.

There are basically three problems, which to some extent are related. Firstly, since the greater part of the Horn-clause theory is derived from a set of equivalences (cf. section 2.4.1) it is easy to wander aimlessly around in the search-space, alternately using "normal" and "backward" Horn-clause versions of equivalences. Secondly, even when this does not happen, it may be the case that there are multiple redundant proofs of a goal. The third problem arises from inefficient ordering of conjunctions. When trying to prove a conjunction, it can sometimes be the case that there are many proofs of the first conjunct (which take a correspondingly long time to generate), but that the second conjunct can quickly be discovered to have either one or no proofs; in this case attempting to prove the conjuncts in a different order can pay dividends. The non-obvious aspects of the search strategy, which we will now describe, stem from attempts to counteract these problems. We describe them one at a time.

```
(All clauses from the interpreter in figure9.2)


prove(G,AssumptionsIn-AssumptionsIn,ImplAssumptions) :-
  member(G,ImplAssumptions).

prove(forall(Vars,impl(LHS,RHS)),AssIn-AssOut,ImplAss) :-
    replace_by_unique_constants(Vars),
    add_implicational_assumptions(LHS,ImplAss,ImplAss1),
    prove(R,AssIn-AssOut,ImplAss1).

prove(forall(Vars,impl(LHS,RHS)),AssIn-AssOut,ImplAss)
    safe_bagof(environment(Vars,AssOutLeft),
               prove(LHS,AssIn-AssOutLeft,ImplAss)
               LeftEnvironmentList),
    prove_forall_cases(LeftEnvironmentList,Vars^RHS,
                       ImplAss,AssOutList),
    union_n_1(AssOutList,AssOut).

prove_forall_cases([],_Vars^_RHS,_ImplAss,[]).
prove_forall_cases([EnvFirst|EnvRest],Right,ImplAss,
                   [AssOutFirst|AssOutRest]) :-
    prove_forall_case(EnvFirst,Right,ImplAss,AssOutFirst),
    prove_forall_cases(EnvRest,Right,ImplAss,AssOutRest).

prove_forall_case(environment(VarsLeft,AssOutLeft),
                  Vars^RHS,ImplAss,AssOut) :-
    copy_term([Vars^RHS,ImplAss],[Vars1^RHS1,ImplAss1]),
    VarsLeft = Vars1,
    prove(RHS1,AssOutLeft-AssOut,ImplAss1).

add_implicational_assumptions(and(P,Q),ImplAssIn,ImplAssOut) :- !,
    add_implicational_assumptions(P,ImplAssIn,ImplAssNext),
    add_implicational_assumptions(Q,ImplAssNext,ImplAssOut).
add_implicational_assumptions(G,ImplAss,[G|ImplAss]) :-
    atomic_goal(G), !.
add_implicational_assumptions(_Other,ImplAss,ImplAss).
```

Figure 9.3: Abductive interpreter for Horn-clauses and implications

9.2.1 Avoiding infinitely recursive branches

We currently use two different approaches to help minimize time spent in infinitely recursive branches of the search-tree. The first approach is based on a standard loop-avoidance method. It can be proved that it is sound to cut off search if a goal is *identical* to one of its ancestors. The inference engine uses this criterion, both in its simple form and in a "look-ahead" mode: when expanding a goal G to a conjunction $B_1 \wedge B_2 \ldots$, it checks before adding the B_i to the stack of pending goals, to make sure that none of them are identical to an ancestor. Perhaps surprisingly (since this is a fairly expensive operation), it turns out that performing the "look-ahead" check increases efficiency quite substantially.

It would be pleasant if the "identity" condition could be relaxed a little, for example by replacing it with a test of unification or subsumption with one of the ancestor goals. Unfortunately, it is easy to prove that weakening the condition is in general not correct. However, a strong syntactic similarity between a goal and an ancestor is often a good indicator that a loop has arisen, and can be taken as a justification for exacting a "penalty", causing the cost limit to be reached more quickly. In the current version, the penalty is applied if the goal is subsumed by an ancestor, a strategy which we have found to work well in practice. The method is sound, since its only effect is to devote less effort to search of the affected branches without cutting them off completely, and appears rather more general than the approach advocated in (Appelt and Hobbs 90), which involves recoding of the meaning postulates.

The second approach to the "looping" problem builds on the assumption that the equivalences should be read as translation rules; the point is to exploit this when using them in their compiled Horn-clause forms. Recall that

(9.2) $P_1 \wedge P_2 \wedge \ldots \leftrightarrow Q_1 \wedge Q_2 \wedge \ldots$

produces one "backward" Horn-clause,

(9.3) $Q_i \leftarrow P_1 \wedge P_2 \wedge \ldots$

for each Q_i. Now suppose that Q_i occurs as a condition when trying to expand some other goal G. We demand that the rule (9.3) only be invoked to prove Q_i if this is consistent with the assumption that Q_i could be produced from the original rule by a chain of translations the last one of which used (9.2). (In other words, (9.3) can only be used to attempt to reconstruct such a chain). What this means in terms of limiting search is that the first goal in (9.3), P_1, must now be proved either by looking it up in the environment of assumptions, or by invoking a further "backward" Horn-clause of which P_1 is the head. The result is that search now proceeds only by chaining "backward" rules, until either a goal can be found in the environment or a dead-end is reached, a process that normally terminates quickly. Without the restriction, any "backward" rule invocation could be followed by a "normal" one, leading to aimless meandering in the search-space that is only terminated by the depth-bound being exceeded. For non-trivial sets of equivalences the restriction on use of "backward" Horn-clauses seems necessary, but also intuitively acceptable.

9.2.2 Avoiding redundant search

The second problem, that of avoiding redundant recomputation, is currently only applied to ground goals, that is goals which contain no uninstantiated variables. We have observed in many cases that, without some kind of restrictions, a great deal of effort is wasted in producing multiple proofs for goals of this kind. Since a proof of a ground goal is basically just a "yes" or "no", it is tempting to adjust the backtracking mechanism so as to block attempts to find new proofs of ground goals that have already been successfully proved. However, this is not quite right; the problem is that the ordering of the search-space may mean that the first proof produced is an expensive one, which leaves the search process on the verge of exceeding the cost limit, and prevents further progress. The correct realization of the idea seems rather to be to block retrying of ground goals only if the cost of the existing proof falls under a given bound, which we have at present arbitrarily fixed as a third of the total cost limit.

9.2.3 Dynamic re-ordering of conjuncts

The time taken to find a proof for a conjunctive goal (or to ascertain that there is no proof) can often be strongly dependent on the order in which the conjuncts are attempted. We have made no attempt to solve this problem in its general form, but two special cases are sufficiently important that the inference engine needs to take account of them. Suppose first that the interpreter is about to attempt to prove the conjunction $L \wedge R$. If it is the case that there is no proof of R, then time lost establishing this can be recovered by omitting the attempt to prove L. Thus it can pay to perform "look-ahead" and search for goals which can quickly be shown to be unachievable. Similarly, if L contains free variables which also occur in R then the time needed to find a proof can be reduced if the number of proofs for R is substantially less than the number of proofs for L, since instantiating free variables nearly always has the function of reducing the number of possible proofs. In particular, if it can quickly be shown that there is only *one* proof of R then it is very likely that it will be correct to attempt it first.

It is possible to include declarations of the form

```
quick_failure_test(<Goal>) :- <Conds>
```

or

```
quick_determinism_test(<Goal>) :- <Conds>
```

where `<Goal>` is an atomic goal and `<Conds>` an arbitrary Prolog form; the intended semantics are that there should be, respectively, no proofs or exactly one proof of `<Goal>` if `<Conds>` hold. For declarations of this kind to be useful, `<Conds>` should be quickly evaluable. In the present version of CLARE, instances of `<Conds>` never do more than check instantiations of variables, or attempt to unify them with other terms. Tests for both cases are applied when a Horn-clause is used to expand a goal `G`; after the head of the clause has been unified with `G`, the goals in the body are examined. If a `quick_failure_test` succeeds then proof of the whole conjunction fails; if a `quick_determinism_test` succeeds on a subgoal, then that subgoal is evaluated first. There are currently about 30 "quick failure and determinism" declarations, all of them for the "conversion" predicates described in section 3.5.10.

9.3 Finding finite proof procedures

In the final section of this chapter we describe the module responsible for searching for finite proof procedures (see also section 2.3.2). Since the inference engine is basically a Prolog meta-interpreter, it is sensitive to the ordering of conjuncts: it attempts to prove a goal of the form `and(P,Q)` by first proving `P` and then proving `Q`. If `P` and `Q` share a free variable `X`, the order can be important. To repeat the example from 2.3.2, if $TRANS/3$ is a database predicate then the strategy "find an X such that $X > 100$, then find values of Y such that $TRANS(john, X, Y)$" is not a finite strategy; however, reversing the order to make the strategy "find X and Y such that $TRANS(john, X, Y)$, then determine whether $X > 100$ holds" is finite.

The search for finite proof strategies is carried out by another extended Prolog meta-interpreter, using an abstract interpretation method. The information needed to deal with the base case of attempting to prove an atomic goal is provided by declarations of the form

```
call_pattern(<Pred>(Arg1,Arg2,...Argn),InArgs,OutArgs)
```

where `<Pred>` is a predicate and `InArgs` and `OutArgs` are subsets of the set {`Arg1, ..., Argn`}. The intended semantics are that there are finitely many proofs of the goal

```
<Pred>(Arg1,Arg2,...Argn)
```

if the variables in `InArgs` are already instantiated, and that each of them will result in the arguments `OutArgs` becoming instantiated. The basic structure of the interpreter is shown in figure 9.4; only the clauses for conjunctive and atomic goals are displayed. In the implemented interpreter, there is a clause for each logical operator of the TRL language. The central idea is to rearrange the expression by using an abstract version of the Prolog "freeze" or delayed-goal

```
rearrange(OldForm,NewForm) :-
    copy_term(OldForm,EvalForm),
    rearrange0(OldForm,NewForm,EvalForm,[]-[],[]-_), !.

rearrange0(Old,New,Eval,FrozenIn-FrozenOut,DoneIn-DoneOut) :-
    rearrange1(Old,New0,Eval,FrozenIn-FrozenNext,DoneIn-DoneNext),
    try_to_thaw(New0,New,FrozenNext-FrozenOut,DoneNext-DoneOut).

rearrange1(and(LOld,ROld),and(LNew,RNew),and(L,R),
           FrozenIn-FrozenOut,DoneIn-DoneOut) :- !,
    rearrange0(LOld,LNew,L,FrozenIn-FrozenNext,DoneIn-DoneNext),
    rearrange0(ROld,RNew,R,FrozenNext-FrozenOut,DoneNext-DoneOut).
rearrange1(OldG,OldG,G,Frozen-Frozen,Done-[goal_executed|Done]) :-
    abstract_call(G),!.
rearrange1(OldG,true,G,Frozen-[frozen(G,OldG)|Frozen],Done-Done).

try_to_thaw(Form,Form,[]-[],Done-Done) :- !.
try_to_thaw(Form,OutForm,
            [frozen(FrozenEval,FrozenOld)|FrozenNext]-FrozenOut,
            DoneIn-DoneOut) :-
    rearrange1(FrozenOld,FrozenNew,FrozenEval,[]-[],[]-[_|_]),!,
    try_to_thaw(and(Form,FrozenNew),OutForm,FrozenNext-FrozenOut,
                [goal_executed|DoneIn]-DoneOut).
try_to_thaw(Form,OutForm,
            [FrozenFirst|FrozenNext]-[FrozenFirst|FrozenOutRest],
            DoneIn-DoneOut) :-
    try_to_thaw(Form,OutForm,FrozenNext-FrozenOutRest,DoneIn-DoneOut).

abstract_call(X=Y) :-
     X = Y, !.
abstract_call(_X=_Y) :- !.
abstract_call(G) :-
     call_pattern(G,InArgs,OutArgs),
     is_instantiated(InArgs),
     make_instantiated(OutArgs).
```

Figure 9.4: Interpreter for finding finite proof strategies

primitive. The arguments of the main predicates `rearrange0` and `rearrange1` are

```
rearrange0/1(Original,Rearranged,Eval,Frozen,Done)
```

where

`Original` is the original expression.

`Rearranged` is the rearranged expression.

`Eval` is a copy of `Original` used to keep track of the variables currently instantiated in the abstract interpretation.

`Frozen` is a difference-list of delayed goals, i.e. goals which were insufficiently instantiated to be executed when first called.

`Done` is a difference-list that keeps track of progress made in evaluating goals, to prevent infinite postponement of goal execution.

the two predicates call each other recursively, with `rearrange0` calling `rearrange1` and then trying to "thaw" any frozen goals that may have been created. Note the special treatment of equality goals: if `Eval` is an equality, its arguments are unified if possible.

Chapter 10

The Preference Mechanism

Linguistic coverage of the CLE subsystem of CLARE has been increasing gradually as the system develops, both in terms of grammatical coverage and in the coverage of referential terms handled during interpretation. While this has the obvious advantage of being able to handle new types of sentences, it also increases the number of analyses and interpretations produced for a sentence. If these alternatives are regarded as equally likely, then for sentences that were already treated by the narrower coverage the chance of selecting the correct interpretation of a sentence can decrease with wider coverage. Clearly this problem can be overcome by providing a mechanism that gives an accurate indication of when to prefer one alternative over another. This chapter describes preference metrics and mechanisms implemented in CLARE (for related approaches, see McCord 1989 or Hobbs and Bear 1990). We start by describing the role of the preference mechanism in the treatment of alternatives in the current architecture, then the general approach to computing preferences. The last three sections describe the actual preferences used by the system.

10.1 Preference and the CLE Architecture

The preference mechanism applies to unresolved semantic analyses as well as to resolved and scoped interpretations. More specifically, the mechanism is currently applied at the following three points in the course of processing input sentences:

- the derivation of a set of unresolved QLFs by the application of semantic rules and sense entries to the packed syntactic analysis records;
- the derivation of resolved QLFs by the application of reference resolution rules;
- the application of the scoping algorithm to resolved QLFs yielding scoped resolved QLFs.

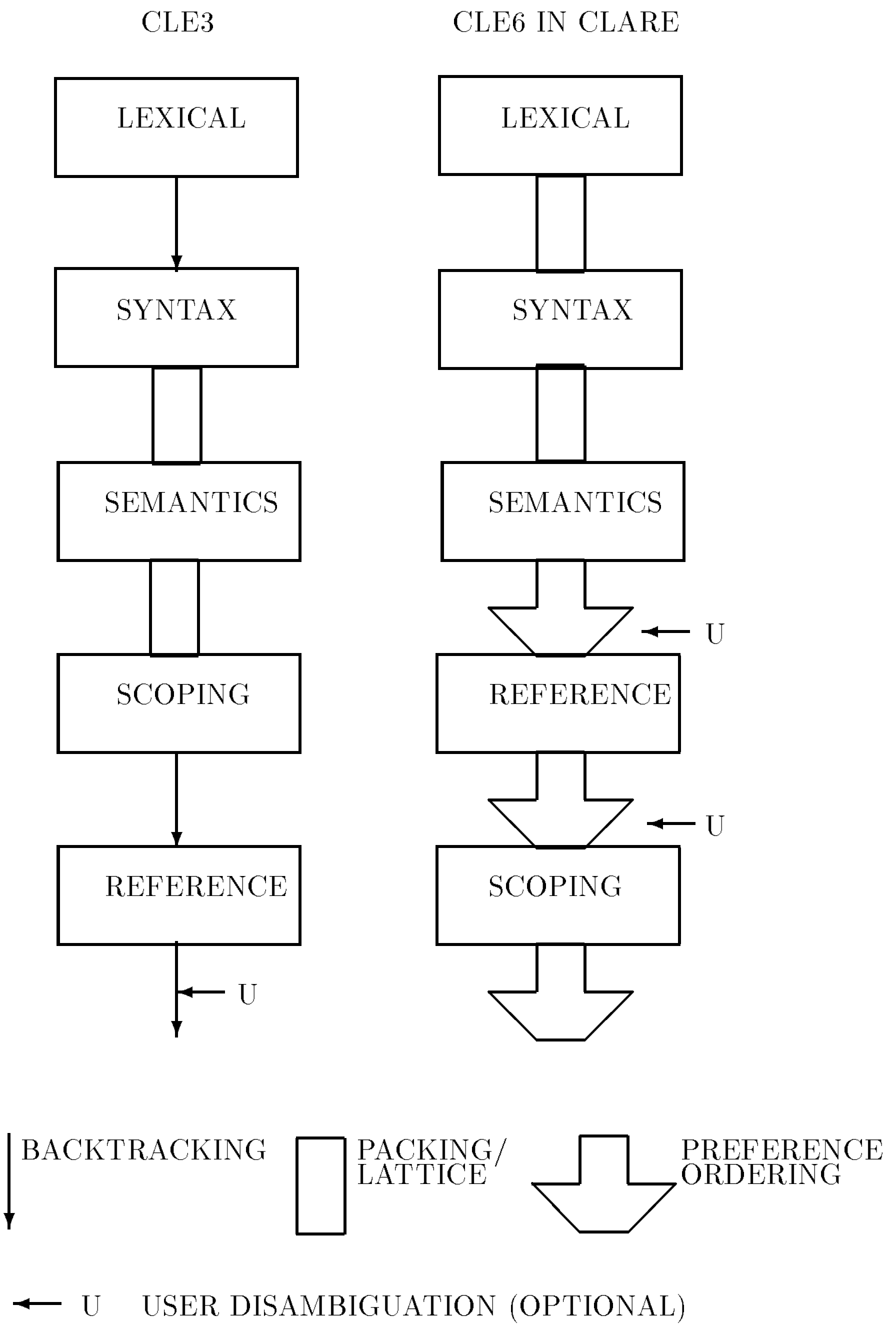


Figure 10.1: Handling of alternatives by main CLE phases

Note that reference resolution takes place before scoping in the revised architecture of the CLE subsystem employed in CLARE. This change was made possible by the implementation of a more monotonic model for semantic interpretation as discussed in detail in Chapter 6. At each of the points listed, the preference mechanism attempts to ensure that alternatives are passed to the next phase of processing starting with the most likely alternative, the next most likely alternative, and so on.

As in earlier versions of the CLE, all syntactic analyses for the sentence are represented implicitly in the packed syntax records, but unlike earlier versions, explicit QLFs are derived directly from the syntax records rather than being first coded as packed semantic records. This change is related to the following considerations: (i) direct computation of QLFs allows a restriction on the form of semantic rules (daughter QLFs being restricted to variables) to be lifted, making the rule set more amenable to efficient generation, (ii) since preferences on QLFs often depend on non-local structural properties, it is difficult to compute preference measures for a set of QLFs represented as records with local ambiguity packing, and (iii) prepositions are now given single vague-relation meanings in unresolved QLF (the relations are refined during reference resolution), so much of the ambiguity which motivated packed semantic records has been removed from this level of analysis.

The overall changes in the handling of alternatives in CLARE compared with the CLE in NATTIE are summarized in the Figure 10.1.

The preference mechanism improves user interaction since alternatives can be presented for confirmation starting with the most likely alternative analysis or interpretation according to the preference measures. It should therefore reduce the amount of interaction that is necessary, or even allow the user to run the system in fully automatic mode.

Figure 10.1 also shows that user interaction is no longer at the end of the whole interpretation process, when a resolved LF was presented to the user, but instead takes place after semantic analysis and reference resolution. These two points were chosen for interactive disambiguation because they involve choices that can often be displayed in an informative way to non-expert users (QLFs and LFs are no longer displayed in "user mode"). After semantic analysis, word sense choices and meaningful phrasal analysis (i.e. identification of phrases corresponding to objects or basic assertions) are presented. After resolution, new entities and relations identified in a possible paraphrase using the referents (examples are presented in Section 10.2).

10.2 User Interaction for QLF Selection

After applying its own preference criteria, the system by default presents the user with information about the various QLFs in order of preference. The first QLF the user accepts is the one used for further processing.

Rather than print the QLF itself, which would be useful only to specialists, the system gives a bracketing of the sentence, showing the noun phrases it is analysed as containing according to that QLF. It also prints paraphrases of word senses, contrasting them with alternatives that may be defined in the lexicon.

Thus inputting the sentence *John met the man in the bank* causes the following to be printed:

```
    Complete sentence with bracketing:

      "{John} met {{the man} in {the bank}}."

    Word senses (unordered):

       meet: encounter (rather than "be adjacent")
       man: male person
       bank: company (rather than "building" or "edge")

 Confirm this analysis? (y/n/c/p/?):
```

The user is being told that according to this analysis,

- the input is being interpreted as a complete sentence rather than an elliptical one; an ambiguity of this type would occur for the input *the men met in the bank*, which can either be a sentence or a noun phrase meaning *the men who were met in the bank*;
- *the man in the bank* is a constituent; i.e. *in the bank* attaches to *man* rather than *met*;
- *meet* is being interpreted as "encounter" rather than "be adjacent" – a sense that would be appropriate for *the two walls meet at the corner of the property*;
- *man* is being interpreted as "male person" – there is no alternative to this;
- *bank* is being interpreted as a kind of company rather than as a building or as the edge of something such as a river.

The user may accept the analysis by typing `y` or reject it with `n`. He or she may also have the QLF itself printed by typing `p` and then make a decision. However, it is also possible to reject it and specify the reasons why it is unsatisfactory by typing `c` (for "constraints"). This facility can greatly reduce the amount of interaction required when there are a large number of QLFs which correspond to the various combinations of decisions on several different ambiguities.

Let us suppose that the user is unhappy with the QLF because of the sense of *bank* chosen, but wants to confirm the bracketing. The response to `c` is a menu:

```
Possible constraints:
    1 This bracketing
    2 Complete sentence (rather than phrase)
    3 "meet" meaning "encounter"
    4 "bank" meaning "company"
Enter some signed numbers (+n to insist, -n to disallow)
separated by spaces:
```

If the user gives the response "`+1 -4`", then QLFs whose bracketings are not as given above will be rejected without the user seeing them, as will QLFs where "bank" means "company". The next QLF that matches the user's constraints is therefore described as follows:

```
   Complete sentence with bracketing:

     "{John} met {{the man} in {the bank}}."

   Word senses (unordered):

      meet: encounter (rather than "be adjacent")
      man: male person
      bank: edge (rather than "building" or "company")

Confirm this analysis? (y/n/c/p/?):
```

This time, the user might simply time `n`, to reject the QLF. If so, a further QLF, with *bank* this time meaning "building" but with the same bracketing, is tried.

In this way, users without an understanding of the QLF language can make intelligent interpretation choices. They can also avoid a lot of unnecessary interaction; in the above example, only three QLFs were presented out of a total of six found by the system. The other three are automatically rejected because of the bracketing constraint.

10.3 Preference Measures

At each of the points at which the preference mechanism is applied, a numerical 'preference measure' is computed for each of the alternative representations being considered at that point, and the alternatives are sorted into an order respecting the preference measure. There are system parameters (`maxqlfs`, etc.) for controlling the maximum number of alternatives considered at each stage, and a minimum preference measure threshold below which alternatives are discarded immediately. Different preference measures are used for the different points at which the mechanism is applied: the measures are meant to reflect the contribution of new choices made during interpretation that have not yet been evaluated by the mechanism.

Most (unscaled) scoring functions are simply integers corresponding to counts of particular constructs in the representation, for example the number of unresolved ellipsis `forms` in QLF. Others, such as the syntax rule score and the semantic collocation metrics are real numbers derived from automatic training processes. Some of the preferences now implemented as scoring functions with relatively high negative scaling factors were previously regarded as inviolable constraints applied, in particular, to scopings and resolutions. We now turn to the scoring functions used, respectively, to compute preference measures for QLFs, resolved QLFs, and scoped resolved QLFs respectively.

A preference measure is itself a weighted sum of a set of scoring functions declared by assertions of the form

```
scoring_function(Identifier,Fi,Ci).
```

Here `Id` is an identifier for the preference measure (e.g. `qlf`, `scope`), `Ci` is a scaling factor (positive or negative) and `Fi` is the name of the scoring function. Examples are:

```
scoring_function(qlf,qlf_left_nesting,-3)
scoring_function(qlf,qlf_sense_weights,10).
```

The scoring function takes a complete representation (QLF or resolved QLF) as input, returning a numerical value `Si`, the preference measure being computed by summing over the product of scaled scores for all scoring functions:

```
C1S1+...+CmSm.
```

10.4 Scaling Factors

When the preference mechanism was first used, an initial set of scaling factors was chosen by hand according to knowledge of the how the particular raw preference metrics were computed and introspection about the 'strength' of the metrics as indicators of preference or dispreference. These initial scaling factors were subsequently revised according to their observed behaviour in ranking analyses in the CLARE PRM application. This tuning process was time consuming but it did eventually lead to reasonably well behaved rankings for this domain and application. It is difficult to quantify these remarks because, for this application, there was no independently collected corpus of examples to use for evaluation.

However, there are a number of disadvantages to manual tuning of scaling factors. These include the effort spent in maintaining the parameters, this effort being greater for those with less knowledge of how the raw preference functions are computed since this increases the effort for trial-and-error tuning. A point of diminishing returns is also reached after which further attempts at improvement through hand-tuning often turn out to be counter-productive. Another problem was that it became difficult to detect preference metrics that were ineffective, or simply wrong, if they were given sufficiently low scaling factors. Probably a more serious problem is that the contributions of different preference factors to selecting the most plausible analyses seem to vary from one sublanguage to another.

These disadvantages point to the need for automatic procedures to determine scaling factors that optimize preference metric rankings for a particular sublanguage. This requires a corpus of sentences that are representative of the sublanguage and some independent measure of the correctness or plausibility of analyses of these sentences. (With a sufficiently large corpus, it may be possible to start with some initial plausibility measure and refine it through re-estimation, although we have not attempted this.) One way of gathering the correctness information is to retain the choices made during interactive disambiguation by operators of information system interfaces or machine aided translation systems. This could result in systems whose automatic preference choices would improve gradually over time.

Work on optimizing the scaling factors for the CLARE preference mechanism has been carried out under a project for building a speech translation system based on CLARE's linguistic components (Section 13.5). These experiments made use of the Air Travel Information System (ATIS) corpus of transcribed speech sentences. We chose this application for these experiments because there existed a hand-parsed sub-collection of this corpus built as part of the University of Pennsylvania "Treebank" project. Further skeletal trees induced by QLF analyses, and verified manually, were collected at Cambridge. We then defined a penalty function on QLFs which reflected the extent to which a QLF analysis for a sentence differed from the treebank analysis. It was then possible to determine an optimal set of scaling factors, with respect to this penalty function, using the least squares minimization: this turns into the straightforward analytic procedure to find the unique solution of a set of linear simultaneous equations. The experiments are described in Alshawi and Carter 1992. In summary, the results from blind testing indicated that despite the crude penalty function used, the method yielded a set of scaling factors that is at least as good as the set derived by labour intensive hand tuning.

10.5 QLF Preferences

The scoring functions contributing to the preference measure for QLFs can be divided into two categories, those expressing structural preferences and those expressing semantic preferences. We will present examples relevant to some of the fifteen or so QLF scoring functions currently in use.

10.5.1 Structural Preferences

QLF structural preferences are often the same as, or very similar to, those proposed in previous work for selecting between possible syntactic analyses (Hobbs and Bear 1990 discuss principles underlying parsing preferences). Although the QLF representation is semantic in nature, it is possible to use such syntactic preference criteria because of the presence of linguistic categories in QLF and because much of the phrasal structure of sentences is implicit in QLFs. Structural QLF preferences include the following:

- Prefer subcategorization to adjuncts. For example, the reading of *John laughed at the school* in which *laugh at* is treated as a verb-particle construction is preferred to the reading in which *at the school* is an optional modifier.

- Prefer low attachment. For example, this prefers the reading of *John saw a cat in a tree in a garden* in which the cat is in a tree that is in a garden.

- Disprefer demonstrative readings of *that* and *there*. This applies to examples like *I know that sugar is expensive* and *What cars are there?*

- Prefer QLFs with structurally balanced conjunctions, for example the more likely reading of *They sell motorcycles in France and caravans in Germany.* (There is also a sort parallelism preference for conjunctions.)

10.5.2 Semantic Preferences

QLF semantic preferences are used to downgrade vague analyses, such as those involving `forms`, or to prefer structures which lexical semantics makes more plausible. Semantic QLF preferences include the following:

- Prefer QLFs which include more frequently observed semantic collocations. This type of metric is considered in more detail in the section below on semantic head collocations.

- Disprefer QLFs with `forms` corresponding to NBAR ellipsis. For example, this disprefers analysing *Mary sold some tickets* as elliptical for, say, *Mary sold tickets to some customers.*

- Disprefer QLFs with `forms` arising from verb phrase ellipsis. For example, this disprefers analysing *Mary is in Cambridge* as elliptical for, say, *Mary is working in Cambridge.*

- Disprefer readings with properties as arguments. The progressive reading of *It is eating* is thus preferred over the equative reading.

- Prefer QLFs which include word senses with higher probabilities indicated by `weight` declarations in the lexicon. These preferences are often application specific. For example, in the PRM supplementary lexicon, the 'money' sense of *pound* is strongly preferred to the 'weight measure' sense.

10.5.3 Semantic Head Collocations

Semantic collocation metrics, especially those determined statistically have recently attracted a lot of attention in computational linguistics (Calzolari and Bindi 1990, Church and Hanks 1990, Hindle and Rooth 1990, Sekine et al. 1992). They are typically derived by observing the occurrences of tuples (usually pairs or triples) that summarize relationships present in a semantic analysis of a text, or relate lexical heads of phrases.

In our case, we use semantic collocations extracted from QLF expressions in the form of `(H1,R,H2)` triples where `H1` and `H2` are the head predicates of phrases in a sentence and `R` indicates the semantic relationship (e.g. a preposition or an argument position corresponding to a case relation) between the two phrases in the proposed analysis. Semantic collocation metrics are especially relevant to choosing between analyses with different prepositional attachments and different word senses; the can be viewed as a statistical analogue of selectional restrictions based on concept hierarchies.

Data collection for the semantic collocation metrics proceeds by deriving a set of triples from each QLF analysis of the sentences in the training set. This is followed by statistical analysis to produce the following functions of each triple in the observed triple population:

- mutual information: this relates the probability of the triple assuming independence between its three fields (`P1*P2*P3`) with the probability `A` estimated from actual observations of triples derived from analyses ranked highest (or joint highest) by the tree bank goodness metric. More specifically we use `ln(A)-ln(P1)-ln(P2)-ln(P2)`.

- chi-squared: compares expected frequency `E` of a triple with the square of the difference between the observed frequency `F` of the triple and the expected frequency. Here the observed frequency is in analyses ranked highest (or joint highest) by the tree bank goodness metric and the "expected" frequency assumes independence between triple fields. More specifically we use `|F-E|*(F-E)/E` (This variant of chi-squared in which the numerator is signed is used so that the function is monotonic making it more suitable in preference metrics.)

- mean distance: the average of the tree bank goodness metric for all QLF analyses (not necessarily highest ranked ones) which includes the semantic collocation corresponding to the triple.

Computation of the mutual information and chi-squared functions for triples involves the simple smoothing technique of adding 0.5 to actual counts. From these three functions on triples we define three semantic collocation preference metrics applied to QLF, in each case by summing over the result of applying the function to each triple derived from a QLF. In the experiments described in Alshawi and Carter 1992 we compared the effectiveness of these semantic collocation metrics. Chi-squared performed best, with mean distance a close second and mutual information performing significantly less well than the other two, despite our impression that it is the most commonly used collocation metric.

10.6 Resolved QLF Preferences

Resolved QLF preferences can be split into two broad types: those that depend on the precise resolution method used, and those that depend solely on the form of the finally resolved QLF. This division arises because more than one resolution may be applicable to an unresolved expression, but one cannot tell which method was used just by looking at the form of the resolved expression. For example, a domain specific and a non-specific resolution method may give rise to exactly the same resolution, but greater weight should be placed on the domain specific case.

To impose a preference ranking over resolution methods, `rule_score` declarations are given. These take the form

```
rule_score(Method,Category,Weight).
```

`Method` names a resolution method, `Category` specifies a category of unresolved expression to which the method can be applied, and `Weight` assigns a preference weighting to the use of the method on that category. (Currently, the category is left unspecified in all declarations). During resolution, a score is accumulated as resolution methods are applied. The overall score is then weighted by the

```
scoring_function(ref(_),ref_rules_score,Weight).
```

declaration.

Other preference metrics operate in the more conventional way by considering just the resolved QLF. These preferences include

- Checking for violations of reflexive and non-reflexive pronoun constraints. Violations are heavily penalised.

- Penalising duplicate resolved QLFs. Duplicates may sometimes arise through distinct resolution methods leading to identical results.
- Penalising sortal violations.
- Dispreferring non-parallel substitutions in ellipsis, e.g. where a non-quantified term is substituted for a quantified one or vice versa.
- Scoring resolutions of determiners to different collective and distributive quantifiers. `det_to_quant` rules are given scores which contribute to the ranking, and certain QLF constructions can be identified as demanding collective (or distributive) quantifiers.

The number of resolutions made in the first place, and the number returned after preference ranking are controlled by setting a resolution limit and a ranked resolution limit respectively. Also, resolution is carried out in two stages, which lead to different metrics being used. The first bout of resolutions deals with 'major' resolutions, e.g. resolution of pronouns, vague relations, ellipsis, and so forth. These resolutions are fairly easy to paraphrase back to the user for confirmation. The second stage involves resolving determiners to quantifiers and deciding on distributive or collective readings, which are not so easily paraphrasable. Separate limits are defined for the two stages.

10.7 Scoping Preferences

Preferences defined for scoped QLFs include

- Penalising redundant scope permutations, e.g. ignoring one of $\exists X \exists Y$ and $\exists Y \exists X$
- Heavily dispreferring violations of domain specific functional information (see Section 6.4).
- Scoring how far quantifiers have been 'raised' through the QLF
- Making pairwise comparisons of quantifier – quantifier scopes, and quantifier – operator scopes. These call on subsidiary definitions stating four way preferences about the relative scopes of specific quantifiers and operators.

Chapter 11

Lexical and Syntactic Analysis and Failure Recovery

11.1 Introduction

In many language processing systems, uncertainty in the boundaries of linguistic units, at various levels, means that data are represented not as a well-defined sequence of units but as some kind of lattice. It is common for speech recognizers to maintain a lattice of overlapping word hypotheses from which one or more plausible complete paths are subsequently selected. Syntactic parsing, in the CLARE and many other systems, makes use of a chart or well-formed substring table because the correct bracketing of a sentence cannot (easily) be calculated deterministically. And lattices are also often used in the task of converting Japanese text typed in kana (syllabic symbols) to kanji; the lack of interword spacing in written Japanese and the complex morphology of the language mean that lexical items and their boundaries cannot be reliably identified without applying syntactic and semantic knowledge (Abe *et al*, 1986).

In contrast, however, most systems accepting typed input have attempted to group their input deterministically into a sequence of words and punctuation symbols to be handed to the parser, possibly performing word-by-word morphological analysis on the way. Such an approach is sometimes adopted even when typographically complex inputs are handled; see, for example, Futrelle *et al*, 1991.

In this chapter we argue that to deal with a range of phenomena that occur in written English, a lattice representation must also be used in front end (pre-parsing) analysis. We describe how the CLARE system uses such a representation, and show how this allows the system to deal straightforwardly with combinations of phenomena that would be difficult or impossible to process correctly under a sequence representation. The chapter also describes changes to the version of the CLE parser described in section 7.1 of Alshawi *et al* (1992), which was current near the beginning of the CLARE project.

For typed input, spaces do not necessarily correspond to boundaries between lexical items, both for linguistic reasons and because of the possibility of typographic errors. This means that a lattice representation, not a simple sequence, should be used throughout front end analysis. As evidence for the performance of the approach taken, we describe an evaluation of CLARE's ability to deal with typing and spelling errors. Such errors are especially common in interactive use, for which CLARE is designed, and the correction of as many of them as possible can make an appreciable difference to the usability of a system.

The word identity and word boundary ambiguities encountered in the interpretation of errorful input often require the application of syntactic and semantic knowledge on a phrasal or even sentential scale. Such knowledge may be applied as soon as the problem is encountered; however, this brings major problems with it, such as the need for adequate lookahead, and the difficulties of engineering large systems where the processing levels are tightly coupled. To avoid these difficulties, CLARE adopts a staged architecture, in which indeterminacy is preserved until the knowledge needed to resolve it is ready to be applied. An appropriate representation is of course the key to doing this efficiently.

When an input is outside the system's grammatical coverage or is judged sortally incoherent, it is much more difficult to make a correction than when a spelling error occurs. This is because the space of possibilities is much larger and less well-defined; even if one could enumerate the possible corrections, choosing between them would be very difficult, and there would often be so many of them that no motivated choice could be made in a reasonable time. CLARE therefore reacts to grammatical and sortal problems not by attempting a recovery but by offering the user an analysis of them which is intended to provide guidance for rephrasing the input or, during development, modifying linguistic data.

11.2 Lattices for Lexical Processing

In general, typing errors are not just a matter of one intended input token being miskeyed as another one. Spaces between tokens may be deleted (so that two or more intended words appear as one) or inserted (so that one word appears as two or more). Multiple errors, involving both spaces and other characters, may be combined in the same intended or actual token. A reliable spelling corrector must allow for all these possibilities, which must, in addition, be distinguished from the use of correctly-typed words that happen to fall outside the system's lexicon.

However, even in the absence of "noise" of this kind, spaces do not always correspond to lexical item boundaries, at least if lexical items are defined in a way that is most convenient for grammatical purposes. For example, "special" forms such as telephone numbers or e-mail addresses, which are common in many domains, may contain spaces. In CLARE, these are analysed using regular expressions (cf Grosz *et al*, 1987), which may include space characters. When such an expression is realised, an analysis of it, connecting non-adjacent vertices if it contains spaces, is added to the lattice.

The complexities of punctuation are another source of uncertainty: many punctuation symbols have several uses, not all of which necessarily lead to the same way of segmenting the input. For example, periods may indicate either the end of a sentence or an abbreviation, and slashes may be simple word-internal characters (e.g. *X11/NeWS*) or function lexically as disjunctions, as in

> *I'm looking for suggestions for vendors to deal with/avoid.* [1]

Here, the character string "*with/avoid*", although it contains no spaces, represents three lexical items that do not even form a syntactic constituent.

And, particularly in running text, a number of different punctuation symbols must be dealt with. While some of these phenomena can be handled fairly effectively on their own without resorting to a lattice representation, they tend to interact in ways that make apparent the inadequacy of modelling the input as a deterministic sequence of tokens.

An example that shows the need for a lattice representation of possible words is as follows:

```
Does anyone out there in net.land know of/sell/use a Centronics
parallel (redundent I know) output board for VME bus?
```

To prepare this sentence for parsing, at least four questions would need to be answered:

- If `net.land` is not in the lexicon, should it be interpreted as `netland` or `net land`? (Note that the dot cannot be an end-of-sentence marker here).

- Is `of/sell/use` to be interpreted somehow as a single lexical item, or do the slashes have a wider syntactic function? If the latter, how? (Other uses of the slash character common in the bulletin board are `X11/NeWS` and `Sun 4/330GX`; in both cases, the slash has no special status inside the lexical item.)

- Do the bracket characters function as parentheses? There are other uses, as in another sentence from the same bulletin board:

  ```
  PS: Dear wnl, outlaw those insipid ":-)" smiley faces now!
  ```

- What is the correct version of the misspelled word `redundent`? Misspellings sometimes involve spaces being deleted, as in tokens like `soundsthe`, or

[1] These two examples are taken from the Sun-spots electronic bulletin board on Sun systems administration.

`witha`. In the latter case, whether the intended input is "with a" or "with" cannot be determined without knowledge of syntax, which is unlikely to be easily applicable before (candidate) words have been identified.

The specific task considered in this chapter is the process of mapping single sentences from character strings to quasi logical form representations. Two kinds of issue are therefore not discussed here. These are the problem of segmenting a text into sentences and dealing with any markup instructions, which is logically prior to producing character strings; and possible context-dependence of the lexical phenomena discussed, which would need to be dealt with after the creation of QLFs.

Many of the aspects of language under discussion here are especially liable to vary between applications. Since CLARE is intended to be tailorable to different domains and tasks without the intervention of expert linguists, it is important that non-linguist application experts be able to write the appropriate rules and lexical entries. This constraint, too, is reflected in the solutions to be discussed.

11.3 CLARE's Lexical Processing Stages

In the analysis direction, CLARE's lexical processing stages are as follows.

Clustering. A sentence is deterministically divided into a sequence of *clusters* separated by white space. An example of a cluster is "`books,`" (including the comma).

Tokenization. Each cluster is divided into one or more *tokens*; for example, "`books`" and "`,`". A token is anything that the parser treats as a terminal symbol; the term therefore includes words, special forms and punctuation symbols. Because tokenization is non-deterministic, a lattice representation is adopted from this stage onwards.

Segmentation. Each token, or sometimes a sequence of several tokens, is analysed nondeterministically as a sequence of one or more *segments*. For words, these segments are morphemes; for example, "`book`" and "`es`".

Recovery. The failure to analyse a token at stage 3 usually means that another division of the cluster it originated in (in stage 2) was the appropriate one. Such tokens are simply pruned away. However, if there is *no* path through the lattice that avoids unanalysable tokens, then some of those tokens are handed to various *failure recovery* modules. These may create new tokens or new definitions for the existing tokens.

Idiom and Abbreviation Recognition. Some words or phrases may have been defined as idioms or abbreviations that are equivalent to other phrases. If so, segmented versions of the latter are added to the lattice.

Word Parsing. The segments making up each token that is a word undergo word parsing with respect to the system's morphological rules to yield one or more syntactic and semantic analyses. Thus "`[book,es]`" is analysed as a plural noun; if "book" were to be defined as a verb, a third person singular present verb analysis would also result. Non-word tokens undergo analogous processes. Word parsing is not further described here as it is unchanged from earlier versions of the system.

Phrase Parsing. Before sentence parsing proper (for which see the final Nattie report), a set of grammar rules known as *phrase rules* are applied to the word analyses.

In synthesis, analogous process are applied. The generator (see Chapter 8) creates a parse tree, and the reverse of the segmentation, tokenization and clustering phases are then applied. The same rules are used as for analysis. In the following sections, the stages of lexical processing are described in the analysis order.

11.4 Clustering

The clustering phase is the simplest of the lexical processing phases. In analysis, the input characters are separated at white spaces into atoms known as *clusters* that contain no white space. Multiple contiguous white space characters are treated just like a single one.

Recognition of the end of a sentence in general may require the application of syntactic and semantic knowledge, because there are situations in which a cluster-final character such as a period or an exclamation mark does not mark the end of a sentence. However, in the interests of practicability, the rules in CLARE, which should be well suited to interactive input, are as follows.

- A sentence can only end at the end of a line of input. That is, a new sentence cannot start on the same line.
- If the last cluster on a line ends in a sentence-terminating final punctuation symbol it is taken as an end of sentence. That is, a sentence can be input over multiple lines as long as the last cluster on a line is not one that can finish a sentence.

The cluster `##` acts as a signal to abandon the sentence currently being input. It is useful if the user notices an error only after continuing a sentence on a new line.

Declustering, the trivial counterpart of clustering in synthesis, consists merely of outputting the clusters generated by detokenization (see below) with space characters between them.

11.5 Tokenization

In earlier versions of the system, tokenization was hard-wired and deterministic. For example, a lexical item that happened to end in a colon could not be recognized because the colon would be stripped off before lexical access took place. In CLARE, however, tokenization is performed by the same code as word segmentation, using rules of the same format. All rules are treated as optional; thus the cluster "`books,`" will yield one tokenization consisting of "`books`" and a comma, and one consisting just of "`books,`" itself. (The latter will normally, of course, fail segmentation). The symbol "`,`", like, say, "`ed`", is defined as a suffix, but one that is treated by the grammar as a separate token rather than a bound morpheme.

Rules for punctuation characters tend to be very simple because no spelling changes are involved. However, the possessive ending "`'s`" is treated as a separate lexical item, not a bound morpheme, in the CLARE grammar to allow the correct analysis of phrases such as "the man in the corner's wife", and spelling changes can be involved here: the possessive of "cheques", for example, is `cheques'` and not `cheques's`.

As well as prefix and suffix symbols, a new class, that of infixes, has been defined. This is to allow symbols such as hyphens and slashes, which typically appear within clusters, to be treated syntactically. In the example given in section 11.2 the cluster sequence "`know of/sell/use`" would best be analysed as something like "know of or sell or use", with the slashes acting independent lexical items.

The CLARE system is, by default, sensitive to the casing of its input; thus if "Bill" (the name) and "bill" (a noun or verb) are both defined, an occurrence of `bill` will access only the latter. Casing errors are dealt with as a simple case of spelling correction (see below). However, if the token (or tokens) at the beginning of the lattice is capitalized, an uncapitalized version of it is added to ensure correct behaviour for sentences like "Bill six hours to the WHIZ project".

In the detokenization phase of synthesis, a token sequence is turned into a cluster sequence. At each token boundary, the system must decide whether to join the tokens together or to start a new cluster. If token `T1` is followed by token `T2`, a new cluster must be started at `T2` if `T1` can only be a root or a suffix and `T2` can only be a root or a prefix.[2] A new cluster is also started if spelling changes prevent the creation of a cluster including both `T1` and `T2` (though such spelling changes are very rare).

[2]The terms "root", "prefix" and "suffix" apply at the cluster level here; thus an English word would be a "root" even if inflected, a comma would be a typical "suffix", and an opening bracket a typical "prefix".

11.6 Segmentation

A fairly simple affix-stripping approach to word segmentation is adopted in CLARE because inflectional spelling changes in English tend not to be complex enough to warrant more powerful, and potentially less efficient, treatments such as two-level morphology (Koskenniemi, 1983). Derivational relationships, such as that between *destruction* and *destroy*, usually involve semantic peculiarities as well, necessitating the definition of words like *destruction* in the lexicon in their own right.

Segmentation rules are applied recursively to cover cases like "workers" (analysed as `work+er+es`) without a special rule for the combination `er+es`. This is useful not only in token segmentation but also in cluster tokenization; the cluster `of/sell/use` referred to above requires two applications of an infix rule to be correctly tokenized.

11.6.1 Special Forms

As in earlier releases, special forms may be defined in CLARE; see sections 6.2 and C.3.2 in the Nattie final report for general principles. In CLARE, however, special forms may include spaces. For example, a car registration number such as `XYZ 987` could be recognized as a single special form, with both halves also perhaps being treated as (unrelated) special forms in their own right. It is difficult to see how this could easily be done if a deterministic word sequence had to be created before parsing. When a special form is recognized, an edge is added to the lattice of possible segmentations; a special form with a space corresponds to an edge connecting non-adjacent vertices.

The passing of information from surface string to logical form has also changed somewhat. The following special form definition illustrates the use of regular expressions and the way information in the items they match is used to create structures in the logical form. It handles dates in the form of three numbers separated by colons or slashes. As before, the pattern matching is based on regular expressions.

```
special_form('\([0-9]+\)\([/:]\)\([0-9]+\)\2\([0-9]+\)',
             a_term(time(date),
                    V,
                    [and,[day_num,V,reg(1)],
                         [and,[month_num,V,reg(3)],
                              [year_num,V,reg(4)]]]),
             parallel('Monday',monday_DayName)).
```

The sequences `\(` and `\)` in the first line of the entry cause the characters matched to be assigned to a register. Thus for a date of the form `26/9/91`, the registers would be given the contents `26`, `/`, `9` and `91`. The characters `\2` in the first line refer to the contents of register two, which is set when the earlier pattern `[/:]` is matched (with either a slash or a colon); this enforces the constraint that the same separator must be used in both positions (`26/9:91`, for example, is not valid). The first, third and fourth register values are substituted for the `reg(`N`)` structures in the `a_term` structure to create the logical constant

```
a_term(time(date),
       V,
       [and,[day_num,V,26],
            [and,[month_num,V,9],
                 [year_num,V,91]]]).
```

This constant later forms part of any successfully constructed QLFs for the sentence in which the date occurs.

In synthesis, information flows in the reverse direction, from the logical constant to the registers, which are matched with the regular expression to compute the appropriate character sequence (in this case, there would be two possibilities, one for each choice of separator).

The final line of the entry, `parallel('Monday',monday_DayName))`, describes the syntactic behaviour of the items being defined. On encountering it, the system looks (at the word parsing stage, in fact) for a definition of the word `Monday` involving the constant `monday_DayName`, and creates syntactic and semantic entries for the special form by analogy with it. This allows the application developer to specify the behaviours of special forms in sentences by reference to existing words which have earlier been defined using VEX.

Other options, allowing more flexibility, exist for the use of more linguistically knowledgeable developers. Instead of a "`parallel(...)`" structure, an atomic "prototype" may be given; this is, as in earlier releases, expected to have its own definitions. Alternatively, if the required syntactic and semantic behaviour follows a known paradigm, a structure of the form `paradigm(Para,Proto)` may be given; this causes the prototype `Proto` to be defined using the paradigm name `Para`.

11.7 Lexical Failure Recovery

When a token identified by tokenization cannot be segmented as a normal word or a special form, and when there is no alternative path through the lattice that avoids such tokens, CLARE has two basic strategies available. One is to assume that the token is as intended but the lexicon is incomplete, and attempt to derive an entry by accessing an external database such the OUP (MRC) lexicon; eliciting information from the user (via VEX); or simply by guesswork. The mechanisms for this are as described in Alshawi, 1992. The other strategy is that the word is not as intended, i.e. that a spelling or typing error has occurred. All of these possibilities correspond to switches that may be turned on and off by users or application developers.

CLARE's token segmentation phase attempts to find analyses for all the single tokens in the lattice, and for any special forms, which may include spaces and therefore span multiple tokens. Next, a series of *recovery methods*, which may be augmented or reordered by the application developer, are applied. *Global* methods apply to the lattice as a whole, and are intended to modify its contents or create required lexicon entries on a scale larger than the individual token. *Local* methods apply only to single still-unanalysed tokens, and may either supply analyses for them or alter them to other tokens. The default methods, all of which may be switched on or off using system commands, supply facilities for inferring entries through access to an external machine-readable dictionary; for defining sequences of capitalized tokens as proper names; for spelling correction (described in detail in the next section); and for interacting with the user who may suggest a replacement word or phrase or enter the VEX lexical acquisition subsystem (Carter, 1989) to create the required entries.

After a method has been applied, the lattice is, if possible, pruned: edges labelled by unanalysed tokens are provisionally removed, as are other edges and vertices that then do not lie on a complete path. If pruning succeeds (i.e. if at least one problem-free path remains) then token analysis is deemed to have succeeded, and unanalysed tokens (such as *with/avoid*) are forgotten; any remaining global methods are invoked, because they may provide analyses for token *sequences*, but remaining local ones are not. If full pruning does not succeed, any subpath in the lattice containing more unrecognized tokens than an alternative subpath is eliminated. Subpaths containing tokens with with non-alphabetic characters are penalized more heavily; this ensures that if the cluster "`boooks,`" is input, the token sequence "`boooks ,`" (in which "`boooks`" is an unrecognized token and "`,`" is a comma) is preferred to the single token "`boooks,`" (where the comma is part of the putative lexical item). The next method is then applied.[3]

One major advantage of the simplicity of the affix-stripping mechanism is that spelling correction can be interleaved directly with it. Root forms in the lexicon are represented in a discrimination net for efficient access (cf Emirkanian and Bouchard, 1988). When the spelling corrector is called to suggest possible corrections for a word, the number of simple errors (of deletion, insertion, substitution and transposition; e.g. Pollock and Zamora, 1984) to assume is given. Normal segmentation is just the special case of this with the number of errors set to zero. The mechanism nondeterministically removes affixes from each end of the word, postulating errors if appropriate, and then looks up the resulting

[3] In fact, for completeness, CLARE allows the application of two or more methods in tandem and will combines the results without any intermediate pruning. This option would be useful if, in a given application, two sources of knowledge were deemed to be about equally reliable in their predictions.

string in the discrimination net, again considering the possibility of error.[4]

Interleaving correction with segmentation like this promotes efficiency in the following way. As in most other correctors, only up to two simple errors are considered along a given search path. Therefore, either the affix-stripping phase or the lookup phase is fairly quick and produces a fairly small number of results, and so the two do not combine to slow processing down. Another beneficial consequence of the interleaving is that no special treatment is required for the otherwise awkward case where errors overlap morpheme boundaries; thus *desigend* is corrected to *designed* as easily as *deisgned* or *designde* are.

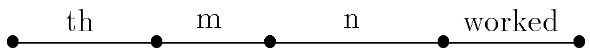
If one or more possible corrections to a token are found, they may either be presented to the user for selection or approval, or, if the number of them does not exceed a preset threshold, all be preserved as alternatives for disambiguation at the later syntactic or semantic stages. The lattice representation allows multiple-word corrections to be preserved along with single-word ones.

It is generally recognized that spelling errors in typed input are of two kinds: competence errors, where the user does not know, or has forgotten, how to spell a word; and performance errors, where the wrong sequence of keys is hit. CLARE's correction mechanism is oriented towards the latter. Other work (e.g. Veronis, 1988, Emirkanian and Bouchard, 1988, van Berkel and De Smedt, 1988) emphasizes the former, often on the grounds that competence errors are both harder for the user to correct and tend to make a worse impression on a human reader. However, Emirkanian and Bouchard identify the many-to-one nature of French spelling-sound correspondence as responsible for the predominance of such errors in that language, which they say does not hold in English; and material typed to CLARE tends to be processed further (for database access, translation, etc) rather than reproduced for potentially embarrassing human consumption. A performance-error approach also has the practical advantage of not depending on extensive linguistic knowledge; and many competence errors can be detected by a performance approach, especially if some straightforward adjustments (e.g. to prefer doubling to other kinds of letter insertion) are made to the algorithm.

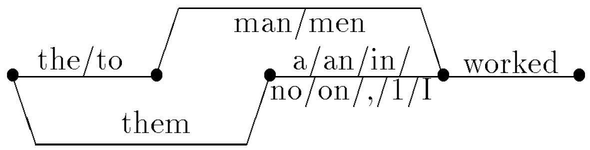
As well as coping quite easily with morpheme boundaries, CLARE's algorithm can also handle the insertion or deletion of word boundary spaces. For the token *witha*, CLARE postulates both *with* and *with a* as corrections, and (depending on the current switch settings) both may go into the lattice. The choice will only finally be made when a QLF is selected on sortal and other grounds after parsing and semantic analysis. For the token pair *nev er*, CLARE postulates the single correction *never*, because this involves assuming only one simple error (the insertion of a space) rather than two or more to "correct" each token individually.

[4]This is the reverse of Veronis' (1988) algorithm, where roots are matched before affixes. However, it seems easier and more efficient to match affixes first, because then the hypothesized root can be looked up without having to allow for any spelling changes; and if both prefixes and suffixes are to be handled, as they are in CLARE, there is no obvious single starting point for searching for the root first.

Multiple overlapping possibilities can also be handled; the input *Th m n worked* causes CLARE to transform the initial lattice



into a corrected lattice containing analyses of the words shown here:



The edges labelled "them" and "man/men" are constructed first by the "global" spelling correction method, which looks for possible corrections across token boundaries. The edge for the token "m" is then removed because, given that it connects only to errorful tokens on both sides, it cannot form part of any potentially optimal path through the lattice. Corrections are, however, sought for "th" and "n" as single tokens when the local spelling correction method is invoked. The corrected lattice then undergoes syntactic and semantic processing, and QLFs for the sequences "the man worked" and "the men worked", but not for any sequence starting with "them" or "to", are produced.

11.8 Idiom and Abbreviation Recognition

However wide the coverage of a grammar, it is inevitable that some constructions will be encountered that go beyond it. This is particularly true when tailoring to applications is considered. There will also be constructions that are within syntactic coverage but whose semantics differ from what the semantic rules would predict. Such constructions are, at least from the standpoint of the system's inevitably incomplete grammar, idiomatic. They present a problem when an application must be developed without help from a linguist competent to extend the grammar in the normal way.

CLARE therefore allows application developers and other users to write lexical entries that define phrases as equivalent to other words or phrases. Flexibility is allowed in affixation but not in the sequence of roots; this is in contrast to a similar facility in the TEAM system (Grosz et al, 1987), where such phrases were defined using regular expressions. Thus in CLARE, a user can specify that occurrences of the phrase *kick the bucket* should, as one analysis possibility, be treated as if the word *die* had been input instead, and that any affixes on *kick* should be transferred to *die*. This allows idiomatic analysis of *John kicked the bucket*, but not of *John kicked the proverbial bucket* or *The bucket was kicked by John*. Such constraints on the applicability of entries can be quite easily understood by non-linguists, who may create the entries through interacting with the VEX lexical acquisition subsystem; this might not be true if regular expressions were used. In our example, the user need only supply the underlined material. He or she does, however, need to choose a target expression that the system can interpret unambiguously in the contexts where the idiom is likely to occur, because if "die" has multiple senses, the system has no way of knowing which one means the same as "kick the bucket".

```
Enter the word or word group to be defined:  kick the bucket
Is this an adjective, noun, verb or idiom in your domain?
Enter one or more of a, n, v or i:  i

I am now defining "kick the bucket" as an idiom.

Enter a phrase with an equivalent meaning:  die
Where should any affix on "kick" be attached?
1 Don't allow any affix.
2 Ignore any affix.
3 Attach it to "die".
Enter a number between 1 and 3:  3

Where should any affix on "bucket" be attached?
1 Don't allow any affix.
2 Ignore any affix.
3 Attach it to "die".
Enter a number between 1 and 3:  1
```

Phrases treated as idioms in this way do not have to be idioms in the usual linguistic sense. Abbreviations and variant spellings can also be treated with this mechanism, defining, say, *i.e.* as *that is* and *color* as *colour*. In the true idiom case, however, provision must be made for literal as well as idiomatic interpretations. It is not correct to replace an occurrence of *kicking the bucket* with the single word *dying*; both must be preserved. The easiest way to achieve the desired effect is to represent both possibilities in parallel in a word lattice.

Users may also define "special form expansions" in which one word sequence, specified using regular expressions, is defined as equivalent to another. For example, the entry

```
special_form_expansion('\$\([0-9]+\)',['\1',dollars]).
```

causes a token such as `$523` to be handled as if it were `523 dollars`, which is more convenient for syntactic reasons.

Idioms and special form expansions are ignored in generation, because in the generation direction they would simply replace one word string by another, perhaps more opaque, one, adding nothing to the coverage of the generator. That is, if it is possible to derive a sentence from a QLF using idiom definitions, it is also possible to do so without using them.

11.9 Phrase Parsing

Just before sentence parsing proper, the *phrase parsing* phase applies a designated set of syntactic rules to the lattice of categories created by word parsing. These rules have exactly the same format as other syntactic rules. However, they are applied in a simple left-corner, bottom-up manner, and may not therefore involve the identification of syntactic "gaps" in the input (although, in contrast to earlier versions of CLARE, they may involve recursion). In the current grammar, phrase rules match such sequences as

how ⟨*adjective*⟩ a
the ⟨*adjective*⟩est
at least ⟨*number*⟩

The distinction between normal and phrasal grammar rules promotes efficiency in two ways. Firstly, the simpler mechanism used to apply phrase rules is inherently more efficient. Secondly, any factorization of the grammar will reduce the search space; phrasal rules obviously cannot apply to the output of the main parsing phase.

When phrase parsing begins, pseudo-edges for the start and end of the sentence are added to the lattice. These allow a more general treatment of words like "however", which, when used sentence-internally as sentential adverbs, appear between a pair of commas, one of which can be viewed as being deleted when the adverb begins or ends the sentence.

The phrase rule mechanism may initially seem to provide similar functionality to the idiom mechanism. However, the two differ in important ways:

- A phrase rule, but not an idiom definition, specifies a pattern that can include syntactic categories as well as words, and may be recursively applied. For this reason, non-linguists are not recommended to write phrase rules. The idiom mechanism was introduced precisely to allow non-linguists to extend the coverage of the system.

- An idiom definition, but not a phrase rule, can map its input onto a sequence rather than a single symbol. Indeed, the members of the sequence may form part of quite different constituents in the ultimate parse.

- Phrase rules, like other syntactic rules, are essential for successful generation of a word string from some QLFs. Idiom definitions are ignored in generation, as stated above.

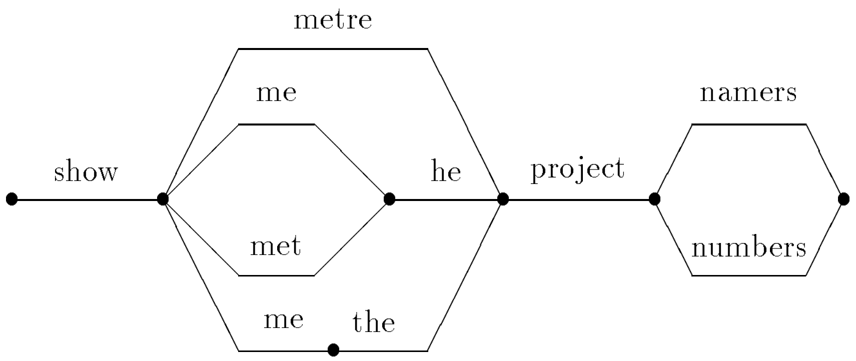
11.10 Examples of Lattice Processing

In CLARE, morphological, syntactic and semantic processing all act on the lattice of word candidates identified by the processes just described, and resulting word-identity ambiguities are resolved as a by-product of the selection (by the user and/or the system) of the appropriate QLF for the sentence, which will correspond to a particular path through the lattice. Thus final decisions are delayed until sufficient knowledge is available to make them.

A simple example of this delaying effect, involving spelling correction, is:

```
Show methe project nambers.
```

where the intended input, typical of those occurring in resource management applications such as the PRM system developed under CLARE, is *Show me the project numbers*. The word lattice, after spelling correction is applied to `methe` and `nambers`, consists of segmentations of the following word candidates:



When this lattice is parsed, four of the eight possible strings (those involving the sequences "met he" and "metre" and finishing with either "namers" or "numbers") are effectively ruled out, because no complete parse of the sentence can be based on those sequences. Semantic processing leads to QLFs being produced first for *Show met project numbers* and second for *Show me the project namers*, but the latter is rejected by users (or, potentially, by the system itself) because the phrase *project namers* is uninterpretable in the PRM domain. The first QLF eventually causes an appropriate list of project numbers to be displayed. All this is done without the user ever being explicitly queried about the required spelling changes.

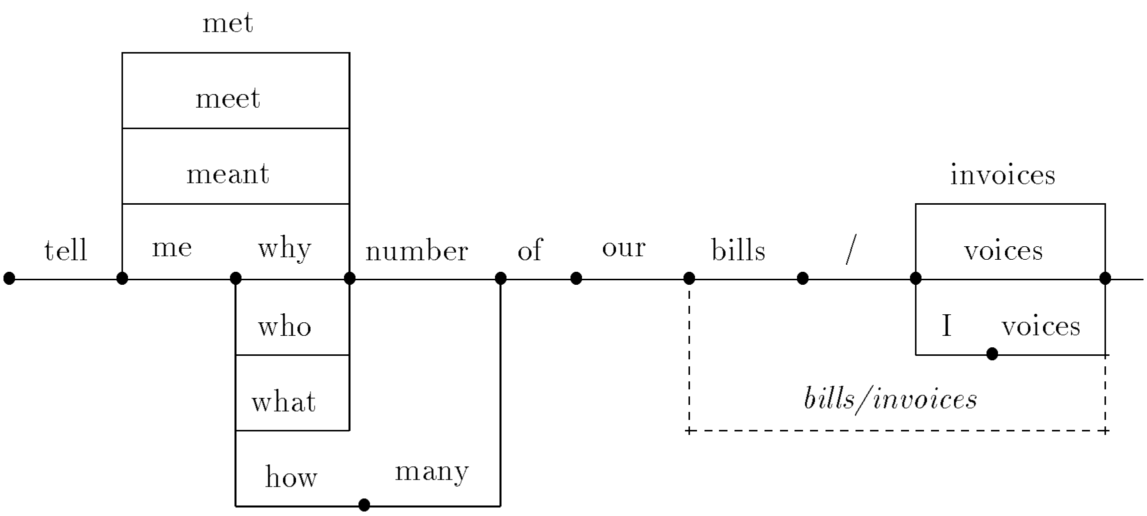
Theoretically, an alternative to the use of a lattice would be to create a list of the possible complete word sequences for the sentence. However, this would involve much repeated work and be very inefficient. Currently in CLARE, the time taken for parsing and semantic analysis depends on the size of the lattice, and not directly on the number of distinct word sequences it represents. For sentences with many word identity ambiguities, this can be a major advantage.

Indeed, it is possible for even fairly short sentences containing instances of the lexical phenomena discussed in this chapter, especially spelling errors, to give rise to lattices containing tens of edges and representing hundreds of possible sequences of morphologically analyses of words.

The above example showed interaction between two lexical processing problems of the same type. However, interacting problems of different types can be handled as easily. Suppose that the phrase *what number* is defined as an idiom meaning *how many*, and the slash character as, potentially, a word separating punctuation symbol defined in the lexicon as a conjunction.[5] Then the input

```
Tell mewht number of our bills/ivoices have not been paid.
```

where `mewht` is a mistyping of `me what` and `ivoices` of `invoices`, causes various spelling corrections to be hypothesized, triggering recognition of the idiom. After spelling correction, the word lattice (with some uninteresting parts omitted) therefore becomes[6]:



The correct word string can again be identified by subsequent processing; only the initial path "tell me how many..." supports any complete syntactic analyses, and the choice between "invoices" and "voices" could be made at QLF level (on domain-specific word frequency grounds, for example).

[5]This example is in fact beyond the coverage by the CLARE/PRM system as delivered, but is included for illustrative reasons.

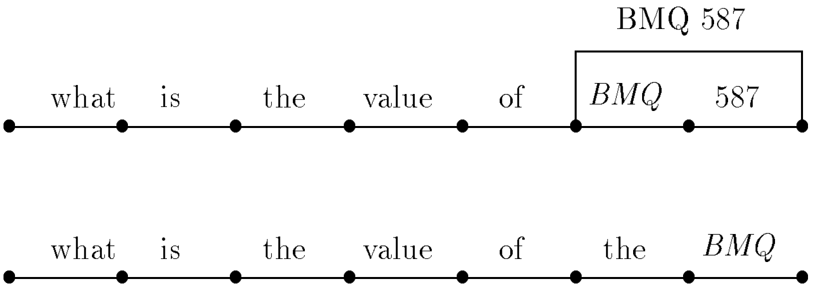
[6]The dashed edge, for `bills/invoices`, would only exist if for some reason that string were defined in the lexicon or otherwise analysable as a single word, and were therefore available as a correction for `bills/ivoices`.

A lattice structure also means that the decision on whether to attempt correction or recovery for unrecognized tokens can be made sensitive to context. If, say, CLARE were to be used to access a database of vehicles, a special form entry might be created to allow recognition of registration numbers of the form `lll nnn` where each `l` is a letter and each `n` a number. Consider the queries

```
What is the value of BMQ 587?
What is the value of the BMQ?
```

Suppose `BMW`, but not `BMQ`, is defined in the lexicon as a count noun denoting a type of car. In processing the first query, `BMQ 587` is recognized as a registration number, and we do not want the system to attempt to correct the spelling of `BMQ`. But in the second, no registration number is recognized, and possibility of correcting `BMQ` to `BMW` should be considered.

When faced with alternative paths through the lattice, such as would arise in the `BMQ 587` example, CLARE therefore prunes any path containing N unanalysable words as long as there is an alternative path with less than N. The lattices for the two examples before correction, with italics denoting unanalysability, are as follows:



In first lattice, therefore, the lower path, where `BMQ` and `587` are separate items, is pruned without any correction being considered, because it contains an unanalysable word and the alternative upper path, with `BMQ 587` as a single special form, does not. In the second lattice, in contrast, there is no alternative path to compete with that along which `BMQ` is unanalysable. The `BMQ` is thus not pruned, and spelling correction and/or user intervention are invoked, depending on the system's switch settings.

11.11 An Evaluation

To assess the usefulness of syntactico-semantic constraints in CLARE's spelling correction, the following experiment, intended to simulate performance (typographic) errors, was carried out. Five hundred sentences, of up to ten words in length, falling within CLARE's current core lexical (1600 root forms) and grammatical coverage were taken at random from the LOB corpus. These sentences were passed, character by character, through a channel which transmitted a character without alteration with probability 0.99, and with probability 0.01 introduced a simple error. The relative probabilities of the four different kinds of error were deduced from Table X of Pollock and Zamora, 1984; where a new character had to be inserted or substituted, it was selected at random from the original sentence set. This process produced a total of 102 sentences that differed from their originals. Some examples are:

```
Andb I agree with it.
Do we or do we not wa nt registration?
She was very goood aboutit.
Les sthan 2,000 now remain.
```

The average length was 6.46 words, and there were 123 corrupted tokens in all, some containing more than one simple error. Because longer sentences were more likely to be changed, the average length of a changed sentence was some 15% more than that of an original one.

The corrupted sentence set was then processed by CLARE with only the spelling correction recovery method in force and with no user intervention. Up to two simple errors were considered per token. No domain-specific or context-dependent knowledge was used.

Of the 123 corrupted tokens, ten were corrupted into other known words, and so no correction was attempted. Parsing failed in nine of these cases; in the tenth, the corrupted word made as much sense as the original out of discourse context. In three further cases, the original token was not suggested as a correction; one was a special form, and for the other two, alternative corrections involved fewer simple errors. The corrections for two other tokens were not used because a corruption into a known word elsewhere in the same sentence caused parsing to fail.

Only one correction (the right one) was suggested for 59 of the remaining 108 tokens. Multiple-token correction, involving the manipulation of space characters, took place in 24 of these cases.

This left 49 tokens for which more than one correction was suggested, requiring syntactic and semantic processing for further disambiguation. The average number of corrections suggested for these 49 was 4.57. However, only an average of 1.69 candidates (including, because of the way the corpus was selected, all the right ones) appeared in QLFs satisfying selectional restrictions; thus only 19% of the wrong candidates found their way into any QLF. If, in the absence of frequency information, we take all candidates as equally likely, then syntactic and semantic processing reduced the average entropy from 1.92 to 0.54, removing 72% of the uncertainty (see Carter, 1987, for a discussion of why entropy is the best measure to use in contexts like this).

Stage	Right cand's	Wrong cand's	Average number
Suggested	49	175	4.57
In any QLF	49	34	1.69
In best-scoring QLF(s)	44	27	1.45

Table 11.1: Correction candidates for the 49 multiple-candidate tokens

When many QLFs are produced for a sentence, CLARE orders them according to a set of scoring functions encoding syntactic and semantic preferences. For the 49 multiple-candidate tokens, removing all but the best-scoring QLF(s) eliminated 7 (21%) of the 34 wrong candidates surviving to the QLF stage; however, it also eliminated 5 (10%) of the right candidates. It is expected that future development of the scoring functions will further improve these figures, which are summarized in Table 11.1.

The times taken to parse lattices containing multiple spelling candidates reflect the characteristics of CLARE's parser, which uses a backtracking, left-corner algorithm and stores well-formed constituents so as to avoid repeating work where possible. In general, when a problem token appears late in the sentence and/or when several candidate corrections are syntactically plausible, the lattice approach is several times faster than processing the alternative strings separately (which tends to be very time-consuming). When the problem token occurs early and has only one plausible correction, the two methods are about the same speed.

For example, in one case, a corrupted token with 13 candidate corrections occurred in sixth position in an eight-word sentence. Parsing the resulting lattice was three times faster than parsing each alternative full string separately. The lattice representation avoided repetition of work on the first six words. However, in another case, where the corrupted token occurred second in an eight-word sentence, and had six candidates, only one of which was syntactically plausible, the lattice representation was no faster, as the incorrect candidates in five of the strings led to the parse being abandoned early.

An analogous experiment was carried out with 500 sentences from the same corpus which CLARE could *not* parse. 131 of the sentences, with average length 7.39 words, suffered the introduction of errors. Of these, only seven (5%) received a parse. Four of the seven received no sortally valid QLFs, leaving only three (2%) "false positives". This low figure is consistent with the results from the originally parseable sentence set; nine out of the ten corruptions into known words in that experiment led to parse failure, and only 19% of wrong suggested candidates led to a sortally valid QLF. If, as those figures suggest, the replacement of one word by another only rarely maps one sentence inside coverage to another, then a corresponding replacement on a sentence *outside* coverage should yield something within coverage even more rarely, and this does appear to be the case.

These experimental results suggest that general syntactic and semantic information is an effective source of constraint for correcting typing errors, and that a conceptually fairly simple staged architecture, where word identity and word boundary ambiguities are only resolved when the relevant knowledge is ready to be applied, can be acceptably efficient. The lattice representation also allows the system to deal cleanly with word boundary uncertainty not caused by noise in the input.

A fairly small vocabulary was used in the experiment. However, these words were originally selected on the basis of frequency of occurrence, so that expanding the lexicon would involve introducing proportionately fewer short words than longer ones. Mistyped short words tend to be the ones with many correction candidates, so the complexity of the problem should grow less fast than might be expected with vocabulary size. Furthermore, more use could be made of statistical information: relative frequency of occurrence could be used as a criterion for pruning relatively unlikely correction candidates, as could more sophisticated statistics in the suggestion algorithm, along the lines of Kernighan *et al* (1990). Phonological knowledge, to allow competence errors to be tackled more directly, would provide another useful source of constraint.

11.12 Responding to Parse Failures

By default, CLARE responds to a failure in syntactic parsing by searching for and displaying the substrings of the input that can be parsed and interpreted as sentences, noun phrases, or other constituents that the application developer may specify. An example is the input

```
Can you tell me what colleges are pretty much within
walking distance of the centre of the city?
```

to which CLARE responds

```
... No parses found.

Minimal sequence(s) of words or phrases have length 4:
can you tell me which colleges are pretty | much within walking |
distance | of the centre of the city
```

Additional information may also be requested. In this case, it is as follows.

```
Maximal entity descriptions:
  ---  you
  ---  me
  ---  which colleges are
```

```
      ---  much within walking
      ---  the centre of the city

    Maximal embedded sentence:
      ---  can you tell me which colleges are pretty

    Shortest sequence(s) of words and complete constituents:

        S:8 NP:3 distance PP:6
```

The parsed substrings are intended to give the user some idea of where the problem parts of the sentence are. In this example, the user should notice that the beginning and end of the sentence were both parsed successfully, but that no significant progress was made on the middle part, "pretty much within walking distance of". This should give the user some guidance in rephrasing his request, e.g.:

```
    Can you tell me which colleges are near the centre of the city?
```

for which parsing and semantic analysis succeed.

Some caution needs to be exercised in interpreting partial parses, however. The embedded sentence in the example, "Can you tell me which colleges are pretty?" is not in fact a constituent of the intended interpretation of either the original input or the subsequent, simplified version; it involves treating "pretty" in its meaning of "attractive" rather than as a modifier of what follows. Nevertheless, this bracketing does give a fairly good idea of what a relevant part of the system's coverage is, indicating, for example, that sentences of the form "Can you ...", and occurrences of "tell" with a dative and a "wh-" sentential complement, are covered.

For the sake of brevity, constituents spanning a strict subpart of the text scanned by constituents of the same type are not reported. Thus although the phrase "Tell me which colleges are" is identified as a sentence, it is not included in the list of sentences found.

Sentences (major category `s` in the grammar) and noun phrases or "entity descriptions" (major category `np`) were selected for display because they seem to correspond to easily identifiable concepts more closely than some other categories do. While naive users can be expected to understand the word "sentence" and to assign some meaning to the phrase "entity description", it is hard to find ways of expressing, say, categories `vp` (verb phrase) or `nbar` in any non-technical way that will not sometimes be misleading. Also, the categories `s` and `np` may appear as complete utterances rather more often than some others, thus making them easier for the user to interpret as sentence fragments. However, the types of constituent reported may be varied if desired.

The initial (failed) parse of a sentence does not necessarily create entries in the well-formed substring table for every possible constituent relevant to partial parsing. This is because during parsing, contextual expectations, derived from the phrases already identified, are applied. To find the complete set of required constituents, it is therefore necessary to parse the sentence starting from every word boundary within it, searching for the desired constituents.

The additional processing seems typically to double the total amount of time needed for parsing, as compared with a successfully parsed sentence of similar length, though there is a fair amount of variation.

If semantic analysis is switched on when a partial parse is produced, one or more partial semantic analyses will also be sought, based on the minimal parsing sequence(s). Only certain constituents (certain types of `np`, `s`, `pp`, `adjp` and `advp`) are used as the starting points for such top-down analysis. The QLF returned is a `form` involving the application of an unspecified predicate to the list of arguments formed by analysing each constituent in the sequence.

11.13 Responding to Sortal Restriction Failures

CLE-3 (Alshawi, 1992) contained a facility for examining logical forms after all analyses of a sentence had failed sortal restrictions, and showing how sortal restriction violations arose. This facility is useful to people familiar with the contents of logical forms and with the sort hierarchy currently in force; however, it is of very limited use to non-experts. Therefore, while it is retained in CLARE, an additional facility is provided that gives English explanations of sortal failures.

An example is:

```
Did the house decide to consider the plan?
```

With only the core lexicon loaded, all semantic analyses fail sortal checks. The CLARE system gives the following output:

```
No sorted semantic analyses found.

Examining ill-sorted analysis 1 (yes/no question)...
Examining ill-sorted analysis 2 (yes/no question)...
The (best) analyses were both ill-sorted for the following reasons
   Analyses 1 and 2:
    A house cannot consider anything.
consider_ThinkAbout ---> animate          => agent;
house_Building      ---> passive_nonliving => nonagent.

   Analyses 1 and 2:
    A house cannot decide to do anything.
```

```
decide_ChooseTo ---> animate           => agent;
house_Building  ---> passive_nonliving => nonagent.
```

The location of the first violation is indicated by the sentence "A house cannot consider anything", and its nature is given on the following two lines. It should be clear that, according to the lexical entries currently loaded, only animate beings are able to consider things, and a house (only understood in the sense of "building") is not animate. Similarly, according to these entries, houses are not able to make decisions.

Sometimes, all interpretations fail sorts, but for different reasons; or there is more than one reason why a given reading fails. The system attempts to summarize in such situations, grouping one reason together under multiple readings when appropriate.

To gather the information necessary to provide these explanations, the sortal restriction checker behaves differently when the relevant system switch is turned on. Each variable in a logical form is associated not just with a sort term (Alshawi, 1992, p174) but also with a list of structures indicating what restrictions have applied to it. For example, when the following QLF for "The man left"

```
[dcl,
 form(l([the,man,left]),verb(past,no,no,no,y),A,
      B^
      [B,
       [leave_Depart,A,
        term(l([the,man]),ref(def,the,sing,l([])),_,
             C^[man_MalePerson,C],_,_)]],
      _)]
```

is checked, the variable `C` is (during checking only) instantiated to

```
(D;
 ss(s(...,animate(human(human_being),male),...),
    [...
     arg(man_MalePerson,1),
     ...,
     arg(leave_Depart,2)|T]))
```

Here, "`ss`" stands for "sourced sort", indicating that both a sort term, `s(...)`, and a list of sources of that sort are represented. The source list here indicates that the variable has had to satisfy the sortal constraints the first (and only) argument of the predicate `man_MalePerson`, and the second (agent) argument of `leave_Depart`. The list finishes with a variable tail, `T`, to allow it to be extended by unification during sort checking.

When two `ss` terms are combined, the source list of each one is extended by the other. If the sort terms themselves unify, nothing further is done. Otherwise, the sort checker does not immediately fail, as it would if explanations were not required. Instead, the sort terms are left unchanged, but the (combined) source list is added to an auxiliary *failure description list* (FDL). If, at the end of checking a logical form, the FDL is empty, the logical form is sortally consistent. Otherwise, it is sortally inconsistent; the sort checker fails, and the FDL is asserted in the Prolog database in anticipation of *all* (quasi) logical forms for the current sentence failing, and explanations being required.

If and when all readings do fail, and explanations are required, the FDL for each reading is retrieved. Every source list in every FDL must contain evidence of at least one *pairwise* sortal violation: that is, there must be two source records that specify conflicting sorts.[7]. The offending pair in each list is extracted, and the system attempts to construct a QLF which, when handed to the generator, will give an appropriate English sentence. For instance, when sortal checking fails for the sentence "The house slept", it is known that there must be a clash between two of the sorts referred to in the list

```
[...,
 arg(sleep_BeNaturallyUnconscious,2),
 ...
 arg(house_Building,1)|_]
```

It is easily verified that the clash is between the two items given here. The system then inspects the lexical entries from which the offending predicates are derived in order to construct a sensible QLF. A constant or a `qterm`, which should be realised as a noun phrase, is created from one of the source terms, if possible. Then the main part of the QLF, into which the `qterm` will slot, is created from the other. The sentence patterns aimed for, with phrases for the clashing terms shown in angle brackets include:

> Nothing can $\langle Verb\rangle$ ... $\langle NounPhrase\rangle$ ....
> $\langle NounPhrase\rangle$ cannot $\langle VerbPhrase\rangle$
> $\langle NounPhrase\rangle$ cannot be $\langle AdjectivePhrase\rangle$
> $\langle NounPhrase\rangle$ cannot be $\langle NounPhrase\rangle$
> Nothing can be $\langle Preposition\rangle$ $\langle NounPhrase\rangle$
> Nothing can be $\langle Adjective + Preposition\rangle$ $\langle NounPhrase\rangle$
> Nothing can $\langle Verb\rangle$ $\langle Preposition\rangle$ anything.

The extra material indicated by dots in the first pattern, and the contents of the verb phrase in the second pattern, depend on the paradigm with which the predicate in question is defined. For an intransitive verb, the verb phrase will be a single word:

[7] Non-pairwise conflicts are not possible in the absence of pairwise ones, because sort terms never contain repeated variables. If $T_1, \ldots, T_n$ are Prolog terms without repeated variables, and if T_i and T_j unify for all i and j, then unification of all of $T_1, \ldots, T_n$ is guaranteed to succeed

```
>> The house slept.
 ...
A house can not sleep.
```

But for a ditransitive verb or one with a sentential complement, it will be more complex:

```
>> John sent Mary an idea.
 ...
Nothing can send anything an idea.
>> The house wants to leave.
 ...
A house cannot want to do anything.
```

When a QLF cannot be constructed – for example, when one of the predicates is for a content word but is not derived from a paradigm definition – or when no QLF leads to a sentence being generated, the system simply states that the two sources clash. This will be comprehensible to a system developer but not to a naive user.

11.14 Summary and Conclusions

This chapter has described the way CLARE reacts to material that is outside lexical, grammatical or sortal coverage.

For the lexical stage, we have demonstrated that no simple deterministic scheme can be adequate for word identification in a system that aims to process real-world texts fully. The solution to the problem adopted in the CLARE system is based on a lattice representation, allowing conceptually clean treatments of individual lexical phenomena to interact smoothly with one another. The treatments selected are also aimed at allowing non-linguist application developers to tailor the system to their needs at a level beyond that of defining individual new words in the lexicon.

The current state of development provides a firm basis for development to proceed on the syntactic and semantic coverage of the phenomena whose lexical dimensions have been dealt with here. This can be quite a major task; for example, it is rather easier to develop algorithms to identify the different possible uses of bracket characters than to write syntactic and semantic rules to account for all the uses of parentheses in English.

More important, perhaps, is the evaluation of the claims made here that non-linguists will be able to extend the system's coverage beyond the lexical level by defining their own special forms and by using VEX to define multiple-word idioms as well as single words. As CLARE is tailored to prototype applications at various sites, the feasibility of such an approach should become clearer.

CLARE's analysis of grammatical and sortal failure is aimed at describing its view of the input so that the user can rephrase his or her input appropriately. Its descriptions, which use bracketing and English statements respectively, are intended to be useful to non-linguists, both users and developers of applications.

Chapter 12

Coverage Extensions and Evaluations

This chapter describes some of the extensions to linguistic coverage made during the CLARE project, and explains how documentation associated with linguistic rules can be accessed to explore this coverage in more detail. It then goes on to describe extensive tests carried out on the delivered CLARE system to evaluate its coverage under different conditions and in comparison to earlier versions of the system. The chapter concludes with a description of some experimental work with word tagging to enhance parsing efficiency.

12.1 Additions to Coverage

This section describes the most important of the changes or additions that have been made to the syntactic and semantic analysis capabilities of the CLE as developed under CLARE. Many smaller changes are not mentioned, (for example, extensions to some new types of lexical item or idiosyncratic construction), even though their impact on corpus coverage figures might actually be rather greater.

12.1.1 Ellipsis

As described in 7.1, considerable attention has been paid to elliptical sentences. The following constructions illustrate some of the types of elliptical construction which CLARE is intended to be able to analyse:

What about Smith?
How many?
Does Jones?
and Bill doesn't want to?
In Cambridge.

3 or 4.

A variety of means are used to analyse such occurrences. Some are regarded as syntactically incomplete sentences. Others are regarded as syntactically complete phrases. Still others are regarded as semi-idiomatic forms requiring special rules. All of them are interpreted with respect to the current model of the context, in particular the immediately preceding linguistic context.

It is worth noting that much of the attention in the linguistic literature has focused on cases of ellipsis where the antecedent is to be found within the same sentence. Examples of this include so called 'gapping' and 'VP-deletion':

John likes fish, and Bill chips.
Joe likes Mary, and Bill does too.

Our treatment of ellipsis will readily extend to examples like this. However, the current version of the system has some of the relevant rules placed in an unused rule group: we found that in the corpora used for evaluation, and in the sample applications we were building, the discourse versions of these types of ellipsis were overwhelmingly more frequent. The slight lack of coverage was more than compensated for by the elimination of the spurious analyses all treatments of ellipsis give rise to.

12.1.2 Comparatives

Compared to CLE, the grammar in CLARE intends to give a wider coverage of comparatives. Two kinds of comparatives are distinguished: 'compositional' comparatives, where the meaning can be arrived at purely on the basis of the sentence itself, and those involving some kind of ellipsis. Examples of the two kinds are as follows:

Mary owns more dogs than Bill owns.
Mare owns more dogs than Bill owns cats.
Mary swims faster than Bill swims.
Mary owns a larger dog than Bill owns.
Mary looks older than Bill looks.

Mary owns more dogs than Bill does.
Mary owns more dogs than Bill.
Mary swims faster than Bill does.
Mary swims faster than Bill.
Mary owns a larger dog than Bill.
Mary looks older than Bill does.
Mary looks older than Bill.

The first group of sentences can all be analysed as containing the appropriate type of gap in the part of the sentence following the comparative marker (`more/less-than`, `as-as`). The usual type of gap threading approach can be used to build up the correct QLF. In the second group of sentences, however, it is more accurate to regard what follows the comparative marker as an elliptical sentence, and to build the appropriate QLF using essentially the same technique as for ellipsis.

12.1.3 NP postmodification

During CLARE the CLE treatment of postmodification of NPs has changed, although no external differences should be apparent to the user. In the original treatment, a phrase like 'every college in Cambridge', or 'every college that was designed by a bishop' was regarded as having the following structure:

[every [college in Cambridge]]
[every [college designed by a bishop]]

This is the simplest structure that is consistent with the syntactic properties found and which allows for the correct semantic representation to be build up compositionally. However, it has several disadvantages: firstly, it will make it difficult to extend this analysis for other types of NP postmodification like

[everyone in the room]
[the college, which was designed by a bishop, ...]

since it is impossible to impose an analogous syntactic structure on these (although they are not semantically identical). The most important disadvantage, though, is that the analysis does not have good properties from the point of view of robust parsing: for example, in the case of parse failure, it would be nice to be able to first of all identify all the 'minimal' phrases that can be found, and then in a later pass assemble these minimal phrases in semantically plausible ways. But on this analysis, in order to combine 'every college' with 'in Cambridge' we would first need to undo the phrase 'every college'.

With a view to making it possible to make the various extensions described, and to further work on robustness, we have changed the analysis of these constructions so that they are of the form:

[[every college][in Cambridge]]

etc. The resulting semantic rules are more complex than before, but we hope that this slight disadvantage will be outweighed by future benefits.

12.1.4 "Big PPs"

An alteration in constituent structure similar in spirit to that described in the previous section has also been made to the handling of prepositional phrases. Previously, an NP or VP postmodified by a sequence of PPs was analyzed in the traditional way, as illustrated by the following two examples:

> [[people [in England]] [during 1991]]
> [He [[arrived [in England]] [during 1991]]]

the NP or VP contained a succession of nested smaller constituents of the same type, one for each modifier. It turns out, however, that there are concrete advantages to be gained from adopting an alternate representation where the sequence of PPs is regarded as a "big PP", a compound prepositional phrase in its own right. Thus the new analysis of the examples above is

> [people [[in England] [during 1991]]]
> [He [arrived [[in England] [during 1991]]]]

The "big PP" rules are supplemented by a rule that allows a temporal NP such as 'today' or 'this Tuesday' also to be treated as a PP. This implies that expressions like

> in London this week
> last year during the election campaign

are also regarded as PPs. These and other "big PPs" can thus appear on their own as elliptical phrases, or be preposed to form expressions like

> In London this week, several people said the same thing.

These increases in coverage turn out to be quite significant in interactive query domains like the SLT translation project (section 13.5). Another point is that introduction of the "big PP' allows simple definition of preference methods (Chapter 10) which reward plausible combinations of PPs, in particular the very common pair *from ... to*.

12.1.5 Adverbial phrases

The coverage of adverbial phrases has been substantially broadened in CLARE. There are two main extensions. Firstly, a number of rules have been added to cover various cases where sentences and verb-phrases function as adverbs, as for example in the following:

John left England [to find a job].
John left [singing a song].
John slept [, I think].

Secondly, the treatent of punctuation with regard to adverbial modification has been properly systematized, by allowing commas to be used to demarcate the boundaries of the phrase; in accordance with normal English usage, commas can appear both before and after in medial position, only after in initial, and only before in final. This is illustrated in the next group of examples:

[Quickly ,] John left.
John [, quickly ,] left.
John left [, quickly].
John [, Mary told me ,] has left.
John has left [, Mary told me].

The implementation of these rules relies on the notional `start_marker` and `end_marker` categories introduced at the beginning and end of the utterance during lexical processing (see section 11.9).

12.1.6 Idioms

It is now possible to define multi-word lexical entries. Thus an entry like

lex([not,many],....)

is possible and will allow the sequence of words 'not many' to be analysed as a single item with the categories given in the entry. Previously such combinations had to be recognized by putting in special syntax rules, one for each such combination. The addition of the new mechanism has allowed us to simplify the grammar by removing these rules.

The same mechanism is used by VEX in defining simple idioms, as documented in the manual (see section 7.2.5 there). A lexical entry of the form

$$\texttt{idiom([}\langle iword_1\rangle\texttt{,...,}\langle iword_i\rangle\texttt{],}\\ \texttt{[}\langle mword_1\rangle\texttt{,...,}\langle mword_j\rangle\texttt{],}\\ \texttt{[}\langle pair_1\rangle\texttt{,...,}\langle pair_k\rangle\texttt{])}$$

defines the phrase $\langle iword_1\rangle,...,\langle iword_i\rangle$ (typically an idiom or an abbreviation) as possibly equivalent to the phrase $\langle mword_1\rangle,...,\langle mword_j\rangle$ (its "meaning").

Some common fixed phrases and idioms have been added, although we are still in the process of discovering from corpora which are the most frequent and useful of such phrases.

12.1.7 Dates, times and numbers

The PRM application, as do many others, demands a comprehensive treatment of dates. We have added the necessary special forms, entries and rules to be able recognize dates in any of these common formats:

27/11/1991.
27 Nov 1991.
27th Nov 1991.
27th of Nov 1991.
27 Nov, 1991.
27th Nov, 1991.
27th of Nov, 1991.
Nov 27, 1991. Nov 27 1991.
Nov 27th, 1991.
Nov 27th 1991.
Nov the 27th, 1991.
Nov the 27th 1991.
27 November 1991.
27th November 1991.
27th of November 1991.
27 November, 1991.
27th November, 1991.
27th of November, 1991.
November 27, 1991.
November 27 1991.
November 27th, 1991.
November 27th 1991.
November the 27th, 1991.
November the 27th 1991.

If the year is omitted, it will be inferred to be the current year. The year can also be abbreviated, for example, to 91.

There is also a fairly comprehensive coverage of the "spelled-out" forms of times and numbers. These have been added primarily for use in the SLT translation project described in section 13.5, but should be of general utility in domains where spoken-language input is used. Examples follow of the types of construction for which coverage exists.

six.
six o'clock.
six a m.
six o'clock a m.
six thirty.

six thirty a m.
six thirty five.
six oh five.
twenty two fifteen.
fourteen hundred hours.
thirty eight.
eight hundred.
eight hundred and ten.
nineteen ninety.
eight three. *(Spelled-out two-digit number)*
eight three five. *(Spelled-out three-digit number)*
eight three five seven. *(Spelled-out four-digit number)*

The main problem with including constructions of this type in the grammar is that single-digit phrases like "two" acquire numerous interpretions, which without extra grammatical constraints can lead to an explosion in the number of analyses produced for complex phrases. For example,

Six dogs.

could potentially mean "a half-dozen dogs", "six o'clock dogs" (like "six o'clock news"), or "dogs with code six" (like "eight ball"). In order to combat this tendency, analyses like the second and third ones above in which a number is used as a pre-nominal modifier are currently blocked by use of special features.

12.2 Grammar Documentation Facilities

CLARE incorporates a facility for documenting the intended behaviour of individual grammatical rules. In addition, documentation fields for each rule provide an index to relevant sentences in a small test corpus which is designed to illustrate and provide a check on the basic coverage of the grammar.

A `doc` field within a grammar rule is used to associate various types of information with the rule. The form of the field is

```
doc(FragmentString,CorpusIDList,CommentString)
```

Each documentation record contains a fragment of the language covered by the rule, a number of corpus sentence identifiers, and a brief comment describing the intended behaviour of the rule.

Rule name identifiers are the same as those used in the relevant grammar file. The CLE command `.dr` (see the software manual) displays the documentation for a particular rule given its identifier. The language fragment is a string consisting of a sentence parseable by the CLE using the core lexicon. The part of this sentence covered by the relevant rule is enclosed in square brackets. A list of corpus sentence identifiers is provided which indicate which sentences in the test corpus are relevant to test or illustrate the behaviour of the rule. The comment string provides a brief description of the intended behaviour and function of the rule. For example, calling `.dr` on the rule `syn s_np_vp_Normal` produces the following output:

```
(describe rule specified by 's_np_vp_Normal')

syn rule s_np_vp_Normal:
  Describes fragment:
    [Wren slept]
  Corpus sentences:
    d1: Wren designed a library.
    d3: The man slept.
    d4: I wonder who you like.
    e1: Wren has.
    e3: Who has.
    q2: What has Wren designed?
    q4: Who has designed a library?
  Comment:
    Covers most types of finite and subjunctive, uninverted clause
```

The bracketed language fragment is shown and the test corpus sentences indexed by the identifiers are listed, and the brief explanation of the rule is shown.

The test corpus consists of a set of sentences parseable by the CLE using the core lexicon. Each sentence has an identifier consisting of one or two (minimally mnemonic) letters followed by a number (a = adjectives, c = coordination, cp = comparatives, d = declaratives / verb phrases, e = ellipsis, m = measure phrases / dates, n = noun phrases, p = prepositional phrases, q = questions, i = imperatives, r = relative clauses). The CLE command for testing a rule on the corpus is `.tr [<ruletype>] <rulename>`. This command invokes the CLE on the language fragment in the documentation field and on the test corpus sentences indexed there.

To date, many of the syntactic rules in the current grammar have been documented and a test corpus constructed illustrating their coverage. The description of the rules has been kept as informal and consistent as possible. Unfamiliar terms should be explained in a compatible fashion in Quirk *et al* (1985).

12.3 Conditions of Coverage Evaluation

In order to evaluate progress during the CLARE project, CLARE-0 (the version of the CLE at the end of the Nattie project), CLARE-1, CLARE-2 and CLARE-3 (the final release) were each run on parts of the LOB (Lancaster-Oslo-Bergen) corpus, a mixture of many different kinds of printed material. This corpus was selected as being the closest available approximation to a general or "unbiased" set of sentences. In addition to the LOB data, CLARE-2 was run on four other text sets; this process is described in section 12.5 below.

Because of the large amount of data involved, a client/server control regime was adopted, with one client process distributing sentences to CLARE server processes on several different machines (using the TCP facilities provided with Quintus Prolog 3) and automatically handling Prolog or UNIX error conditions by restarting servers when necessary.[1]

The sentences selected for evaluation were the members of the text sets that contained only "legible" characters from CLARE's point of view: that is, alphanumerics, hyphens, commas and, in sentence-final position only, full stops, question marks and exclamation marks. Although the mechanism of CLARE is capable of processing sentences containing other characters (see Chapter 11), its lexicon and grammar have not been significantly extended in this direction. The "legible" sets were further restricted by including only sentences of twenty words or fewer, because longer sentences stand little chance of being successfully processed by the system.

12.3.1 Limiting the vocabulary

For some of the experiments, a further subset of sentences was taken: those consisting only of words within the coverage of the CLARE-2 core and PRM lexicons. For the LOB corpus, over 78% of the word occurrences were within coverage. However, this meant that only around 9% of the sentences consisted only of words that were within coverage.

A plot of the distribution of sentence lengths in the LOB corpus is given in figure 12.1. For each length between 1 and 100, the graph shows the number of sentences of that length in the whole LOB corpus, in the "legible" subset, and in the vocabulary-limited subset.

When vocabulary-unlimited text sets were processed, an interface to the MRC Psycholinguistic Database (Coltheart, 1981; Alshawi, 1992, p125) was used for words not in the core or PRM lexicons. It was also used to provide additional definitions for major categories (parts of speech) not appearing in the main lexicons. For example, "burn" is defined in the core lexicon as a verb, but in the MRC database as both a verb and a noun. When this word was used, a noun definition was created on the basis of the MRC data. Proper name inference (Alshawi et al, p118) was also used to define capitalized words as potentially either noun phrases or count nouns.

CLARE-2 and earlier versions interface to the earlier form of Coltheart's

[1]The code implementing this is not an official part of the final CLARE release, but is included with it in files named `tcp*.pl`.

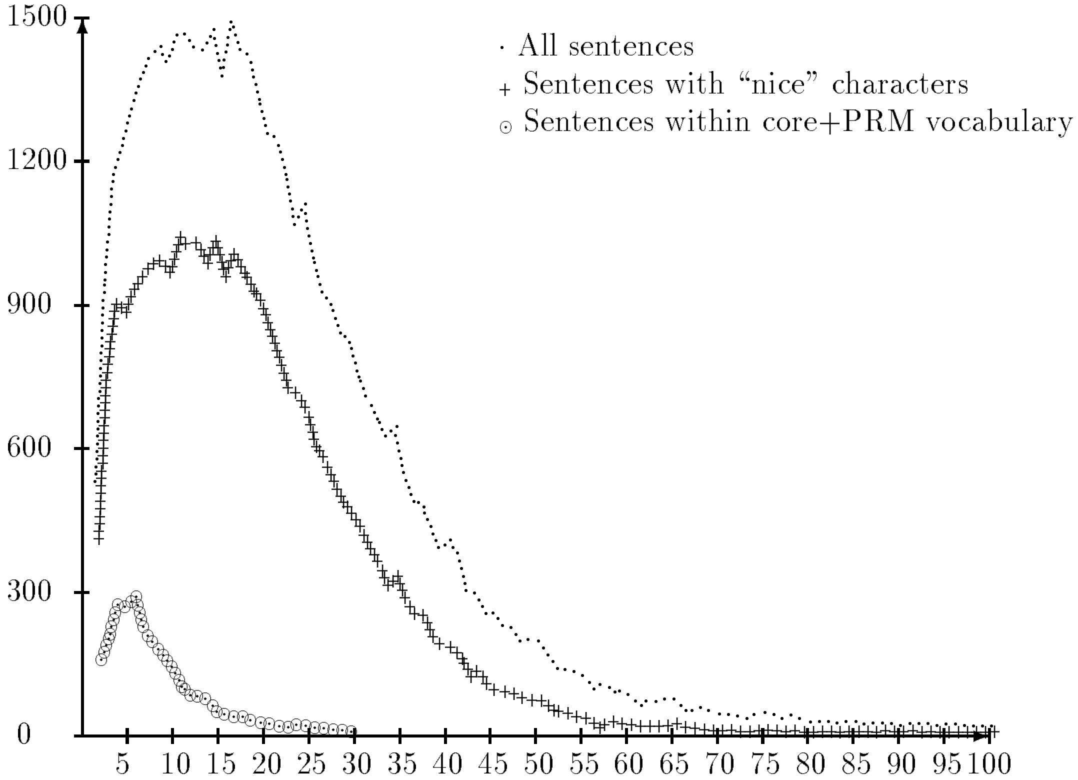


Figure 12.1: Sentence length distribution for LOB corpus

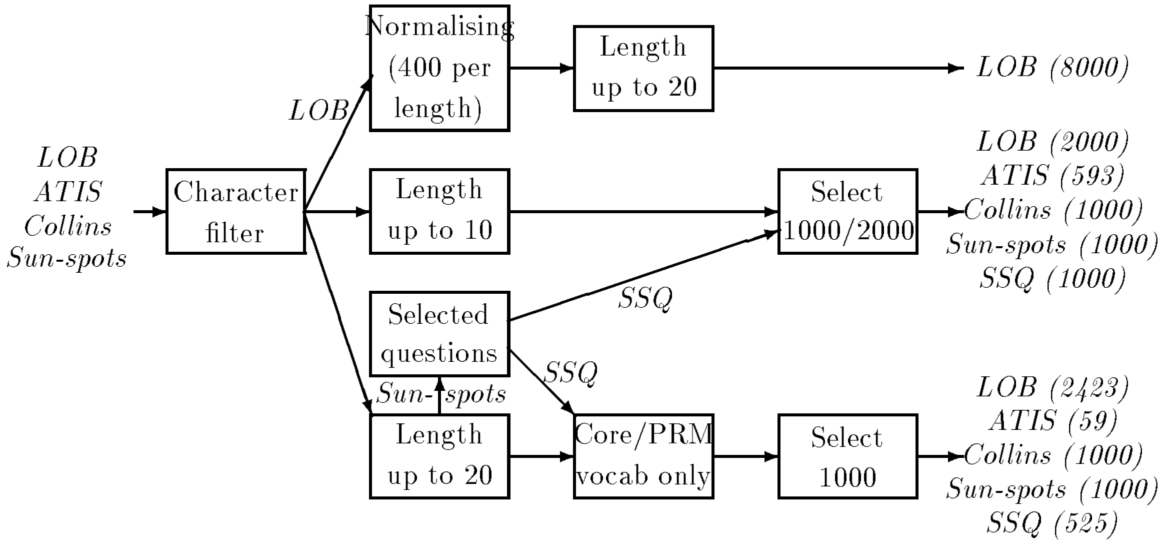


Figure 12.2: Creation of text sets

database. The only useful information in this database is for major categories of words. Thus the fact that a word can be a verb is recorded, but distinctions between different kinds of verb are absent. For this reason, the derived lexical entries are fairly crude, representing only the most common behaviours, and many errors and omissions occur as a result; most other machine-readable lexicons provide significantly richer information. CLARE-3 uses the version of the MRC database released by Oxford University Press under the title of the OUP Psycholinguistic Database (Quinlan, 1992). In this version, intransitive and transitive verbs are distinguished, as are mass and count nouns; this led to a noticeable improvement in the accuracy of QLFs derived using the external lexicon.[2]

The filtering applied to the LOB corpus, and to the other text sets which will be described in more detail later, is summarized in figure 12.2. Each filter is applied to each possible text set except where indicated otherwise. In some cases, a random subset of the data (1000 or 2000 sentences) was selected if the input was larger than this, to reduce the amount of processing needed to manageable dimensions.

[2]At the time of writing, the OUP database is available with a Macintosh interface at £175 from Oxford University Press. The CLARE-3 release is also accompanied by a less developed interface to the rather richer WordNet database (Miller, 1990), which is available by anonymous ftp from `clarity.princeton.edu`.

12.4 Development over Time

In the first set of experiments, versions 0, 1, 2 and 3 of CLARE were run on the vocabulary-limited subset of the LOB corpus and on the vocabulary-unlimited (but still legible) subset. In the latter case, the interface to the MRC Psycholinguistic Database was used to provide word definitions when necessary.

As expected, later versions showed considerable improvements over earlier ones.

It was not realistic to process the sentences in context-dependent ways. This was because

- modelling context for general language input, unconstrained by task and domain, is well beyond the state of the art;
- some of the sentences in the original discourses were ignored because they were not "legible";
- some of the remaining sentences failed to parse, so that their contributions to a context model could not in any case have been evaluated.

Thus only stages up to and including the construction of QLFs by semantic analysis were evaluated. The criteria used were the proportion of sentences getting a QLF, and the proportion of sentences getting a "good" first-choice QLF. The first of these values can be calculated automatically, but the second involves human inspection, and hence was only carried out for a small proportion of sentences.

By a "good" QLF, we mean a QLF that would be the correct one in some context judged to be reasonably plausible. Obviously there is an element of subjectivity in such a judgment. Furthermore, such figures are only a rough guide to the performance of CLARE in a discourse application. The production of a good first-choice QLF by the system in the absence of contextual processing is neither necessary nor sufficient for the successful processing of that sentence in a full discourse-based application. It is not necessary, because contextual processing may reject the first QLF and accept a later one, or cause the preference ordering of QLFs to differ so that another is produced first. And it is not sufficient, because even if the QLF is good in some context, it may not be correct in the context in which the application operates; and even if it is, later processing may introduce errors (in reasoning or reference resolution, for example). Nevertheless, it is probably safe to assume that the greater the proportion of good first-choice QLFs produced by a version of CLARE, the more capable it will be of supporting a full application.

In CLARE-2 and CLARE-3, but not earlier versions, preposition meanings are represented in QLF by `form`s that do not attempt to distinguish their senses. To allow comparison between versions, implausible preposition senses were ignored in assessing the goodness of a QLF derived from CLARE-0 or CLARE-1. However, all other sense distinctions were treated as significant.

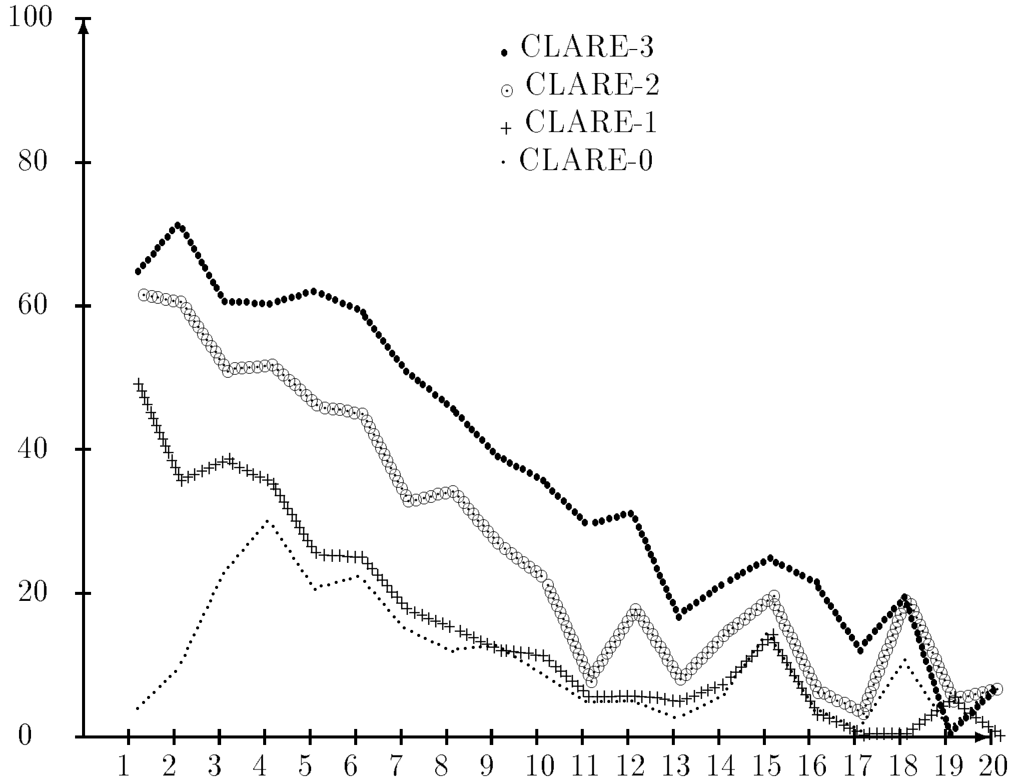


Figure 12.3: QLF production rates. LOB corpus, limited vocabulary, non-cumulative.

12.4.1 Proportions of QLFs produced

Figures 12.3 to 12.6 show the percentages of sentences of given lengths (in words) for which QLFs were produced. Sentence length is shown along the horizontal axis in each figure. Figure 12.3 is for the limited (core plus PRM) vocabulary sentences, with no external lexicon access. It shows that, for CLARE-3, the proportion falls from around 65% for very short sentences to around 40% for sentences of ten words. For longer sentences, success rates were under 40%; it is not possible to be more exact because the numbers of sentences of these lengths were fairly small (see figure 12.1).

The performance of CLARE-3 is consistently better than that of CLARE-2 for sentences of up to ten words, just as CLARE-2 improves on CLARE-1. This reflects the way that coverage extensions were made during the last two years of the project in response to shortcomings identified in processing various (non-LOB) texts. CLARE-1 is in turn superior to CLARE-0 for very short sentences, but only a little better thereafter. This is because the main coverage extensions in the first year of the project were for elliptical phenomena, and very short sentences are often elliptical in nature.

Figure 12.4 presents the same data in a cumulative way: that is, for each

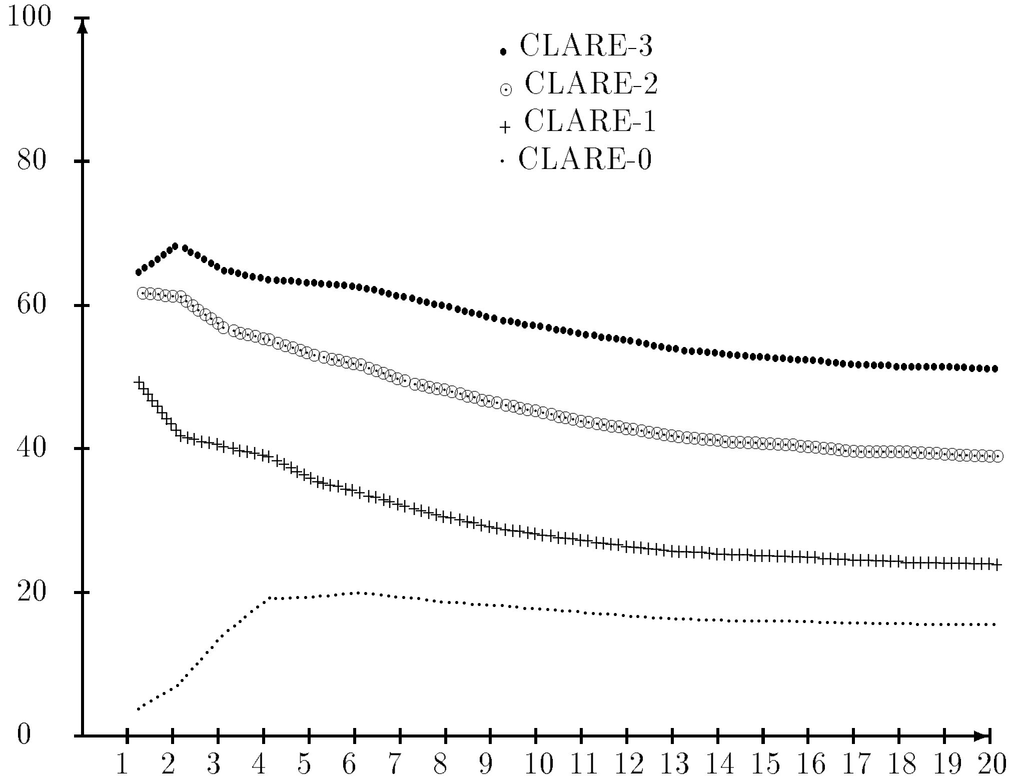


Figure 12.4: QLF production rates. LOB corpus, limited vocabulary, cumulative.

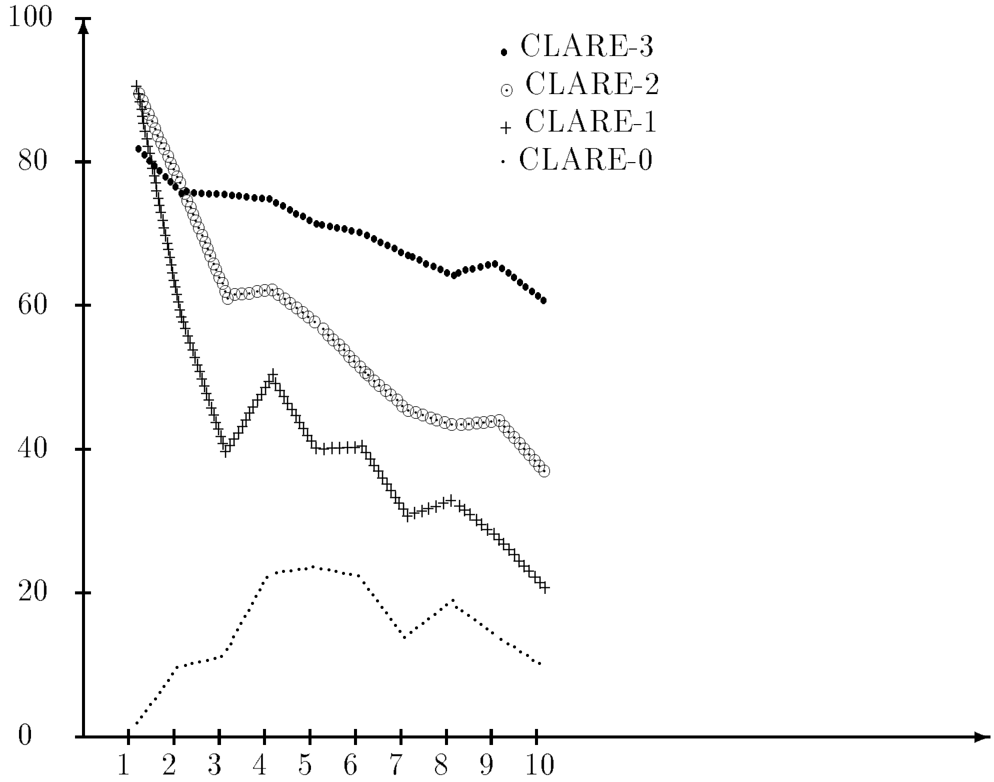


Figure 12.5: QLF production rates. LOB corpus, unlimited vocabulary, non-cumulative.

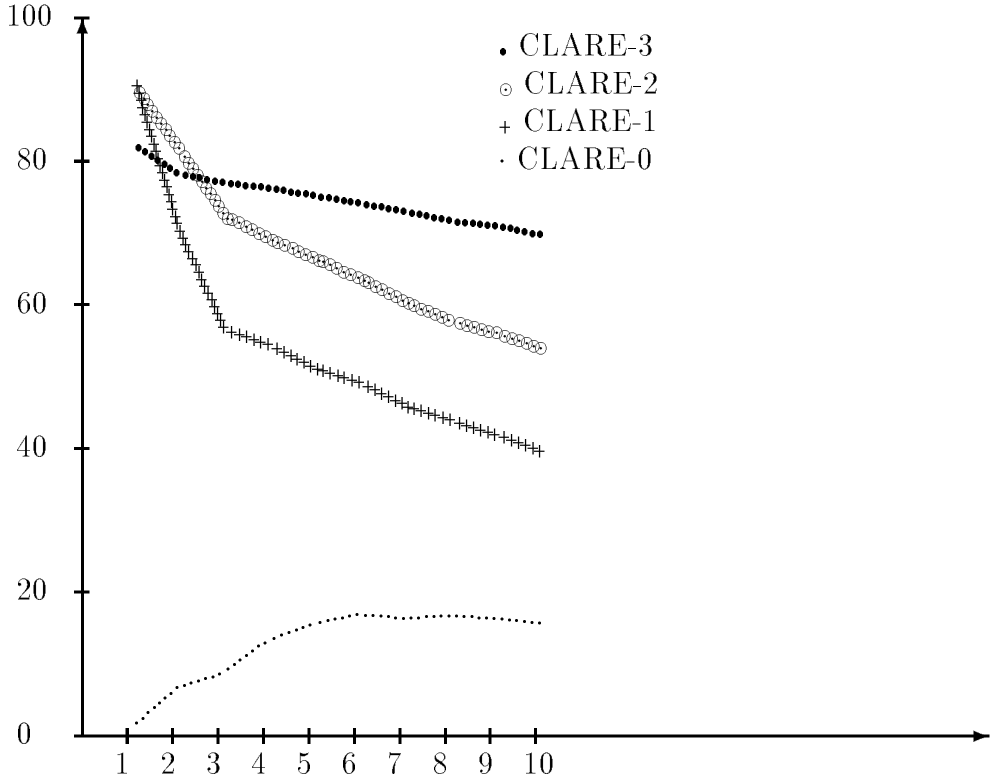


Figure 12.6: QLF production rates. LOB corpus, unlimited vocabulary, cumulative.

length l, the percentage of sentences of length l or less that received a QLF is shown. This again makes the improvement in the second and third years of the project apparent.

The QLF production rates for this restricted vocabulary set should be viewed very much as a pessimistic lower bound on the performance CLARE is likely to exhibit in a domain to which it has been tailored. This is because in any natural language, core vocabulary items tend to be those with the most complex behaviour. Evidence for this is the fact that one third of the Longman Dictionary of Contemporary English consists of definitions for the two thousand words in the "basic vocabulary" identified by the dictionary writers, although those words represent only a small minority of the tens of thousands of words that the dictionary defines. Thus the restricted vocabulary sentences contain many words with a large range of syntactic and semantic behaviours, only the most frequently occurring of which are represented in the core lexicon.

In contrast, sentences pertaining to a specific domain will typically contain many words with comparatively simple behaviours, and many core lexicon words will only occur in a subset of their possible uses. If the domain lexicon is properly set up, it is to be expected that a much lower proportion of sentences will contain word uses not foreseen in the lexicon.

Unfortunately, no sufficiently large corpus of sentences is currently available to assess this likely improvement in the PRM domain. However, the performance of the various CLARE systems on the unlimited vocabulary set, with MRC external lexicon access switched on, is illuminating; see figures 12.5 and 12.6. Under these conditions, the proportion of words with complex behaviours more closely reflects the language as a whole and therefore presumably, likely behaviour in applications. The advantage of domain lexicon tailoring is clearly not available, however. Nevertheless, significant improvements are apparent: for sentences of up to ten words, success rates range from 80% down to 60% for CLARE-3, compared to the range of 65% down to 40% for the restricted vocabulary case. Similar improvements are apparent for earlier versions. The relatively large increase between CLARE-2 and CLARE-3 is due in part to the introduction of the OUP Database. Thorough exploitation of a more sophisticated external lexicon would improve the figures still further.

Performance figures for an early version of an application of CLARE to an air travel inquiry domain, detailed in Section 13.5, lend weight to the claim that much better performance can be expected when the system is properly tailored. In that application, 92% of sentences of up to 10 words produced a QLF, as did 91% of sentences up to 15 words. These figures are much superior to any derived from running CLARE on the general LOB corpus.

The LOB figures turn out to be quite sensitive to the range of characters that are treated as being legible. For example, commas currently have only a partial treatment in the grammar, and hyphens often occur (in LOB) in pairs signifying dashes, which are not treated at all. If commas and hyphens are excluded, leaving

System	% of good QLFs		% of sentences good	
	Limited	Unlimited	Limited	Unlimited
CLARE-0	89	79	15	12
CLARE-1	94	57	26	22
CLARE-2	85	61	38	33
CLARE-3	78	83	44	58
Air Travel	84		77	

Table 12.1: Good QLF percentages for CLARE versions

only alphanumerics, white space and final punctuation characters as legible, then for the unrestricted vocabulary case the proportion of sentences of up to ten words that get a QLF from CLARE-2 rises by 5.9%, from 53.4% to 59.3%. For restricted vocabulary, the rise is more modest, by 2.9% from 44.5% to 47.6%.

Part of these increases are because decreasing the set of legible characters tends to decrease the average sentence length; however, this effect on its own should cause no more than a 1% rise in QLF rates. In fact, the effect of eliminating commas and hyphens from the sentences in the "normalised" LOB sentence set discussed in section 12.6 below is to increase the percentage of QLFs produced by an average of 4% *at each sentence length* between 1 and 10, i.e. after removing the bias caused by the length change.

12.4.2 Proportions of good QLFs

The preceding discussion says nothing about the appropriateness of the QLFs produced. The goodness or otherwise of a QLF has to be assessed by human inspection, a time-consuming process which meant that only a small proportion could be inspected.

One hundred QLFs from each of the eight conditions (CLARE-0, CLARE-1, CLARE-2 and CLARE-3, each for the limited and unlimited vocabulary cases) were inspected. The results are given in table 12.1, along with those from an analogous experiment on SRI's air travel enquiry application of CLARE (see Section 13.5). The two "percent of good QLFs" columns show the numbers of good QLFs found in the inspections; the other two columns show these figures multiplied by the proportions of sentences of ten words or less for which QLFs were produced at all under the conditions in question.

Because of the small sizes of the samples, these figures should be interpreted with caution. The margin of error in a count of, say, 90 is around 6 (at the 5% confidence level) and for a count of 60, it is around 10.

Nevertheless, it is possible to conclude that a large majority of first-choice QLFs are good in the limited-vocabulary cases. The decrease (from 94 to 85 and then to 78) from CLARE-1 to CLARE-3 is due to the coverage of later versions being so much greater, which introduces scope for more "false positive" interpretations. The preference mechanisms have an important role to play here in depreferring readings that are linguistically unlikely (and hence not really plausible in any context). That they are fulfilling this role is shown by the increase in the estimate of the proportion of *sentences* producing good QLFs, which rises from 26% in CLARE-1 to 38% in CLARE-2 and 44% in CLARE-3, a marked improvement.

The figures in the vocabulary-unlimited case also show an improvement for later versions of the system in the proportion of sentences yielding a good QLF. The proportion of good to bad QLFs fell up to CLARE-2 but then rose to their highest level in CLARE-3. The fall is largely because MRC-derived entries tend to be quite general and so, as the syntactic coverage of the system increases, the risk of a false parse resulting from overgeneral entries increases. The rise in CLARE-3 is due mainly to the more discriminating external lexicon and the improved preference metrics.

The intermediate results for the air travel application, given for comparison, suggest that the fairly high proportions of good QLFs (84%) can be maintained even when domain-specific linguistic coverage is increased to over 90% of the inputs, giving an overall 77% of sentences with a good first choice QLF. Although it is difficult to generalize from one domain to another, this figure shows that, for some domains at least, domain-specific additions to the grammar and lexicon can greatly improve performance.

12.4.3 Development over Time: Conclusions

It is interesting that in CLARE-3, the proportion of sentences yielding good QLFs is higher for the vocabulary-unlimited case in spite of the relative paucity of external lexicon information. This may partly be because whereas the core and PRM lexicons distinguish between senses for content words, the MRC-derived entries only do so where there is a syntactic difference. Thus the MRC-derived QLFs tend to be vaguer, and hence have a better chance of being judged good. However, working against this is the fact that the unlimited sentences were on average longer and therefore should have been harder to analyse.

More important, however, is the fact that, as already discussed, the vocabulary-limited sentences are relatively hard ones because they contain a preponderance of frequent, and therefore often idiosyncratic, words. Thus the level of performance in a domain to which CLARE has been tailored should be significantly better than either the vocabulary-limited or the vocabulary-unlimited figures given here, as it has the advantages of both and the disadvantages of neither. Domain constraints will often make it possible to do a thorough job of representing a much greater proportion of the word behaviours that do occur in the domain in question. The importance of tailoring CLARE's vocabulary to the domain is therefore underlined.

12.5 Dependence on text type

In the second experiment, the CLARE-2[3] system was run on five different text sets, in order to test the dependence of the system's performance on text type. The text sets were as follows. The names used to refer to them elsewhere in the chapter are given in bold text.

- The **LOB** corpus, described earlier. Because of the nature of its original medium, printed text, sentences tend to be fairly long, and ungrammaticality is rare.

- A set of spoken enquiries about air travel information (the **ATIS** task), taken from version 1 of the CD-ROM recently release under the Association for Computational Linguistics' Text Encoding Initiative. This material was selected because it represents interactive dialogue, because the domain is quite constrained, and because, unlike the other text sets used, it originated in speech. Sentences are fairly short, and, as with most spoken material, there are many syntactic and semantic errors.

- The example phrases and sentences from the (parsed) version of the **Collins** English dictionary, provided on the ACL CD. This is an artificial set, but covers the lexicon of English quite thoroughly. It contains few or no errors, but fragmentary sentences are common, and the average sentence length is low.

- Sentences from the **Sun-spots** electronic bulletin board, whose purpose is to allow users of Sun Microsystems products to share problems and solutions. Although this medium for this sentence set is typed, the material is informal and "semi-interactive" in flavour; there are, for example, many instances of participants quoting someone's question and then answering it or commenting on it as if the dialogue were taking place in real time. Sun-spots material contains many errors and sentence fragments and much specialized vocabulary.

- A set of Sun-spots sentences identified as questions (henceforth **SSQ**) by virtue of their terminating with a question mark. Questions are, perhaps, of particular interest because query answering is an important possible use for CLARE. In addition, the performance of the CLARE system on this

[3]This experiment was not repeated for the CLARE-3 system. However, informal observations of performance on some of the test sets involved suggest the CLARE-3 percentages will typically be around 10% higher. The conclusions are likely to be unaffected.

Text set	Sentences	% legible	Avge len all	Avge len legible
LOB	45382	40.1	21.0	10.9
ATIS	1015	80.3	9.9	9.0
Collins	18813	92.4	4.5	4.4
Sun-spots	24356	39.0	15.4	9.8
SSQ	1800			9.0

Table 12.2: Sentence characteristics of text sets

Text set	Mean Length	Median Length	% QLFs	% QLFs good	% sents good
ATIS	4.22	4	27.6	100	28
Collins	4.59	4	42.6	87	37
LOB	5.12	5	44.5	85	38
Sun-spots	4.54	4	47.4	83	40
SSQ	5.54	5	51.6	90	46

Table 12.3: CLARE-2 performance for vocabulary-limited case

sentence set has been monitored throughout the project, and some coverage extensions were made as a result.

Table 12.2 shows the number of sentences in each text set, the percentage of them that were "legible", and the average sentence length of all the sentences and of just the legible ones. Some figures for the SSQ set are omitted because this set was derived from the legible part of the Sun-spots set.

The results are given in tables 12.3 and 12.4, for sentences of up to ten words. The mean and median lengths of sentences are given because performance tends, other things being equal, to be better for shorter sentences. The other three columns give the percentage of sentences that produced QLFs, the estimated percentage of first-choice QLFs that were good, and, as the product of these, the estimated percentage of sentences producing a good first-choice QLF.

Because of the large number of cases, the percentage of good QLFs was estimated from a sample of only 30 in all but one case. With a sample of 30, the results can be considered accurate to within about 5%. However, only 12 QLFs were available in the ATIS vocabulary-limited case, so the result of 100% good is rather unreliable.

With these caveats in mind, the conclusions must be fairly tentative. However, excluding the ATIS set from consideration, it does not appear that there is any very strong dependence of performance on the text types. The SSQ set gives a

Text set	Mean Length	Median Length	% QLFs	% QLFs good	% sents good
ATIS	6.53	7	68.5	63	43
Collins	4.21	4	74.4	53	40
LOB	5.95	6	53.4	61	33
Sun-spots	5.88	6	58.6	57	33
SSQ	6.29	6	59.7	77	46

Table 12.4: CLARE-2 performance for vocabulary-unlimited case

rather better score than the Sun-spots set; this is probably due to the addition to the CLARE-2 grammar of constructions such as "anyone else" and "out there", which are very common in SSQ but less so in the main Sun-spots list or in the other text sets. The difference points to the significant improvements that can be gained from making grammar development sensitive to the frequency of occurrence of the phenomena considered.

In the Collins text set, the fact that the sentences tended to be error-free and fairly short was partly offset for the vocabulary-limited case by the fact that examples were provided of word uses irrespective of their frequency. Thus relatively rare uses, which were less likely to be defined in the core lexicon, occurred more often than they would in other kinds of text set.

In the ATIS set, the small percentage of QLFs for the vocabulary-limited case is due to a high frequency of domain-specific phrases such as "class M" which happen to fall within lexical but not syntactic coverage. A few extensions to the grammar would improve matters significantly.

12.6 System components and sentence length

In the final experiment, CLARE-3 was tested against a subset of the LOB corpus to determine, for sentences of different lengths, the success rate for the various system components up to and including semantic analysis. Because of the numbers of sentences needed to ensure reliable results, the vocabulary-unlimited condition, with the MRC external interface (OUP version), was used.

The corpus consisted of 400 sentences, selected at random, of each of the lengths 1 to 20 inclusive (this is the process referred to as "normalising" in figure 12.2). Because of improvements in the external lexicon interface and in the TCP-based corpus processing code, very few sentences timed out or failed at the lexical level. Figure 12.7 therefore shows the percentages of sentences of each length for which a full parse and a full semantic analysis, respectively, were found. Values are accurate to within about 2.5% at the usual 5% confidence level.

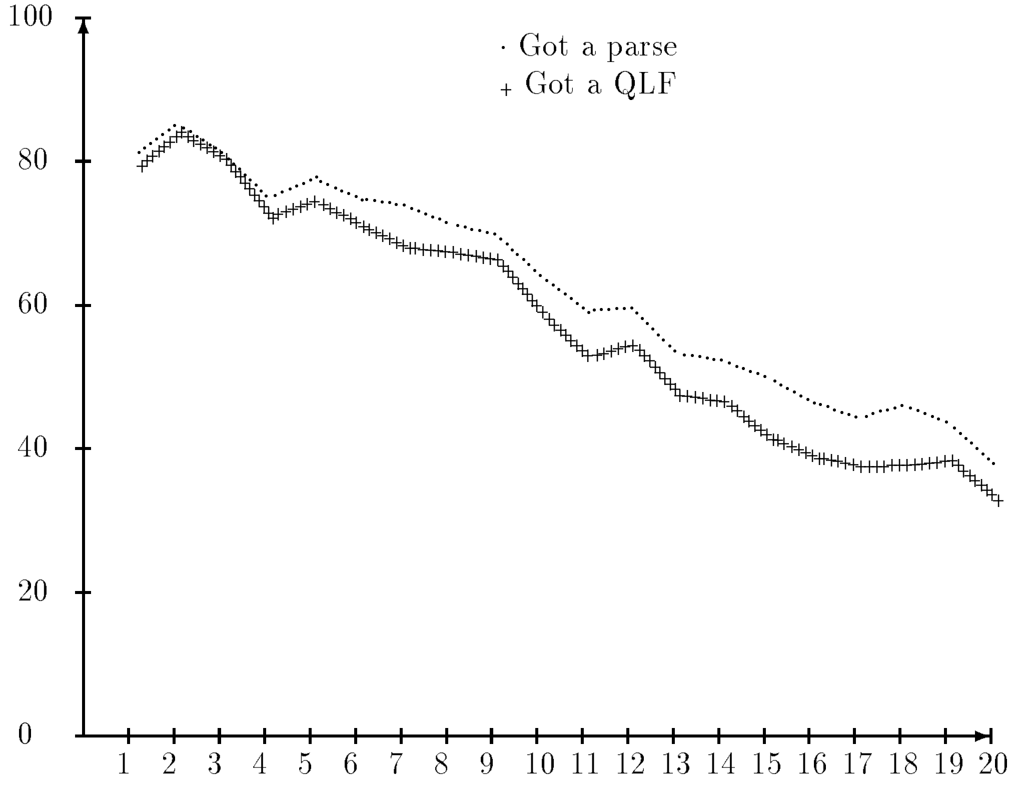


Figure 12.7: Sentence fates according to length

It is clear from the figure that, unsurprisingly, parsing becomes increasingly likely to fail as sentence length increases. Semantic analysis only rarely fails when parsing succeeds; most such failures are on sortal grounds.

Because of the coverage work done during the third year of the project, the success rate falls off more slowly with sentence length than it did a year ago. More than half of sentences at all lengths up to twelve words receive an analysis, and even at twenty words, nearly a third of sentences do. Again, it should be emphasised that tailoring to a domain sublanguage would significantly increase these figures.

12.7 Tagging for parsing efficiency: a preliminary experiment

This section is based on a talk given by Stephen Pulman to a meeting of the DTI Speech and Language Technology Club on Sublanguages, in Edinburgh, January 1992. We are grateful to the participants there for their comments, and to David Elworthy for providing both software and help.

12.7.1 Introduction

One of the crucial factors affecting parsing and analysis speeds for large rule-based systems is the degree of lexical ambiguity found in the input sentence. Verbs, in particular, often have many syntactically distinct entries even for similar senses, because of their differing subcategorisation requirements. But other ambiguities like whether something is a verb or a noun can also have a significant effect on efficiency, because they lead to locally valid parsing hypotheses that may not be ruled out for some time. The combinatoric effect of these local ambiguities might be quite large, and in a large coverage descriptive grammar and lexicon the degree of lexical ambiguity is usually very large.

Ideally, one would like only to take into account those lexical entries for words which are capable of playing a part in the overall parse, not even looking at those which cannot be integrated into a larger structure. One might want also want to discriminate against those lexical entries which *can* play a part in an overall parse, but which are relatively unlikely to do so. In the first case, we are interested in producing all parses, but not wasting time on local red herrings. In the second case we might be trying to tune a general purpose natural language system to some sublanguage, taking advantage of regularities characteristic of that sublanguage, so that valid but unlikely analyses will not be produced.

For both these purposes, the most obvious way to achieve the goals in question is to introduce a statistical element in analysis. In the cases we shall be talking about, this will mean statistical information about the frequency with which a word appears with a particular syntactic category, and about bi-gram probabilities between pairs of categories. However, if the necessary data was available, it would be sensible to do something analogous for semantic information too. Ignoring or down-grading senses of words that are irrelevant or unlikely with respect to a given domain, even where there is no syntactic distinction, is a cheap and cheerful way of accomplishing a task that could otherwise require sophisticated contextual reasoning.

The best way to introduce statistical information into the analysis process is to associate statistical information with rules and lexical entries and make the parsing algorithm sensitive to that information in the desired way. Ideally, this would be done in a comprehensive and systematic manner, allowing for information from different sources to be combined. In the long run, this will be the best approach. In the short term, a very simple way to introduce a statistical element into parsing is to use 'tagging' as a way of disambiguating lexical categories. Tagging is the assignment of lexical categories to a stream of words based on statistical cooccurrence data of the type that can be modelled using Hidden Markov Models or the simple subcases of these as described e.g. in DeRose (1988) or De Marcken (1990).

A tagger of the type described in these references will assign to each word in a sentence a part of speech in such a way as to maximise the overall probability of that sequence of assignments in the sentence. We can then use this information so that the parser can ignore all the lexical entries for a word that are inconsistent with this assignment of tags. (See De Marcken (1990) for a description of this type of approach in aiding deterministic parsing of a large corpus). In the ideal case, we would discover that this information still allowed the parser to build the most likely overall analysis for the sentence.

This method will not guarantee overall correctness or completeness, of course. There are several sources of error possible. Firstly, accuracy rates for taggers are usually in the region of 95% word accuracy. This sounds high, but means that every 20 word sentence is likely to contain at least one error. Secondly, some sentences may be genuinely intended as ambiguous: many advertising slogans, for example, rely on this for their effect. A deterministic assignment of tags will mean that such examples will only get one analysis, if the ambiguity in question involves different choices of lexical category. (This is unlikely to happen very frequently in most applications). Thirdly, and most simply, it may be the case that syntactic probability does not equate with semantic plausibility. The most syntactically likely analysis may not always correspond to the most semantically plausible.

It is clear that any kind of pruning strategy will affect completeness. The question is one of whether the trade-off between completeness and efficiency is an acceptable one. The rest of this section describes two small experiments in which we attempted to investigate what this combination of tagging and rule-based analysis would give rise to: in particular, to measure exactly what the above trade-off was. This experiment used a tagger and the CLARE system.

12.7.2 Tagging

Since no existing tagger was available to us, we implemented a simple bigram-based system of the type described in DeRose (1988). Although his presentation does not make the connection, such a system can be seen as the bigram case of a Hidden Markov Model, where the bigram probabilities represent probabilities of transitions between states in a set of two-state models, his 'relative tag probability' measure corresponds to output probabilities of symbols on transitions, and the 'optimal path' computation described is an application of the Viterbi algorithm.

Transition and output probabilities were obtained by processing the tagged version of the LOB corpus. The resulting tagging system achieved accuracy rates of about 94% when tested on closed data, and about 88% when tested on open data. This could be improved on by moving to trigrams, and by doing some preprocessing to identify fixed phrases, but the necessary resources (of time) were not available to me.

The LOB tag set is rather large, and does not coincide with the set of lexical categories used in the CLARE natural language processing system. Furthermore, even where the relevant set of tags is identical, the tagged LOB corpus does not always agree with the CLE grammar over the allocation of specific words to lexical categories.

In order to use the tagging for the purpose intended, the output of the tagger was mapped to a set of CLE categories. This process is relational rather than functional: for example, the CLE grammar does not distinguish syntactically between quantificational and non-quantificational determiners, and so 'any' and 'the' are members of the same CLE category. LOB distinguishes these two. Conversely, a word like 'both' in LOB has a special single category, whereas in the CLE grammar 'both' can be either a conjunction or a determiner.

Ideally, the experiments should have used a corpus of sentences successfully parsed by the CLE to derive the statistical information. Although we hope soon to be in a position to carry this out, using an existing tagged corpus was at the time the only way of getting sufficiently large numbers of examples for the statistics to be reliable. However, the mismatch between category sets means, among other things, that the results of any experiments can only be regarded as suggestive. On no account should they be taken as conclusive.

12.7.3 Experiment One

The main purpose of this experiment was to use the output of the tagger to prune the lexical entries seen by the parser. Ideally, one would hope that processing speeds would be faster, but that accuracy would not be too badly compromised.

The first issue to be solved is: how do we measure accuracy? In CLARE, the output of syntactic and semantic analysis is a set of QLFs which are ranked according to various measures of plausibility, both *a priori* and empirically derived. The *a priori* estimates are based on properties of the structure of QLFs, including sortal restrictions, enabling the system to attach preferences to, for example, low attachments, properly balanced conjunctions, and so on. We have found that in practice these rankings are surprisingly effective in ensuring that the most likely interpretation of a sentence is the one that is most highly rated.

As a first approximation, therefore, we selected a set of sentences from the LOB corpus for which the CLE had provided, as its first choice, a QLF that we regarded as a plausible one in the absence of any kind of biasing context. The sentences were up to 12 words long: in general, the CLE produces QLFs for about 67% of sentences up to 10 words long in LOB, with coverage gradually declining after that. There were 124 such sentences (representing about 1 day's manual QLF inspection effort).

Next we arranged for each of these sentences to be tagged, as described above, the result mapped to a set of CLE tags, and for the CLE then to be run on the same sentences, using only the non-null intersection of the lexical entries for each word from these two sources. In the case where there was no intersection between the two sets of tags, the CLE tags were left. (The LOB and CLE tag sets are not subsets of each other). The motivation for this method of pruning was in order to try to minimise the bad effects of the non-identity of the tag sets described earlier. Then for each sentence the QLF produced by the system using the pruned tag set was compared with that produced earlier. Relative timings and various other statistics were also collected.

Effects of the sort that we had been hoping to find can be illustrated by the following sentence, where the lexical entries produced by the CLE with the 100,000 word lexicon are shown for each word in figure 12.8.

The verb 'read' has 5 distinct subcategorisations in the CLE. Of these, three are capable of appearing in the passive form. On morphological grounds, therefore, we have a total of 23 distinct possibilities for the morpheme 'read'. (These entries contain semantic information: if this were factored out some collapsing of entries might be possible. But the general point will still remain).

Entries signalled with `>>` are those that were pruned out because they did not overlap with the tags produced for that sentence. Singular nouns ('can') do not usually follow plural determiners. It is statistically very unlikely that a simple past, a non-third-person-singular present, or an infinitive form of a verb will be found immediately following 'be'. The LOB tag set does not distinguish passive from perfective forms of verbs (they are all 'VBN') and so both the perfective and the passive CLE forms are retained.

This illustrates one kind of lack of constraint caused by the mismatch in tag sets: if the tagger had been trained on CLE categories it would have eliminated the perfective ('vform=en') alternative as this only occurs (or should occur) fol-

Figure 12.8: Pruning by tagging

```
these: det(...)
can: v(...)
>> can: n(...)
be: v(...) (3 times)
read: [read,ed] v_v_ed_Perf v(...) (5 times)
>> read: [read,ed] v_v_ed_Past v(...) (5 times)
read: [read,ed] v_v_ed_Passive v(...) (3 times)
>> read: [read] v_v_Not3ps v(...) (5 times)
>> read: [read] v_v_Inf v(...) (5 times)
in: p(...)
their: possdet(...)
proper: adj(...)
place: n(...)
```

Figure 12.9: Experiment 1

```
ACCURACY
Unaffected  Pruned  No Parse   Bad QLF  %accuracy
   35          89      12          4         87

SPEED    (successful parses only)
Averages         #Entries  Parsing  #Parses Sem #QLFS
Without Pruning   17.7      15.3     8.8     5.9  3.6
With Pruning      13.7      9.9      7.6     4.8  2.8
% Difference      22        35       13      18   22
```

lowing a form of ‘have’: correspondingly, following ‘be’ a passive (‘vform=passive’) is much more likely.

Parsing and semantically analysing the original sentence took 50.4 and 1.1 seconds respectively, on a Sun Sparcstation 1 (a machine two generations behind current hardware). Parsing and analysing the pruned version of the sentence took 16.4 and 0.5 secs. In both cases only one QLF was obtained - the correct one.

Unfortunately, this splendid result was not uniformly obtained. The averages over all 124 sentences are summarised in figure 12.9.

As can be seen, for almost 30% of the sentences, tagging produces the same set of entries as the CLE does. In cases where some pruning takes place, two things can go wrong. Either no parse can be found for the sentence at all, since an entry crucial to a successful parse has been pruned out, or a parse and a QLF are found, but the ‘wrong’ one: i.e. not the one ranked top by the system on the previous run. For the sentences taken overall this gives an error rate of almost 13%.

Against this, in those cases where analysis is successful, we get an improvement in parsing speed of 35%, and a respectable improvement in speed of semantic analysis. While useful, my own reaction to this finding was that the tradeoff between speed and accuracy was that it was not profitable enough. Losing over 12% accuracy for only 35% speed up does not seem like a very good trade in most applications, especially when one remembers that we have in any case excluded those cases where the parser fails even where all the lexical categories are present.

One optimistic note is that the number of bad QLFs that were obtained was relatively small. The biggest contribution to overall accuracy are the large number of cases for which no parse at all can be obtained. Inspection of a few of these cases revealed that in some of these cases the tagger had made a decision based on local preferences, that was not always optimal when the sentence was taken as a whole. N-gram tagging techniques are of course always 'shortsighted': there will always be cases where a local cooccurrence possibility is overwhelmingly more likely than the next best alternative on local grounds, but actually less plausible when a bigger context is taken into account.

12.7.4 Experiment Two

One way of overcoming the deficiency noted above would be to move to a trigram-based system. However, even a 3 word window will not always be sufficient to make the correct decision. A simpler alternative would be to be less ruthless in the elimination of tags, on the assumption that by relaxing the requirement on local plausibility, enough (more globally plausible) candidates would be retained so that the correct parse can still be obtained.

In the first experiment, assignment of tags to a word by the LOB based tagger was deterministic: one word, one tag. In the mapping to CLE categories, this determinism was not always preserved, but nevertheless no 'close seconds' were transmitted from the tagger.

De Marcken (1990) describes how to use thresholding to vary the number of hypotheses delivered by a tagger. The system assigns a tag to a word in the usual way, and then adds to the chosen tag all those tags that come within some threshold of the winner's score. The thresholding is achieved by multiplying the score of potential alternatives by some factor, and doing a comparison with the score of the actual winner.

If the factor is 1, the result is as for the deterministic case, since this will never increase an alternative's score. By setting the factor to be very large, no elimination of tags at all will be achieved, because every alternative tag will then exceed the threshold. Between these two extremes it ought to be possible to retain only those tags for a word that are reasonably likely, while still eliminating those that have only a very small chance of contributing to the final parse.

Figure 12.10: Tagging at different factors

```
BEFORE:
 and: conj(...)
 then: conj(...)
 then: advp(...)
 all: det(...)
 failed: [fail,ed] v_v_ed_Perf v(...) (3 times)
 failed: [fail,ed] v_v_ed_Past v(...) (3 times)
 failed: [fail,ed] v_v_ed_Passive v(...)

AFTER (Factor= 1):
 and: conj(...)
 then: advp(...)
 all: det(...)
 failed: [fail,ed] v_v_ed_Perf v(...) (3 times)
 failed: [fail,ed] v_v_ed_Passive v(...)

AFTER (Factor=300):
 and: conj(...)
 then: advp(...)
 all: det(...)
 failed: [fail,ed] v_v_ed_Perf v(...) (3 times)
 failed: [fail,ed] v_v_ed_Past v(...) (3 times)
 failed: [fail,ed] v_v_ed_Passive v(...)
```

Thus for the second experiment we modified the tagger so as to be able to vary the factor governing how many tags for each word would be retained. Again, the hope was to determine an acceptable tradeoff between accuracy and efficiency. Ideally, one would hope to find that tagging with one particular factor, and then using that for pruning, would retain all the lexical entries that were necessary for finding the correct parse, while still filtering out enough to give a useful gain in parsing speed.

As an illustration of the effects hoped for, consider the example in figure 12.10. With no pruning we have 2 entries for 'then', and 7 entries for 'failed'. Tagging deterministically, with a factor of 1, eliminates the unlikely entry for 'then' (this is not the 'then' of 'if-then') but also, for some reason presumably to do with the character of the LOB corpus, eliminates the simple past entries for 'failed', preferring the entry appropriate in phrases like 'all failed politicians'. (As before, this entry yields two CLE possibilities). Unfortunately, there is no parse for this sentence with these entries.

Tagging with a factor of 300 retains all those entries which are such that their

Figure 12.11: Experiment 2

ACCURACY

Factor	Unaffected	Pruned	No Parse	Bad QLF	%accuracy
1	35	89	12	4	87
300	41	83	9	3	90

SPEED

Factor=1 (successful parses only)

Averages	#Entries	Parsing	#Parses	Sem	#QLFS
Without Pruning	17.7	15.3	8.8	5.9	3.6
With Pruning	13.7	9.9	7.6	4.8	2.8
Difference	22	35	13	18	22

Factor=300 (successful parses only)

Averages	#Entries	Parsing	#Parses	Sem	#QLFS
Without Pruning	17.5	14.5	8.3	5.7	3.5
With Pruning	13.9	10.2	7.2	4.7	2.8
Difference	20	29	13	17	20

score, if multiplied by 300, equals or exceeds that of the winner. [4] In this case, the 'conjunction' entry for 'then' is correctly eliminated, while the simple past entry for 'failed' is retained.

The experiment was repeated with several different factors. The results are given here for a factor of 300. However, while 300 was the best number we could find, it did not prove to be a trade-off that was spectacularly better than the results of the previous experiment. The results are summarised in figure 4, with those of the previous experiment repeated for easy comparison.

In general these results are in the right direction, though they do not go far enough, unfortunately. Roughly speaking, we get accuracy of a little over 90%, with efficiency gains of around 30%. It is still the case that the major source of inaccuracy is failure to find a parse.

These figures conceal one small fact which is worth remarking on. It was not the case that the parse failures on the second run were a subset of those on the first run. This somewhat curious state of affairs is explained by the pruning regime used. Recall that for each word, the set of categories derived from the tagger were intersected with those assigned by the CLE. Where the intersection was not null, only the intersection was retained. Where it was null, the CLE categories were retained. On several occasions it was the case that a previously null intersection became, on the second run, a non-null intersection, and so the set

[4] This seems like a big number. But probabilities get very small very quickly. Factors of 50 or more are needed, at least with the statistics we were using, to make any difference at all.

Figure 12.12: Over eager pruning

```
BEFORE:
 what: det(...)
 had: v(...) (4 times)
 he: np(...)
 told: [tell,ed] v_v_ed_Perf v(...) (6 times)
 told: [tell,ed] v_v_ed_Past v(...) (6 times)
 told: [tell,ed] v_v_ed_Passive v(...) (6 times)
 them: np(...)

AFTER (Factor=300):
 what: det(...)
 had: v(...) (4 times)
 he: np(...)
 told: [tell,ed] v_v_ed_Past v(...) (6 times)
 them: np(...)
```

of categories for the word seen by the CLE was actually smaller on the second run than on the first. It was thus possible for a sentence which had previously been parsed correctly to now fail, and this indeed did happen on several occasions.

Inspection of the remaining parse failures revealed that again the major cause of error was the shortsightedness of the tagger, even when relaxed as described. A typical example is shown in figure 5.

There are six different subcategorisations for 'tell' in the CLE lexicon. This means that there are 18 semantically and syntactically distinguishable possibilities for the morpheme 'told'. Here, pruning removes all but the simple past entries for 'told', even at the relatively generous factor of 300. This is incorrect, for the 'told' is linked with 'had' and thus is a perfective form. Unfortunately, 'had' and 'told' do not fall within the bigram 'window' and so the local preference of a finite form following a proper name overwhelms the correct possibility.

Of course, the correct form will be retained if the factor is increased. Unfortunately, we discovered that if the factor is increased to 400 or above, then while this achieves the desired result of retaining the correct tag, it also achieves the undesired result that *all* the other tags are also retained. Thus (on short sentences like these, at least) the only way to preserve accuracy is to abandon any increase in efficiency.

Notice that in the example above (and most of the others that failed for this reason) the relevant dependency would fall within a trigram window '... had he told ....'. In the general case, of course, there will always be some dependencies that will not do so: a similar sentence could be constructed with arbitrarily long NPs intervening between 'had' and 'told'. Thus although moving to a trigram based tagger would improve performance on examples like these, there would always be a residue of cases where analogous myopia would set in. In these cases the tagger is likely to go wrong systematically and we will be back in the same position.

12.7.5 Conclusions

This was in every way a preliminary study, and its findings should be treated with caution. It should be replicated using a tag set the same as that used by the parser, and using statistics derived from a corpus that has been analysed in a way consistent with the way the parser would analyse it. This would avoid the problems caused by the mismatch of tag sets. A larger number of examples should also be used so that the results would be more reliable statistically. Sentence length should also be controlled to see what interactions emerge by comparing performance of short and long sentences. (We tried to do this here but the numbers of sentences involved were so small that nothing reliable emerged.)

We would be very surprised if the results of a more accurate study were not somewhat more encouraging than those found here. However, we would also expect that there would still be a sizeable residue of cases involving non-local dependencies which would still not be covered properly. For some corpora or applications this residue might be small enough that the efficiency/accuracy tradeoff would be favourable. For others, this would not be the case.

There are several obvious places where statistical information could be used to good effect which the experiments described above did not tackle. The most obvious fact about the few example sentences discussed is the large number of subcategorisations given for each verb. Tag sets like that used by LOB and others do not distinguish between different subcategorisations for verbs, for the good reason that this is not usually determinable on a local bi- or tri-gram basis. However, if it were possible to eliminate unlikely subcategorisations this would have a large impact on efficiency. (Of course, it might also have a large impact on accuracy).

Chapter 13

Applications Development Reports

13.1 DRA Malvern

Sue Browning

The Speech Research Unit at DRA Malvern is primarily concerned with fundamental research into speech recognition techniques and algorithms. In 1989 we produced a demonstrator of speaker-dependent, task specific continuous speech recognition with a vocabulary of around 500 words, and we have just completed a more advanced, speaker independent version of that system. We are currently working towards task-independent large vocabulary recognition, among other things. As vocabulary size and task complexity grow we are becoming increasingly concerned with how speech can be integrated into a complex man-machine interface, and we see NLP as of vital importance to the future effectiveness of such interfaces.

Our interest in NLP therefore stems from the need to model (spoken) language in order to both constrain the recognition and to interpret the recognised speech in terms of the task domain. Currently most of our direct involvement in NLP is through projects contracted out to universities and others (including SRI). These are mainly investigations into the suitability of various techniques for language analysis and generation, with the emphasis on techniques that deal with "real" language, rather than artificial examples, as well as on those which are extendable to deal with the spoken, as well as the written, word. We are also extending our interest to look at speech dialogue management techniques, in which it is assumed that "natural" language can be both interpreted and generated.

As an application in which to develop and evaluate future speech and language technology, we have chosen Autoroute, a commercially available software package for route planning. The user supplies the start and destination points of a car journey in the UK, along with any places to be visited or avoided along the way, and Autoroute calculates the best route according to other user-specified variables, such as preferred road types and driving speeds. It was thought that this application would provide a relatively constrained environment, but one which would be rich enough to be interesting and challenging. Therefore, it is around the Autoroute application that most of our involvement and experience with CLARE revolves.

The first project which involved the use of CLARE at DRA involved the building of an interface between CLARE and Autoroute. The project was completed in May 1991 by Gordon Brown, a student working in the SRU. Gordon spent some time at SRI learning about the CLARE software and then was able to build a prototype of the interface fairly easily, although he had had no previous experience of either Prolog or NLP.

Version 1 of CLARE was used on a DEC 3100 workstation running Quintus Prolog v2.5, which was connected, by means of a commercially available program called pcANYWHERE, to Autoroute running on a Dell PC. It was necessary to extend the lexicon to cover the many new words in the route planning task, and this was done quickly and easily using VEX.

The interface took the TRL output from CLARE and searched it for predefined patterns of words to fill in templates containing the information required by Autoroute to process a query. These were then passed to the PC as control codes for Autoroute. The reply from Autoroute was easily returned because pcANYWHERE transmits the PC screen back to the Decstation. It was also possible to change the display type from map to table, but not all of the functionality of Autoroute was built into the prototype interface.

A small number of genuine queries elicited by questionnaire were collected as a sample on which to test the combined system. Detailed results are available in Gordon's report, which is part of the CLARE project documentation, but briefly, only around 9% of the 146 unique queries submitted got an appropriate answer from Autoroute, 74% failed to get through CLARE, and 17% failed at the interface stage. The majority of CLARE failures were due to there being no parse for the input,and this was usually caused by the use of non-standard (telegraphese) and ungrammatical constructions in the queries, and showed up important areas where the CLE could be extended to cover this type of language. Interface failures were almost always due to the pattern matching not being sophisticated enough, and this would clearly need extending in any future application.

It was clear from this experiment that a much larger body of data about the kinds of queries that would have to be handled would be crucial to further development of the Autoroute application. Therefore, we are currently producing a corpus of genuine spoken queries to the Autoroute system. Using "Wizard of Oz" techniques we simulate powerful speech recognition, natural language understanding and speech output in a genuine telephone-based route planning enquiry system, and have already recorded a quite sizeable corpus of spoken interactions. This corpus not only allows the techniques used in systems like

CLARE to be tested on and adapted to "real" spoken language (in a restricted domain), but also allows the SRU to study the speech produced by callers to such systems, both in terms of what kinds of things people say, and in terms of how they react to such a system, and therefore how the dialogue can be designed to accommodate them.

A second aspect of our involvement with CLARE stems from our concern with how speech and language technology can best be integrated. There are two basic ways in which NLP can aid speech recognition; by working in parallel with the recognition algorithms (or in some sense integrated into them) to make predictions that might constrain the recognition task, or by taking the output of the recogniser and analysing it to find the most likely interpretation or reject unlikely interpretations. The ASLU (Architectures for Spoken Language Understanding) project which the DRA supports at SRI is investigating how CLARE might be integrated with the SRU's speech recognition techniques and is using the route planning corpus.

More recently, we have had a very short exploratory project to investigate techniques for spoken dialogue management. The exchange between the user and Autoroute is viewed as a conversation in which the dialogue manager directs the conversation so as to extract the information necessary to construct a valid Autoroute query from the user in as efficient a manner as possible. The strategy employed to do this will depend on the user's degree of familiarity with the task and the system, and the quality of the communications link etc. The dialogue manager would also control the error handling and correction facilities which will be needed if the input is naturally spoken queries, due both to errors in the recognition process, and to the potential ambiguity of natural language and the inevitable lack of coverage in the NLP component. The project assumed the availability of a fairly sophisticated NL analysis and generation system (to generate appropriate queries and confirmation messages). Ultimately the intention would be to use something like the CLE, but due to lack of time in this prototype the NL component was simulated by a very simple pattern matching parser. Although it was very simple, some interesting ideas about generic dialogue management techniques were developed, which we intend to explore more fully in the future.

As well as further exploiting the route planning corpus to develop and evaluate generic and task specific speech and language technology, we also foresee that our interest in CLARE and NLP in general will have a slightly different focus. Many of our possible speech recognition applications could involve interrogation, manipulation, and maintenance of large complex databases, such as are becoming increasingly available in (military) command and control situations. Many of these involve quite restricted task domains with medium-sized vocabularies which have a rather rich structure. As we extend the power and flexibility of our speech recognition technology to these new domains there will be a need to automatically produce linguistic models targeted at these specific applications (something like sublanguage modelling). Although each model may be rather simple, the ability to customise the linguistic analyser automatically to new applications could be crucial to the take up of our speech technology.

13.2 BP Research

Dave Wolstenholme

13.2.1 The context for natural language processing

Perceived Business need for natural language processing

Information is the life-blood of a large organization such as BP. Without an adequate supply of good-quality information, individuals, teams and other organizational units are unable to function to their best ability; this clearly has a detrimental effect on the functioning of the overall organization. In our society, the volume and complexity of information and data is continually increasing, and it is necessary to make use of computers for storing it—frequently in databases. The problem with storing information in computer databases is that the people who might need the information frequently have great difficulty in obtaining it.

This problem is, in part, related to a more general problem associated with user-interfaces: that users of computer systems are often expected to learn unnatural, formal command or query languages. This need to communicate in machine-oriented terms is widely perceived as adversely affecting the acceptability of computer systems and as reducing the effective use made of them by users; the means of communication should be more user-oriented, e.g. through natural spoken or written English.

The problem of interacting with databases is, however, deeper than that. When using a formal database query language, such as SQL, a user is expected to have a fairly detailed knowledge of the way in which the data are stored, such as the names of tables and fields. This puts an unfair load on database users, particularly on those who need to access several databases, possibly using different query languages. The more casual user is, of course, at a serious disadvantage, and may, indeed, give up trying to obtain the data. There can be very few users, even so-called 'computer literate' ones, who have not been either unable to obtain the information they require or frustrated and delayed in the attempt. The information stored is thus being under-utilized, and is less valuable than it might otherwise be.

The problems identified above point to the need for a different way of interacting with databases. A good model might be the way we would obtain information from a human intermediary who has that information stored in some way, whether in paper form or on computer. We would ask for the information in English; we would not need to know where, or how, the information was stored (it may be in more than one physical place)—this would normally not concern us. Something the human intermediary also does is use common sense or specialized knowledge to bring together the various individual items of data needed to answer a particular question. So, for example, if the human intermediary were asked how long Fred's boss had worked for BP, he might look at a paper-based organigram to see who Fred's boss was, then query a computer database to see when that person joined BP, and, finally, use general knowledge of the current date to determine how long this corresponded to.

If we look at the model of interaction with a human intermediary, we see that the interaction is seldom a simple ask-question-get-answer type. Apart from the dialogue required by social conventions—saying hello, asking after each other's health, discussing the weather, etc.—there will normally be dialogue preliminary to the main question, concerned with establishing the background and reason for the question. This provides the context in which the question is set, and may be useful for helping the intermediary to answer the question. There may also be subsidiary dialogue associated with the question; this may be instigated by the intermediary in order to clarify any imprecisions or to sort out any ambiguities in the question. The answer too may not be straightforward. It may be that the question is outside the scope of the intermediary's knowledge, or that there is some other reason, e.g. security, why the answer cannot be given. This must be conveyed to the questioner. Even if a real answer can be given, certain qualifications and provisos may be given with it. For example, the intermediary may make explicit certain assumptions upon which the answer is based.

The CLARE Project, with its emphasis on interactive natural language processing and on contextual reasoning and cooperative response, was seen as a good vehicle for implementing NL database interfaces with interaction modelled on that of a human intermediary. These would address the two main problems identified above: first, providing interaction that is clearly human- rather than machine-oriented, and, second, ensuring that the user need not be aware of the way in which data are stored. They would also enable a human-computer dialogue to exhibit many of the features found in a complex human-human dialogue, as discussed above, such as using knowledge to bring together raw data to answer a question, handling context, and interacting to resolve ambiguities and explain in-built assumptions.

The main perceived benefits that would accrue to BP Businesses from the use of natural language database interfaces were clear and simple: a) information stored on computer would be more accessible and, hence, more useful and valuable; and b) individuals needing to access data would feel less frustrated, more in control, and less alienated from the ever-spreading computers. There would also be direct IT benefits, in that the separation of the user from the direct form of data representation would enable modifications to the representation to be made transparently.

BP involvement in CLARE and NLP

Those directly involved in the CLARE Project in BP have come from the Information Management (IM) team, one of several teams in the Information Science and Engineering (ISE) branch at BP's Research Centre, Sunbury-on-Thames. The teams in this branch are involved with advanced IT research and development. The IM team is concerned with many areas of information and knowledge management, including knowledge-based systems, database interfaces, computer-supported cooperative working, distributed systems, multimedia and process modelling. In addition, it employs psychologists for applied human factors work.

Apart from participation in an earlier Esprit project, LOKI, which had some NL content, the CLARE Project is BP's only involvement with natural language processing.

13.2.2 BP's experience with the CLARE Project

Development of a NL interface to a mock HR database

BP has three core businesses: exploration, oil and chemicals. One thing that all three have in common is human resources. For this reason, and because the domain was felt to be more complex than a technical one, and hence more challenging and more likely to raise interesting issues, it was decided that a natural language interface to a mock HR database should be built.

The database is modelled on BP's Sunbury research centre, and contains data about various object types including divisions, branches, teams, buildings, rooms, projects and staff. Details of an individual member of staff include date of birth, sex, whether married, number of children, address, skills, and telephone numbers. In addition, relationships between the various objects are stored, e.g. organization structure, locations of individuals and rooms, managerial posts and line management relationships.

The interface was firstly built on top of the result of the earlier Nattie Project, CLE. CLE was supplied with some demonstration interfaces, including one to a Prolog database about Cambridge colleges. Whilst this proved a useful basis for developing the interface to our HR database, which was also to be built first in Prolog, the general approach taken in it was seen to be inadequate for our intended purpose—to build an interface to a database held in a commercial DB system, accessed via SQL. The main reasons for this inadequacy were:

- identical representation of objects and relations in CLARE and the Prolog database program. Whilst useful for building a quick demonstration, this would be quite unrealistic for a real system, where the database representation might be quite different from the internal CLARE representation.

- the concept of events is important within the CLARE system, where the first argument of a verb represents an event. This representation was carried over directly into the colleges database, where each event was explicitly named, and the time of each event was recorded separately in the database. Again, this would be unrealistic in a real database.

- clauses for a particular relationship in the Prolog database were a mixture of general rules and database facts. In a real system, it is likely that only raw facts would be stored in the database, while general rules about them would be stored elsewhere, e.g. in a knowledge base.

The following clauses from the colleges database demonstrate these three points.

```
build_make(Event, Person, Building) :-
           found1(Event, Person, Building).
build_make(event40, wren1, trinity1).

found1(event1, beaufort1, christs1).

in_time(event1, year(1148)).
```

Therefore, the interface was designed with the following characteristics:

- the database containing the basic data (e.g. who is in which room, which room is in which building), was conceptually quite separate from the knowledge base. The knowledge base included general rules for inferring information from the raw data, e.g. which person is in which building.

- clean interfaces between CLARE and the knowledge base, and between the knowledge base and the database. All objects and relationships were mapped between different representations at these interfaces. Although both the knowledge base and database, as well as CLARE, were initially implemented in Prolog, this design allowed an easy porting to an Oracle database later.

- a function-like representation of events in arguments of relationships in the knowledge base. The following clauses demonstrate these.

```
clare_KB_pred(work_on_IsEmployedOn, work_on).


work_on([work_on, Person, Objective], Person, Objective) :-
    isa(Objective, objective),
    'works on'(Person, Objective).
work_on([work_on, Person, Objective], Person, Project) :-
    isa(Project, project),
    'objective of project'(Objective, Project),
    'works on'(Person, Objective).
```

```
'works on'(Person, B) :-
    db_query('works on_db',
             [[person, Person], ['work object', B]]).


db_query(Pred, [[T1, Var1], [T2, Var2]]) :-
    'KB_DB_object'(T1, Var1, V1),
    'KB_DB_object'(T2, Var2, V2),
    Term =.. [Pred, V1, V2], call(Term),
    'DB_KB_object'(T1, V1, Var1),
    'DB_KB_object'(T2, V2, Var2).
```

The interface was later modified for a setup where the basic data were held in an Oracle database (see Figure 13.1). Apart from creating and linking to the database itself, this involved making changes only to the database query procedure and to the relevant object conversion procedures. The Oracle database was resident on an HP machine, while CLARE and the knowledge base ran under Quintus Prolog on a Sun. The database query procedure made use of a Quintus Prolog-Oracle SQL interface package, ProDBI (now from Keylink Computers Ltd), for generating SQL queries to Oracle for solving the lowest-level relationships. The link across the network between the Sun and the HP was made via SQLNet. No optimization of SQL queries was carried out; whenever a knowledge-base relationship needed information from the database to be solved, an SQL query was made, in the same way that a Prolog query would normally have been made. This approach would not be suitable for large databases. This approach contrasts with that taken later in the CLARE Project where several queries are effectively bundled into one more complex query.

The system was later ported onto CLARE, version 1. This conversion was not trivial, as there were many internal changes between CLE and CLARE 1. CLARE 1 allowed handling of ellipsis and paraphrase generation. A typical sequence of questions and answers is shown next:

Where does Dave Smith work?
Information Management, ISE, Room 25, Building 200, Sunbury, etc

Which room does he work in?
Room 25

What year was he born in?
1953

What is his boss's phone number?
3256

Who is Humphrey's boss's line manager?

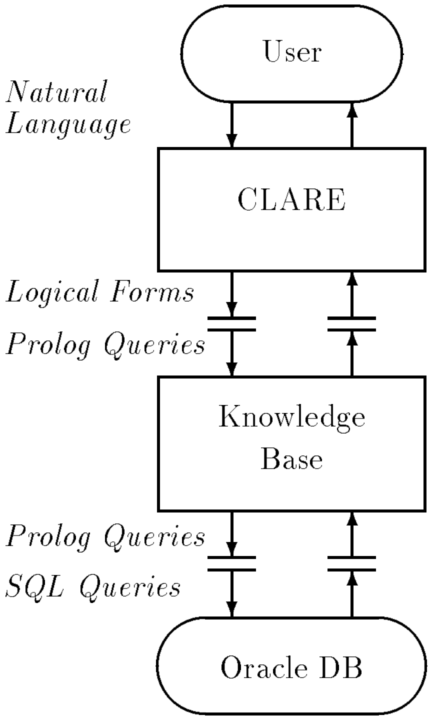


Figure 13.1: Architecture of CLARE-Oracle prototype

Jim Boon

Who works in Information Management and knows about fuel technology?
John Corner

Is he married?
No

Who is the oldest person in the ISE?
Eric

Who is older than his boss?
Eric

How many women are there in Jim's branch?
5

In which team are more than two women working?
Information Management

Does Mark know about HCI?
Yes

Logic programming?
No

Does Dave?
Yes

The system has been demonstrated to many parts of BP, and to external companies, and has been favourably received. The ability to ask complex queries with a certain amount of indirection in them, e.g. *What is his boss's phone number?*, and *ad hoc* queries for which it would be unlikely that pre-defined templates/screens would be provided, *e.g. In which team are more than two women that know about HCI working?* was particularly liked. The fact that the architecture does not restrict the data source to be just one database, so that the knowledge base could be given access to several databases in order to answer a query, was also appreciated.

The architecture chosen, where objects and relationships needed to be represented in several different ways (in English, in CLARE, in the knowledge base and in the database), led us to develop tools to aid system construction. These tools were written in MacProlog, where advantage could be taken of its multi-windowing environment. The first tool aided the development of a 'raw database', i.e. definitions of the class hierarchy, objects, atrributes and basic binary relationships between them. The second provided for taking this raw database and producing all the Quintus Prolog code required for the system, i.e. the CLARE definitions, the knowledge base, the database and the interface/conversion procedures. The code resulting from this second tool was a combination of Prolog code and CLARE definitions generated from the raw database and directly written code. The system required both that a domain model be built and that many extensions to the lexicon be provided, and the tool greatly aided their provision. For example, it provided easy ways of defining synonyms for nouns, adjectives and verbs, and of defining type declarations and relational forms of nouns. The following are example declarations given within the tool, from which CLARE declarations for the given nouns and verbs were directly generated.

```
type_sort(section_leader, count, section_leader_LeaderOfSection, person).
rel_pred(section_leader, section_leader_SectionLeaderOfPerson, person).
rel_pred(section_leader, section_leader_SectionLeaderOfSection, branch).
synonyms(section_leader, [[section, leader],[leader], [project, leader],
         [group, leader], [team, leader]]).

type_sort(computing, mass, computing_ComputingSubject, subject).
synonyms(computing, [[computing], [computing, science],
                                  [computer, science]]).

clare_verb(know_about, know_about_IsFamiliarWith, [person, subject]).
verb_synonyms(know_about, [v_subj_specpp([know, about]),
                           v_subj_obj([understand])]).
```

Because of the perceived need to generate new code automatically when changes were made, the VEX tool was not used in developing the system. The system did not require changes to the built-in sortal hierarchy, and changes to the built-in grammar were not made directly. However, development of the prototype did highlight several shortcomings of the grammar, which was modified accordingly by SRI. For example, the construction where an object is named by its type followed by a subname/number, e.g. 'building 200', 'room 12A', or 'project lionheart', was found to be a very common requirement which could not originally be handled.

No direct use has been made of versions 2.0 or 2.5 of CLARE, except for demonstrations, which have, again, been well-received.

The future of CLARE and NLP in BP

Although CLARE and the prototypes built using it have been well-received, none of the BP Businesses have allocated direct funding for any further development work, or for continued research into NLP. The general perception is that

- the costs of developing NL database interfaces is high, whilst the benefits obtained are difficult to quantify;
- much work is still required to provide an acceptable HCI;

- NL database interfaces face competition from other means of access, e.g. graphical interfaces. Whilst these may not provide the naturalness of NL interfaces, and generally do not hide the users from the database structure, they are found to be acceptable and flexible enough for the domains of interest to BP.

Related to this last point, BP do see benefit in NLP related to text files, where there is less competition from other means. Potential applications include mail filtering and summarization, first step translation of foreign language documents, and checking specifications and operator manuals for ambiguity.

Future use of CLARE, and of NLP work generally, within BP has been put further into doubt by recent events. BP has taken a strategic decision to concentrate on its core Businesses and to halt all internal IT research and outsource IT development. This has led to the disbanding of ISE from the end of 1992.

13.3 BT Labs

Gavin Meggs and Sandra Williams

BT Labs. has maintained active support and interest in SRI Cambridge's CLARE project throughout its lifetime. Lack of resources and manpower have restricted the amount of work carried out at BT Labs using CLARE, except for the three short-term projects described below:

13.3.1 Sept - Nov 1990 Geoffrey Sampson's evaluation of the July 1989 version of the CLE

Geoffrey Sampson spent 12 weeks working at BT Labs on a short-term fellowship. He evaluated the CLE using a small corpus of naturally occurring queries elicited from employees of BT Labs. Although his corpus was too small to obtain statistically significant results, he found that the performance of the CLE compared favourably with that of other major NLP parsers and semantic analysers. He performed an extensive investigation of the coverage of the lexical, grammatical and semantic components of the CLE. He criticised the many gaps in coverage, the idiosyncrasies of the system, and the lack of robustness, mentioned above. As quantified in the coverage evaluation chapter, linguistic coverage has been considerably extended under CLARE and we hope the ARLA project will bridge other gaps.

13.3.2 Summer 1990 - April 1991 Interfacing CLARE to BT's SPUD database

This work was carried out in an attempt to direct some of the work of BT Labs. Natural Language Processing Group towards a practical application. It was a small-scale project carried out by one person over 9-10 months and called BARNACLE (Business Application of a Relational Network database with the Aid of the Core Language Engine). Since the work was to use a new database which was being developed in another part of BT, a great deal of time was spent liaising with the database developers and learning about its proposed structure. BT's SPUD (Service Protection Utilities Database) is a relational database implemented in ORACLE. For the CLARE application, a "snapshot" of the database under development was used which contained 37 database relations. The SPUD developers provided a corpus of 40 database queries in English and the equivalents of the first five in SQL The database "snapshot" was simulated using the Quintus Prolog database interface to represent the database relations as Prolog clauses. The BARNACLE system was designed with three major parts: a Prolog translator to translate CLARE's LFs (Logical Forms) to RXs (relational query expressions); application-dependent extensions to CLARE; and software to control the system. CLARE's TRL was not used because at the time, summer 1990, it was too ill-defined. The LF to RX translator was implemented successfully with the LFs simplified into as near flat lists as possible prior to translation. The control software was never implemented in the short project life. The emphasis of this project was more linguistic than that of the project described above, with more work on application-dependent extensions to CLARE. A domain-dependent lexicon was constructed and an interface to the Oxford Advanced Learner's Dictionary was investigated. Effort was directed towards trying to resolve gaps in CLARE's coverage within the restricted domain. The conclusions as a result of this work were that CLARE ought to be made more robust. There are large gaps in the coverage and the system (at the time) fell over if it came across anything it did not recognise. Analysis of most sentences in the sample corpus failed. The robustness problem will hopefully be addressed in the next phase, ARLA.

13.3.3 April-May 1992: The Use of CLARE in Intelligent Query

Research into the field of Intelligent Query necessitates the study of systems which provide the user with an interface which is easily understood, yet has the complexity to reason with the query so that relevant information may be accessed. The most natural choice for a language which the user will understand is natural language, but until recently natural language systems have been incapable of providing adequate reasoning capabilities. CLARE, at first glance, appears to overcome this restriction.

CLARE-2's PRM domain model, and associated literature, shows how a natural language processing system may be interfaced with a fictional database of Prolog facts. Preliminary evaluations of CLARE-2 have shown that it is also possible to interface CLARE-2 with a relational database and to build a simple domain model based around the structure of the PRM model.

The PRM data was translated into relations which were stored in a relational database, using the ORACLE database management system. An interface was then designed which took the low-level database queries made by CLARE-2 and transformed them into SQL queries. These queries were made via ORACLE to the (now relational) PRM database. Performance was seen to be almost identical to that of the original system, with the exception of recursive table queries which gave error messages. It should be noted that these errors were not a fault of CLARE-2, but of the method of translation from Prolog queries to SQL queries and should be overcome by the forthcoming SQL/CLARE-2 interface being developed by SRI.

The idea of Project Resource Management (PRM) was taken up as a suitable test-bed for building a domain model. Data on local projects and individuals was collected and stored both as Prolog facts and relations within a relational database. The sequence of steps outlined in the CLARE-2 software manual (Oct. 1991) and end of year report (Nov. 1991) were then followed in order to build a new domain model. The resultant complete domain model can be used to extract information about real projects and real individuals in a real domain using queries typed in English.

The steps for building a domain model explain how to build a declarative representation of the domain that can be used for reasoning by the CLARE-2 system. The majority of the steps were found to be quite straightforward and their implementation was simple. Some difficulty was encountered in defining assumptions for the domain, but by far the most difficult step was the definition of 'equivalence rules'. For the implementation of this step it was necessary to make heavy use of the PRM model as a template from which to build new equivalence rules. It should also be noted that in building the new domain use was made of the PRM lexicon files and so no measure of the difficulty of building a domain specific lexicon was made.

Since the development of the above system additional literature has been provided by SRI which describes in more detail the process for building a domain model. This information both clarifies the use of assumptions in query processing and goes some way to simplifying the process of building equivalence rules. Using the updated 'guide to building a domain model' we hope to use CLARE to interface with a more complex domain database, which has yet to be identified.

13.4 British Aerospace Involvement with the CLARE Project

Phil Sims

This section briefly describes the background to the Sowerby Research Centre's (SRC) involvement with the CLARE project. It also describes the SRC's experience to date with the software and the future plans to continue work in the area of Natural Language Processing and related applications.

British Aerospace joined the CLARE consortium on the basis of its experience as a member of the previous NATTIE project. The results gained in the development of the Core Language Engine (CLE) had been extremely promising, and the goal of the CLARE project to show the CLE working in an embedded application, and incorporating further contextual reasoning was in line with the company's own view of where the next progress should be made.

Each version of the CLARE software has been installed and runs on the Sun Network at the Sowerby Research Centre. It has been demonstrated to a large number of senior management in the course of visits made to the Centre. An early demonstration of queries to an possible in-house project management system was used to demonstrate the basic principles behind the system, and time was spent with SRI working on this application. Further plans have been drawn up to extend this work to a full demonstration of the software's capability by using it to access the Project Management software which has recently been installed and which is now used throughout the Research Centre to plan and monitor all of its projects.

Two other pieces of work currently under discussion with the British Aerospace may also lead to a future use of the software:

Automatic Document Generation British Aerospace produces a number of extremely complex systems. The documentation of such systems often runs into many volumes. Whilst much of this documentation often goes unused it is a contractual requirement, and represents a significant amount in project's overall cost. This is particularly true when modifications are made to the system, or slight variants are made to an existing product for a new customer. A new set of complete documentation is required and current methods of production are being investigated to improve cost efficiency. As a result, three lines of research have been proposed in this area:

- The automatic production of documentation from appropriate models of the relevant system - Standard Logical Form, and the generation software in CLARE will be relevant.
- Producing documentation in an electronic as well as paper form.
- Building interfaces to any resulting electronic version of the documentation - Natural Language will be considered in conjunction with

HyperText interfaces.

Message Interpretation Another area of British Aerospace business is in providing communications systems. Recent developments in the area mean that in order to stay competitive, systems must provide some degree of extra processing. In the first instance this will be parsing and error correction in the the structured headers used to introduce and close messages, but the next step will be to extend this approach to an analysis of the free text included within the message, for error correction, paraphrasing and recording in a historical database. Again, we hope to deploy some of the CLARE based software and technology in this area.

The Sowerby Research Centre feels that it has gained appreciably through its involvement with the CLARE consortium, both in terms of the software it has received, and from the more general transfer of natural language processing technology from SRI, Cambridge University and the other partners. Although the SRC has been resource limited throughout the project, thus restricting its own ability to fully exploit the work, it is hoped that new areas of work which are taking place in conjunction with the operating companies will lead to an increased level of commitment to exploiting the obvious technological lead which CLARE could give the company.

13.5 SRI and Swedish partners: A speech-to-speech translation system

In this final section, we briefly report on a new project, aimed at construction of a prototype speech-to-speech translation system, in which the Bilingual Conversation Interpreter (BCI) appears as a subsystem. The BCI itself consists of two copies of the CLE, for the source and target natural languages, together with a transfer component for converting between QLFs representations for these languages. The other subsystems of the speech translation system are the SRI DECIPHER speech recognizer (Murveit, Butzberger and Weintraub 1991), and the Swedish Telecom Prophon speech synthesizer (Bäckström, Ceder and Lyberg 1989, Ceder and Lyberg 1990). The project is being sponsored by Swedish Telecom; it started in August 1992, and is initially planned to run for one year. At the end of this period, the intention is to be able to demonstrate a prototype system which will translate spoken English into spoken Swedish, using a vocabulary of between 700 and 1000 words. The proposed architecture of the system is sketched out in Figure 13.2.

The system's domain will be defined by the well-known ATIS corpus, a collection of about 10000 sentences relating to air travel information. Typical examples of sentences from the corpus are displayed below.

```
LIST FLIGHTS FROM DENVER TO BALTIMORE.
I'D LIKE TO GO AT FIVE FORTY.
WHAT MEALS ARE SERVED ON EASTERN FLIGHT SEVENTY.
SHOW ME THE MORNING FLIGHTS FROM BOSTON TO PHILADELPHIA.
I WOULD LIKE TO PLAN A FLIGHT ON AMERICAN AIRLINES.
HOW MUCH DOES IT COST TO USE THE AIR TAXI.
WHAT IS THE LEAST EXPENSIVE FLIGHT BETWEEN BOSTON AND SAN FRANCISCO.
SHOW ME THE ROUND TRIP FARE FOR THE U S AIR FLIGHT.
WHAT FLIGHTS GO FROM PHILADELPHIA TO SAN FRANCISCO WITH A
                                     STOPOVER IN DALLAS.
```

It is still somewhat early to make any definite promises about the outcome of the project, but it should be noted that the BCI has several concrete strengths in this type of application when compared to other MT architectures. Firstly, the high quality of the translation output becomes doubly important when dealing with spoken language, since post-editing is in the nature of things impractical. Secondly, the fact that the Core Language Engine is basically a general natural-language processing device greatly simplifies the task of switching domains (in this case, from car hire to airline reservations); in fact, the preliminary adaptation of the English CLE to the ATIS domain already produces plausible analyses for over 80% of the corpus sentences under 15 words in length.

The performance of a preliminary version of the system for tests on unseen material carried out on the 25th of November are shown in the tables in Figures 13.3 and 13.4. (These are scores for English analysis only, NOT translation).

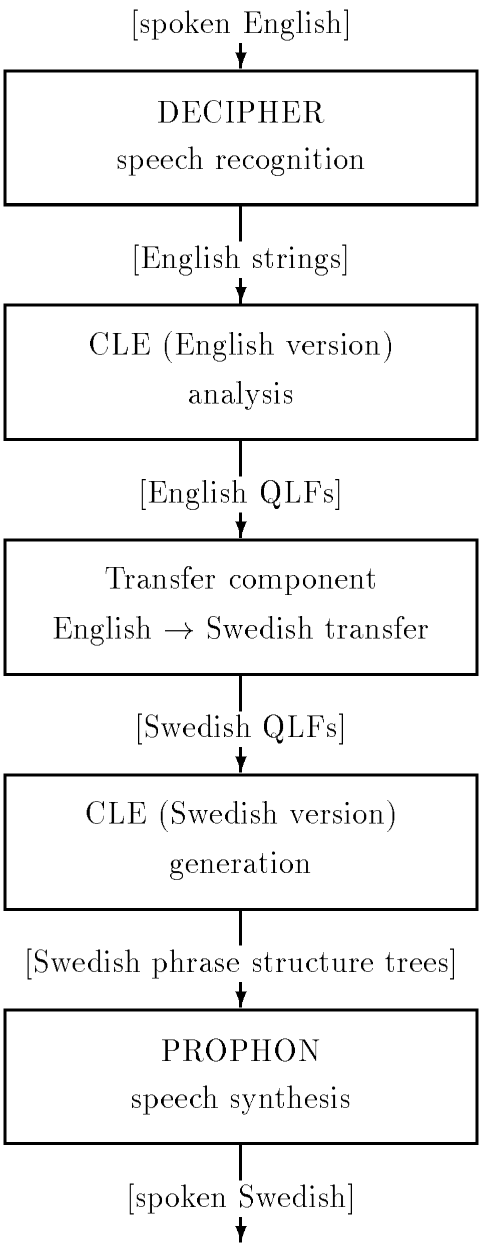


Figure 13.2: Architecture of the Spoken Language Translation System

Length	Number of sentences	Highest ranked QLF correct	Some other QLF correct	All QLFs incorrect	No QLFs produced
1-10	114	88	12	5	9
11-15	62	36	10	10	6
15+	24	2	4	5	13

Figure 13.3: Test of English CLE grammar and preference heuristics on 200 unseen ATIS sentences, 25/10/92.

Length	Some QLFs produced	Some QLF correct	Highest ranked QLF correct
1–10	92%	88%	77%
1–15	91%	83%	70%
all	86%	76%	63%

Figure 13.4: Cumulative scores

Chapter 14

NLP Evaluation Study – Report Digest

Evaluating Natural Language Processing Systems

J R Galliers and K Sparck Jones

Computer Laboratory, University of Cambridge
New Museums Site, Pembroke Street, Cambridge CB2 3QG, UK

December 1992

This report is a comprehensive study of NLP system evaluation. It provides a detailed analysis of the concepts and issues involved, an extensive review of the state of the art as represented both by actual evaluations to date and by published discussions of evaluation methodology in, for example, workshop proceedings, and suggestions for sensible approaches to evaluation.

Part 1 establishes a descriptive framework for addressing evaluation. This refers on the one hand to systems and their subsystems and components, and on the other to systems within their settings, as together forming 'setups'. These are then examined in detail for their elements and aspects relevant to evaluation, via carefully developed examples. This emphasises, for instance, the importance of system objectives and setup purposes in relation to evaluation. The report then dissects evaluation to identify and characterise its many constituent notions drawing distinctions, for example, between evaluation criteria, measures and methods, and again illustrating their application via a set of related examples. The discussion emphasises the need to distinguish a system and its parameters from a system's environment and its variables. Part 1 also includes discussions, for comparative purposes, of evaluation in information retrieval, in speech processing, and in the social sciences in general; and it considers the particular problems presented by 'generic' system evaluation, i.e. how to evaluate language processors independent of task applications, and by the provision of general-purpose test data.

Part 2 is devoted to a detailed account of the present situation in NLP evaluation. Interest and activity in evaluation has grown rapidly in the last few years, as evinced by the DARPA Conferences and by a number of workshops specifically addressing evaluation. The description of actual evaluations covers major or representative cases and is grouped by four main areas: machine translation, message understanding, database query, and speech understanding. One of the most important issues involved is the extent to which NLP evaluation has to be task specific. The description of explicit discussions of evaluation methodology (as opposed to its exhibition in use) is devoted in particular to accounts of and comments on the 1991 Berkeley and 1992 Edinburgh Workshops; and the DARPA evaluations, which are having a noticeable impact on evaluation, are collectively considered for their overall contribution to evaluation methodology. This part also briefly reviews the current state of speech processing evaluation and of the design and provision of test resources like corpora and toolkits. The report lists the main points emerging from this review, for instance that evaluation has so far focussed on systems without sufficient regard for their environments, and finds that NLP evaluation methodology is still relatively underdeveloped, for instance in frequently failing to be clear about the precise remit and bounds of evaluation, and in not addressing factor decomposition with enough rigour.

Part 3 addresses the future. The suggested approach to evaluation is based on two major grounds: the first is that NLP evaluation narrowly conceived is, if not a chimaera, of very limited value; the second is that evaluation techniques cannot be at once specific and generally applicable. The only approach to NLP evaluation is through a proper evaluation *strategy*, which has two main components: a complete statement of the evaluation *remit* and a thorough decomposition of the whole system/setup into all its *factors*. These together imply a systematic *grid* design for evaluation and thus in general not a single test but a test *programme*. Particular test and evaluation *techniques* can only be chosen and applied after this explicit and detailed characterisation of what is given and what is required has been provided. These points are again illustrated by worked examples. Thus the report conclusion is that while there are no magic bullets for NLP evaluation, and the present standard must be raised, this can be done by wholehearted pursuit of the strategy described. In the process this will provide the more specific, applicable knowhow needed to carry out actual evaluations that are well-conducted and thus informative ones advancing NLP itself.

(Approx, 175 pages, with Bibliography and Figures; available from the Bookshop, Computer Laboratory, University of Cambridge, early 1993)

References

Abe, M., Y. Ōshima, K. Yuura and N. Takeichi. 1986. "A Kana-Kanji Translation System for Non-Segmented Input Sentences Based on Syntactic and Semantic Analysis", *Proceedings of the Eleventh International Conference on Computational Linguistics*, 280–285.

Alshawi, H. and D.M. Carter. 1992. "Optimal Scaling of Preference Metrics". Paper submitted for presentation at the 31st Annual Meeting of the Association for Computational Linguistics.

Alshawi, H. 1990. "Resolving Quasi Logical Forms". *Computational Linguistics* 16:133–144.

Alshawi, H., ed. 1992. *The Core Language Engine.* Cambridge, Massachusetts: The MIT Press.

Alshawi, H., D.M. Carter, M. Rayner and B. Gambäck. 1991. "Translation by Quasi Logical Form Transfer", in *Proc. 29th ACL*, Berkeley, California.

Alshawi, H., Rayner, M., and Smith, A.G. 1991. "Declarative Derivation of Database Queries from Meaning Representations", in Society for Worldwide Interbank Financial Telecommunications S.C. (ed.) *Proceedings of the 1991 BANKAI Workshop on Intelligent Information Access*, Elsevier, Amsterdam.

Appelt, D. and J.R. Hobbs. 1990. "Making Abduction more Efficient", in *Proc. DARPA Workshop on Speech and Natural Language*, Morgan Kaufmann, San Mateo.

Bäckström, M., K. Ceder and B. Lyberg 1989. "Prophon – an Interactive Environment for Text-to-Speech Conversion", *Proceedings of the European Conference on Speech, Communication and Technology*, Paris, CPC Consultants Ltd, Edinburgh, UK.

Barwise, J., and R. Cooper. 1981. "Generalized Quantifiers and Natural Language". *Linguistics and Philosophy* 4:159–219.

van Berkel, B., and K. De Smedt. 1988. "Triphone Analysis: A Combined Method for the Correction of Orthographical and Typographical Errors", *Proceedings of the Second Conference on Applied Natural Language Processing*, pp. 77–83.

Bronneberg, W.J.H.J., H.C. Bunt, S.P.J. Landsbergen, R.J.H. Scha, W.J. Schoenmakers and E.P.C. van Utteren. 1980. "The Question Answering System PHLIQA1". In L. Bolc (ed.), *Natural Language Question Answering Systems.* Macmillan.

Calzolari, N. and R. Bindi. 1990. "Acquisition of Lexical Information from a Large Textual Italian Corpus". *Proceedings of the 13th International Conference on Computational Linguistics*, 3:54–59.

Church, K.W. and P. Hanks. 1990. "Word Association Norms, Mutual Information, and Lexicography". *Computational Linguistics* 16:22–30.

Carter, D.M. 1987. "An Information-theoretic Analysis of Phonetic Dictionary Access", *Computer Speech and Language*, 2:1–11.

Carter, D.M. 1989. "Lexical Acquisition in the Core Language Engine", *Proceedings of the Fourth Conference of the European Chapter of the Association for Computational Linguistics*, 137–144.

Ceder, K. and B. Lyberg 1990. "The Integration of Linguistic Levels in a Text-to-Speech Conversion System", *Proceedings of the International Conference on Spoken Language Systems*, Kobe, Japan.

Clark, K.L. 1978. "Negation as Failure". In H. Gallaire and J. Minker (editors), *Logic and Data Bases*, Plenum, New York.

Coltheart, M. 1981. "The MRC Psycholinguistic Database". *Quarterly Journal of Experimental Psychology*, 33A:497–505.

Dalrymple, M., S. M. Shieber, and F. C. N. Pereira. 1991. "Ellipsis and Higher-Order Unification". *Linguistics and Philosophy*, 14:399–452.

Davidson, D. 1967. "The Logical Form of Action Sentences". Reprinted in Davidson, 1980, *Essays on Actions and Events*, Oxford University Press.

De Marcken, C. G.. 1990. "Parsing the LOB corpus", *Proceedings of the 28th ACL*, Pittsburgh, 243–251.

DeRose, S. 1988. "Grammatical Category Disambiguation by Statistical Optimisation", *Computational Linguistics*, 14:31–39.

Emirkanian, L., and L.H. Bouchard. 1988. "Knowledge Integration in a Robust and Efficient Morpho-syntactic Analyser for French", *Proceedings of the Twelfth International Conference on Computational Linguistics*, pp 166–171.

Futrelle, R.P., C.E. Dunn, D.S. Ellis and M.J. Pescitelli, Jr. 1991. "Preprocessing and Lexicon Design for Parsing Technical Text", *Proceedings of the Second International Workshop on Parsing Technologies*, pp. 31–40.

Grosz, B. J., D. E. Appelt, P. Martin, and F. Pereira. 1987. "TEAM: An Experiment in the Design of Transportable Natural-Language Interfaces". *Artificial Intelligence* 32: 173–243.

Hirschbühler, P. 1982. "VP Deletion and Across the Board Quantifier Scope". *Proceedings of NELS 12*, GLSA, University of Massachusetts.

Hindle, D. and M. Rooth. 1990. "Structural Ambiguity and Lexical Relations". *Proceedings of the DARPA Speech and Natural Language Workshop, Morgan Kaufman*, 257–262.

Hobbs, J. R. and J. Bear. 1990. "Two Principles of Parse Preference". Vol. 3, Proceedings of the 13th International Conference on Computational Linguistics, Helsinki, 162–167.

Hobbs, J.R., M. Stickel, P. Martin and D. Edwards. 1988. "Interpretation as Abduction", in *Proceedings of the 26th ACL*, Buffalo, New York.

Hobbs, J. R., and S. M. Shieber. 1987. "An Algorithm for Generating Quantifier Scopings". *Computational Linguistics* 13:47–63.

Huet, G. 1975. "A Unification Algorithm for Typed Lambda Calculus". *Theoretical Computer Science* 1:27–57.

Kamp, H. 1981. "A Theory of Truth and Semantic Representation". In *Formal Methods in the Study of Language*, edited by J. A. G. Groenendijk, T. M. V. Janssen, and M. B. J. Stokhof. Mathematisch Centrum, Amsterdam.

Kernighan, M.D., K.W. Church, and W.A. Gale. 1990. "A Spelling Correction Program Based on a Noisy Channel Model", *Proceedings of the Thirteenth International Conference on Computational Linguistics*, pp 205–210.

Konolige, K. 1981. *The Database as Model: A Metatheoretic Approach*, SRI technical note 255.

Korf, R. 1986. "Search: A Survey of Recent Results" in *Exploring Artificial Intelligence: Survey Talks from the National Conferences on Artificial Intelligence*, Morgan Kaufmann, San Mateo.

Koskenniemi, K. 1983. *Two-level morphology: a general computational model for word-form recognition and production.* University of Helsinki, Department of General Linguistics, Publications, No. 11.

Lewin, I. 1990. "A Quantifier Scoping Algorithm without a Free Variable Constraint", Proceedings of COLING 1990.

McCord, M.C. 1989. "A New Version of Slot Grammar". IBM Research Report RC 14506, IBM Thomas J. Watson Research Center, Yorktown Heights, New York.

McCord, M.C. 1987. "Natural Language Processing in Prolog", in A. Walker (ed.) *Knowledge Systems and Prolog.* Addison-Wesley, Reading, MA.

Miller, G.A., ed. (1990). "WordNet: An OnLine Lexical Database", *International Journal of Lexicography*, 3:4.

Moran, D. B. 1988. "Quantifier Scoping in the SRI Core Language Engine". Proceedings of the 26th Annual Meeting of the Association for Computational Linguistics, State University of New York at Buffalo, Buffalo, New York, 33–40.

Murveit, H., J. Butzberger and M. Weintraub (1991). "Speech Recognition in SRI's Resource Management and ATIS Systems". *Proceedings of the Speech and Natural Language Workshop*, Morgan Kaufmann.

Pereira, F. C. N. 1990. "Categorial Semantics and Scoping", *Computational Linguistics* 16:1 1–10.

Pereira, F.C.N. and M.E. Pollack. 1991. "Incremental Interpretation", *Artificial Intelligence* **50** 37-82.

Pereira, F.C.N. and S.M. Shieber. 1985. *Prolog and Natural-Language Understanding*, CLSI Lecture Notes.

Perrault, C. Raymond and Barbara J. Grosz. 1988. "Natural Language Interfaces" in *Exploring Artificial Intelligence: Survey Talks from the National Conferences on Artificial Intelligence*, Morgan Kaufmann, San Mateo.

Pollock, J.J., and A. Zamora. 1984. "Automatic Spelling Correction in Scientific and Scholarly Text", *Communications of the ACM*, 27:4, pp 358–368.

Quinlan, P. 1992. *The Oxford Psycholinguistic Database.* Oxford University Press.

Quirk, R., S. Greenbaum, G. Leech and J. Svartvik. 1985. *A Comprehensive Grammar of the English Language.* Longman.

Rayner, M. and S. Janson. 1987. "Epistemic Reasoning, Logic Programming, and the Interpretation of Questions". In *Proc. 2nd International Workshop on Natural Language Understanding and Logic Programming*, North-Holland.

Rayner, M. and S. Janson. 1989. "Finding Out = Achieving Decidability", in *Working notes: IJCAI workshop on Knowledge, Planning and Perception*, Detroit, Michigan. Also available as Research Report R89015, Swedish Institute of Computer Science, Kista, Sweden.

Scha, R.J.H. 1983. *Logical Foundations for Question Answering*, Ph.D. Thesis, University of Groningen, the Netherlands.

Sekine, S., J.J. Carroll, S. Ananiadou, and J. Tsujii. 1992. "Automatic Learning for Semantic Collocation". *Proceedings of the Third Conference on Applied Natural Language Processing*, ACL, 104–110.

Shieber, S. M., G. van Noord, F. C. N. Pereira, and R. C. Moore. 1990. "Semantic-Head-Driven Generation". *Computational Linguistics* 16:30–43.

Stallard, D.G. 1986. "A Terminological Simplification Transformation for Natural Language Question-Answering Systems". *Proceedings of the 24th Annual Meeting of the Association for Computational Linguistics*, ACL, 241–246.

Stickel, M. E. 1986. "A Prolog technology theorem prover: implementation by an extended Prolog compiler.", *Journal of Automated Reasoning*, 4, 353-380.

Sterling, L. and S. Shapiro. 1985. *The Art of Prolog*, Addison-Wesley, Reading, MA.

Veltman, F. 1990. "Defaults in Update Semantics", in H. Kamp (ed), *Conditionals, Defaults and Belief Revision*, DYANA deliverable R2.5.A.

Veronis, J. 1988. "Morphosyntactic Correction in Natural Language Interfaces", *Proceedings of the Twelfth International Conference on Computational Linguistics*, pp 708–713.

Webber, B.L. 1983. "So What Shall we Talk About Now". Reprinted in Grosz, Sparck-Jones and Webber (eds), 1986, *Readings in Natural Language Processing*. Morgan Kaufmann.