

A Corpus-Consulting Probabilistic Approach to Parsing: the CCPX Parser and its Complementary Components

Michael David Day

Department of Computer Science
University of Wales
College of Cardiff

June 2007

A DISSERTATION
SUBMITTED IN PARTIAL FULFILLMENT
OF THE REQUIREMENT FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

UMI Number: U585093

All rights reserved

INFORMATION TO ALL USERS

The quality of this reproduction is dependent upon the quality of the copy submitted.

In the unlikely event that the author did not send a complete manuscript and there are missing pages, these will be noted. Also, if material had to be removed, a note will indicate the deletion.



UMI U585093

Published by ProQuest LLC 2013. Copyright in the Dissertation held by the Author.
Microform Edition © ProQuest LLC.

All rights reserved. This work is protected against
unauthorized copying under Title 17, United States Code.



ProQuest LLC
789 East Eisenhower Parkway
P.O. Box 1346
Ann Arbor, MI 48106-1346

Abstract

Corpus linguistics is now a major field in the study of language. In recent years corpora that are syntactically analysed have become available to researchers, and these clearly have great potential for use in the field of **parsing natural language**. This thesis describes a project that exploits this possibility. It makes four distinct contributions to these two fields.

The first is an **updated version** of a corpus that is (a) analysed in terms of the rich syntax of **Systemic Functional Grammar (SFG)**, and (b) annotated using the eXtensible Mark-up Language (XML).

The second contribution is a native XML **corpus database**, and the third is a sophisticated **corpus query tool** for accessing it.

The fourth contribution is a new type of parser that is both **corpus-consulting and probabilistic**. It draws its knowledge of syntactic probabilities from the corpus database, and it stores its working data within the database, so that it is strongly database-oriented.

SFG has been widely used in **natural language generation** for approaching two decades, but it has been used far less frequently in parsing (the first stage in **natural language understanding**). Previous SFG corpus-based parsers have utilised traditional parsing algorithms, but they have experienced problems of efficiency and coverage, due to (a) the richness of the syntax and (b) the challenge of parsing unrestricted spoken and written texts.

The present research overcomes these problems by introducing a new type of parsing algorithm that is 'semi-deterministic' (as human readers are), and utilises its knowledge of the rules - including probabilities - of English syntax.

A language, however, is constantly evolving. New words and uses are added, while others become less frequent and drop out altogether. The new parsing system seeks to replicate this. As new sentences are parsed they are added to the corpus, and this slowly changes the frequencies of the words and the syntactic patterns. The corpus is in this sense **dynamic**, and so simulates a human's changing knowledge of words and syntax.

Acknowledgements

There have been many people who have helped make this work a success. I would especially like to thank the following:

- **Robin Fawcett, my Ph.D. supervisor -**
 - First for introducing me to computational linguistics in 1992, and for providing me with the necessary challenge and enthusiasm that has made this work a success.
 - Second, for giving an excellent research environment at Windsor Crescent.
 - Third, for assistance in the research, particularly in the work towards the new corpus and a new parser.
 - Finally, but most importantly, for being a friend.
- **Andrew Jones, my second supervisor -**
For his rapid readings of the chapters and his constructive criticisms of it (often at short notice and at times when his work loads were high).
- **Margaret Fawcett -**
For allowing the Fawcett home to become a research centre.
- **Past and present Members of the COMMUNAL team -**
I am very grateful for the help of the COMMUNAL team both past and present. Particularly Gordon Tucker, who provided much of the inspiration along with Robin Fawcett in the creation of ICQF. To past COMMUNAL members for the many stimulating discussions in past years, in particular: Ruvan Weerasinghe, Yuen Lin, Amy Neale, Fiona Ball, and Victor Castel.
- **Ruvan Weerasinghe and Clive Souter -**
For their pioneering research in SFG parsing. Their findings strongly influenced the direction of this work.
- **My professional colleagues -**
Thanks must go to Rolls-Royce for providing the support and release to be able to embark on this research.
- **Victoria Ichizli-Bartels -**
For her reading of the chapters and constructive comments.
- **My family -**
My parents. Patricia for starting me on this path in 1991. Matthew, especially for his understanding during the latter years.

Glossary

abstract mark up	Mark up method that uses generic identifiers that have general names with attributes to qualify the meaning (eg <code><unit class="ngp"/></code>).
AECMA	See ASD.
AMALGAM	Automatic Mapping Among Lexico-Grammatical Annotation Models
ARK Corpus	A corpus that was automatically generated by GENESYS, and was used as one of the sources of probabilities for Weerasinghe's POP.
ASD	European Association of Aerospace Manufacturers (formerly AECMA)
ATN	Augmented Transition Network
attribute	A mark up attribute, further qualifies a mark up element
BNC	British National Corpus
BRILL tagger	A part of speech tagger developed by Eric Brill
BUS	Backward Unit Structure (a probabilistic table /query used by the CCPP)
backtracking	In parsing terms, this occurs when the parser has to go back after a failed analysis
built structure	A (maybe partial) parse tree that represents what the CCPP has built at a particular point of the parse
business rule (BR)	A rule that states how an SGML/XML element or attribute should be used in a given context
business rule checker	A program that checks that business rules are properly applied in an SGML/XML document
C / C++	Programming language
CALS	Continuous Acquisition and Life-cycle Support (formerly Computer Aided Logistic Support). A United States Department of Defense initiative for electronically capturing military documentation.
candidate structure	A partial parse tree that the CCPP is attempting to join or grow at a particular point of the parse
CCPP	Corpus-Consulting Probabilistic Parser
CCXP	Corpus-Consulting Probabilistic XML Parser
CELEX	The Centre for Lexical Information (resource used by Souter 1996)
CLAWS	Constituent Likelihood Automatic Wordtagging System
CMS	Content Management System - see native XML database
CNF	Chomsky Normal Form
COBUILD	Collins Birmingham University International Language Database (corpus)
COMMUNAL	Convivial Man-Machine Understanding through NATural Language - a project based at the University of Wales, College of Cardiff
computational backtracking	Backtracking that involves going back to the next untried path in sequence
concordance	A report format derived from a corpus that shows word usage in the context of the words around it (see KWIC)
concordancer	A program that produces a concordance report

co-ordination	SFG relationship: two or more units that fill an element are said to be co-ordinated
corpus	large body of text (plural corpora)
corpus index tables	The indexes to the corpus tables
corpus query tool	A program that can be used to extract information from a corpus
corpus tables	The native XML tables used to store the corpus in this project
corpusDB	The Corpus Database (part of this project)
corpus database	See corpusDB
corpusSearch	A corpus query tool
CQL	Corpus Query language (used in ICQF)
createdDTD	A program that creates a DTD given a batch of SGML/XML instances
CVS	'Correct' Vertical Strip (part of the XML vertical strip parser scoring algorithm)
DARPA	The Defense Advanced Research Projects Agency. Funded work on the SWITCHBOARD corpus.
DBMS	Database Management System
DCG	Definite Clause Grammar
descriptive mark up	Mark-up method that uses describes itself. Generic identifiers have meaningful names (eg <ngp>).
discontinuity	A unit is discontinuous when its elements are interrupted in their position by other units.
DITA	Darwin Information Typing Architecture
DOM	Document Object Model - programming methods for accessing XML objects
DTD	An SGML or XML Document Type Definition which, for example, determines what mark up elements can exist and in what order
dynamic (parsed) corpus	A corpus that grows. The probabilities of syntax relationships changes as new parsed sentences are added.
E2U	Element up to Unit (a probabilistic table used by the CCPP)
element	A mark up element, part of an SGML or XML document
element	(of structure) - a Cardiff Grammar category
ellipsis	Elements of structure may be ellipped; that is, they are present in their unit but are not expounded by a lexical item or filled by a unit.
ENGCG	English Constraint Grammar
ENGTWOL	English morphological analyser
exponence	SFG relationship: an item is said to expound an element
filling	SFG relationship: a unit is said to fill an element
FPD	Fawcett-Perkins-Day Corpus
FUF	Functional Unification Formalism
FUG	Functional Unification Grammar
FUS	Forward Unit Structure (a probabilistic table / query used by the CCPP)
GEIG	Grammar Evaluation Interest Group
GENESYS	GENERates SYStematically - the natural language generator component of

	COMMUNAL
GI	Generic Identifier (name of a mark up tag)
GPSG	Generalised Phrase Structure Grammar
I2E	Item up to Element (a probabilistic table / query used by the CCPP)
I2E2U2E	Item up to Element up to Unit up to Element (a probabilistic table / query used by the CCPP)
ICE	International Corpus of English
ICECUP	The ICE Corpus Utility Program (a corpus query tool)
ICE-GB	British component of the ICE corpus
ICQF	Interactive Corpus Query Facility - a corpus query tool
ICQF+	The latest version of ICQF
ID/IDREF	Cross referencing method in SGML / XML
ID/LP	Immediate Dominance (ID) rules and Linear Precedence (LP) rules
IDAS	Intelligent Documentation Advisory System (an NLG application - see Reiter, Mellish and Levine (1995))
ILEX	Intelligent Label Explorer (an NLG application - see O'Donnell (1996).
instance	An SGML document or an XML document that is said to be an instance of a document type.
ISO	International Standards Organisation
item	A Cardiff Grammar category. An item may be a lexical ('vocabulary') item, such as a noun, or a grammatical item such as an article or part of the verb be .
IVS	Initial Vertical Strip
IVS-ELEM	Initial Vertical Strip without item (a probabilistic table used by the CCPP)
IVS-ITEM	Initial Vertical Strip with Item (a probabilistic table used by the CCPP)
Java	Programming language
KWIC	KeyWord In Context
linguistically motivated backtracking	Backtracking that involves going back to a point in the search-space that is determined by linguistic reasoning (e.g. knowledge of a failed attachment option)
LMS	Left-most strip (vertical strip)
LOB	The Lancaster-Oslo/Bergen Corpus
MARKUPINDEX	The main corpus index table (that indexes mark up data)
MDC	Mark up Declaration Closed (normally '>')
MDO	Mark up Declaration Open (normally '<')
metadata	Data about data
MIT	Massachusetts Institute of Technology
MSXML	Microsoft's XML parser
native XML database	An XML 'aware' database management system which stores XML data in terms of elements, attributes and PCDATA etc
NEGRA	A corpus of German newspaper texts
NLG	Natural Language Generation

NLU	Natural Language Understanding
NLP	Natural Language Processing
parsed corpus	A corpus that is fully syntactically analysed
parser	A program that checks SGML or XML is valid
parser	A natural language parser
Parser WorkBench	The development environment of the Corpus-Consulting Probabilistic Parser
parser workflow	The states through which groups of built and candidate structures flow during the parsing of a sentence in the CCPP
parser working tables	The part of the corpus database where the parser's working data is stored
PARSEVAL	The parser evaluation project
PARSIFAL	Marcus (1980) deterministic parser
Part of speech tagger	See word tagger
PCDATA	Parsed Character Data - text that can appear as content of mark up tags
PDF	Portable Document Format - page-oriented format used in Adobe Acrobat
PG1.5 Corpus	COMMUNAL Prototype Grammar version 1.5 Corpus (used in O'Donoghue's VSP)
POP	Probabilistic Online Parser - Weerasinghe's SFG probabilistic chart parser (Weerasinghe 1994)
POS	Part Of Speech
POW	Polytechnic of Wales Corpus
PR	Participant Role
PROLOG	(a) a logic programming language commonly used in artificial intelligence applications; (b) the front part of an SGML / XML document that declares (for example) the DTD or the schema.
prescriptive mark up	Mark up method that describes, for example, a book as containing chapters and sections, rather than describing the data contained within
probabilities tables	The tables used by the parser in this project
PROLOG	Declarative programming language
PVS	Parser Vertical Strip (part of the XML vertical strip parser scoring algorithm)
RAP	Realistic Annealing Parser (Atwell et al 1988b)
PSG	Phrase Structure Grammar
PST	Parser State Table - one of the parser working tables
raw corpus	A corpus that contains only text, and no part-of-speech or other syntactic analysis
REVELATION	COMMUNAL's semantic interpreter (O'Donoghue 1991b)
rewrite rule	A grammar rule typically used in rules-based parsers. They take the symbol on the left-hand side and rewrite it by the symbols on the right-hand side
RMS	Right-most strip (vertical strip)
RTN	Recursive Transition Network
RULEINDEX	Old name for the main corpus index table used in this project
S1000D	Specification for Technical Publications Utilizing a Common Source Database

SARA	SGML Aware Retrieval Application - a corpus query tool
Schema	(a) the structure of a database in terms of its tables and fields and the relationships between them; (b) an XML schema (see XML schema).
SFG	Systemic Functional Grammar
SFL	Systemic Functional Linguistics
SGML	Standard Generalized Mark-up Language
SHRDLU	A natural language understanding system developed by Winograd (1972)
SQL	Structured Query Language
SSN	Simple Synchrony Networks
Star Parser	A prototype parser developed in this project, so called because of its Star-shaped data structure.
STM	State Transition Machine
stylesheet	The rules for how an XML document should be formatted or converted
SUSANNE	Surface and Underlying Structural ANalyses of Naturalistic English (corpus)
SWITCHBOARD	Spoken corpus containing telephone conversations
tag	An SGML or XML tag (eg <para>)
TAG	Tree Adjoining Grammar
tagged corpus	A corpus that contains part-of-speech analysis - usually annotated automatically using a word tagger
TEI	Text Encoding Initiative
TGREP	A corpus query tool for extracting parse trees from a corpus
TIGERSearch	A corpus query tool developed with the NEGRA corpus
TigerXML	An XML Schema used in the NEGRA corpus and the TigerSearch application.
TOSCA tagger	A part-of-speech tagger developed by the TOSCA Research Group at the University of Nijmegen and used in the ICE project
treebank	term given a large collection of parsed trees
U2E	Unit up to Element (a probabilistic table used by the CCPP)
UCREL	University Centre for Computer Corpus Research on Language (based at Lancaster University) - responsible for the development of CLAWS
unit	A Cardiff Grammar category
VB	Visual BASIC (programming language)
VIQTORIA	a VItual Query Tool fOR sYntactically Annotated corpora (corpus query tool)
VSG	Vertical Strip Grammar (O'Donoghue 1991a)
VSN	Vertical Strip Network (O'Donoghue 1991a)
VSP	Vertical Strip Parser (O'Donoghue 1991a)
W3C	World Wide Web Consortium
WAG	Workbench for Analysis and Generation (O'Donnell 1993, 1994)
word tagger	A program that determines parts of speech for words in a sentence normally through the investigation of the words that surround it (an example is CLAWS)

XARA	Corpus query tool - the XML version of SARA (now called Xaira)
Xaira	New name for XARA (renamed due to the fact that another unrelated software has the name XARA)
XML	eXtensible Mark-up Language
XML Schema	An XML file that provides the same function as a DTD but with many enhancements
Xquery	XML Query Language
XSL	XML Stylesheet Language
XSL-FO	XSL- Formatting Objects
XSL-T	XSL Transformations
XTAG	A project responsible for TAG

List of parts and chapters

CHAPTER ONE: Introduction	1
PART ONE: THE CONTEXT OF THIS WORK	6
CHAPTER TWO: Computational linguistics	7
CHAPTER THREE: Corpus linguistics	25
CHAPTER FOUR: The syntax of the Cardiff Grammar	45
PART TWO: THE CORPUS DATABASE	60
CHAPTER FIVE: Towards a corpus annotation scheme and a method of representing syntactic relationships	62
CHAPTER SIX: Marking up language texts	74
CHAPTER SEVEN: Defining the corpus database schema	99
CHAPTER EIGHT: Enhancing the Interactive Corpus Query Facility	114
CHAPTER NINE: Updating the corpus	137
PART THREE: PARSING NATURAL LANGUAGE: SOME RELEVANT ANTECEDENTS	151
CHAPTER TEN: Concepts used in parsing - a selective history	152
CHAPTER ELEVEN: Earlier parsers that use Systemic Functional Grammar	176
CHAPTER TWELVE: Early attempts at parsing in this project	194
PART FOUR: THE NEW PARSER	214
CHAPTER THIRTEEN: Introducing the Corpus Consulting Probabilistic Parser	215
CHAPTER FOURTEEN: The parser's probabilities tables, working tables and data structures	226
CHAPTER FIFTEEN: The parsing algorithm	230
CHAPTER SIXTEEN: The parser WorkBench	262
CHAPTER SEVENTEEN: Testing and evaluation	282
CHAPTER EIGHTEEN: Further work and conclusions	323
BIBLIOGRAPHY	337
APPENDICES	346

Table of contents

Abstract	iii
Acknowledgements	iv
Glossary	v
CHAPTER ONE: Introduction	1
1.1 Preamble.....	1
1.2 The aims of the work.....	2
1.3 This work in its context.....	3
1.4 How this thesis is organised.....	4
PART ONE: THE CONTEXT OF THIS WORK	6
CHAPTER TWO: Computational linguistics	7
2.1 The field of computational linguistics.....	7
2.2 Linguistic theories	8
2.2.1 Some formal theories of language	8
2.2.2 Some description based theories of language	9
2.2.3 Formal and descriptive theories in the context of parsing	9
2.3 Some applications of linguistic theory in computational linguistics	10
2.3.1 Theories used in natural language understanding applications.....	10
2.3.2 Theories used in natural language generation applications	11
2.4 An overview of the Cardiff Grammar	12
2.4.1 Background	12
2.4.2 The place of syntax in a systemic functional grammar.....	13
2.5 An introduction to the COMMUNAL Project.....	15
2.5.1 The natural language generation components in COMMUNAL.....	15
2.5.2 The natural language understanding components in COMMUNAL	22
2.6 Summary	24
CHAPTER THREE: Corpus linguistics	25
3.1 Definitions and classifications.....	26
3.1.1 What is a corpus?.....	26
3.1.2 Classifying corpora.....	26
3.2 A survey of some of the major corpora	28
3.2.1 COBUILD	28
3.2.2 The British National Corpus (BNC)	29
3.2.3 The Penn Treebank.....	29
3.2.4 The SUSANNE Corpus	30
3.2.5 The Lancaster-Oslo/Bergen (LOB) Corpus	30
3.2.6 The International Corpus of English (ICE).....	31
3.2.7 The SWITCHBOARD Corpus	31
3.2.8 The NEGRA Corpus.....	31
3.2.9 The Fawcett-Perkins-Day Corpus.....	32
3.3 Part-of-Speech taggers	37
3.4 Corpus query tools	37
3.4.1 Products relating primarily to items	38
3.4.1.1 Word lists	38
3.4.1.2 Concordances and collocations	39
3.4.2 Products relating to syntax	41
3.4.2.1 Unit-up-to-element tables (U2E).....	41
3.4.2.2 Element-up-to-unit tables (E2U)	42
3.4.3 Tools for querying part-of-speech tagged corpora.....	42
3.4.4 Tools for querying parsed corpora.....	42
3.5 Summary	44

CHAPTER FOUR: The syntax of the Cardiff Grammar	45
4.1 Definitions	45
4.1.1 Units, elements and items	45
4.1.2 The sentence element	45
4.1.3 Relationships	46
4.1.4 Unfinished units	46
4.1.5 Replacement elements	46
4.1.6 Formalisms used in diagrams of syntax	46
4.2 The syntactic units of the Cardiff Grammar	48
4.2.1 The Clause (Cl)	49
4.2.1.1 Definition	49
4.2.1.2 Conflated elements in a clause	49
4.2.1.3 Participant Roles in the clause	50
4.2.2 The nominal group (ngp)	50
4.2.3 The prepositional group (pgp)	53
4.2.4 The quality group (qlgp)	53
4.2.5 The quantity group (qtgp)	55
4.2.6 The genitive cluster (genclr)	56
4.2.7 Text	57
4.3 Places, potential structures and discontinuity	57
4.4 More on ellipsis	58
4.5 Summary	59
PART TWO: THE CORPUS DATABASE	60
CHAPTER FIVE: Towards a corpus annotation scheme and a method of representing syntactic relationships	62
5.1 Requirements	62
5.2 Traditional methods of representing syntactic relationships	64
5.2.1 Phrase Structure Grammar style rewrite rules	64
5.2.1.1 Definition	64
5.2.1.2 Use in SFG and the suitability for this project	64
5.2.2 Transition networks	66
5.2.2.1 Definition	66
5.2.2.2 The use in SFG and the suitability for this project	66
5.2.3 Tree Adjoining Grammar (TAG)	68
5.2.3.1 Definition	68
5.2.3.2 The use in SFG and the suitability for this project	69
5.2.4 Vertical Strip Grammar (VSG)	70
5.2.4.1 Definition	70
5.2.4.2 The use in SFG and the suitability for this project	71
5.2.5 Definite Clause Grammar (DCG)	71
5.2.5.1 Definition	71
5.2.5.2 The use in SFG and the suitability for this project	71
5.2.6 The decisions made for this project	72
5.3 Summary: Representing the corpus text and syntax structures	73
CHAPTER SIX: Marking up language texts	74
6.1 The use of mark up languages in this project	74
6.1.1 The requirements for a corpus annotation scheme	74
6.1.2 The background of mark up languages, and their use in this project	75
6.1.3 Other benefits of using mark up languages	76
6.1.4 Towards a mark up scheme	77
6.1.4.1 Definitions	77
6.1.4.1.1 Prescriptive mark up	77
6.1.4.1.2 Descriptive mark up	77
6.1.4.1.3 Abstract mark up	78
6.1.4.2 An abstract mark up scheme for the Cardiff Grammar	78
6.1.4.3 Evaluating the abstract model	80

6.1.5 Using a DTD to specify a rule-based grammar	81
6.1.5.1 Creating a DTD manually from a linguist's model of language.....	81
6.1.5.2 The chosen mark up scheme.....	81
6.1.5.3 An SGML Parser as a natural language parser	83
6.1.5.4 The automatically created DTD	84
6.1.6 Satisfying the design criteria for Systemic Functional Grammar	85
6.1.6.1 Handling discontinuous units	87
6.1.6.2 Handling ellipted elements	87
6.1.6.3 Handling questionable (or unknown) units, elements and items	88
6.1.7 Using the mark up scheme to represent syntactic relationships.....	89
6.2 Other natural language projects that are use mark up languages.....	89
6.2.1 The Text Encoding Initiative (TEI)	89
6.2.2 Other non-TEI projects that use mark up languages.....	92
6.2.3 Comparing these mark up schemes	94
6.2.4 What disadvantages are there with our chosen scheme?	94
6.3 Summary	98
CHAPTER SEVEN: Defining the corpus database schema.....	99
7.1 Mark up and databases - surveying the fields.....	101
7.1.1 Applications in Industry	101
7.1.1.1 The data model approach	101
7.1.1.2 The native approach	101
7.1.2 Applications in research	104
7.1.3 Choices made for this project.....	105
7.2 The database schema: the corpus and the corpus index tables	106
7.2.1 The corpus tables.....	106
7.2.2 The corpus index tables	109
7.3 Summary	112
CHAPTER EIGHT: Enhancing the Interactive Corpus Query Facility.....	114
8.1 History.....	114
8.2 Towards ICQF+	115
8.3 Querying in ICQF+ - the Corpus Query Language	116
8.3.1 Queries about items	116
8.3.2 Queries about syntax	119
8.3.2.1 Standard queries about syntax	119
8.3.2.2 Wildcards	119
8.3.2.3 Advanced queries about syntax.....	120
8.3.3 Finding Sentences by their DOCUMENT ID or POW CELL.....	121
8.3.4 Restricting queries	121
8.3.4.1 Restricting queries by selecting parts of the sentence's cell identifier.....	122
8.3.4.2 Restricting queries by initial in sentence and unit	122
8.3.4.3 Restricting queries by ignoring ellipsis	123
8.4 Using ICQF+	123
8.4.1 Retrieving the results of a query.....	123
8.4.1.1 Displaying sentences	124
8.4.1.2 Navigation	124
8.4.1.3 Editing and deleting sentences	124
8.4.2 The concordancer	128
8.4.3 The reports.....	132
8.5 Corpus modification programs.....	135
8.6 Summary	135
CHAPTER NINE: Updating the corpus.....	137
9.1 Why the changes are needed	138
9.2 The overall strategy.....	139
9.3 Stage one: how the changes were made	139
9.3.1 Details of the change process	140
9.3.1.1 Automatic changes	140
9.3.1.2 Manual changes.....	140

9.3.1.3 Errors discovered in the POW Corpus	141
9.3.2 Stage One: example changes	141
9.3.2.1 A simple change of a mark up element	142
9.3.2.2 A simple change of a mark up element and its parent	143
9.3.2.3 A complex example with mark up element insertion	144
9.3.2.4 Example showing a more complex query	147
9.4 Stage Two	149
9.5 Summary	149

PART THREE: PARSING NATURAL LANGUAGE: SOME RELEVANT ANTECEDENTS..... 151

CHAPTER TEN: Concepts used in parsing - a selective history	152
10.1 The goals of parsing	152
10.2 Some early classifications used in parsing	154
10.2.1 Mode of operation	154
10.2.1.1 Top-down	154
10.2.1.2 Bottom-up	155
10.2.1.3 Mixed-mode	155
10.2.2 Search strategies	155
10.2.2.1 Breadth-first versus depth-first	155
10.2.2.2 Best-first and beam search (n-best)	155
10.2.3 Deterministic and non-deterministic parsing	156
10.2.4 Backtracking	158
10.2.5 Incremental parsing (on-line parsing)	158
10.2.6 Classifying the parser described in Part Four	159
10.3 Parsing algorithms	160
10.3.1 Some common algorithms	160
10.3.1.1 Augmented Transition Networks (ATNs) and Recursive Transition Networks (RTNs)	160
10.3.1.2 Truly deterministic parsing	160
10.3.1.3 Shift-reduce parsing	161
10.3.1.4 Definite Clause Grammar (DCG) parsers	162
10.3.1.5 Chart parsers	162
10.3.1.6 CYK parsers	165
10.3.2 Other relative parsing algorithms	166
10.3.2.1 Vertical strips parsing	166
10.3.3 Concepts used in the current work	166
10.4 Other techniques	169
10.4.1 Left-corner parsing	169
10.4.2 Head-driven parsing	169
10.4.3 Shallow parsing	169
10.4.4 Pre-tagging	170
10.4.5 Word lattice parsing and neural networks	170
10.4.6 The relevance of these techniques to the present work	171
10.5 Probabilistic and statistical approaches	171
10.6 Corpus-based approaches	172
10.7 Applying the algorithms to Systemic Functional Grammar	173
10.8 Summary	174

CHAPTER ELEVEN: Earlier parsers that use Systemic Functional Grammar	176
11.1 Winograd's SHRDLU	176
11.2 Earlier work at Leeds University	177
11.3 Kasper's NIGEL-based parser	178
11.4 O'Donnell's work	178
11.5 O'Donoghue's Vertical Strips Parser	179
11.5.1 The data structures	179
11.5.2 The algorithm	180
11.5.3 Evaluation	181

11.6 Weerasinghe's Probabilistic On-line Parser (POP).....	184
11.6.1 The algorithm	184
11.6.2 The probabilistic scoring of edges.....	185
11.6.3 Modifications to the standard chart parsing algorithm	186
11.6.4 Evaluation.....	186
11.7 Souter's Corpus-Trained Parser.....	187
11.7.1 The algorithm	187
11.7.2 The probabilistic scoring of edges.....	188
11.7.3 Modifications to the standard chart parsing algorithm	188
11.7.4 Evaluation.....	191
11.8 Summary	192

CHAPTER TWELVE: Early attempts at parsing in this project..... 194

12.1 Experiments with a chart parser	194
12.1.1 Background	195
12.1.2 Implementing a chart parser	195
12.1.2.1 The chart data structure	195
12.1.2.2 The chart parsing algorithm.....	198
12.1.2.3 Corpus queries.....	199
12.1.2.4 The probabilistic scoring of edges.....	201
12.1.2.5 The value of the database-oriented approach	201
12.1.2.6 The value of expressing the results in XML.....	201
12.1.3 A comparison with the work of Weerasinghe (1994) and Souter (1996)	202
12.1.3.1 The model of syntax and the lexicon.....	202
12.1.3.2 The chart data structure	203
12.1.3.3 Methods used to improve the efficiency of the parser.....	204
12.1.3.4 Probabilistic scoring.....	206
12.1.3.5 Comparing the programming methods.....	207
12.1.4 Evaluating the chart parser reported here	207
12.2 Beyond a chart parser	209
12.2.1 The Star Parser	209
12.2.1.1 The Star data structure.....	209
12.2.1.2 The Star parsing algorithm.....	210
12.2.1.3 Evaluating the Star Parser	211
12.3 Summary: Towards the Corpus-Consulting Probabilistic Parser	212

PART FOUR: THE NEW PARSER..... 214

CHAPTER THIRTEEN: Introducing the Corpus

Consulting Probabilistic Parser.....	215
13.1 Aims	216
13.2 The better model: a partial simulation of the human parsing process	218
13.2.1 The deterministic approach	218
13.2.2 The corpus-based approach	220
13.2.3 The probabilistic approach	220
13.2.4 A comparison with the traditional ways of classifying a parser	220
13.3 The overall method of research	221
13.4 The relationship of the implemented research reported here and the work of future phases of the project.....	221
13.5 Backtracking	222
13.6 Introducing the new parsing process	223
13.7 Summary	225

CHAPTER FOURTEEN: The parser's probabilities tables,

working tables and data structures.....	226
14.1 The probabilities tables and their queries: Version One.....	226
14.2 The development of the Version Two tables.....	228
14.3 The parser data structures.....	229
14.4 Summary	229

CHAPTER FIFTEEN: The parsing algorithm	230
15.1 The basic concepts defined and illustrated	230
15.1.1 Candidate structure trees and built structure trees	230
15.1.2 Tree operations and tree pairs	232
15.1.3 Start and end positions, active and inactive trees	234
15.1.4 The workflow of the parsing process	234
15.1.5 The parser state table	238
15.1.6 The n-best approach	242
15.1.7 Node and level probabilities	243
15.2 A full specification of the algorithm	243
15.2.1 Stage 0: Initialising the parser	243
15.2.2 Stage 1: Parsing the initial item	244
15.2.3 Stage 2: Building a candidate structure	246
15.2.4 Stage 3: Attempting to join a candidate structure to a built structure (joining two sibling elements in a unit)	247
15.2.5 Stage 4: Growing the candidate structure	255
15.2.6 Stage 5: Joining by the co-ordination of units	256
15.2.6.1 An overview	256
15.2.6.2 Stage 5: Calculating the co-ordination joining score	256
15.2.6.3 Stage 5a: Making the co-ordination Joins	258
15.2.6.4 The join cycle	258
15.2.7 Stage 6: Backtracking	259
15.2.8 Stage 7: Determine if more analyses are required	260
15.3 The Parser WorkBench	261
15.4 Summary	261
CHAPTER SIXTEEN: The Parser WorkBench	262
16.1 Step-by-step incremental parsing	262
16.2 The value of the step-by-step incremental approach	263
16.2.1 How it works	263
16.2.2 Its use in the development of the new parser	263
16.2.3 Speeding-up research	263
16.2.3.1 Changing a sentence in mid-parse	263
16.2.3.2 Altering probabilities and joining scores	264
16.2.3.3 Altering the configurable parameters in mid-parse	265
16.2.4 Answering 'what-if' questions	265
16.3 The configurable parameters	265
16.3.1 The number of trees to create in Stage 1 (Figure 16.1, Item 1)	265
16.3.2 The value of n for best-n joins (Figure 16.1, Item 2)	266
16.3.3 The value of n for best-n to grow (Figure 16.1, Item 3)	266
16.3.4 The number of cycles of Stages 3, 4, 3 and 5 to allow before backtracking	266
16.3.5 The threshold probability for accepting / rejecting join attempts (Figure 16.1, Item 6)	266
16.3.6 The unit-join probabilities (Figure 16.1, Item 7)	267
16.3.7 The join score formula parameters to be used (Figure 16.1, Item 1)	267
16.4 The user interface	267
16.4.1 The configurable parameters	267
16.4.2 Starting a step-by step incremental parse	269
16.4.2.1 Stage 0	269
16.4.2.2 Stage 1	271
16.4.2.3 Stage 2	274
16.4.2.4 Stage 3: Joining a candidate structure to a built structure	275
16.5 The commit button	279
16.6 Summary	280
CHAPTER SEVENTEEN: Testing and evaluation	282
17.1 Testing the parser	282
17.1.1 Establishing the optimum configurable parameter settings	282
17.1.1.1 Establishing an optimum value of n in n-best	284
17.1.1.2 Establishing the best configuration of the joining score formula parameters	287

17.1.1.3	Establishing the best join threshold value	292
17.1.1.4	Performing the basic tests.....	294
17.1.1.5	Testing the parser's ability to handle co-ordination	296
17.1.1.5.1	Calculating the co-ordination joining score.....	296
17.1.1.5.2	Determining the value of n in n-best co-ordinated joins to take forward	297
17.1.1.5.3	The results of the co-ordination tests.....	297
17.1.2	Tests for special cases.....	299
17.1.2.1	Tests for a sentence involving attachment ambiguity.....	299
17.1.2.2	Tests using a sentence that required backtracking.....	300
17.1.3	Extensive testing for measuring the accuracy, efficiency and speed of the parser	301
17.1.3.1	Measuring the accuracy of the parser.....	302
17.1.3.1.1	Current approaches to evaluating parser outputs.....	302
17.1.3.1.2	Selecting a set of test sentences.....	302
17.1.3.1.3	Defining the methods used for scoring parses.....	303
17.1.3.1.3.1	XML Differencing.....	304
17.1.3.1.3.2	The XML vertical strip scoring algorithm.....	305
17.1.3.2	Summary of the results of the tests using the FPD test set.....	309
17.1.3.2.1	Tests for the accuracy of the output from the parser	310
17.1.3.2.2	Tests for the efficiency of the parser	310
17.1.3.2.3	Tests for the speed of the parser	312
17.1.3.2.4	Opportunities for improvement	314
17.2	An evaluation against the project's goals.....	315
17.2.1	How far is the output from a richly annotated syntax diagram.....	315
17.2.2	How far is the parser deterministic?	318
17.2.3	Does the corpus-consulting, database-oriented approach lead to better parsing?.....	319
17.2.4	Improvements in the speed, efficiency and accuracy of the parser.....	319
17.3	The results of the evaluation	320
17.4	Summary	321
CHAPTER EIGHTEEN: Further work and conclusions		323
18.1	The Corpus Database and ICQF+: improvements and further work	323
18.1.1	The native XML tables	323
18.1.2	ICQF+.....	324
18.1.2.1	A graphical query builder for ICQF+.....	324
18.1.2.2	XML tree grapher.....	325
18.1.2.3	Controlling access to the corpus editor.....	325
18.1.2.4	Multi-user and web-enabled support	325
18.1.3	Using the Text Encoding Initiative (TEI) standard.....	325
18.1.3.1	Export in today's TEI format	326
18.1.3.2	Import from today's TEI format	326
18.1.3.3	Extending the TEI model to provide a more usable format.....	326
18.1.4	Summary	327
18.2	The parser: improvements and further work	327
18.2.1	Improvements for Phase Two and Phase Three.....	327
18.2.2	Testing and refining the Version Two probabilities tables	327
18.2.3	Improved item recognition: detecting multi-word items etc.....	328
18.2.4	Improved punctuation treatment.....	328
18.2.5	Intelligent tree growing	329
18.2.6	Linguistically motivated backtracking	330
18.2.7	Participant roles	331
18.2.8	Making use of morphological information	331
18.2.9	Units that handle names, dates, times etc.	332
18.2.10	Other improvements	332
18.2.10.1	Web-enabled parsing.....	332
18.2.10.2	The recognition of unknown items	332
18.2.10.3	The use of semantic features	333
18.2.10.4	Modelling the dynamic properties of language	334
18.3	Conclusions	334
18.4	The final word	336

BIBLIOGRAPHY	337
APPENDIX A: The Cardiff Grammar	346
A.1 The Cardiff Grammar Units	346
A.2 The Clause (C1)	346
A.3 The nominal group (ngp).....	347
A.4 The prepositional group (pgp).....	347
A.5 The quality group (qlgp)	347
A.6 The quantity group (qtgp)	348
A.7 The genitive cluster (genclr)	348
A.8 Elements that can occur in any class of unit.....	348
APPENDIX B: An alphabetical list of the Cardiff Grammar units and elements	349
APPENDIX C: Marking up language texts	352
C.1 Defining Mark up	352
C.1.1 Definitions	352
C.1.2 Document Type Definitions and Schemas	353
C.1.3 Document Type Definitions (DTDs)	354
C.1.4 Using mark up languages for non-hierarchical structures.....	355
C.2 Artefacts from the experiments with mark up languages	356
C.2.1 The nominal group.....	356
C.2.2 The quality group.....	357
C.2.3 The quantity group.....	357
C.2.4 The prepositional group	358
C.2.5 The genitive cluster.....	358
C.2.6 The text unit	358
C.2.7 Sentence	359
C.2.8 The Clause	359
C.3 Summary.....	360
APPENDIX D: The database schema	361
D.1 The corpus tables.....	361
D.2 The corpus index tables	361
D.3 The probabilities tables	362
D.4 The parser working tables	362
APPENDIX E: Example ICQF+ reports	370
APPENDIX F: Modifying the corpus: details of the changes made in Stage One	373
APPENDIX G: Modifying the corpus - a table of syntax token mappings used in Stage Two	382
APPENDIX H: The probabilities tables, parser working tables and data structures	385
H.1 The probabilities tables and their queries	385
H.1.1 Types of table and their queries.....	385
H.1.2 Queries to the initial vertical strip tables	387
H.1.2.1 Rationale in creating the tables.....	387
H.1.2.2 The structure of the tables	388
H.1.2.3 Criteria for including items in the Initial Vertical Strip Item table.....	389
H.1.3 Item-up-to-element (I2E) and item-up-to-element-up-to-Unit-up-to-Element (I2E2U2E) tables	389
H.1.4 Queries to the element-up-to-unit (E2U) table	391

H.1.5 Queries to the unit-up-to-element (U2E) table	392
H.1.6 An overview of the unit structure (US) tables and their queries.....	392
H.1.6.1 Queries to the forward unit structure tables (FUS).....	392
H.1.6.2 Queries to the backward unit structure table (BUS).....	393
H.2 The parser data structures	394
H.2.1 Trees and Nodes	394
H.2.2 Forward and backward predictions.....	395
H.2.3 The initial vertical strip table	396
H.2.4 The parser state table	396
H.3 Tree functions.....	397
H.4 Summary	398
APPENDIX I: The Version Two probabilities tables	399
I.1 The development of the Version Two tables	399
I.1.1 Weaknesses in the Version One tables.....	399
I.1.2 How the Version Two tables were developed.....	402
I.2 Summary.....	404
APPENDIX J: The British National Corpus tag set and its mapping to Cardiff Grammar elements	405
APPENDIX K: Parser walkthrough	408
K.1 The test sentences.....	408
APPENDIX L: Ideas for linguistically motivated backtracking.....	416
L.1 Linguistically motivated backtracking.....	416
L.2 Summary.....	418
APPENDIX M: Parser - testing and evaluation using the FPD corpus test set	419

Chapter One

Introduction

1.1 Preamble

The process of transforming sentences into rich syntactic structures is called natural language parsing, and despite over half a century of research, this continues to present a major challenge to the field of computational linguistics. This is especially so when the goal of the parsing process is raised beyond that of producing a minimal, purely formal syntactic structure, to producing one that is rich with functional information that will facilitate the next stage of Natural Language Understanding (NLU).

In the last couple of decades, researchers have begun to exploit the statistical information that can be determined from large bodies of texts (called corpora) as they become increasingly available in a computerised form, and some of the most successful of the current approaches to natural language parsing are those that arrive at their analyses by using **probabilities** that are derived from such corpora.

Raw, unanalysed corpora are now widely used in the field of corpus linguistics, i.e. as an aid in the study of languages. Some corpora contain part-of-speech information that has been supplied by **word taggers**, in order to enhance the results that are gained from tools that can exploit this data. Although the number of fully analysed corpora is steadily increasing, their number unfortunately remains low, due to the substantial effort involved in creating them.

Probabilistic corpus-based parsers are parsers that complement the standard probabilistic approach by the use of information about the syntactic structures that have been derived from corpora. Some probabilistic parsers have given promising results. This present study is intended to constitute a contribution to this field of research.

Language is not static. It changes with time. New words and new meanings of existing words are continuously being created. The frequency of use of some words also changes with time. Certain words become more frequently used while other words may become less frequent. In this project, we introduce the concept of a **living dynamic corpus**. In such an approach, the system's knowledge of the frequencies of given syntactic phenomena changes as new sentences are parsed and added to the

corpus. In this way, the model changes so that it reflects the fact that language is constantly changing.

This work describes the implementation of this idea and it is intended as a new offering to the fields of corpus linguistics and the natural language parsing.

1.2 The aims of the work

Because of its functional nature, the parsing of Systemic Functional Grammar has proved to be more problematical than the parsing of less richly annotated formalisms. In a probabilistic parser the task is made even more difficult when these are extracted from naturally occurring texts of a corpus rather than being manually created.

In the research reported here, our primary aim is to prove that it is possible to parse texts successfully in terms of Systemic Functional Grammar, by (a) using probabilistic data that is automatically drawn from a dynamic corpus database in which the probabilities may be modified as new sentences are satisfactorily parsed, and (b) introducing knowledge of functional syntax into the parsing algorithm. This thesis will describe such an approach. I will show that a parser can be built which is **database-oriented** - i.e. it uses a relational database to (a) retrieve its knowledge of syntactic relationships to be used in a parse, and (b) store its working data. I will show that this provides a good environment for developing a parser.

To achieve these goals, we need a suitable corpus and a research tool that can assist in creating the linguistic knowledge that we will build into the parser's probabilistic model. I will show how mark-up languages can be used for the corpus annotation of a rich functional syntax, and how such corpus data can be stored in a relational database. I shall further show that this provides an environment that allows the efficient modification of a corpus (in order to update it to another annotation scheme, or to modify the analyses that it contains), and that it provides an environment in which a parser can operate.

Figure 1.1 shows how the new parser operates within this environment.

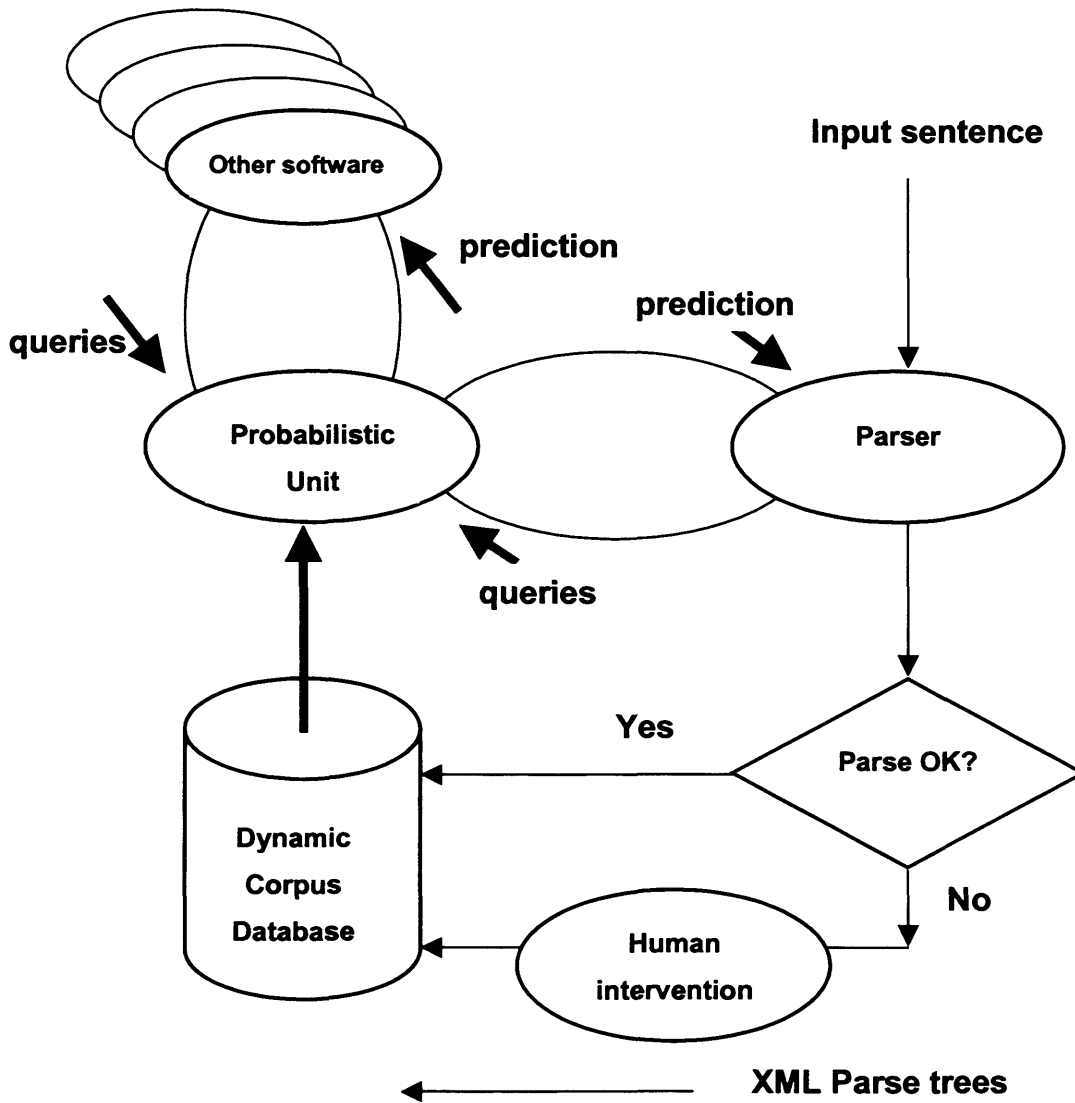


Figure 1.1: An outline of our environment

1.3 This work in its context

The context of the research reported in this thesis is the COMMUNAL Project (which will be described in Section 2.5 of Chapter Two), a major project in the generation and understanding of natural language based in the Computational Linguistics Unit of Cardiff University and directed by Professor Robin Fawcett. At the time of planning the research to be described here, I had recently completed an MSc dissertation (Day 1993a) in this framework, and in my view, it represented an interesting and challenging context in which to develop and evaluate the ideas behind this new type of parser.

For this type of project to succeed, there needs to be close co-operation between a computer scientist and a linguist (myself and Professor Fawcett in the present case). In such a working relationship, each partner is continually constructively challenging and complementing the ideas and suggestions of the other, in order to reach the optimal solution.

The database and the parser that constitute the major products of this research are the results of three types of input, in terms of the contributions made by Professor Fawcett and myself. Given that the present work is a PhD thesis, it is important to make it clear from the start which contributions came from Professor Fawcett, which from myself, and which came from joint work. In general terms, the input that draws on linguistics comes from Professor Fawcett and his colleagues in the COMMUNAL Project (see Section 2.5 of Chapter Two), while I am responsible for the computational input. It should be emphasised that the latter is not merely an implementation of the linguistic concepts, but a major contribution from computer science that complements the input from linguistics. Indeed, the most original parts of the work are those that combine the two, and in these areas, the computational inputs have been at least as original as the linguistic inputs. This thesis focuses on the computational and the joint aspects of the research. At the relevant points of the thesis, I shall identify which contributions came from myself and which came from our joint research.

1.4 How this thesis is organised

This work is organised in three parts. Following this introductory chapter, Part One provides the necessary background to the field. It has three chapters. Chapter Two provides an introduction to computational linguistics. It describes the sub-fields of natural language understanding and natural language parsing before discussing some of the major research projects. Chapter Three introduces the rapidly expanding field of corpus linguistics and includes (a) descriptions of the most popular corpora that are being used for research today, and (b) a review of some corpus query tools. Chapter Four gives the reader the necessary background information about the Cardiff Grammar, i.e. the particular version of Systemic Functional Grammar that is used in this work.

Part Two describes (a) the corpus used in this work, and (b) the database which has been created to hold the corpus in a suitable form for the parser to consult. This is as central a component of the model as the parser itself. Part Two contains five

chapters. Chapter Five describes the traditional methods that have been used for specifying a rule-based grammar, and discusses their suitability for use in this project for both: (a) a **corpus annotation scheme**, and (b) a **method of representing syntactic relationships**. Chapter Six describes our choice of annotation scheme and how we translated an existing corpus into it. Chapter Seven outlines the design of the corpus database schema, and Chapter Eight the new **corpus query tool** that I created for use in our research (ICQF+). Chapter Nine details the changes we made to the Polytechnic of Wales (POW) corpus to create the new Fawcett-Perkins-Day (FPD) Corpus.

Part Three describes the process of parsing itself. Chapter Ten starts by looking at the traditional approaches and surveys the current approaches to parsing, and this is followed in Chapter Eleven, by a description of approaches to parsing Systemic Functional Grammar. Chapter Twelve describes the early attempts at parsing in this project.

Part Four describes the new parser. Chapter Thirteen introduces it, and Chapter Fourteen describes the **probabilities tables** and the **parser's working tables** and its **data structures**. Chapter Fifteen describes the parsing algorithm in detail and, since our parsing algorithm is quite different from other approaches, I have provided a walkthrough using sample sentences in Appendix K. The parser operates in a configurable environment called the **Parser WorkBench**, and this is described in Chapter Sixteen. Chapter Seventeen describes the procedures used to evaluate the parser. My conclusions and a summary of further work to be undertaken can be found in Chapter Eighteen.

PART ONE

The Context of this Work

This part of the thesis sets the scene for the description of the parser and its complementary components by providing brief overviews of the three major fields of study on which this project draws.

Chapter Two provides an introduction to the field of Computational Linguistics. It discusses some of the major theories of language used in this field, focussing in particular on the theory of language upon which the work described in this thesis is based: Systemic Functional Grammar (SFG), and within that, on the version used in this project, i.e. the Cardiff Grammar. This is followed by an introduction to the fields of Natural Language Generation, and Natural Language Understanding. Chapter Two concludes with an outline of the components of the COMMUNAL Project, to which this work makes a major contribution.

Chapter Three gives an overview of the rapidly growing new field of Corpus Linguistics. After introducing the main ways in which corpora have been classified, it provides overviews of the major corpora used in the field. It then discusses the requirements for tools that can extract information from these corpora for use in both (a) the general study of language, and (b) the specific role of providing data that can be used by a natural language parser of the type described in Part Four of this thesis.

Chapter Four provides the reader with the necessary background in the linguistic theory used in this project, namely that of the Cardiff Grammar, focussing in particular on its syntax.

Chapter Two

Computational linguistics

This chapter gives a necessarily selective overview of the field of computational linguistics - the field to which the research reported here seeks to contribute. We start with some definitions, and then look at some of the different linguistic theories that have been used. This is followed by a brief survey of some applications of these theories in computational linguistics systems, and finally we will look in detail at the COMMUNAL systems for natural language generation and understanding, since this is the overall model in which the research described here is set.

2.1 The field of computational linguistics

Computational linguistics (or Natural Language Processing) is the name given to the field that studies the processing of natural languages such as English, French, German, Italian etc in a computer. It can be divided into two main fields: **natural language understanding** (NLU) and **natural language generation** (NLG), although there are others such as machine translation, document summarising and so on.

In this work, we are mainly concerned with Natural Language Understanding. This is the process of determining what the performer of a text has said, or written. As a first approximation, it can be divided into **sentence parsing**, **semantic interpretation** and the processing of information in the **higher levels of beliefs and reasoning**.

The understanding process starts either with speech (in which the first stage is speech analysis) or with the words of a written text. The classic approach to parsing in the second half of the last century is that a **natural language parser** takes as its input a string of written words and determines if it forms a sentence according to the rules of the grammar. Normally, it will deliver one or more syntactic representations of the sentence in the form of a **tree diagram**, which is annotated with quite widely varying degrees of richness.

The **semantic interpreter** takes the syntax tree delivered by the sentence parser (or trees, in the case of ambiguous sentences) and turns it into a semantic representation that can then be processed by the higher NLU components. Normally, the output from a semantic interpreter is a logical form, which is then added to the belief system, and, if appropriate, responded to.

The field of natural language generation is concerned with the production of natural language texts. The process typically involves three levels of planning – **overall planning**, **discourse planning** and **sentence planning**. The output from the generation system is typically a structured set of sentences with rich syntactic and intonation labelling, and this may in turn become (a) a written output and / or (b) an input to a speech system.

2.2 Linguistic theories

In terms of the philosophy of science, linguistics is still at the stage when there are competing theories of language, and so competing schools of linguistics (Kuhn 1962/70). In general, most theories of language reflect the influence of either formal language theory or the tradition of descriptive linguistics. This section provides a brief introduction to some major theories of language that have been influential in computational linguistics.

2.2.1 Some formal theories of language

The major influence in formal linguistics, which evolved from the fields of mathematics and logic, has been Noam Chomsky - the creator of Transformational Generative Grammar (Chomsky 1957). This model, which was constantly re-worked in the later part of the century, greatly influenced a number of other models that have been used for parsing, such as Generalised Phrase Structure Grammar (GPSG) and Tree Adjoining Grammar (TAG).

In most approaches to parsing, Phrase Structure Grammars (PSGs) are assumed to be 'the correct way' to represent the syntactic structure of a sentence (eg Gazdar et al 1985). PSGs use the concept of **rewrite rules**, such as $S \rightarrow NP VP^1$. They take the symbol on the left-hand side and rewrite it by the symbols on the right-hand side. This type of formalism is commonly used in parsing systems which are based on formal linguistics (these are discussed further in Chapter Five). PSGs predominantly show classes of **units** (as we shall term them here) foregrounding as an essential concept 'the phrase' (which we shall here term **group**). Thus they foreground the **units** in a language rather than the **functions** served by those units. In the Cardiff Grammar, both are given an equal weight, as we shall see in Chapter Four.

¹ This example states that a sentence (**S**) is to be rewritten as a noun phrase (**NP**) followed by a verb phrase (**VP**).

Chomsky's (1965) Transformational Generative Grammar added to the concept of a set of phrase structure rules the concept of transformational rules that 'transform' one phrase structure into another. The concept of a PSG was extended in Generalised Phrase Structure Grammars (GPSG) in such a way as to cut out the transformational rules (Gazdar et al 1985). Like other grammars of the later twentieth century, GPSGs include features such as word agreements, verb classifications and dependencies. Rules are separated into Immediate Dominance (ID) rules and Linear Precedence (LP) rules, and a slash category is added to allow GPSG to generate sentences with unrealised elements termed 'gaps'.

Another theory that is quite widely used in parsing is Tree Adjoining Grammar (TAG) (Joshi 1985). This specifies its rules in the form of elementary trees, and uses the tree operations of substitution and adjoining; TAG is described in Chapter Five and a good illustration of its use is provided by the XTAG project (Xtag 1995).

2.2.2 Some description based theories of language

In contrast with formal theories of language, which focus primarily on the study of syntactic forms and mathematical models, descriptive theories have grown from the study of texts and so, typically, they take a functional approach to language. Major influences have been Quirk and Halliday in Great Britain and, in the United States, Pike. Most modern description-based approaches to language are therefore also **functional**. This means there is a major distinction in current linguistics between **formal** and **functional** approaches to language.

Firth (Firth 1957), laid the groundwork for Halliday when he developed a theory of language as a social function. Halliday, who attended lectures by Firth, Robins and others of the 'London School', laid the foundation for his theory with his system and structure description of Chinese (Halliday 1956/76). Halliday developed this into Scale and Category Grammar (Halliday 1961) and then later, Systemic Functional Grammar (SFG) (Halliday 1985, 1994). It is Fawcett's further development of this (Fawcett 1980, 2000a) that provides the linguistic basis for the present project.

2.2.3 Formal and descriptive theories in the context of parsing

In its syntax structures SFG, by its very nature, employs functional (or semantic) labels to describe language. For example, nominal groups contain determiners, modifiers and heads of various types, each being labelled according to their function

within the unit. The Clause, for example, has various different types of Adjunct, all being identified in functional terms, and the Main Verb predicts the presence of various configurations of **Participant Roles**, which specify the semantic roles of the Subject and Complement.

In contrast, most implementations of PSGs are based on descriptions only in terms of units, e.g. the noun phrase, the verb phrase and the prepositional phrase, normally with little or no semantic labelling. When compared to a typical PSG, the semantically-oriented representation given in an SFG presents a more demanding challenge to the computational linguist who is constructing a parser, because the parser has to be additionally concerned with the function that a syntax element is fulfilling. The reward for the extra effort, however, is a semantically richer annotated syntax tree, and this in turn is of far greater use as an input to the higher levels of processing in a natural language understanding system than an analysis in PSG terms.

The next section provides a brief overview of some uses of these linguistic theories in computational linguistics.

2.3 Some applications of linguistic theory in computational linguistics

While a wide range of linguistic models has been explored in Natural Language Generation, most approaches used since the early 1990s have used a form of Systemic Functional Grammar. In contrast, formal grammars, because of their emphasis on syntax, have enjoyed greater popularity in Natural Language Understanding - with the notable exception of Winograd (1972), as we shall see in Chapter Eleven in Part Three.

2.3.1 Theories used in natural language understanding applications

The main focus of research for systems which process natural language has been on building natural language parsers. These are described more fully in Chapters Ten and Eleven in Part Three. But here we can say that their task is to take a string of words of a written and / or spoken input and to convert this into a syntax tree. This tree will then be used by a **semantic interpreter**, so that the more richly annotated with functional information it is, the more helpful it will be.

Every natural language understanding system, however, requires some form of semantic interpreter. In this area of research, much less work has been done, but one notable example of a semantic interpreter is the one built for the COMMUNAL project by O'Donoghue (1991b). This is the REVELATION system. It illustrates what

is required for the component that takes the output from the parser (documented in this thesis) and delivers it to the next stage in the NLU process. Specifically, REVELATION reverses part of the process of natural language generation by turning the syntax structures into the set of semantic features that would have been chosen in a system network in order to generate the syntactic structures.

This, then, is the semantic interpreter to which the outputs of the parser described in this thesis are passed.

We turn next to consider the linguistic formalisms used in NLG.

2.3.2 Theories used in natural language generation applications

As reported by O'Donnell and Bateman (2005), Henrici (1965) was the first to use system networks in NLG. But the first complete NLG system was that of Davey (1978). He created the PROTEUS system, which used a systemic lexicogrammar to generate commentaries on a game of noughts and crosses.

Two major NLG projects followed, both of which used Systemic Functional Grammar. The first was the PENMAN Project (Mann and Matthiessen 1985), and the second was the COMMUNAL Project, to which the work described here is a major contribution. This is described in Section 2.5.

PENMAN, which was the work of Mann, Halliday and Matthiessen, has the NIGEL grammar at its core. O'Donnell and Bateman (2005:360) report that this contains approximately 700 grammatical systems and 1500 grammatical features.² PENMAN was adapted to cover other languages (e.g. German, in the KOMET project, as described in Teich (1999)), and Mann and Thompson (1988) use Rhetorical Structure Theory for discourse planning within PENMAN.

Other notable projects in NLG are McDonald's MUMBLE (1983) and Meteor's (1989) SPOKESMAN generator, but while these show some influence from SFG, they use as their model of syntax, a formal tree adjoining grammar (Joshi 1985) as a basis for syntactic processing (Reiter 1994:2).

Reiter, Mellish and Levine (1995) demonstrate a practical application of NLG for written texts in controlled English in the Intelligent Documentation Advisory System (IDAS). IDAS was supported by Racal Instruments Ltd, and used SFG techniques to generate text for use in technical manuals from a knowledge base. The production of

² For a full description of the PENMAN project see Matthiessen and Bateman (1991).

technical documentation to support a maintenance system is an expensive process, and compliance to restricted English (ASD Simplified English (ASD 1985)) is a mandatory requirement). IDAS, although it relied to some degree on canned text, demonstrates some success in this field. O'Donnell, in 1996, created the WAG natural language generator. This has practical applications, including one in the ILEX project for automatically creating web pages for museum exhibits (O'Donnell 1996).

Rambow and Korelsky (1992) developed JOYCE, which was based on yet another theory of language namely, Mel'cuk's Meaning Text Theory (1998). One widely used system was that of Elhadad (1993). He extended Functional Unification Grammar (Kay 1979), and drew on the SFG work of Fawcett in his Functional Unification Formalism (FUF) for his natural language generator. A practical application of FUF was ADVISERII, which was used to advise students on which courses to take (Elhadad 1993).

Langkilde and Knight (1998), based at the University of Southern California, created NITROGEN, which used probabilities extracted from a corpus to assist in the process of text generation. The system, which maps meanings into word-lattices (a concept also used in parsing, see Chapter Ten), uses statistical information created by its corpus statistical extractor to rank the top-n best paths to determine the best sentence to present to the user. This system, which is only described in very general terms in the published work, is the only NLG system that I have come across (other than the GENESYS system in the COMMUNAL Project) that makes use of probabilities.

Next, in order to provide an overview of the system in which the parser described in this thesis is embedded, we turn to consider the COMMUNAL Project, a major component of which consists of an implementation of the Cardiff Grammar. We shall start by introducing the Cardiff Grammar, and then we will look in more detail at its implementation in the COMMUNAL project.

2.4 An overview of the Cardiff Grammar

2.4.1 Background

In Section 2.2, we saw that there is a wide range of different approaches to modelling language within linguistics and the purpose of the present section is to give an outline of the particular theory in which the present project is set.

This theory is systemic functional linguistics (SFL), of which the major architect is Halliday (e.g. Halliday 1976, 1985, 1994). This is the best known of various theories of language which are 'description-based' and 'functional', in the terms introduced in Section 2.2. Within the overall model of language that is given in SFL, there is the component that is concerned with generating and understanding sentences. This is the **grammar** of the language, or more properly, as we shall see, the **lexicogrammar**.³

However, as a result of the extensive work by Fawcett and his team at Cardiff since the late 1980s, there are now two distinct versions of SFL. See Fawcett (2000a) for a full account of the development of the two versions of the theory. Nonetheless, Halliday has stated that the Cardiff Grammar is 'based on the same systemic functional theory' as his own (Halliday 1994:xii) and Fawcett would agree with this statement, at least with respect to the way in which the grammar operates. The two versions of the theory are commonly referred to in the literature as 'the Sydney Grammar' and 'the Cardiff Grammar'. Butler (2003a and 2003b) provides good descriptions of both versions (and also of two other 'structural-functional' models). He reaches the conclusion that 'in my view the Cardiff model represents a substantial improvement on the Sydney account' (2003b:471).

The research reported in this thesis uses the Cardiff Grammar, as developed in the COMMUNAL Project and as described in Fawcett (2000a), and as used in the Fawcett-Perkins-Day Corpus (FPD) (see Chapter Nine).

2.4.2 The place of syntax in a systemic functional grammar

I shall now provide a brief overview of how an SFG works, in order to show the place of syntax within the overall grammar. The following description is based on that given by Fawcett in Chapter Three of Fawcett (2000a). Here we assume a model in which it is the **system network** that models the 'meaning potential' of a language and constitutes its **semantics**.⁴

³ It may be helpful to remind the reader that a grammar does not necessarily have the form of a 'phrase structure grammar' as described in Section 2.2.

⁴ See Chapter Five of Fawcett (2000a) for a discussion of the similarities and differences between the generative versions of the Sydney Grammar and the Cardiff Grammar.

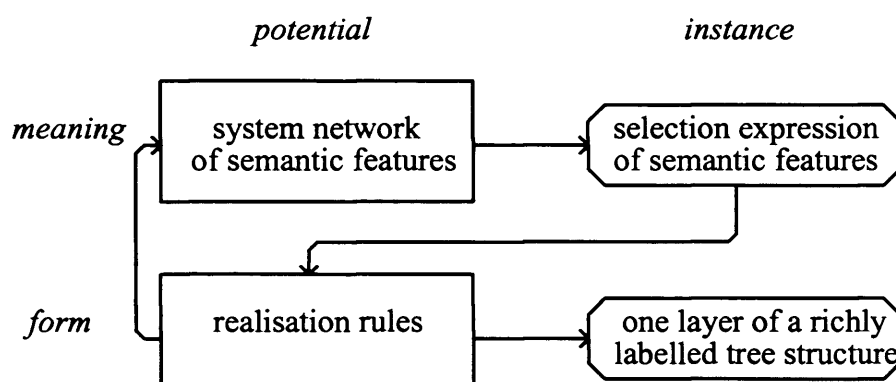


Figure 2.1: The main components of a Systemic Functional Grammar and their outputs

Figure 2.1 shows the two main components of the grammar of a language (on the left), and their outputs (on the right). The grammar contains two ‘potentials’: one at each of the two levels of **meaning** and **form**. The process of generation is controlled by the **semantic system network**, which models the language’s ‘meaning potential’ (see Figure 2.3 for an example).

Figure 2.1 also shows the two types of ‘instance’ - i.e. the outputs from each of the two components. Each traversal of the network results in a **selection expression** of the features chosen on that traversal. The **realisation rules** refer to these as they specify the output. An output consists of a **syntactic unit**, its **elements**, and the **items** that expound them - unless one of its elements is to be filled by a further **unit**, in which case the rule specifies **re-entry** to the network to generate a further unit (note the arrow on the left).⁵

Each traversal of the network chooses features relevant to one semantic unit, and each generates one syntactic unit. The first traversal typically generates a Clause, and later traversals generate the nominal groups (and other units) that will fill some of its elements.

Thus a ‘grammar’ is essentially a model of the sentence-generating component of a full model of language and its use. So the term ‘grammar’ is used here as a short form for **lexicogrammar** - a term that usefully reminds us that the system network covers meanings realised in lexis, as well as in syntax and in grammatical items (as in a narrower sense of ‘grammar’). Given that an SFG operates in this way, it is

⁵ See Chapter Four for the definitions of the Cardiff Grammar’s units, elements and items.

important to note that that the parser does not operate in this generative framework. The part of what has been described above that is relevant to a parser is the output at the level of form, i.e. a knowledge of the complete set of structures that might be generated through the use of the SFG. The framework for describing these structures for the Cardiff Grammar will be the subject of Chapter Four.

The COMMUNAL implementation of the Cardiff version of SFL is described briefly in the next section, and this includes an illustration in an introductory manner, of its use in natural language generation. As for the use of SFL in natural language understanding, we had a brief introduction to the concept of a semantic interpreter in Section 2.3.1 and the description of its use in a parsing process will be described at main points in later chapters. The next section, however, will illustrate the place of all these components in the overall model.

2.5 An introduction to the COMMUNAL Project

The research reported in this thesis is part of the COMMUNAL (CONvivial Man-Machine Understanding through NATural Language) project based at Cardiff University under the direction of Professor Robin Fawcett.

The aims of COMMUNAL are 'to enable computationally naïve people to interact easily and naturally with intelligent knowledge base systems' (Fawcett 1988).

COMMUNAL (see Figure 2.2) integrates both NLG and NLU components. In principle, the COMMUNAL system is able to accept input as a string of text (which may have been pre-processed from a speech analyser), and give a response (if one is required). Details of the processing within the understanding and generation components are given in the sections that follow.

In the early phases of the project, Cardiff had a sister team who were based at Leeds University; in general, Cardiff were responsible for the generation components and Leeds the understanding components. It was at Cardiff that Weerasinghe (1994) produced a probabilistic parser for COMMUNAL. The valuable work by Atwell, Souter and O'Donoghue at Leeds and by Weerasinghe at Cardiff will be described in Chapter Eleven.

2.5.1 The natural language generation components in COMMUNAL

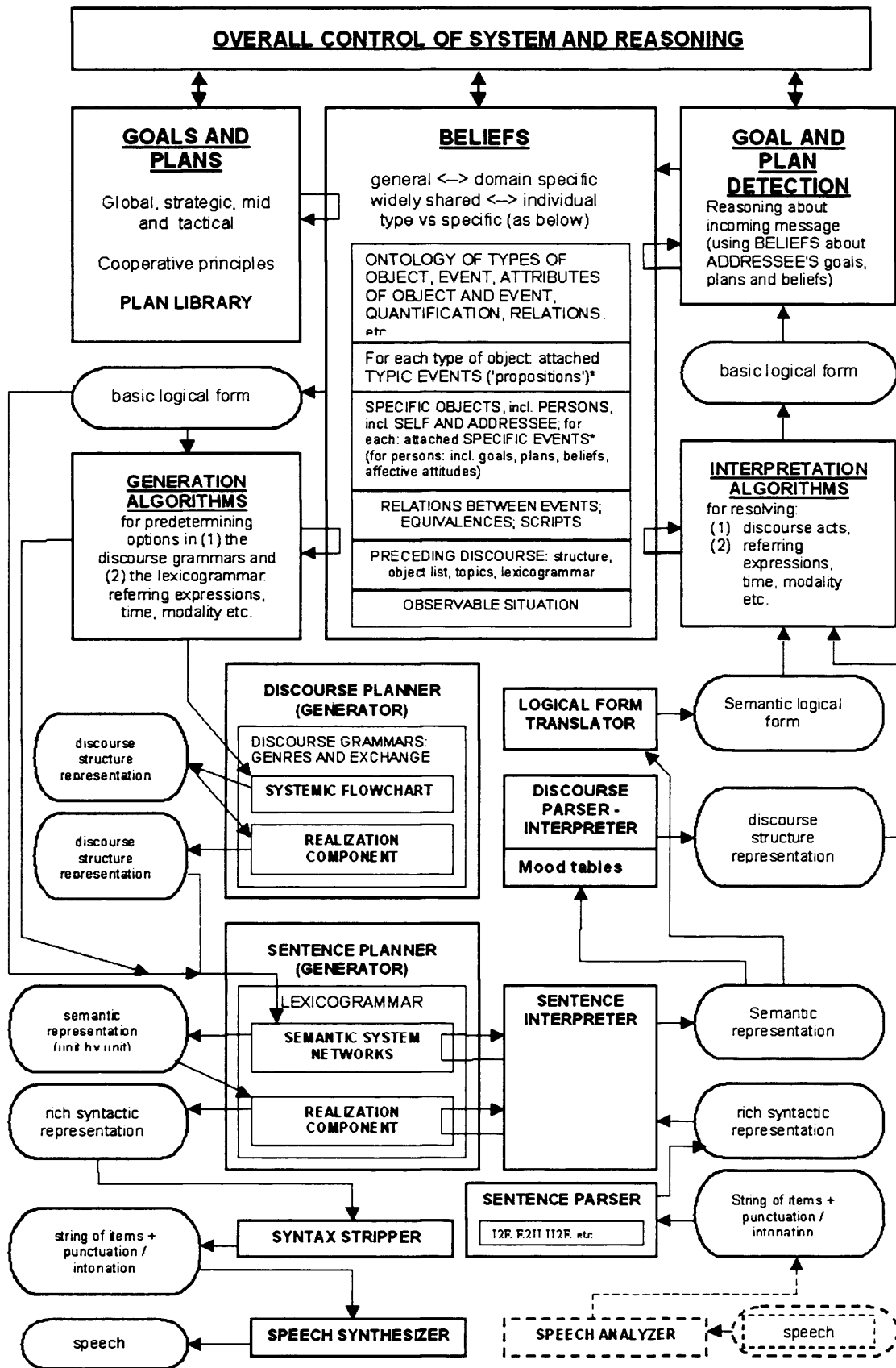
See Figure 2.2 for an overview of the COMMUNAL model. As is generally accepted in the field of NLG, the process of generating a sentence (or utterance) involves three main levels of planning (as Figure 2.2 shows): **overall planning, discourse planning**

and **sentence planning**. Fawcett (1994) describes the application of this overall architecture in COMMUNAL.

Overall control involves the creation of high level plans using goals, reasoning and beliefs about objects and situations which are stored in COMMUNAL's belief system.

The **discourse planner** takes the high level plan and turns it into a discourse structure. This consists of **genre** structure and either **exchange** structure (for texts involving two or more persons) or **rhetorical** structure (for monologues, including most written texts), or a combination of both (Fawcett and Davies 1992). Here an instruction, such as **solicit information**, may be created as an input to the **sentence planner**.

The system networks, together with their realisation rules, form the **lexicogrammar**. This is at the heart of the COMMUNAL's model of language and its use, and is called GENESYS (because it GENERates SYStematically). As we saw in Section 2.4.2, there is a traversal of the system networks which results in a rich syntactic representation of the generated sentence. This requires several steps. Firstly, it involves selecting a pathway (or pathways) by taking branches through the system network (e.g. as in Figure 2.3). The grammar, based on the features in each system, makes choices based on the decisions made by the higher level planning. In addition, the system network also contains 'and' nodes. An 'and' node is represented by a left-hand curly bracket "{", and it means 'follow all these sub-networks in the order given'. The result of the traversal of the system network is a **selection expression of semantic features**, and these are then turned into a syntactic structure by executing the realisation rules.



Key: = system component, = input / output

* For each event (typic or specific): event type, participants, circumstances, time, aspectual type, confidence level, effective attitudes.

Figure 2.2: Fawcett's diagram of the COMMUNAL System

As an example, let us consider how to generate a sentence 'The girl destroyed the book'. We will use the simplified system network shown in Figure 2.3 (taken from Fawcett 1994). The traversal of the system networks are carried out in cycles with each cycle building a new syntactic unit.

The first pass involves taking the '**situation**' branch in the network. Figure 2.3 shows that both **TRANSITIVITY** and **MOOD** networks are to be followed, so building a clause which will, as a result of earlier planning decisions, contain a Subject which has: (a) an Agent as an associated Participant Role, (b) a process of '**destroying**', and (c) a Complement with an Affected Participant Role. Upon entering the **TRANSITIVITY** network, GENESYS must choose whether the process will be either (a) '**material**', (b) '**mental**', or (c) '**environmental**', and the path chosen is '**material**'. Having selected '**material**', a further choice of one of either (a) '**one-role**' or (b) '**two-role**' has to be made, and in this case it is a '**two-role**' process. In the '**two-role**' network, two sub-networks have to be followed. The first is for **VOICE**, and here the planner has selected **agent-S-theme**, which will conflate the role of Agent by Subject. The second of the two systems gives options in the specific type of process, and '**destroying**' is chosen. Having exhausted the routes through the **TRANSITIVITY** network, the generator then follows the **MOOD** network. Here planning determines that it has to select '**information**' (rather than '**proposal for action**') and then, finally, it selects '**giver**' and '**past**' from the two parallel sub-networks.

Thus, the paths taken through the networks for the first pass generates, for **TRANSITIVITY**, this selection expression:

```
[referent,situation,material,two_role,agent_S_theme,destroying]
```

And, for **MOOD** and **TENSE**, the selection expression is:

```
[information-giver,past]
```

It should be emphasised that this is an example from a 'toy' grammar; the full grammar requires many more choices (see Fawcett, Tucker and Lin 1993).

The **realisation rules** (see Figure 2.4) state that as '**situation**' was chosen, a clause (C1) is created in the output structure.

The feature '**giver**' creates a Subject (S) and locates it in Place 3 in the clause, and **agent-S-theme** creates an **Ag** next to the **S** to represent the participant role of

agent. The process **destroying** and the choice of '**giver**' and '**past**', generates a Main Verb (**M**) at place 7 and expounds it by the item **destroyed**.⁶ On the same cycle, the grammar creates a Complement (**C**) and locates it at Place 8 in the Clause, and an affected (**Af**) participant role that is conflated with it. The grammar then re-enters the network to generate a unit, its elements and its items to **fill** both the **agent** and the **affected**.

The choices the planner makes for Agent are :

[**thing,cultural_classification,recoverable,person,girl,singular**]

The realisation rules are then applied to the selection expression as follows. The feature '**thing**' generates a nominal group (**ngp**); '**recoverable**' generates a deictic determiner (**dd**) within the nominal group and expounds it by **the**, and generates a head (**h**) which, because **singular** is selected, it is expounded by **girl**.

The choices that planning makes in re-entry for the affected role are very similar :

[**thing,cultural_classification,recoverable,object-book,singular**]

As before, '**thing**' generates a nominal group (**ngp**) and '**recoverable**' generates a deictic determiner (**dd**) expounded by **the**, within the nominal group a head (**h**) is generated that, because **singular** is selected, it is expounded by **book**. The sentence generated in this example is shown in Figure 2.5.

⁶ If the choice had been **seeker**, an operator (**O**) would also have been generated in Place 2, giving '**did destroy..**'.

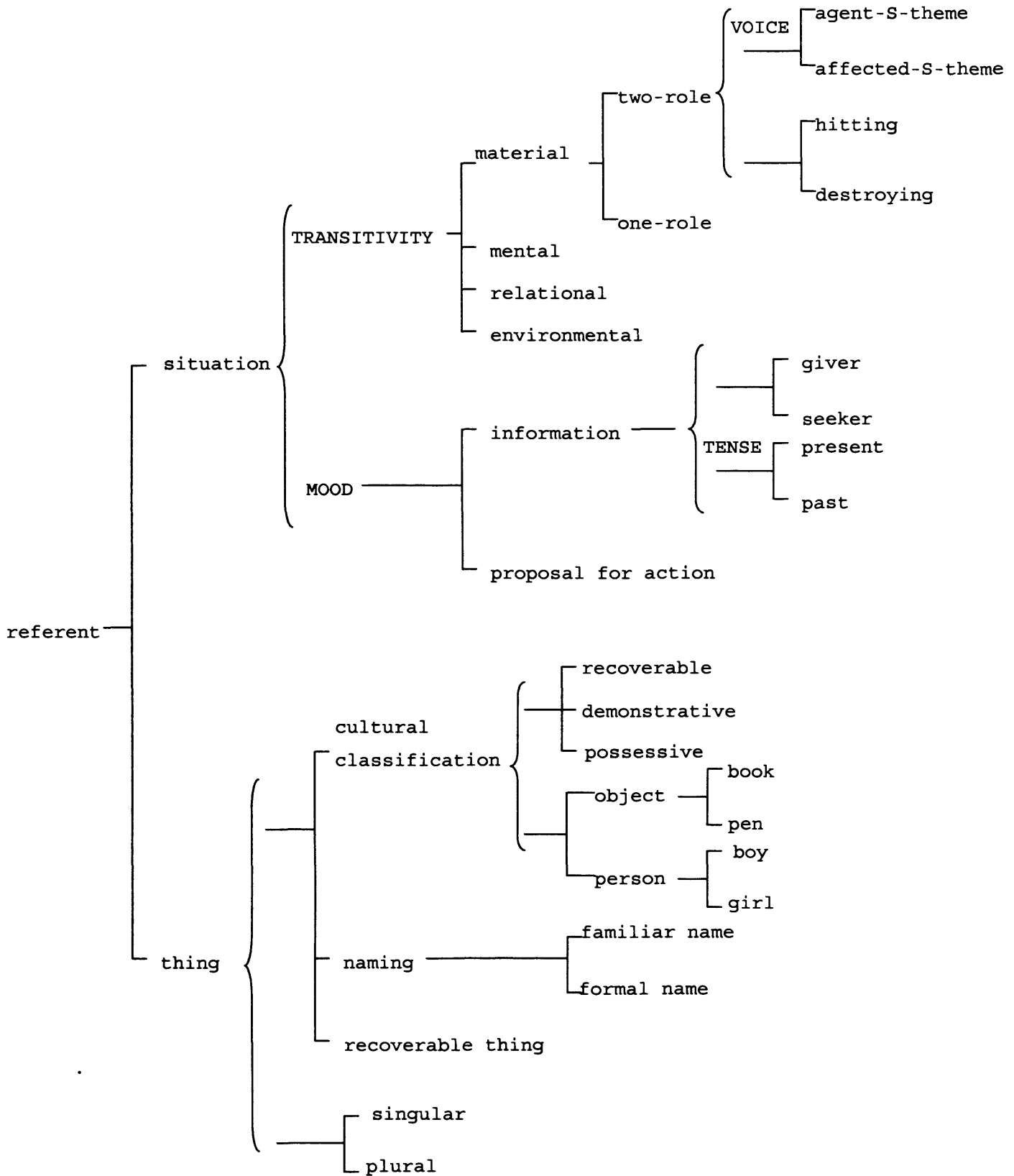


Figure 2.3: A highly simplified system network (from Fawcett 1994)

FEATURE	CONDITIONS	REALISATION
situation	past < giver seeker	C1
giver		S @ 3
agent-S-theme		Ag by S
		For Ag, re-enter at [thing]
		C @ 8, Af by C
		For Af, re-enter at [thing]
destroying		M @ 7 < 'destroyed'
		O @ 2 < 'did'; M@7 < 'destroy'
thing		ngp
recoverable		dd < 'the'
book		h < 'book'
girl	h < 'girl'	

Figure 2.4: Realisation rules to accompany the sample network

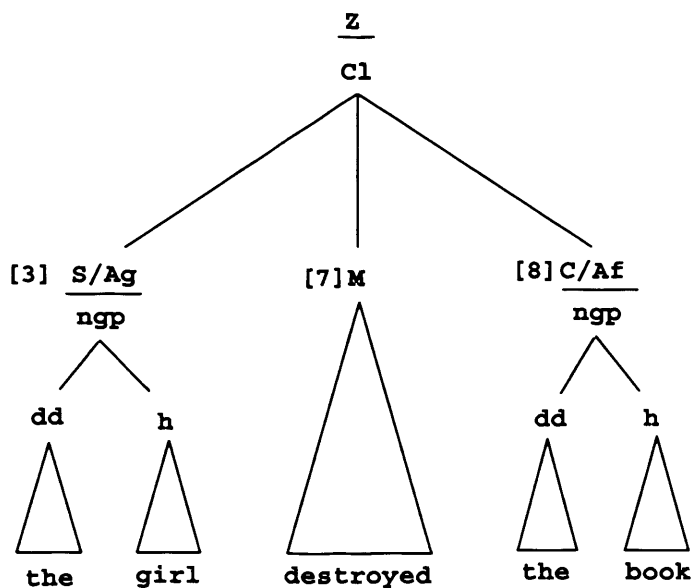


Figure 2.5: The output from the process of generation

This then, is the form that an SFG takes. Such grammars have been very widely used in NLG, as we have seen, and the problem that confronts the researcher whose goal is to build a parser is that of drawing on it in an appropriate manner.

In the next sub-section, we will give a brief overview of the problems associated with using SFG in an understanding process and how they have been resolved in COMMUNAL.

2.5.2 The natural language understanding components in COMMUNAL

We turn now to the right-hand side of the diagram in Figure 2.2.

When processing the spoken words, the understanding process starts with sounds or written symbols. If the text is spoken, a speech analyser is responsible for transforming sounds into a string of written words (items) and these are normally marked up for **intonation**. A written input is simpler, and is supplemented by punctuation. These are two possible forms of input to the parser. For typed written texts then, there is no need for a preliminary process.

The task of the parser is to turn the string of items into a richly annotated syntactic structure, i.e. a tree diagram, complete with syntax labels for units, elements and items, which show the syntax structure of the sentence.

The **semantic interpreter** (Section 2.3.1) takes the syntax structure and works alongside the **discourse interpreter** to derive a **semantic representation**. The **logical form translator** will then produce a **semantic logical form**. The higher level components include the interpretation algorithms which interpret discourse acts and referring expressions, and the **goal and plan detector** which is used to determine the purpose of the incoming message, in consultation with the **belief system**. When understanding is successful, new beliefs are created and, if necessary, the process of formulating a response is initiated in COMMUNAL's generation system.

In this thesis, however, we shall focus entirely on the parser. In Chapter Eleven, we shall look at the various alternative parsers using SFG that have been proposed. But we can say at this point that it is the syntax of the output from a generator of the sort described in Section 2.5.1 that provides the data upon which the parser to be described in Part Four draws. It is the semantic interpreter, in fact, which comes closest to being the 'reverse' of the generator - and not the parser (as is widely assumed). See Fawcett (1994) for the reasons why an NLU system cannot simply be the reverse of a generation system, as some formal language theorists have claimed (e.g. in papers in Strzalkowski 1994).

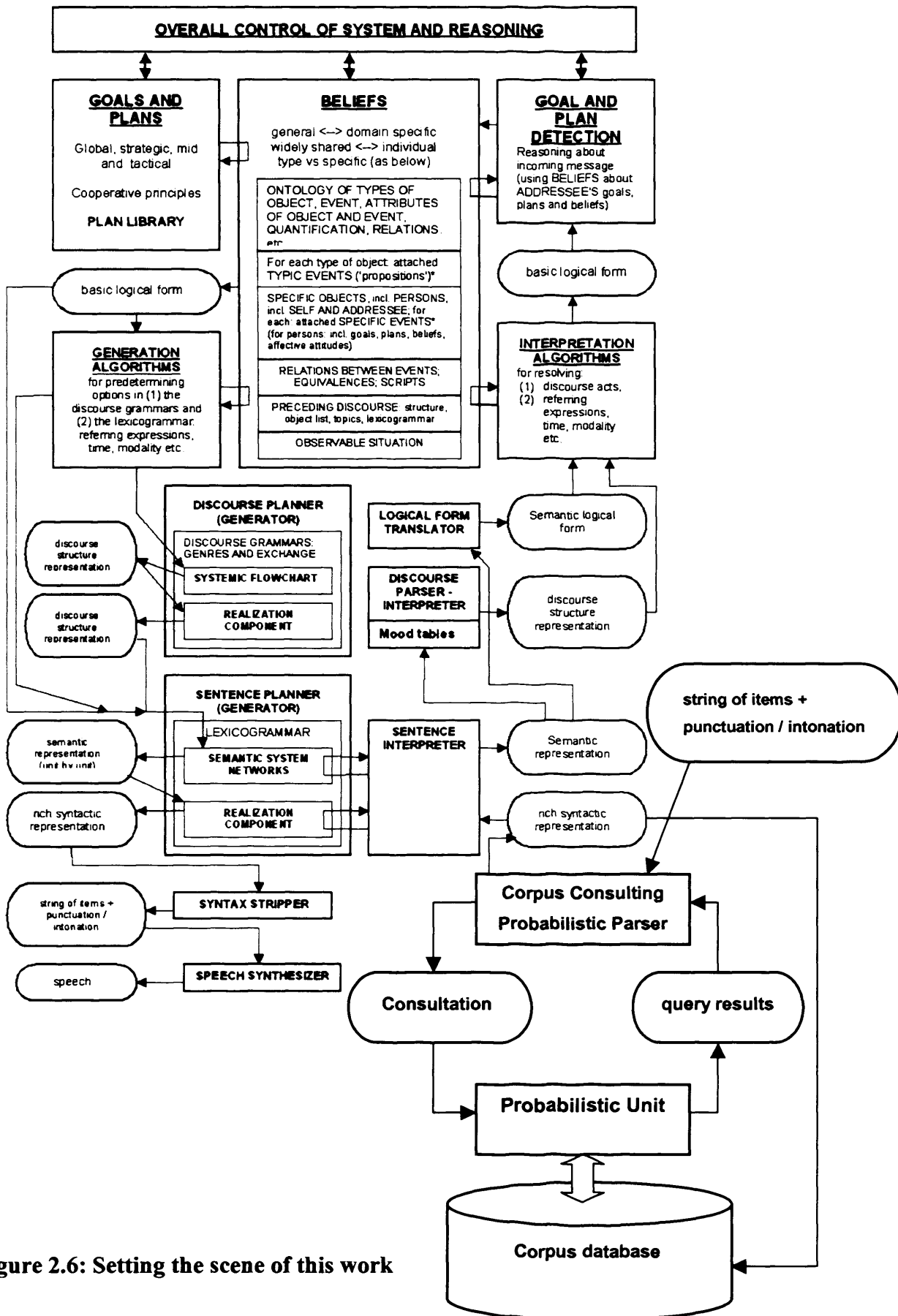


Figure 2.6: Setting the scene of this work

2.6 Summary

In this Chapter, I have provided an overview of the field of computational linguistics in general, and the COMMUNAL project in particular. We began by surveying a number of different linguistic theories, and we have discussed the differences between formal approaches and descriptive approaches.

After looking briefly at some applications of linguistic theory in natural language understanding and generation, we focussed on the particular application of Systemic Functional Linguistics (SFL) within which the research reported here is set, i.e. the COMMUNAL Project. This system provides for both natural language understanding and natural language generation, as it 'understands' what has been said, and 'generates' a response. Figure 2.6 provides the framework for understanding and shows where the components described in this thesis fit within the COMMUNAL Project, and sets the scene for what follows. The project described in this thesis therefore provides a new natural language parser for COMMUNAL of a new type, drawing its knowledge from a **corpus database**.

In Chapter Three, we will introduce the field of corpus linguistics, and, in Chapter Four, the model of language used by this project, the Cardiff version of SFL as described in Fawcett (2000a).

Chapter Three

Corpus linguistics

Corpus linguistics is the study of language when it is based on facts (including statistics) that are derived from naturally occurring texts that are held in a body of text (called a **corpus**). Before computer corpora were available, language students had to provide their own examples, which were based on intuition and personal observation, and this often led to emphasis being placed on relatively rare linguistic phenomena. Corpus linguistics gained popularity at the beginning of the 1990s, as more and more texts became available electronically. Today, the student of linguistics is provided with powerful software tools, which can unlock data from vast databanks, and these are providing new insights into the nature of language.

Corpora are being used increasingly in the field of computational linguistics. Natural language parsers, such as the one described in this thesis, use statistical data to aid their analysis, and even data that can be regarded as syntax rules that have been extracted from a corpus. Natural language generators may use the facts gained from corpora to determine the best route through the networks of a sentence planner.

In the 1990s, there were only a few corpora that included the syntax annotations (**parsed corpora**). This meant that most of the available software tools were only able to provide concordances, collocations and word lists. At this time, the Polytechnic of Wales (POW) Corpus was one of the few parsed corpora available, and the Interactive Corpus Query Facility (ICQF) (Day 1993a) was developed as a tool for use in the COMMUNAL project. Although parsed corpora are still small in number, there are now more of them, and we are beginning to see more powerful corpus query tools being developed. One such tool is the second version of ICQF, which forms part of this present project, and it is described in Chapter Eight.

The present chapter provides an insight into the subject of corpus linguistics. It gives details of some of the major corpora being used in the field and also those that are applicable to this work. It outlines the type of information they contain, and it shows how they are annotated. We will look at types of query and presentation that are useful to linguists and researchers, and at the tools that are being used to deliver these requirements.

3.1 Definitions and classifications

3.1.1 What is a corpus?

A corpus is a large body of naturally occurring text. It can either be:

- (a) **parsed** (fully syntactically analysed),
- (b) **tagged** (labelled with parts-of-speech), or
- (c) **raw** (with no additional annotation).¹

3.1.2 Classifying corpora

Researchers have been building corpora of language texts in electronic form for over two decades. While in recent years more attention has been paid to controlling and classifying them in terms of agreed linguistic variables, most are seriously deficient in this respect.

Halliday (1976) has suggested that there are two main types of linguistic variable in texts: (a) variation according to **register** and (b) variation according **dialect**, and by and large his framework is the one that is most widely used in the field. He defines **register** as a particular configuration of meanings that is associated with a particular situation type (Halliday 1975). He suggests that the main variables are:

- (a) **field** (which includes subject matter),
- (b) **mode** (whether the performer of the text is delivering it in written or spoken form),
- (c) **tenor** (the social and personal relationships between the interactants).

Field defines the text's **technicality** and the **subject** matter of the discourse. An aircraft engine maintenance manual, for example, contains texts that are written in technical English that contain information about how to service a particular engine; contrast this with a novel which would be written in non-technical English containing subject matter relating to the story being told. Interestingly, the same item can have different meanings in different fields. For example, the word **bank** is likely to have different meanings in texts relating to fishing, to aircraft manoeuvrability or to finance text. Further, the frequencies of occurrence of a word when it is functioning as a specific syntactic category is likely to be very different (eg **bank** as a noun and as a verb).

¹ Today, following the advances of part-of-speech taggers, most corpora used in computational linguistics are at least tagged.

Mode refers to the classification of the text as either **written** or **spoken**. Spoken texts will typically contain somewhat different items - and even syntax structures - from those in written texts. For example, spoken texts typically have a great deal of unfinished structures and **ellipsis**, i.e. words that are recoverable, either in the course of rapid speech, or from what has gone before. Furthermore, spoken texts tend to contain a large number of what we shall term **replacement elements**. These occur when the speaker wants to reinforce and make clear what has been said earlier in the same sentence.² Written text tends to be more formal, and therefore the number of unfinished units and replacement elements tends to be greatly reduced.³

Tenor reflects the social relationship between the performer and the addressees (ie the speaker/listener and the writer/readers). For example, an aerospace engineer familiar with the configuration of a particular piece of equipment will speak differently to a colleague who shares his knowledge from the way he would talk to an apprentice who is not yet familiar with the equipment. Similarly, one would speak differently to one's best friend from how one would speak to one's boss.

Quirk et al (1985:27) use a scale to classify the tenor of a text as follows:

very formal <-> formal <-> neutral <-> informal <-> very informal

While **register** variation is variation according to the context of situation, variation in **dialect** is variation according to who the performer is. Here the main variables are (a) geographical, (b) social class and (c) time. However, most modern corpora record texts that are in modern Standard English, with the main variable being the distinction between American English and British English, so that the dialectal dimension of variation is much less important in corpus studies than register.

A further important variable between corpora is their size. This affects the reliability of the probabilities that can be extracted from it. A small corpus may give undue weight to rare words or structures because they have occurred a few times in a relatively small sample; such risks tend to be evened out in a larger sample.

Ideally, because the register and dialect variables typically affect probabilities, the ideal parser would be able to draw on statistics that are based on the type of text being parsed (i.e. by extracting information from specialised corpora or sub-corpora). In

² Approximately 2% of sentences in the Fawcett-Perkins-Day corpus contain replacement structures, an example being **Do some flappers, door flappers (10abpstg#208)**.

³ An unfinished unit is common in speech. They occur when a speaker has started an utterance and either has replaced it with another construct or has been interrupted.

order to do this, corpora need to be collected, classified and annotated according to these criteria.

The larger corpora available today contain texts from a variety of sources and they represent a variety of linguistic variables. However while many corpora classify texts in broad terms according to register variables (e.g. Biber et al 1999), most corpora are strongly biased to written texts, because these are easier to obtain.

3.2 A survey of some of the major corpora

This section describes some of the main corpora that are currently available or that have special relevance to this current project. Table 3.1 provides a summary of the details of these corpora.

3.2.1 COBUILD

The Collins Birmingham University International Language Database (COBUILD), was (and is) based in Birmingham University, and was led by Sinclair and funded by Collins (Sinclair 1990). The outcome was the Bank of English corpus.

In 2006, the corpus contained over 524 million words of English texts. It was created from British, American and Canadian sources and includes text books, novels, newspapers, guides, magazines and web-sites. It is a 'dynamic corpus' as it is continuing to grow as new texts are added to it.

The COBUILD linguists and lexicographers use specially developed software tools to undertake research in the Bank of English corpus. This has resulted in the Collins COBUILD dictionaries and a series of derived reference books. A notable feature of these dictionaries is that their entries are always supported by the evidence of extracts from the corpus.

Jarvinen, working under the direction of Sinclair (in Birmingham) and Karlsson (in Helsinki), annotated the corpus with part-of-speech tags, using a combination of the English morphological analyser (ENGTWOL) and the English Constraint Grammar (ENGCG) parser (Jarvinen 1994). Similar approaches are commonly used to transform raw corpora into more useful tagged corpora. Although more recent taggers are demonstrating greater accuracy, their analyses are still not completely accurate. Many will mark words as 'unclassified' and / or assign multiple categories to the same item. This is true, for example, of the British National Corpus (BNC) of which extensive use is made in the project described in this thesis. The BNC corpus is described in the next section.

3.2.2 The British National Corpus (BNC)

The BNC (www.natcorp.ox.ac.uk/corpus), which was compiled between 1991 and 1994, contains ninety million words of written text and ten million words of spoken text. It is a 'balanced' corpus, in that it is designed to represent a wide cross-section of British English from the later part of the 20th Century.

The written part of the corpus includes extracts from local and national newspapers, periodicals and journals, academic books and fictional works, letters and memoranda, and school and university essays. The spoken part contains a large amount of informal conversation from volunteer speakers of different ages, social classes and situations. Also included is a range of formal spoken texts from, for example, meetings, radio shows and 'phone-ins.

The BNC is encoded in the Standard Generalized Mark-up Language (SGML) (ISO 1986), according to the guidelines of the Text-Encoding Initiative (TEI) (see Chapter Six). A word tagger called CLAWS (see Section 3.3), has been used to provide annotations for part-of-speech.

The BNC is **monolingual** in that it contains modern British English. It is synchronic as it covers British English from the late twentieth century, and it is **general** in that it contains many different styles and varieties and is not limited to subject, genre, field or register, and contains both spoken and written texts. It is also said to be a **sample** corpus as its words are taken from various parts of single-author or speaker texts.

Two sub-corpora have been produced from the BNC since 1994: the BNC Baby and the BNC World. Versions of these sub-corpora have been produced in the eXtensible Markup Language (XML) format (W3C 2004a) and can be used with the XARA corpus query tool (see Section 3.4.3).

3.2.3 The Penn Treebank

The Penn **Treebank**⁴ project, located at the University of Pennsylvania under the direction of Marcus (Marcus et al 1993), provides a corpus that contains over 4.5 million words of American English. This includes the Brown Corpus and the Wall Street Journal Corpus, among others.

The Penn Treebank has been annotated using part-of-speech information, and a large proportion of it has been analysed and annotated with skeletal syntactic structure. The part-of-speech tagging was performed automatically using a reduced Brown

corpus tag-set. Marcus et al (1993:2) argue that many of the Brown tags are not necessary as their meanings can be lexically recovered or, because the treebank is syntactically analysed, this can be recovered from the syntactic context. Early part-of-speech tagging was performed by AT & T's PARTS algorithm (Church 1988), but subsequently this was replaced by other tagging software that reduces the error rate to 2-6%. The automatic tagging, however, was followed by a manual correction stage.

Higher level syntactic structure labels (for example, noun phrases and verb phrases) which were based on those of the Lancaster treebank project, were added by a parsing process; this again was followed by a manual check. The parser, named 'Fidditch' was developed by Hindle (Marcus et al 1993).

The syntactic structures, which have been described as skeletal (Marcus et al 1993), are presented in bracketed form and provide considerably less information than the functional syntax used in this present project, as described in Chapter Four.

3.2.4 The SUSANNE Corpus

The Surface and Underlying Structural ANalyses of Naturalistic English (SUSANNE) parsed corpus has been created with the sponsorship of the Economic and Social Research Council (UK) as part of the process of developing a comprehensive NLP-oriented taxonomy and annotation scheme for the grammar of English (Sampson 1995). The corpus, which is a 128,000 word subset of the Brown Corpus of American English, is annotated with the SUSANNE analytic scheme.

3.2.5 The Lancaster-Oslo/Bergen (LOB) Corpus

The LOB corpus contains 1 million words of British written English dating from 1960 (www.comp.lancs.ac.uk/computing/research/ucrel/corpora). It comprises 15 genre categories, and is tagged with the CLAWS part-of-speech tagger. The Leeds-Lancaster and the Lancaster Parsed Corpora were derived from the LOB corpus. A sample sentence tagged using the CLAWS scheme is shown in Figure 3.1 (note that parts of speech tokens are concatenated to the items).

```
hospitality_NN is_BEZ an_AT excellent_JJ virtue_NN ,, but_CC not_XNOT  
when_WEB the_ATI guests_NNS have_HV to_TO sleep_VB in_IN rows_NNS  
in_IN the_ATI cellar_NN !_!
```

Figure 3.1: Example of CLAWS tagging used in the Leeds-Lancaster Corpus⁵

⁴ A **treebank** is term given a large collection of parsed trees.

⁵ Parts of speech are shown in bold in this example to help the reader; the corpus does not contain this formatting and is ASCII text. A key to the CLAWS tags can be found at <http://www.comp.lancs.ac.uk/computing/research/ucrel/claws1tags.html>.

3.2.6 The International Corpus of English (ICE)

The key concepts behind the International Corpus of English (ICE) (www.ucl.ac.uk/english-usage) project were created by Greenbaum in the 1980s when he envisaged a project where corpora would be collected and analysed globally.

The ICE-GB corpus is the British component of ICE. The project, which began in 1990, contains over a million words of spoken and written English texts. The ICE-GB corpus is fully grammatically analysed, with approximately 83,000 parse trees.

The corpus was automatically annotated in three distinct stages (www.ucl.ac.uk/english-usage). The first stage, following text collection, involved marking the boundaries of sentences, paragraphs and headings. For spoken texts, this involved identifying pauses, discourse markings, false starts, hesitations and speaker turns. The second stage involved marking parts of speech using the TOSCA tagger developed by the TOSCA Research Group at the University of Nijmegen. The third stage involved using a parser to mark the higher syntactic levels of the parse tree.

The annotation scheme used in the corpus was based on that of Quirk et al (1985). The corpus is stored in such a way that it can be edited and queried using the ICECUP corpus query tool (see Section 3.4.4).

3.2.7 The SWITCHBOARD Corpus

The SWITCHBOARD corpus (Godfrey et al 1992) contains spontaneous conversations which were collected at Texas Instruments in a project funded by DARPA. It has 2430 conversations and 3 million words from 500 speakers of both sexes from every major dialect of American English.

Approximately a third of the SWITCHBOARD corpus is provided in parsed form within the PENN Treebank (see Section 3.2.3). Calhoun et al (2005) took a proportion of the corpus and added discourse annotations using an XML schema.

3.2.8 The NEGRA Corpus

The NEGRA corpus contains approximately one million words of German newspaper texts. The corpus can be queried using the TigerSearch corpus query tool. It is annotated using XML, and the approach can be compared to the method that we use for this project in Chapter Six.

3.2.9 The Fawcett-Perkins-Day Corpus

The Fawcett-Perkins-Day Corpus (FPD), which will be described more fully in Chapter Nine, is a revised version of the Polytechnic of Wales (POW) Corpus. This was first developed by Fawcett and Perkins in the 1980s (Fawcett and Perkins 1980), and it was one of the first corpora to be fully analysed in terms of its syntax. A powerful corpus query tool that was used with the POW Corpus was provided by Day (1993a); this work led ultimately to the enhanced tool used in this project and operating on the FPD Corpus as described in Chapter Eight.

The corpus consists of spoken texts taken from 120 children from South Wales arranged in groups of three according to their age, sex and social class. Each group was recorded in two types of situation: (a) as they worked together on a building task, using LEGO, and (b) in conversation with an adult about the building task, a game they liked to play, a television programme, a film, or something that they would like to do in the future. The recordings were then (a) manually transcribed, and (b) syntactically analysed. Each of these tasks brought its own range of problems to be overcome and together they required several years of work by Fawcett, Perkins and a team of sixteen transcribers. Despite a rigorous regime for checking and correcting errors in the analysis, a surprising number of mistaken analyses were found in the course of the present project.

Electronic copies of the analyses were made available as ASCII text files with a header (which we shall refer to as the **cell identifier**). This identifies the child (in terms of age, sex, class and type of situation). In Figure 3.2, **10dgi sm** shows that the child is ten years old, belongs to social group **D**, is female and has the initials **I . S . M**. The annotation method includes provision for discontinuous units, which is rare in such corpora.

Each sentence has a sentence number which is followed by a series of groups of tokens, each of which is preceded by an identifier which represents the group's parent; if no identifier appears before the token, then the token before it in the group is its parent. In sentence 1 in Figure 3.2, the root element is a sentence (**Z**), which is filled by a clause (**CL**).⁶ All 'children' nodes of the **CL** will have a **1** before them i.e. an Adjunct (**A**), a Subject (**S**), a Main Verb (**M**) and a Complement (**C**). The **A** also has a **CL** filling it and children of that **CL** are identified by a **2** before them, the elements

being a Binder (**B**) followed by a subject (**S**) a Main Verb (**M**), and a Complement (**C**). Ellipted elements are indicated by angled brackets <,> and round brackets (,) which are omissions due to rapid speech and recoverable from the previous text respectively.

```

**** 44 10 3 8 1 52
10dgism
1 [HZ:WELL] Z CL 1 A CL 2 B WHEN 2 S NGP HP YOU 2 M WENT 2 C PGP 3 P OUT-OF
  3 CV NGP 4 DD THE 4 H ROOM 1 S NGP HP WE 1 M DECIDED 1 C CL 5 I TO 5 M
  BUILD 5 C NGP 6 DQ AN [NA] 6 H HOUSE
3 [NV:EM] [FS:ABOUT] Z 1 CL 2 C? NGP 3 DQ A 3 MO QQGP AX BIG 3 H FAMILY 2
  CREPL? NGP 4 DQ QQGP 5 T ABOUT 5 AX SEVEN 4 H CHILDREN 1 CLUN & AND
4 Z CL F YES
5 [HZ:WELL] Z CL 1 S NGP HP WE 1 OM WOULD 1 X HAVE [NV:EM] [RP:WE-WOULD-HAVE]
  1 M BUILT 1 C NGP 2 DQ A 2 MO QQGP AX LITTLE 2 H SWIMMING-POOL

6 Z CL 1 & AND [NV:EM] 1 S NGP HP WE 1 OM WOULD 1 X HAVE 1 M BUILT 1 C 2 NGP
  3 DQ A 3 MO QQGP AX LITTLE 3 H SLIDE 2 NGP 4 & AND [RP:AND] 4 <DQ> 4 H SWING
  2 NGP 5 & AND 5 DQ A 5 H SEE-SAW 2 NGP 6 & AND 6 DQ A 6 H ROUNDABOUT

```

Figure 3.2: An excerpt from the Polytechnic of Wales Corpus

This valuable corpus can be queried with the Interactive Corpus Query Facility (Day 1993a) (see also Chapter Eight). However, it has been modified so that it is expressed in terms of the latest version of the Cardiff Grammar, since then it has been renamed as the Fawcett-Perkins-Day (FPD) corpus. For a fuller description of the creation of the new version, see Chapter Nine.

⁶ Note that in the original POW Corpus, there was a restriction that the whole file was in uppercase, due to the computer environment used. However, in the FPD Corpus, we have introduced lowercase where appropriate following the conventions summarised in Fawcett (2000a).

Name	Description	Type	Classification	Size (words unless otherwise stated)	Annotation scheme	Remarks	Reference
COBUILD Bank of English (see Section 3.2.1)	Mainly written. English texts from Britain, Australia, Canada includes books, novels, newspapers, guides and web-sites.	tagged	General, mono-lingual	524,000,000			www.collins.co.uk
British National Corpus (BNC) (see Section 3.2.2)	Written (90%): Extracts from newspapers, periodicals and journals, school and university essays. Spoken (10%): (a) informal conversations from speakers of different ages and social class, (b) formal spoken texts from meetings, radio 'phone-ins etc.	tagged	General, mono-lingual	100,000	TEI	Corpus Query Tool: SARA / XARA	www.natcorp.ox.ac.uk
PENN Treebank (see Section 3.2.3)	Mainly written. American English. Large range of sources including IBM manuals, agricultural documentation, novels from various authors etc.	50% ⁷ skeletal-parsed 50% tagged	General, mono-lingual	4,500,000	bracketed tree / PSG	Corpus Query Tool: TGrep	www.cis.upenn.edu/~treebank/home.html

⁷ The proportion of tagged and parsed data for the PENN treebank were reported by Marcus et al (1993).

Name	Description	Type	Classification	Size (words unless otherwise stated)	Annotation scheme	Remarks	Reference
SUSANNE (see Section 3.2.4)	Written. Subset of the Brown Corpus which has been parsed to the SUSANNE scheme.	parsed	General, mono-lingual	128,000	SUSANNE scheme		eg www.grsampson.net/RSue.html
SWITCHBOARD (see Section 3.2.7)	Spoken. contains 2,400 telephone conversations.		Specialised, mono-lingual	3,000,000 (2,400 telephone conversations)			
NEGRA (see Section 3.2.8)	Written. German newspaper texts.		General. mono-lingual (German)	20,602 sentences	XML / PSG	Corpus query tool: TigerSearch	http://www.coli.uni-saarland.de/projects/sfb378/negra-corpus/
Fawcett and Perkins (Polytechnic of Wales) (see Section 3.2.9)	Spoken. Children's spoken text. Ages 6, 8, 10 and 12 of three social classes in play and interview situations.	parsed	Specialised, mono-lingual	67,000	Numerical tree / SFG	Corpus query tool: ICQF (see Chapter Eight)	

Name	Description	Type	Classification	Size (words unless otherwise stated)	Annotation scheme	Remarks	Reference
Fawcett, Perkins and Day (see Section 3.2.9 and Chapter Nine)	Spoken. Updated version of the POW corpus. Annotated in XML and to the latest Cardiff Grammar. See Chapter Nine.	parsed	Specialised, mono-lingual	67,000	XML / SFG	Corpus query tool: ICQF+ (see Chapter Eight)	
International Corpus of English (ICE) (see Section 3.2.6)	Spoken: direct conversations, telephone calls, classroom lessons, broadcasts etc. Written: Essays, business letters, academic writing, fiction.	parsed	General, mono-lingual	1,000,000		Corpus query tool: ICECUP	http://www.ucl.ac.uk/english-usage/projects/ice.htm
Lancaster Oslo/Bergen (LOB) (see Section 3.2.5)	Written: dating from 1960s of various genres.	tagged	General, mono-lingual	1,000,000	ASCII (word tags appear after words)		
Lampeter Corpus of Early Modern English Tracts	Written: words of English pamphlet literature covering the years 1640-1740.	partly tagged	Historical, mono-lingual	1,000,000			
Lancaster / Leeds Treebank	Written: parsed subset of LOB corpus.	parsed	General, mono-lingual	45,000			

Table 3.1: A summary of some of the major corpora

3.3 Part-of-speech taggers

Many part-of-speech taggers have been produced for use in corpus linguistics. They typically use Markov models to calculate the most likely part of speech for a particular word in a given input sentence. Perhaps the most well known of these are: (a) CLAWS and (b) the BRILL tagger.

The Constituent Likelihood Automatic Word-tagging System (CLAWS) (Garside et al 1987) (Leech et al 1994) was developed by the University Centre for Computer Corpus Research on Language (UCREL) at Lancaster University. A 96-97% accuracy has been claimed for it, when used to annotate the BNC corpus (www.comp.lancs.ac.uk/claws), and it has been fairly widely used to convert raw corpora into tagged corpora.

Eric Brill (1992) developed a tagger that works by automatically recognising its weaknesses and thereby incrementally improving its performance, through a consultation of a large corpus. Brill also introduced the process of what he terms 'transformation-based, error-driven learning' and, apart from applying the techniques to his tagger, he also experimented with the method in a parsing process (Brill 1995). The technique involves tagging a text into an initial state and then comparing it with another text that is known to be accurate (called 'the truth'). The results of the initial attempt can then be compared to 'the truth' and a set of transformation rules can be established to help the tagger achieve results that are closer to 'the truth'. Using these techniques, Brill claimed success rates of 97%.

3.4 Corpus query tools

In this section, we will contrast and compare the corpus query tools that are available today. We start by looking at the typical query and presentation requirements that some of these tools produce.

The types of query that are useful to researchers in language depend on the nature of the work. Lists of words together with examples of use and frequencies are very useful when investigating word uses. The COBUILD team use outputs from the Bank of English to compile the Collins COBUILD suite of books. These range from dictionaries, and thesauruses through to books on usage intended for students of English as a second language. Section 3.4.1 provides a list of typical reports that are of use in such work. Concordances are another important tool for the researcher as they provide examples of word usage together with contexts.

Tools that operate on **parsed corpora** can provide some valuable additional information to help a linguist in determining the nature of the text and in a study of syntax. Queries that involve the higher levels of syntax can tell the linguist more about the nature of texts and the functions particular structures perform. This is denominated by the work presented here and by others in developing either the 'rules' for a rule-based parser, or for producing a probabilistic model for use in the parsing process.⁸

Example lists are provided in Section 3.4.1.

3.4.1 Products relating primarily to items

We will start by looking at the report presentation formats.

3.4.1.1 Word lists

Sinclair (1991) suggests that frequency lists act as a quick guide to the way words are distributed in a text. By examining such lists, one can often typically gain an indication of what further questions are worth asking. An alphabetically sorted list of words and their frequencies not only provides useful information about any particular word, it also is useful, for example, when comparing one text with another. In general, Sinclair notes that the most frequent items tend to have a stable distribution across different text types and any change between texts is likely to be significant and suggests the need for further study.

When extracting such lists from a parsed or tagged corpus, there is also the opportunity to make separate entries for each part-of-speech (i.e. element of structure) that the item expounds. A similar list, called the **item-up-to-element table**, is used extensively by the parser, which is the subject of this thesis (see Part Four, Chapter Fourteen).

As an example, consider the list shown in Figure 3.3. This shows the frequencies of the functions of the item **the** distinguished in the Cardiff Grammar, as they are found in the FPD corpus. It also shows the two functions served by the item **them**. Such report formats are provided in ICQF; see Day (1993a) and Chapter Eight.

⁸ For example, Weerasinghe used ICQF (Day 1993a) in the development of his parser.

Item	Total Freq.	Element	Freq./Elem.	Probability
the	2228	dd (deictic determiner)	2165	97.17%
		qld (quality group determiner)	50	2.24%
		qtd (quantity group determiner)	13	0.59%
...				
them	331	dd (deictic determiner)	35	10.57%
		h_p (pronoun head)	296	89.42%

Figure 3.3: Extract from the item-up-to-element table (from the FPD Corpus)

3.4.1.2 Concordances and collocations

We turn now to the type of output from a corpus query tool known as a **concordance**. A concordance is a set of occurrences of a word or item that is shown within its textual context. Concordances have been very widely used in the study of language. Long before computer corpora were available, concordances had been compiled and used for many purposes. These include the identification and confirmation of authorship of a work and the compilation of dictionaries as well as other types of research.

A **concordancer** is a software tool that searches a corpus and returns a concordance for a given target item (or **key-word**). Some packages, for example, the COBUILD concordancer, offer sophisticated search tools. In its simplest form, the target item is a single word, while in more advanced concordances, it could be a set of words. For example, the user could be interested in all forms of a verb (**sing, sings, sang, sung** and **singing**), or occurrences of similar verbs (**run, ran, runs, running, walk, walked, walks, walking**) or semantic classes of words (**end, stop, terminate, halt, finish,...**).

The concept of a 'wildcard' is used widely in concordance searches. For example, such a concordancer would be able to find all occurrences of a word that contains certain letters: **make*** could return **make, makes, maker**. When wildcards are used to represent whole words, the search tool can provide the words that typically precede or follow the **key-word**. Such searches are called **collocations**.

Typically, the output from a concordance search comes in a format **called Key Word In Context (KWIC)**. Here the target item(s) are clearly displayed in the centre of the concordance as in Figure 3.4.

Some concordance queries allow the user to adjust the number of words or characters that are displayed to the left and right of the target item, while others simply

will print out all that will fit onto a screen or an A4 sheet either in 'portrait' or 'landscape' format.

didn't play it at-all I only had one	run	and it found it boring I
I'd like to	run	a stable
When all the monies have	run	out you just add your up
I 've	run	out of money
	Run	by a battery underneath
d the first pick the other team gets	run	out
sometimes we	run	about
n the middle of the yard and you got	run	across without him catch
(S) (OM) when some people	run	away and they look for y
ll around and if you drop it you got	run	around all the people a
(S) just	run	around
(M) (MEx) (C) til we	run	out of red
lace and (B) (S) say release you can	run	out again
ch-other and if you drop one you got	run	right round all the play
We	run	round 'n we pick daisies
and the bird	run	down the runway and flew
d you hold onto it and-then you must	run	
if they tried to	run	one side they couldn't g
and if somebody 's on it and you	run	all around if they kit
and-then he could	run	away see

Figure 3.4: KWIC style concordance (ordered by occurrence) from the FPD corpus

The ability to sort the sequence of entries in a concordance according to the words to the left or to the right of the target item is useful when investigating the colloquial relationships between the key-word and the text surrounding it (see Figure 3.5).

When performing a concordance on a parsed or tagged corpus, it is possible to confine the search to target words that expound a particular syntactic element and to indicate the syntax token next to the words in the concordance. It is further possible to restrict the concordance to a key-word (or key-words) that expound a particular syntax token. The next section shows how this concept can be taken even further.

Concordances, then, have a considerable value as an aid to those studying the grammar of English, and increasingly, other languages and they have contributed notably to the current version of the Cardiff Grammar (e.g. as described in Fawcett 2000a).

```

e nut? and they? onto them? and they      run
          I 'd like to      run  a stable
          Sometimes we     run  about
n the middle of the yard and you got      run  across without him catch
  and if somebody 's on it and you      run  all around if they kit y
didn't play it at-all I only had one     run  and it found it boring a
          (S) just          run  around
ll around and if you drop it you got      run  around all the people a
  (S) (OM) when some people      run  away and they look for y
          and-then he could  run  away see
          and the bird      run  by battery underneath
          if they tried to  run  down 'e runway and flew
d the first pick the other team gets     run  one side they couldn't g
lace and (B) (S) say release you can     run  out
          I've              run  out again
          (M) (MEx) (C) til we  run  out of money
          (M) (MEx) (C) til we  run  out of red

```

Figure 3.5: KWIC style concordance ordered by word to the right

3.4.2 Products relating to syntax

This section describes a number of useful types of table that can be extracted from a **parsed corpus**, where it is possible to extract probabilities for the relationships between syntactic categories. These tables are equally as valuable to the researcher as those lists based on items presented in Section 3.4.1. They play a central role in the operation of the parser developed for this work (see Part Four, Chapter Fourteen).

3.4.2.1 Unit-up-to-element tables (U2E)

A table can be provided from a parsed corpus that shows which elements are most likely to be filled by a given class of unit.⁹ An example of this type of list is shown in Figure 3.6 which shows that, for example, in the FPD Corpus 40.64% of all nominal groups fill a Subject (S) and 32.08% fill a Complement (C).

Unit: ngp - nominal group (occurs 22248)			
Fills		Frequency	Probability
S	Subject	8882	40.64%
C	Complement	7012	32.08%
cv	completive	2737	12.52%
A	Adjunct	681	3.11%
Voc	Vocative	375	1.66%
...

Figure 3.6: Unit-up-to-element table for the nominal group

⁹ It is also possible to create a list that shows the units that can fill a particular element; this type of list is not shown in this chapter as it is not relevant to this work.

3.4.2.2 Element-up-to-unit tables (E2U)

The second type of table is simply a list of units in which a particular element of structure appears. In the Cardiff Grammar, almost all of the elements of structure occur in only one unit so the information provided is limited.¹⁰ Figure 3.7 shows an example of this type of table for one element that can occur in more than one class of unit, the linker (**Lnk**), and shows that in the FPD Corpus 84.48% of them occur as elements in a Clause (**C1**).

Element: Lnk - Linker (occurs 2351)			
In unit		Frequency	Probability
Cl	Clause	1829	84.48%
ngp	nominal group	301	13.74%
qlgp	quality group	33	1.40%
pgp	prepositional group	9	0.38%

Figure 3.7: Element-up-to-unit table for linker

3.4.3 Tools for querying part-of-speech tagged corpora

The British National Corpus provides a corpus query tool that allows the search and retrieval of concordances and collocations. The SGML Aware Search Application (SARA) can be used on the BNC World corpus. A new XML version (XARA) is now available.

COBUILD comes with a corpus query tool that allows the researcher to generate concordances and collocations. These can be based on words, or combinations of words (e.g. the query, **dog+4bark** returns all instances of **dog** plus **bark** with up to four words between them). Inflected forms of words can be retrieved (e.g. **blew@+away** will return **blew away**, **blown away**, **blows away** and **blowing away**). Sets of words can be strung together (e.g. **cut | cuts | cutting**). It can also include part-of-speech tags such as 'noun'.

3.4.4 Tools for querying parsed corpora

ICQF (Day 1993a) was one of the first parsed corpus query tools that provided both a graphical interface and a powerful query language. It has been developed for research on the Polytechnic of Wales corpus in the Universities of Cardiff, Leeds and Sheffield.

¹⁰ Nonetheless, when used in the parser described in Part Four, this table is vital to its efficient operation.

However, since 1993, a number of other parsed corpora have been developed and with them a number of other corpus query tools have emerged.

One corpus query tool is TGREP2 (Rohde 2005). This is a set of UNIX based tools that were developed at Stanford University, and it allows the user to extract parse trees from syntactically parsed or part-of-speech tagged corpora that match a given tree pattern. TGREP, which takes its name from the well-known UNIX pattern matching utility, has its own corpus query language, but does not provide a graphical user interface. One limitation of TGREP is that a corpus needs to be annotated in the Penn Treebank notation (see Section 3.2.3 for an account of the Penn Treebank).

Like TGREP, CorpusSearch (Randall, 2000 and Taylor 2003) is a command line tool that does not provide a graphical interface, and that works with the Penn Treebank notation. It also contains a powerful query language.

VIQTORYA (a **VI**sual **Q**uery **T**ool **f**OR **s**Yntactically **A**nnotated corpora) (Steinar and Kallmeyer 2002) is a corpus query tool for parsed corpora. It shares its design goals with ICQF in that it is designed to use any corpora (although it is acknowledged that some adaptation work is required).¹¹ VIQTORYA, which was developed to work with the Tübingen Treebank Corpus (Stegmann et al 2000), uses topological fields. VIQTORYA is able to handle grammars that may define some structures as not being trees, and like ICQF, it can handle tree structures with branches that cross and therefore may be able to record discontinuous units (see Section 4.3 of Chapter Four).

Like ICQF+, VIQTORYA stores its corpus data in a relational database and has a supporting query language which is translated into SQL statements; the main difference between the database structures is that ICQF stores data as XML and VIQTORYA has a proprietary schema.

TIGERSearch (König, Lezius and Voormann 2003) is interesting in that, like the work described here, it incorporates an XML format for its corpus annotation (TIGER-XML). It has a very powerful graphical query builder and a corpus query language. TIGERSearch was developed to work with the NEGRA corpus (see Section 3.2.8).

The ICE Corpus Utility Program (ICECUP) is a corpus query tool for the International Corpus of English (ICE) (see Section 3.2.6). It allows the user to perform searches on simple words, or to perform KWIC style concordances. Query

¹¹ It is acknowledged that similar adaptation work would be required for ICQF+ to be able to use a corpus other than the FPD Corpus.

restrictions are provided through ICE's corpus headers. The user can also search for tree structures.

This concludes our survey of corpus query tools. ICQF, as we shall see in Chapter Eight, is at least as sophisticated as any of these.

3.5 Summary

In this chapter, we have defined the field of Corpus Linguistics, and we have looked at the different corpora that are available, and how they are annotated, and the tools that are available to query them.

We have seen why, for our work in creating an SFG parser, the most suitable corpus is the revised version of the Polytechnic of Wales (POW) Corpus - known as the Fawcett-Perkins-Day (FPD) Corpus. This was created for use in this project and it is annotated in SFG terms using the latest version of the Cardiff Grammar. The conversion work was performed because the version of SFG in the POW Corpus is an earlier version of the Cardiff Grammar, and we wanted to benefit from the improvements to Fawcett's model that have been implemented since the 1980s (which are useful for the parsing process). We also needed to make our parser compatible with the other COMMUNAL components. These are the primary reasons why Fawcett and I have modified the POW Corpus so that it is now modified and annotated to conform to the current version of the Cardiff Grammar (as defined in the next chapter). The major task of converting the POW Corpus to the latest version of the Cardiff Grammar in the FPD Corpus is described in Chapter Nine, and as we shall see, this was a complex task which involved both structural and annotation changes

The next chapter, Chapter Four, concludes Part One by introducing you to the categories and relationships of the syntax of the Cardiff Grammar (as used in the FPD Corpus).

Chapter Four

The syntax of the Cardiff Grammar

As we saw in Chapter Two, the research reported in this thesis is set within the framework of the Cardiff Grammar. This is a Systemic Functional Grammar (SFG) as defined in Fawcett (2000a).¹

The purpose of this chapter is to provide the information about the syntax of the Cardiff Grammar that is needed to enable the reader to understand the chapters that follow.

In terms of Figure 2.3 of Chapter Two, the descriptions that follow apply to the **outputs** of the lexicogrammar. The reason for focussing on this part of the grammar, as we saw in Section 2.4 of Chapter Two, is that it is the equivalent of the syntax and items of a generative grammar that provide the output from a parser.

4.1 Definitions

Fawcett's theoretical model of syntax consists of two basic types of entity: **categories** and **relationships**. We will discuss the definitions of categories such as **unit**, **element** and **item** and describe the relationships between them.

4.1.1 Units, elements and items

The **class of unit**, usually referred to simply as a **unit**, is a key category in the Cardiff Grammar. There are seven different **classes of unit**, each of which has a potential internal structure made up of **elements of structure** (or simply **elements**). Each element may in turn either be **filled** by another unit (or by more than one **co-ordinated** unit), or be **expounded** directly by an **item**. An item may be a lexical ('vocabulary') item, such as a noun, or a grammatical item such as an article or part of the verb **be**.

4.1.2 The sentence element

The sentence is represented by an element of structure, but it is unique in that the unit in which it functions is not a syntactic unit, but a unit of discourse grammar. Fawcett (2000a) uses the symbol for Sigma, i.e. Σ , but in this project it is represented by **Z** (due to XML naming restrictions).

¹ The reader should note that the names of some of Fawcett's categories have been changed from those given in Fawcett (2000a). This is simply to avoid clashing with the naming conventions used in the eXtensible Mark-up Language (XML) (see Chapter Six).

4.1.3 Relationships

As introduced in Section 4.1.1, **filling** is the relationship between a **unit** and an **element**, such that a **unit** is said to **fill an element of structure**. **Componence** is the relationship between the parent **unit** and its constituent **elements of structure**. **Exponence** is the relationship between an **item** ('lexical' or 'grammatical') and the **element of structure** above it. When more than one unit fills an element, the units are in a relationship of **co-ordination**.

4.1.4 Unfinished units

Unfinished units are very common in spoken texts. They are not complete in that they lack their head elements (and possibly other elements too). They occur because of interrupted utterances. In the FPD Corpus they are represented by the unit name followed by **_un** (e.g. **C1_un** represents an unfinished clause).

4.1.5 Replacement elements

Replacement elements occur when the speaker wants to replace an element that has been uttered by a different set of words. In the Cardiff Grammar, these are represented by the element name followed by **Repl**. For example, **C_Repl** for the replacement complement in this example:

Trying to find a big one, a big long one²

4.1.6 Formalisms used in diagrams of syntax

The Cardiff Grammar uses the following conventions in its syntax diagrams. **Filling** between a unit and the element of structure above is shown by a line below the element and above the unit. Thus $\frac{S}{ngp}$ represents a nominal group (**ngp**) that fills a Subject (**S**), and $\frac{C}{ngp\ ngp}$ represents two co-ordinated nominal groups that fill a complement (**C**). Componence is represented by lines from the parent unit to the child elements of structure (e.g. $\begin{array}{c} C1 \\ / \quad \backslash \\ S \quad M \quad C \end{array}$ is a Clause (**C1**) with, as its component elements of structure, a Subject (**S**), Main Verb (**M**) and Complement (**C**)). Elements that are expounded by an item are shown with the element of structure at the apex of an 'exponence' triangle and the item at the base (e.g. $\begin{array}{c} h \\ \triangle \\ table \end{array}$ represents a head (**h**) expounded by **table**).

² This example is taken from the FPD Corpus (10abpstg#32).

Figure 4.1 provides a generalised tree diagram that shows (a) units, elements, and items and (b) their relationships. Figure 4.2 shows a sample sentence represented in these terms.

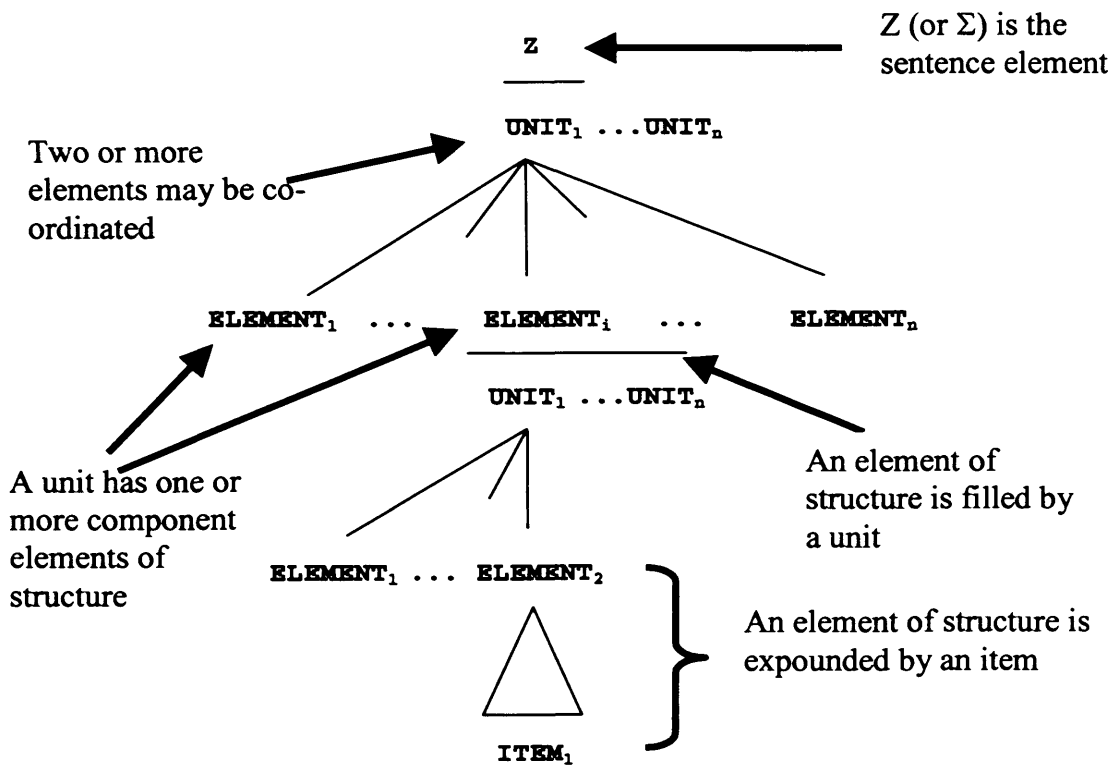


Figure 4.1: An abstract view of the structure of a Cardiff Grammar sentence

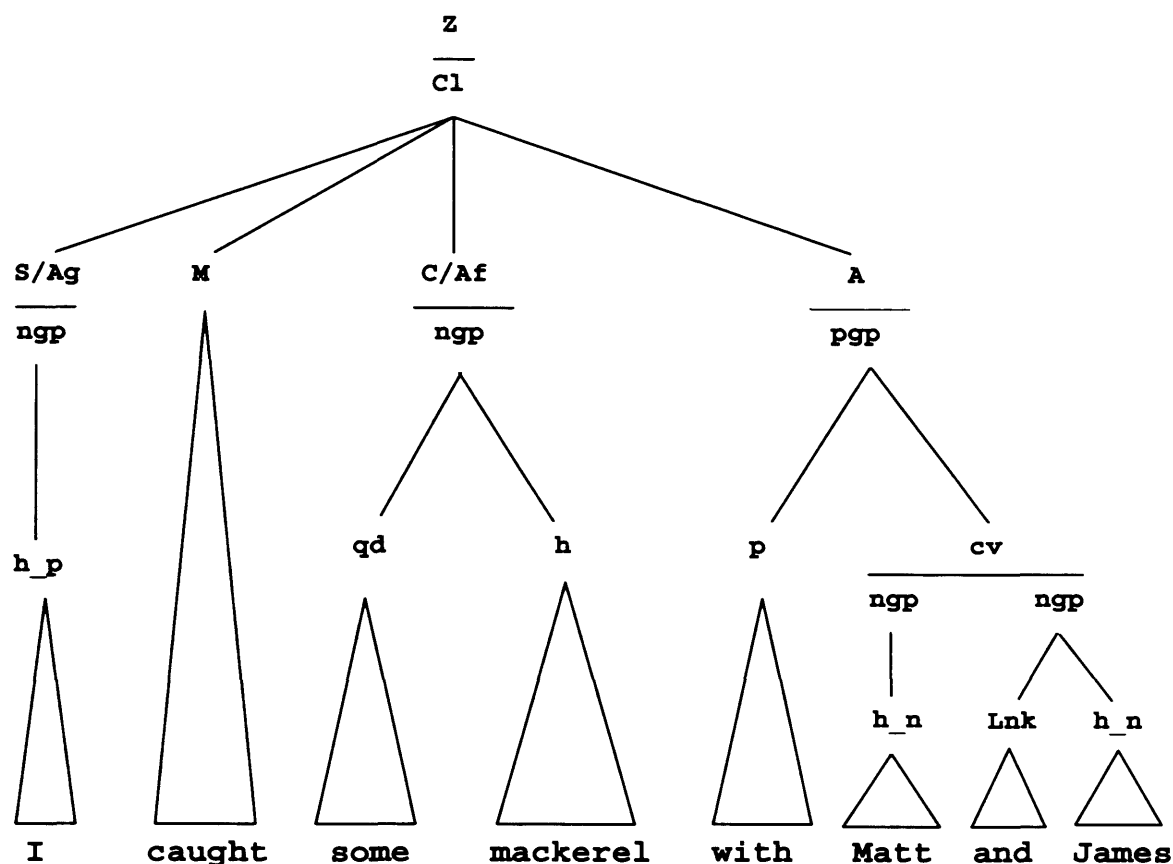


Figure 4.2: A simple sentence represented by a syntax diagram

4.2 The syntactic units of the Cardiff Grammar

This section will look at each unit in turn, and specify the elements that it can contain and their probabilities. The probabilities were derived from the FPD Corpus, using Day's Interactive Corpus Query Facility (ICQF), and also from other corpora, e.g. via the corpus-based grammars of Quirk et al (1985) and Biber et al (1999). These probabilities are summarised in Appendix A of Fawcett (2000a).

The main classes of unit in the Cardiff Grammar are as follows:

- (a) Clause (**Cl**),
- (b) nominal group (**ngp**),
- (c) prepositional group (**pgp**),
- (d) quality group (**qlgp**),
- (e) quantity group (**qtgp**),
- (f) genitive cluster (**genclr**), and
- (g) text (**TEXT**).³

³ There are other units i.e. the human proper name cluster and the address, date and clock time clusters (Fawcett 2000a:253), but these are not used in this present project.

4.2.1 The Clause (Cl)

4.2.1.1 Definition

Virtually all linguists agree that the clause is the central unit of English syntax. It corresponds (although not always in a one-to-one relationship) to Fawcett's term **situation**, as this is used in the semantic system network (see Figure 2.3 of Chapter Two), and to an **event** in the COMMUNAL belief system (Fawcett, Tucker and Lin 1993). The elements that occur most frequently in the main Clause are the Subject (**S**), the Main Verb (**M**) and one or more Complements (**C**). The clause can contain Adjuncts of different types, some of which are differentiated in the version of the Cardiff Grammar used in this project, such as Affective Adjunct (**A_Aff**) for **unfortunately**, **luckily** etc. Fawcett (2000a) suggests that there are over sixty different types of Adjunct. Auxiliary Verbs (**X**) and Main Verb Extensions (**ME_x**) are also common in a Clause. Linkers (**Lnk**)⁴ and Binders (**B**) associate one clause with another in a relationship of co-ordination or sub-ordination respectively. Operators (**O**), if they are present, normally occur before the Subject in an information-seeking clause and after it in an information-giving clause.

The Clause most commonly fills a sentence (**Z**); this occurs in about 85% of cases for all types of text. The probability that a Clause fills a Complement (**C**) is 7% and an Adjunct of any type (**A***), 4%. It can fill a qualifier (**q**) with a 2% probability, and a finisher (**fi**) 0.5% probability. It can also fill, with less than 0.5% probability, a scope (**sc**), a quantity group finisher (**qt_f**), a Subject (**S**), a Main Verb Extension (**ME_x**), a completive (**cv**) or a possessor (**po**).⁵ These are generalised probabilities across all text types, and in many cases there is considerable variation between text types.

4.2.1.2 Conflated elements in a clause

Elements of structure may be **conflated** with other elements, including **Participant Roles** (PRs) (see Section 4.2.1.3). An example of two conflated elements occurs when the Main Verb (**M**) is 'fused' with an Operator (**O**), as seen in the following example (Figure 4.3), where **is** expounds the conflated element **O/M**.

⁴ In Fawcett (2000a), linkers are named **&**.

⁵ The source of these figures is Fawcett (2000a); Fawcett used ICQF to help derive these values.

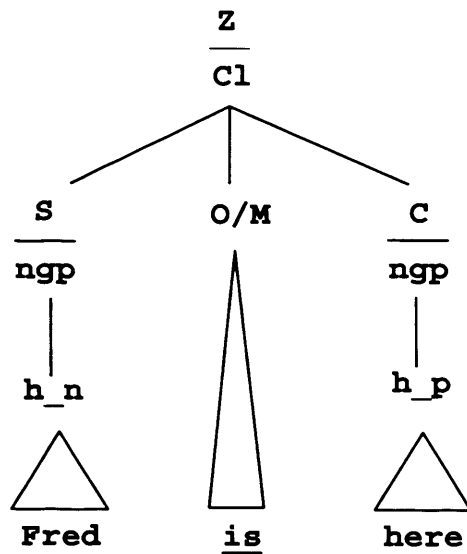


Figure 4.3: An example of a conflated element O/M

4.2.1.3 Participant Roles in the clause

Each process (expressed by the Main Verb (**M**)) has an associated configuration of **Participant Roles** (PR)s. These are conflated with the Subject (**S**) and Complement (**C**) of the clause. PR values include **Agent (Ag)**, **Affected (Af)**, **Carrier (Ca)** or **Attribute (At)**. Thus **S/Ag** represents a Subject which is also the Agent of the process, and **C/Af** represents the conflation of an Affected with a Complement. As a further example, consider Figure 4.2 of Section 4.1.6, which shows an Agent (**Ag**) that is conflated with a Subject (**S**) and an Affected (**Af**) that is conflated with a Complement (**C**).

4.2.2 The nominal group (ngp)

An example of the more complex type of nominal group is shown in Figure 4.4. This unit typically - but not invariably - has a **head**. The head is the 'pivotal element' in the unit, and in this project, we distinguish three types of head: these are expounded by (a) a noun (**h**), (b) a pronoun (**h_p**), and (c) a proper name (**h_n**).⁶ This, as we shall see in Chapter Fourteen, helps greatly in predicting what elements, if any, are likely to precede or follow the head of a nominal group, and so increases the efficiency of the parser.

the aardvark (*aardvark* is a (noun) head)

I ate mackerel (*I* is a pronoun head)

⁶ Note that an **h_n** may consist of more than one word, e.g. **Mr. Jones**, since the parser does not analyse the internal structure of a human proper name cluster.

Cardiff is the capital of Wales (*Cardiff* is a name head)

Often, the nominal group has one (or occasionally more) **determiners**. These are, in the order that they typically occur: typic determiner (**td**), representative determiner (**rd**), partitive determiner (**pd**), quantifying determiner (**qd**), fractionative determiner (**fd**), superlative determiner (**sd**), ordinative determiner (**od**), and deictic determiner (**dd**). These normally appear before any modifiers. Here are a few examples of nominal groups which contain determiners:

The strong bags (*the* is a deictic determiner)

The photo of a mackerel (*the photo* is a representative determiner)

The rim of the glass (*the rim* is a partitive determiner)

The best of the players (*the best* is a superlative determiner, and the second *the* is a deictic determiner)

Several **modifiers** (**mo**) can occur in the same nominal group, each being of a different type. They are typically filled by a quality group, but may also be filled by a nominal group, and even by 'truncated' clauses. The function of a modifier is to describe the referent of the nominal group, as in the following examples where **amazing**, **strong** and **red** are modifiers:

An amazing beer

The strong bags

The red shirts

Now we turn to the **qualifiers**. These serve the same describing function as the modifiers, but these usually contain more words and are typically filled by a prepositional group (**pgp**) or an embedded clause (**cl**). But can also be filled occasionally by a quality group (**qlgp**), or a nominal group (**ngp**). Here are some examples:

An amazing beer that I had in Hartlepool (an embedded clause)

The bags on the table (a prepositional group)

The red shirts worn by Bristol City (a truncated clause)

The last element of the nominal group to be introduced is the linker (**Lnk**). Its function is to attach a nominal group to a preceding nominal group in a co-ordination relationship, as in the following examples:

Matt and his friend (*Matt* and *his friend* are in different nominal groups, the linker *and* is the first element in the second nominal group)

A penny or a pound (*or* is a linker that co-ordinates two different nominal groups, and *or* is analysed as an element of the second one).

The main elements filled by a nominal group are the following, in order of probability: Subject (S) (45%), Complement (C) (32%), completive (cv) (15%), Adjunct (A) (3%) and modifier (mo) (2%).

However, nominal groups can also fill Main Verb Extensions (MEx), Vocatives (Voc), determiners of various types (especially *td*, *rd*, *pd* and *qd*), qualifiers (*q*) and possessives (*po*).

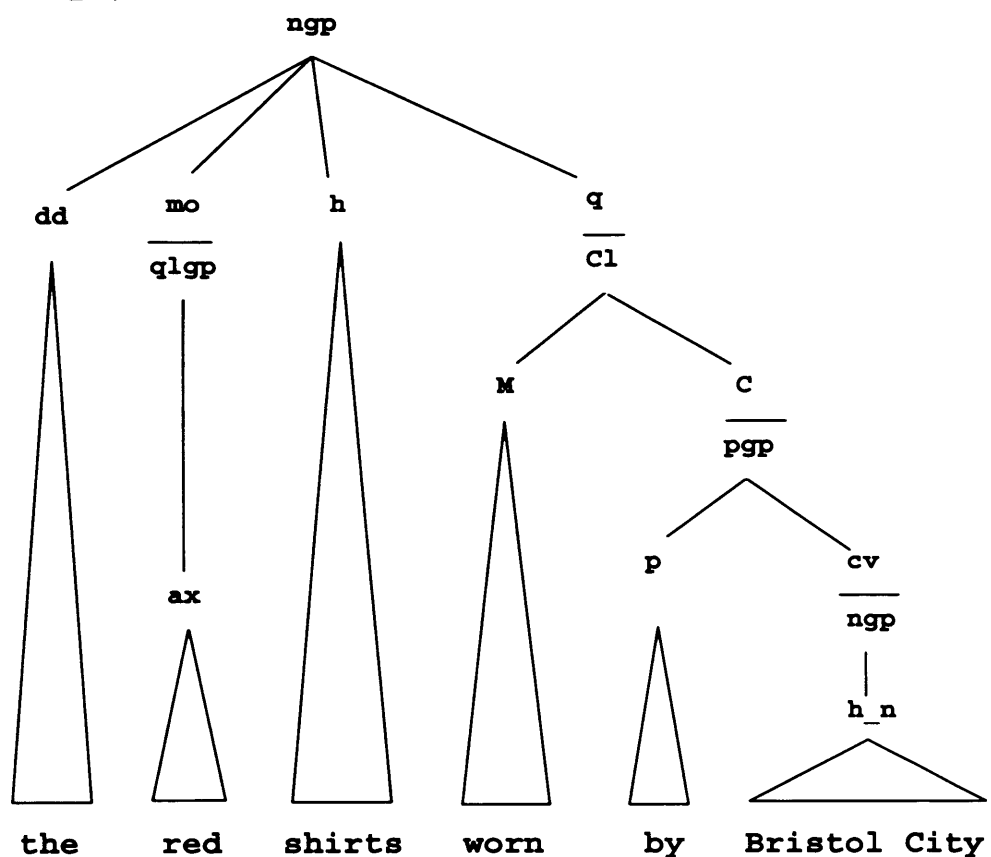


Figure 4.4: A nominal group with a modifier and a qualifier

For a fuller picture of the determiners in a nominal group, and the reasons for using this approach, see Fawcett (2007b) and a fuller of the nominal group as a whole see Fawcett (forthcoming 2007).

4.2.3 The prepositional group (pgp)

Figure 4.5 shows an example of a prepositional group (**pgp**). The pivotal element in the prepositional group is the preposition (**p**). This realises the meaning of 'minor relationship to a thing' (Fawcett 2000a). Most prepositional groups have only two elements: a preposition (**p**) (which is expounded, for example, by **from**, **to**, **in** and **with**) and a completive (**cv**), which is most likely to be filled by a nominal group. Examples of this common type of prepositional group are:

(I am going) to Cardiff
 (Charles fished) with feathers
 (The midfield play) behind the forward line

Sometimes the prepositional group takes a prepositional temperer (**pt**) before the preposition as in:

(the men were) up on the roof (up is a prepositional temperer)

In one fairly common case, the preposition follows the completive, and is, strictly speaking, a 'postposition' e.g.

(I saw her) two weeks ago (ago is a preposition)

The elements most commonly filled by a prepositional group are: Complement (**C**) (55%), Adjunct (**A**) (30%), qualifier (**q**) (12%) scope (**sc**) (2%). It can also fill a Main Verb Extension (**ME_x**), a Subject (**S**), a Completive (**cv**), or a finisher (**fi**).

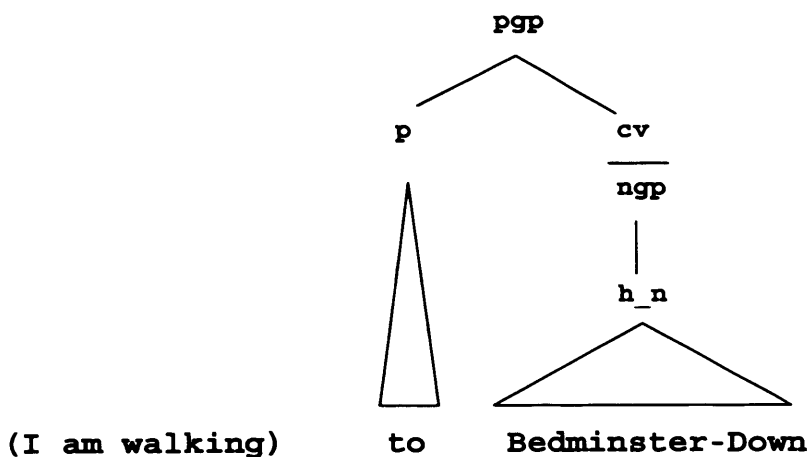


Figure 4.5: An example of a prepositional group

4.2.4 The quality group (qlgp)

Figure 4.6 shows an example of a quality group. The quality group (**qlgp**), which corresponds to the semantic unit of a 'quality', has an apex (**ax**) as the pivotal element

of the unit. Fawcett (2000a) points out that the quality expressed may be an adjective when it expresses the 'quality' of a 'thing', as in:

a very clever man (*clever* is an apex).

or it may be a manner adverb when it expresses the quality of a situation, as in:

he carefully retrieved the line (*carefully* is an apex).

Most quality groups contain just an apex, as in (**the clever man**), or an apex with a temperer, as in:

the very clever man (*very* is a temperer and *clever* is an apex).

In the superlative version of the group, it typically starts with a quality group deictic determiner (**qld**), as in:

the most important (of his reasons).

But occasionally, such structures also contain a quality group quantifier (**qlq**), as in:

the five most important (of his reasons).

Quality groups can contain a scope (**sc**) and / or a finisher (**fi**), e.g.:

more skillful at casting
more skillful at casting than I am

Quality groups typically fill these elements: Complement (**C**) (38%), a modifier (**mo**) (36%), an Adjunct (**A**) (24%), and a superlative determiner (**sd**) (0.5%). They can also fill Main Verb Extensions (**MEx**), Auxiliary Verb Extensions (**XEx**), ordinative determiners (**od**), qualifiers (**q**), temperers (**t**), prepositions (**p**) and, very rarely, Subjects (**S**).

In earlier versions of the Cardiff grammar, i.e. the version which the syntax of the Polytechnic of Wales (POW) Corpus used, the quality group and the quantity group were treated as variants of a single unit called the quantity-quality group (**QQGP**). The task of upgrading the corpus to reflect the recognition of the quality group as a unit in its own right was a major challenge, and is reported in Chapter Nine. See Fawcett (2000a) for a fuller description of the quality group.

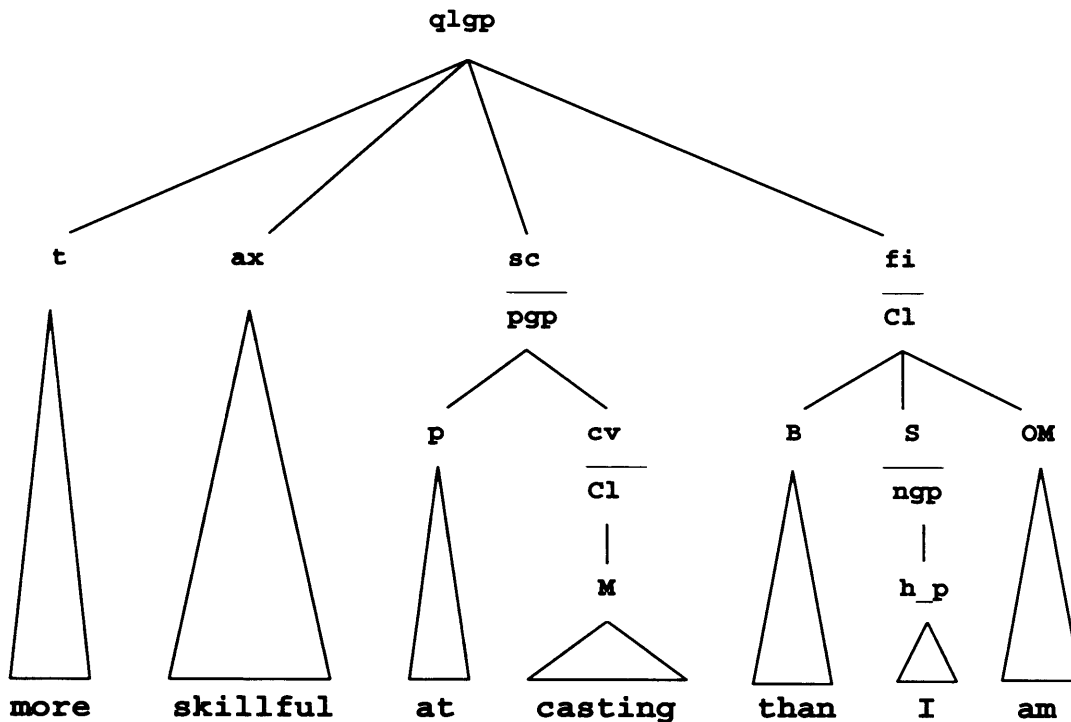


Figure 4.6: A quality group with a temperer, scope and finisher

4.2.5 The quantity group (qtgp)

The quantity group represents the semantic unit of 'quantity'. Fawcett (2000a) states that the quantity group is used to express the quantity of a thing, a situation, a quality or another quantity. Figure 4.7 shows an example of a quantity group.⁷

The head of the quantity group is the amount (**am**); it typically also has an adjustor (**ad**) as in **about sixty**. However, if **sixty** occurs on its own, it is shown directly expounding the element above, e.g. a quantifying determiner (**qd**). Sometimes, the quantity group has a finisher (**qtf**), as in **very much indeed**.

The quantity group most commonly fills these elements: a quantifying determiner (**qd**) (85%), an Adjunct (**A**) (8%), and a degree temperer (**dt**) (6%). It may occasionally also fill a binder (**B**), a preposition (**p**), an adjustor (**ad**), a fractionative determiner (**fd**), or a superlative determiner (**sd**).

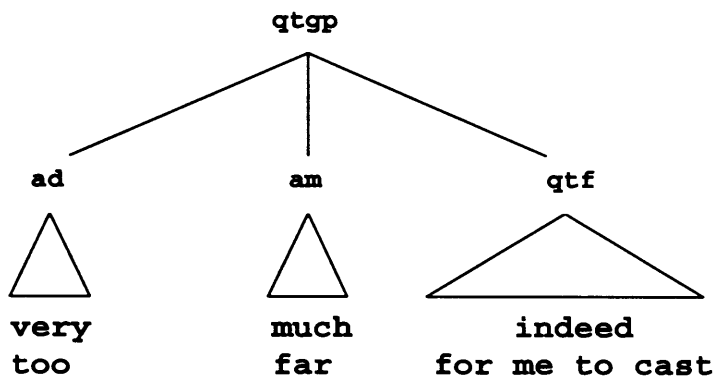


Figure 4.7: A quantity group with an adjuster and a finisher

4.2.6 The genitive cluster (genclr)

The genitive cluster realises the semantic relationship of 'possession', and it is most typically used in a deictic determiner (**dd**) (99%). But can also be found filling a head (**h**), a modifier (**mo**) or a determiner in a quality group (**qld**). Its elements are: a possessor (**po**), the genitive element (**g**), which represents the apostrophe, plus the letter **s** (**'s**). Occasionally, it also contains an owner element (**own**).

An example genitive cluster is shown in Figure 4.8. See Fawcett (2000a) for a fuller description of the genitive cluster.

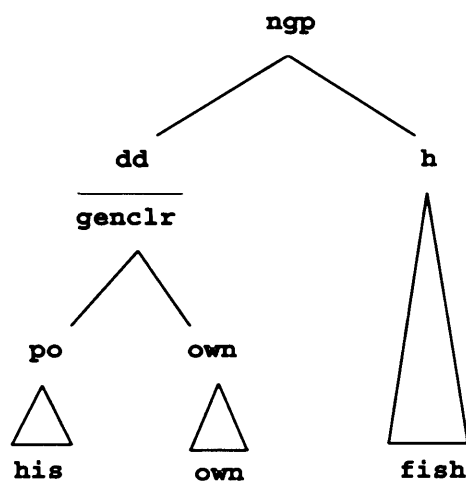


Figure 4.8: A genitive cluster

⁷ As we have already noted, the quantity group was merged with the quality group in the quality-quantity group (**QQGP**) in earlier versions of the Cardiff grammar.

4.2.7 Text

The text unit (**TEXT**) is used typically for passages of quoted text, as in:

"Feathers are the best way to catch mackerel" said the experienced fisherman.

A text, in this sense of the term, is an unusual type of unit, because it can only contain one type of element, i.e. the sentence element (**Z**). Occasionally, it may contain more than one sentence, as in:

The sign reads "No fishing with feathers is allowed on the pier. Offenders will have their tackle confiscated and will receive a £15 fine."

4.3 Places, potential structures and discontinuity

When COMMUNAL generates a **unit**, elements of structure are positioned at numbered locations within it, termed **places**. This, as Fawcett points out (2000a:224), enables it to handle certain types of what transformational grammarians call 'raising phenomena'.

A unit is discontinuous when its elements are interrupted in their position by other units. One of the most frequent types is illustrated in Figure 4.9, where the completive of the prepositional group comes first in the clause.⁸ Places are used to indicate when discontinuity occurs. The numbers represent positions in the Clause.

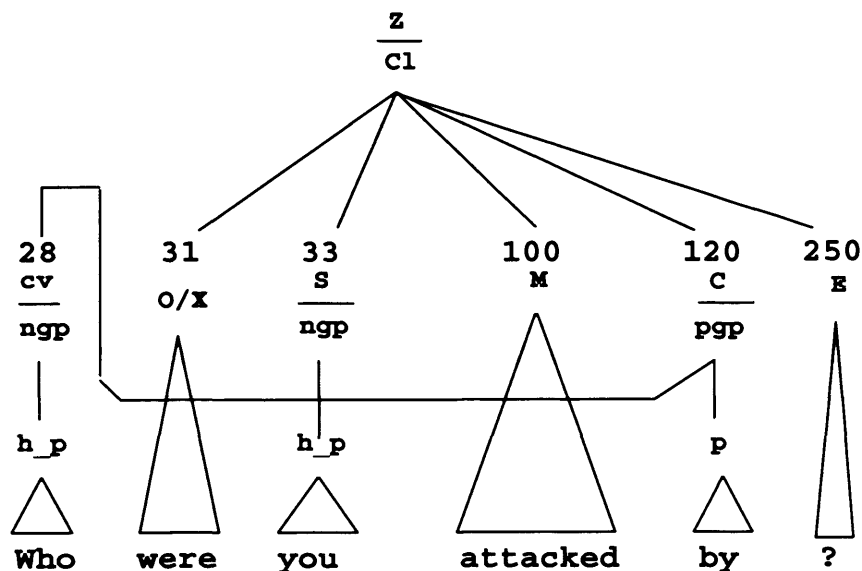


Figure 4.9: Showing discontinuity in the syntax diagram

⁸ Figure 4.9 shows a question mark which, like the full-stop and comma, when they end a unit, are defined as an ender element (**E**).

4.4 More on ellipsis

Elements of structure are said to be **ellipted** if they are expounded by an item that is missing from a previous part of the input sentence. This may be (a) because it is recoverable from a previous part of the text or (b) because it is missing due to rapid speech. In the syntax diagrams used in this project, ellipted elements are represented by round brackets () for rapid speech, and angle brackets < > for previous text. Figure 4.10 shows a sentence with ellipted elements; this sentence, which was a response to the question **Which country was that in?**, is taken from the FPD Corpus.⁹ The ellipted elements (S) and (O/M), if they were not ellipted, would have probably been expounded by the words **that was** or **it was**, and the ellipted preposition (p) by **in**.

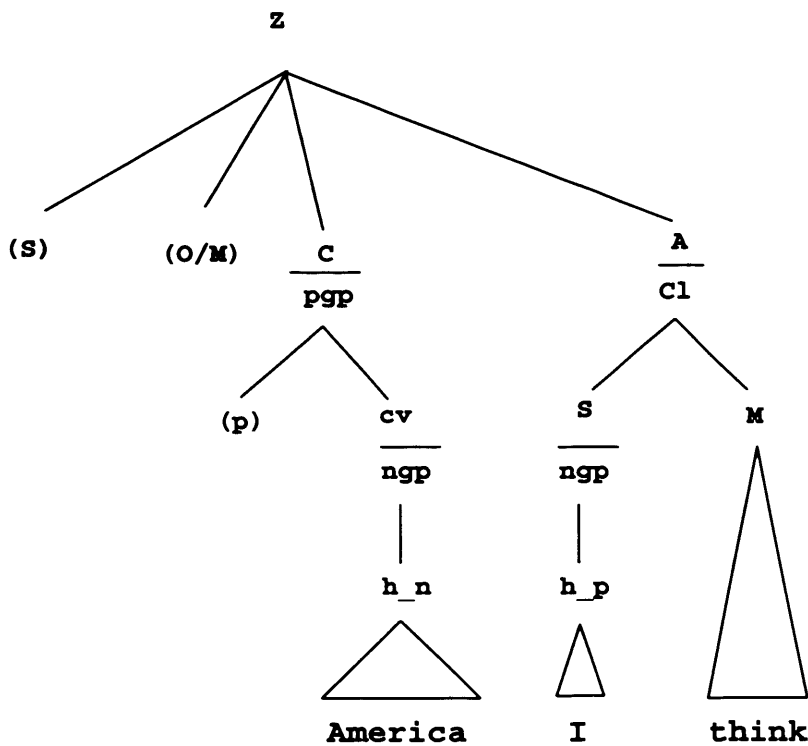


Figure 4.10: A sentence with ellipted elements (shown in brackets)

⁹ This sentence reference in the corpus is **9abiaw#25**.

4.5 Summary

This chapter has introduced the main categories of the Cardiff Grammar (units, elements and items) and shown the relationships between them. We have also looked at the related linguistic phenomena of discontinuity and ellipsis.

A full list of units and their elements of structure is given in Appendix A.

This concludes Part One. We have surveyed three major fields of study, and brought out the essential information about each that is necessary for the reader to understand the more detailed descriptions that follow in Parts Two, Three and Four.

In Part Two, we shall describe in some detail, the first of the two major components in the present project: the **corpus database**. In Part Three, we will discuss the field of natural language parsing in more detail and, in Part Four, the second major component in the present project: the **parser** itself.

Much of what is described in Part Two relates directly back to the present chapter. In Chapters Six and Seven, for example, we discuss the problems of representing and storing a parsed corpus in a database. As you will see, I have been careful to ensure that (a) the design of the method of representation, and (b) the method of storing the data both faithfully represent the categories and relationships of the Cardiff Grammar and also have the ability to annotate ellipsis and discontinuity.

PART TWO

The Corpus Database

This part of the thesis describes the first of two major components that contribute to the success of the parsing procedure to be discussed in Part Four. This is the **corpus database**. It also describes the **corpus query tool** that we used to create an updated version of the parsed corpus, and how we accomplished this difficult task.

Database Management Systems (DBMSs) are ideal for both storing, and providing rapid access to enormous volumes of data, as applications in industry and research demonstrate, and the present project therefore uses a DBMS.

Before we can store a corpus in a database, however, a suitable annotation scheme is needed. Chapter Five therefore surveys the various formalisms that have been used in the past to represent syntax, in the light of their suitability for parsing and for corpus annotation. The chapter concludes by introducing the concept of **mark up languages** as beneficial way to annotate a corpus, and the syntactic relationships within it.

Chapter Six discusses the use of mark up languages (SGML and XML) for annotating corpora, and it explains the method selected for this project and why it was chosen.

Chapter Seven shows how a marked up corpus can be stored in a **native XML** database. The tables that contain the marked up data are called the **corpus tables**. The parser can access the information quickly by using the **mark up index tables** (called the **corpus index tables**), these embody information about a syntax node's ancestors, descendants and siblings, together with their relationship to any other syntax elements in the same parse tree.

The next requirement is a tool that can query the data and review the results, in order to investigate alternative approaches that the parser might use. This tool, which is a much-improved version of the Interactive Corpus Query Facility (Day 1993a), and is now called 'ICQF+', is described in Chapter Eight.

After our earlier attempts at parsing (see Chapter Twelve), we decided that the early version of SFG syntax used in the original POW Corpus should be replaced by the latest version of the Cardiff Grammar (as described in Fawcett (2000a) and Chapter Four), in order to obtain a superior functional analysis of its syntax. Chapter

Nine describes how this difficult goal was achieved and the types of changes that were made.

Apart from serving as a suitable base for the parser, the new corpus, renamed the Fawcett-Perkins-Day (FPD) Corpus, offers a new version of an invaluable resource to the field of Corpus Linguistics.

Chapter Five

Towards a corpus annotation scheme and a method of representing syntactic relationships

This chapter discusses the methods that have been traditionally used for:

- (a) a **corpus annotation**, and
- (b) representing **syntactic relationships**.

Section 5.1 discusses the needs of this project in broad terms for both of these requirements.

Section 5.2 describes some of the traditional methods that have been used to represent syntactic relationships and discusses their use for representing the syntax of a Systemic Functional Grammar.

Section 5.3 introduces a new method of representing both the corpus itself, and the syntactic relationships within it.

5.1 Requirements

Two basic types of information are needed for use by the **parser** and by the **corpus query tool**, these are:

- (a) **sentences** - represented as analysed parse trees, and
- (b) **syntactic relationships** between the items, elements and units that make up the parse trees and their associated **probabilities**.

As we will see in Chapter Eight, the user of the **corpus query tool** needs to be able to ask questions **about items** and **about syntactic relationships**. The results need to be in the form of lists of items and syntactic relationships or as a set of parse trees that can then be displayed in a **sentence viewer**. Furthermore, as we will see in Chapter Nine, in order to update the corpus, there is a requirement to be able to extract and modify sentences and then return them to the database for our work in creating the new version of the corpus.

The **parser** needs access to similar information, and once the parse is successful, it needs to be able to add the new parse trees to the corpus, in order to satisfy the

requirements of implementing a **dynamic corpus** (as discussed in Section 1.2 of Chapter One).

The speed of access to the information in the corpus is of crucial importance to the parser. As we shall see in Chapter Eleven the number of syntactic 'rules' has an impact on the performance of the parser. Previous research by Atwell et al (1988b), Souter (1996) and Day (1993a) (among others) found that there were many thousands of syntactic 'rules' within the naturally occurring texts of a corpus. Furthermore, we can expect the number of 'rules' to significantly increase as the size of the dynamic corpus increases. This is because a relatively small number of 'rules' (with a given structure) occur many times, while a greater number occur relatively few times.

The number of 'rules' is an important reason for adopting a database-oriented approach to parsing; database management systems are designed to store and provide rapid access to large numbers of records. As the parser will need to potentially ask hundreds of questions about syntactic relationships as it parses sentences, the speed of access to the information it needs is crucial. To improve performance further, it is sensible to provide **indexes** to the information in the corpus.

In conclusion, this project needs access to both the data of the corpus and the syntactic relationships within it. Therefore, the corpus needs to be annotated and stored in such a way that easily allows sentences to be:

- (a) located, extracted and displayed,
- (b) added, and
- (c) modified and / or replaced.

Further, indexes are required to be able to provide statistical information about the **syntactic relationships** within the sentences of the corpus data.

The numerical tree method used to annotate the original POW Corpus (see Section 3.2.9 of Chapter Three) is not a format that can easily be adapted to these needs and therefore, a new annotation method is sought. Details of our chosen methods are introduced in Section 5.3 before they are defined in Chapter Six.

Before the corpus annotation method is discussed, we start with an investigation into the traditional methods that have been used for the representation of syntactic relationships in Section 5.2. The description of these methods serves a secondary purpose by providing the necessary background material for the discussion on parsing which is given in Chapter Ten in Part Three.

5.2 Traditional methods of representing syntactic relationships

This section describes the various methods that have been used to represent syntax in the field, and is particularly concerned with the methods used in natural language parsing.

Here, the most common methods are listed, and it is shown where appropriate, how these formalisms can be (or have been) adapted to Systemic Functional Grammar (SFG).

5.2.1 Phrase Structure Grammar style rewrite rules

5.2.1.1 Definition

Phrase Structure Grammars (PSGs) provide computational linguistics with the concept of a set of rewrite rules (of the form $t_1 \rightarrow t_2, \dots, t_n$) that are used to indicate that one token (on the left-hand side of the rule) can be 'rewritten' by one or more tokens (on the right-hand side of the rule). Tokens that have items on their right-hand side (e.g. $N \rightarrow \text{"mackerel"}$) are called **terminal tokens** and tokens that may be rewritten by one or more other tokens are called **non-terminal tokens** (for example, $NP \rightarrow \text{ART, N}$). Terminal tokens are normally **parts-of-speech** (such as verb, noun and article) and non-terminal tokens represent higher syntactic structures such as nominal groups and clauses.

Chomsky (1957) introduced what is called 'Chomsky Normal Form' (CNF). Here, the grammar is 'normalised' such that the right-hand side has just two tokens (one representing a token and the other a sequence of one or more tokens). This form of notation is a requirement for the CYK parser (for a description, see Grune and Jacobs (1990) and Part Three, Chapter Ten).

Generalized Phrase Structure Grammar (GPSG) (Gazdar et al 1985) is a context-free grammar which extend PSG by introducing **features** and **slash categories**. Features include, for example, verb-agreement and transitivity. Lexicalised phrase structure grammars extend the use of PSGs by being able to identify a lexical head in a unit; these are often used in **head-driven** approaches to parsing (see Chapter Ten).

5.2.1.2 Use in SFG and the suitability for this project

Table 5.1 shows a limited set of rewrite rules that can be used to represent the sentence shown in Figure 5.1.

Z->C1	A Sentence (Z) can be rewritten (filled) by a Clause (C1)
C1->S M C	A Clause (C1) can be rewritten by a Subject (S) followed by a Main Verb (M) and a Complement (C) (in a componence relationship)
S->ngp	A Subject (S) can be rewritten (filled) by a nominal group (ngp)
ngp->dd h	A nominal group (ngp) can be rewritten by a deictic determiner (dd) followed by a head (h) (in a componence relationship)
dd->"the"	A deictic determiner (dd) can be rewritten (expounded) by "the"
h->"seagulls"	A head (h) can be rewritten (expounded) by "seagulls"
M->"ate"	A Main Verb (M) can be rewritten (expounded) by "ate"
C->ngp	A complement (C) may be rewritten (filled) by a nominal group (ngp)
h->"mackerel"	A head (h) can be rewritten (expounded) by "mackerel"

Table 5.1: A set of rewrite rules for the sample sentence given in Figure 5.1

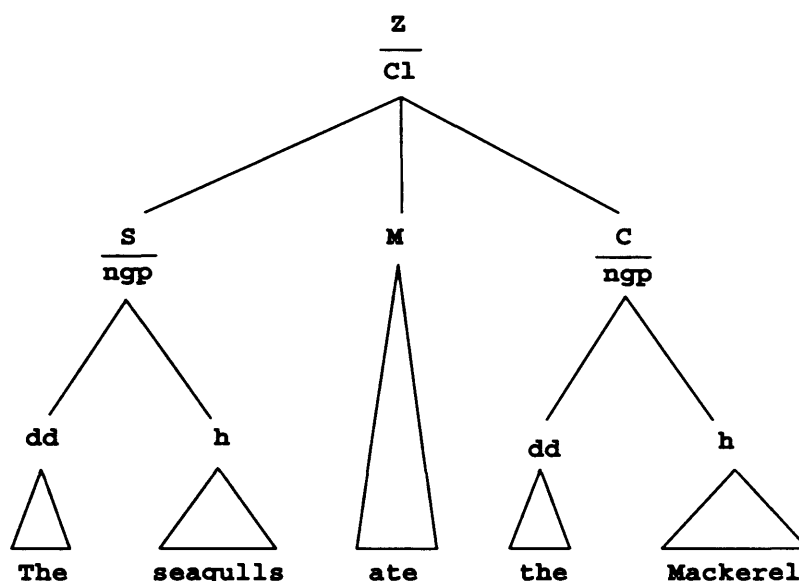


Figure 5.1: A Cardiff Grammar tree representation of a sample sentence

Atwell et al (1988b), Day (1993a), Weerasinghe (1994) and Souter (1996) all showed how the basic concepts of rewrite rules can be used to describe the syntax of a Systemic Functional Grammar with varying degrees of success.

A set of rewrite rules used as described above in Table 5.1 does not adequately maintain the relationships required for the Cardiff Grammar unless they are supplemented by some additional information. In the approach shown, these relationships are all described in the same notation:

h->"mackerel" **exponence - mackerel expounds a head (h)**

ngp->dd h **componence - the deictic determiner (dd) and a head (h) are components of the nominal group (ngp)**

S->ngp **filling - the Subject (s) is filled by a nominal group (ngp)**

S->ngp ngp **co-ordination - two nominal groups (ngp) are co-ordinated and fill the Subject (S)**

There are a number of ways of overcoming this limitation, for example:

- (a) by keeping a record of which syntax tokens represent elements and which tokens represent units,
- (b) by using the level of the 'rule' in the parse tree (i.e. level one represents an item, level two an element, level three a unit and so on),
- (c) by changing the rewrite symbol to characters that have a meaning (e.g. exponence: **h < "mackerel"**, componence: **dd->h**, filling: **S | ngp** and co-ordination **S | ngp ngp**).

In conclusion, the PSG style rewrite rule notation can be used to represent the syntax of a SFG provided that it is supplemented with some modifications that ensure that the SFG relationships are maintained.

5.2.2 Transition networks

5.2.2.1 Definition

Transition networks have been used to model the parsing process since the 1970s, see, for example Woods (1970) and Woods (1973).

Here, a 'grammar' is stated within a set of transition networks which consist of **nodes** and **arcs** (or **states** and **actions**). Each network has a start node, an end node and a set of intermediate nodes, and a **state** which represents a position in the network after a choice has been made. Normally, a network represents a syntactic unit, such as a noun phrase, and the states and actions represent the **events** of having seen an article, noun etc. in the input string. Exiting the network via the exit node means that a valid syntactic unit has been encountered.

5.2.2.2 The use in SFG and the suitability for this project

Figure 5.2 shows how a network could be applied to a SFG, and represents a simplified nominal group (**ngp**). As we traverse the network, we move from state to

state by carrying out an action. The action of finding a deictic determiner (**dd**) in the input string means that we move to state 1, where we are 'looking for' either a thing modifier (**moth**) or a head (**h**) to move to state 2. In the model, modifiers contain **ngps**, and when in state 2, the network calls itself recursively to find an embedded **ngp** before it moves to State 2 after finding an **h**. Note that the JUMP action consumes no input.

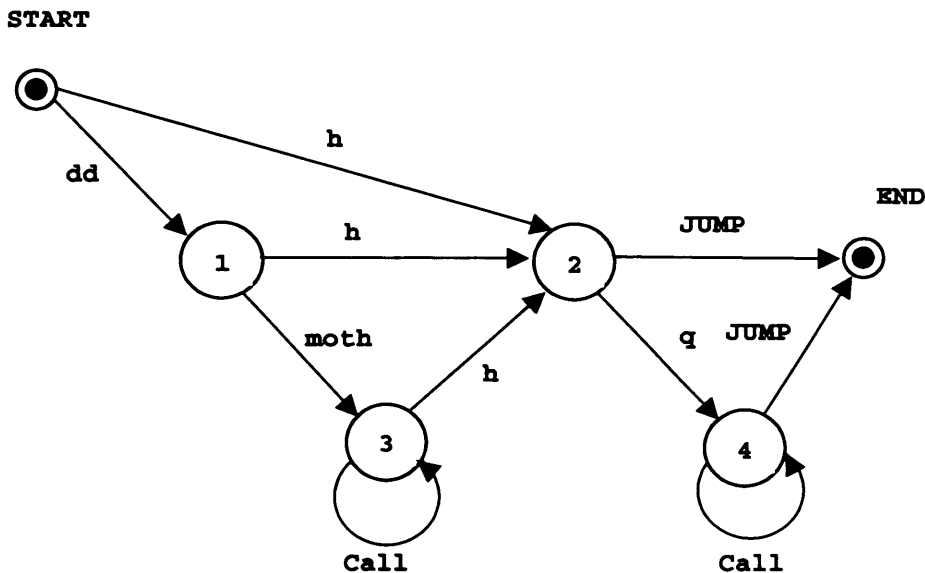


Figure 5.2: A recursive transition network for a simplified nominal group

As shown in Figure 5.2, transition networks can indeed be used to represent the syntax of a SFG. However, as Johnson (1983) points out, although transition networks provide a neat and compact way to represent grammars, they become difficult to manage when the grammar becomes complex. The number of syntactic relationships found in a naturally occurring text is large and the inter-relationships between them are complex. While rewrite rules are relatively straightforward to modify, the same cannot be said for a complex network as one has to consider not only the states, but the paths between the nodes. The networks automatically extracted from a corpus will be large and complex, and, as we need to update the corpus regularly, this makes network annotations not suitable for use in this project.

5.2.3 Tree Adjoining Grammar (TAG)

5.2.3.1 Definition

Tree Adjoining Grammar (TAG) as first defined by Joshi et al (1975), is best described from the computational viewpoint, in the XTAG report (XTAG 1995).¹ TAG concepts are of particular interest to this project as we use similar tree operations in our parser (see Chapter Fourteen and Appendix H).

TAG provides tree-building rules, which are used to build syntax trees. A TAG is defined as quintuple (Joshi and Schabes 1997):

$(\Sigma, \mathbf{NT}, \mathbf{I}, \mathbf{A}, \mathbf{S})$, where:

- Σ is a finite set of terminal symbols,
- \mathbf{NT} is a finite set of non-terminal symbols,
- \mathbf{S} is a distinguished non-terminal symbol,
- \mathbf{I} is a finite set of finite trees called **initial trees**,
- \mathbf{A} is a finite set of finite trees called **auxiliary trees**.

Initial trees (\mathbf{I}) are used to describe minimal linguistic structures that contain no recursion, for example noun phrases and prepositional phrases. These have **internal nodes** that are labelled by non-terminal tokens and **leaf nodes** labelled with terminal tokens (or by non-terminal tokens that have an indication that they may be substituted).

Auxiliary trees represent constructs that are adjuncts to the basic structures (for example, an adverbial); these trees have internal nodes that are labelled with non-terminal tokens and leaf nodes that are labelled with terminal tokens or tokens marked for substitution. Auxiliary trees have a root node (in TAG, called the **foot node**) which allows the tree to be adjoined to another.

There are two basic operations (shown in Figure 5.3) that operate on two sub-trees to give a third; these are **substitution** and **adjunction**. Trees that are the result of a tree operation are called **derived trees**.

Substitution allows the root node of an initial tree to be merged into a non-terminal node leaf node that is marked for substitution (and has the same token) in another initial tree.

¹ See also Joshi and Schabes (1997) and Joshi (1985) for a good introduction to tree adjoining grammars.

Adjunction allows an auxiliary tree to be **grafted** onto a non-terminal node of an initial tree; the root node of the auxiliary tree must match the node to which it is grafted in the initial tree.

5.2.3.2 The use in SFG and the suitability for this project

Next we turn to the suitability of applying TAG representation methods to a SFG. As far as my research has shown, no model of SFG syntax has been implemented in a TAG format. Like other representations, it is, in principle, possible to do this provided that the SFG relationships are not lost in the model. Figure 5.4 includes an attempt at describing a TAG substitution operation on two SFG trees.²

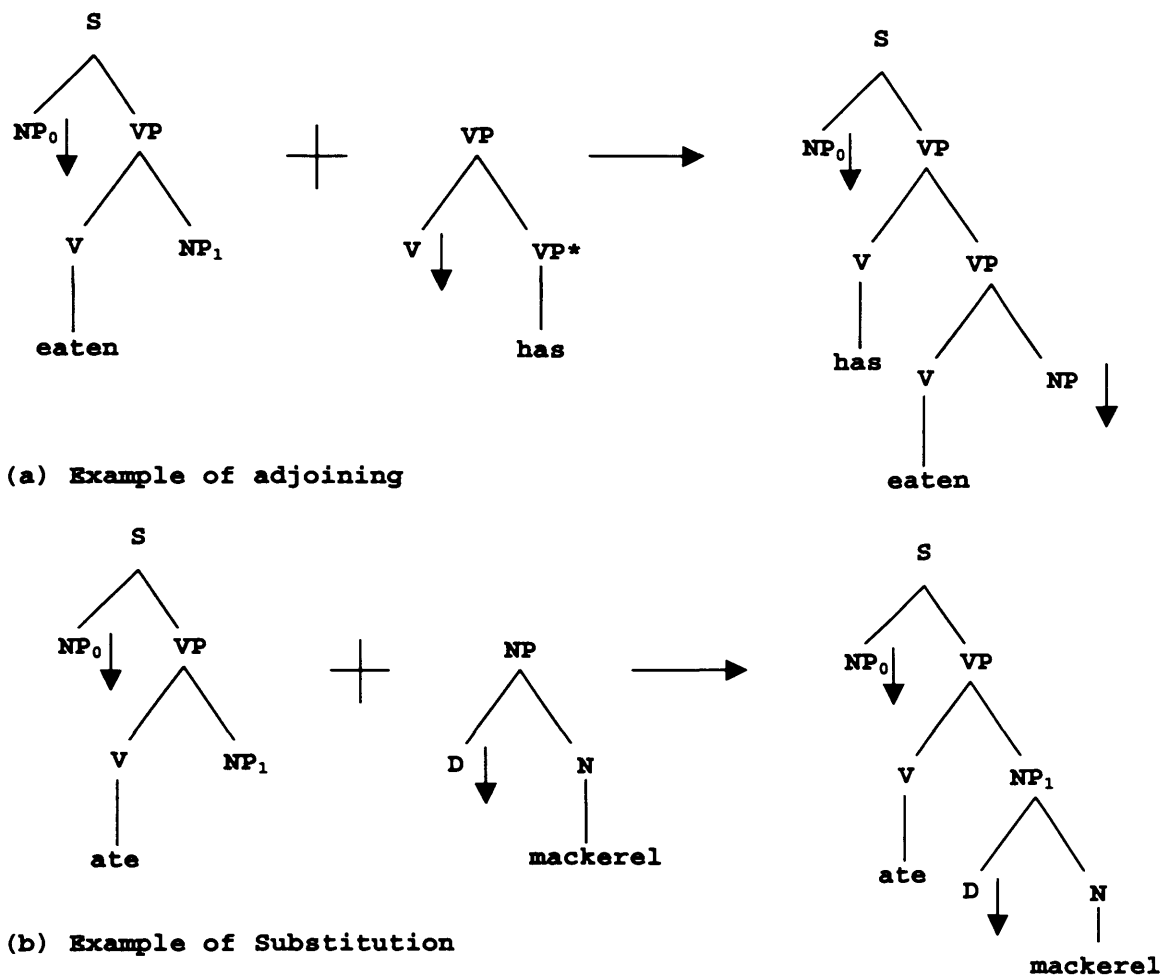


Figure 5.3: TAG - combining operations (adjoining and substitution)

² Note that I have chosen to change the representation of the syntax trees to cover filling and that this is not a standard feature of a TAG.

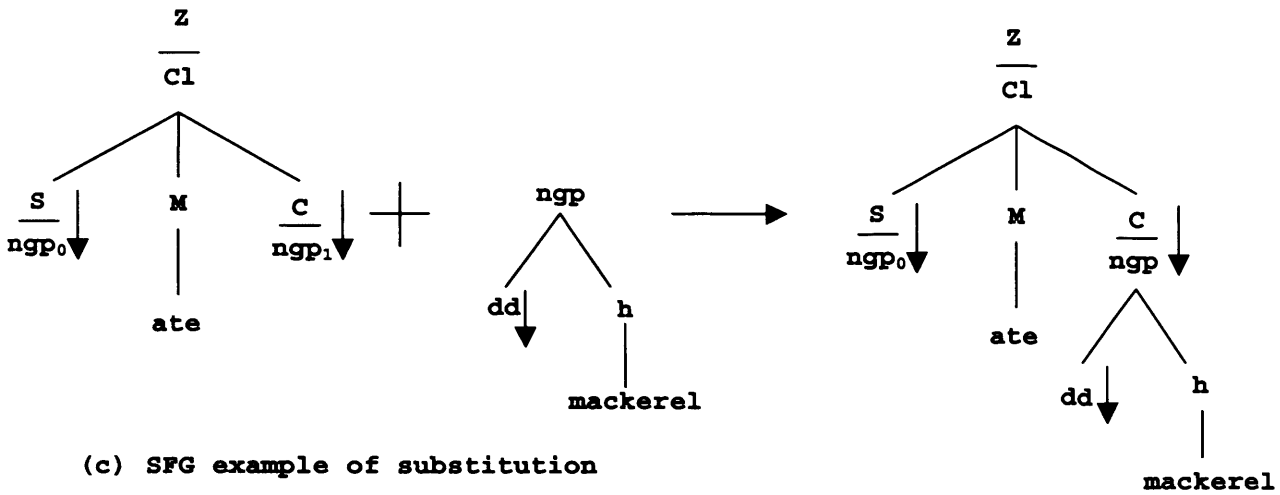
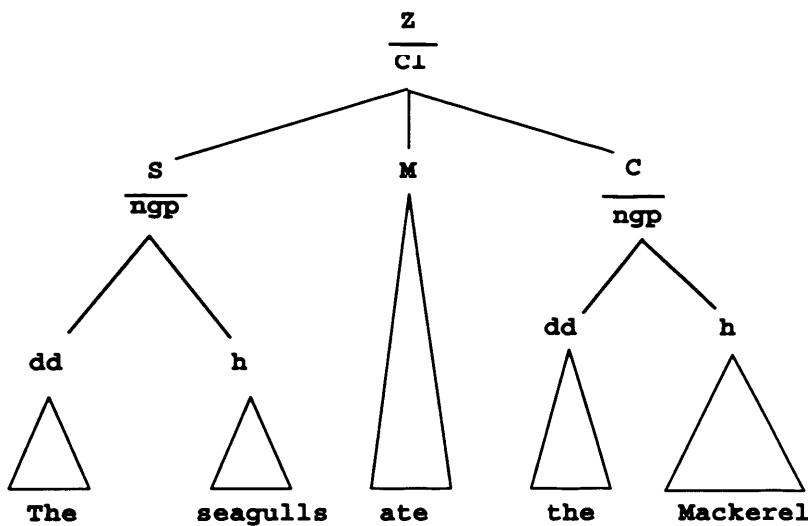


Figure 5.4: TAG - an SFG example of combining operations (adjoining and substitution)

5.2.4 Vertical Strip Grammar (VSG)

5.2.4.1 Definition

As shown by O'Donoghue (1991a), syntax can be represented as a set of vertical strips where a vertical strip is defined as a set of tokens extracted vertically from a parse tree; this is best shown by an example (see Figure 5.5).



Vertical strips extracted from the sentence:

- # dd ngp S Cl Z @
- # h ngp S Cl Z @
- # M Cl Z @
- # dd ngp C Cl Z @
- # h ngp C Cl Z @

= leaf and @ = root

Figure 5.5: A sample sentence and a set of vertical strips extracted from it

O'Donoghue used a form of network to represent his **vertical strip grammar** in order to tell his parser which strips are allowed to follow others. Although being able to state the inter-relationships between strips in this way is considered to be an advantage of network approaches, the disadvantage of networks (as we saw in Section 5.2.2) is that they are difficult to update, as they are complex when they are used to model the syntactic relationships found in naturally occurring texts of a corpus.³

5.2.4.2 The use in SFG and the suitability for this project

O'Donoghue (1991a) first did this in the COMMUNAL project when he extracted the vertical strips automatically from a generated corpus (see Chapter Eleven), and it is therefore suitable to describe the syntax of an SFG.⁴ It is possible to use O'Donoghue's concept of vertical strips without using his network, as we will see in Part Four, Chapter Fourteen. Furthermore, it is also possible to use partial vertical strips that cover a number of vertical elements and units but do not necessarily extend up to the root or down to the leaf of the parse tree.

5.2.5 Definite Clause Grammar (DCG)

5.2.5.1 Definition

Before we leave this survey of 'grammars' that have been widely used to model the parsing process, for completeness, I should mention Definite Clause Grammar (DCG). These are closely associated with the declarative programming language of PROLOG, and they have been quite widely used (Clocksin and Mellish 1994). PROLOG provides the ability to define a program as a set of facts (termed 'Horn clauses') and provides an automatic backtracking mechanism; this made it attractive in the early days of parsing.

5.2.5.2 The use in SFG and the suitability for this project

An example DCG representation of a set of grammar rules defined as facts is shown in the example below:

³ The same advantage of being able to record the inter-relationships between syntactic structures is also given by transition networks.

The fact:

Z(the seagulls ate the mackerel)

Is true (i.e. parses) provided that some sub-facts are also true:

Z:-Cl	ngp:-dd h	M:-"ate"	h:-"mackerel"
Cl:-S, M, C	dd:-the	C:-ngp	
S:-ngp	h:-"seagulls"	ngp:-h	

In the example, a **fact** and a **rewrite rule** are synonymous, therefore DCGs can be applied to represent a Systemic Functional Grammar but have the same limitations as rewrite rules. Because a DCG extracted from the naturally occurring texts of a corpus will have many thousands of facts, there must be some doubt whether PROLOG has the ability to cope with a DCG of this size within the limitations of the computer being used.

5.2.6 The decisions made for this project

We now turn to the question about which of these representations may be of use to this project. As we saw, rewrite rule notations have been used before to represent the syntax 'rules' of a SFG. However, they operate on only two levels of a syntax tree.

Because a rewrite rule can only represent a single syntax relationship, it is not possible to express vertical information that spans more than one level of the syntax tree (e.g. that a head (**h**) is in a nominal group (**ngp**) that fills a subject (**S**)), and this is something that is needed for the new parser that will be described in Part Four.

The concept of transition networks can be extended, as was shown by O'Donoghue (1991a), so that they can be used to model the inter-relationships between different syntactic structures. Their use, however, is not considered for this project due to the fact that they are too complex when they are used to model the syntactic relationships found in unrestricted naturally occurring texts, and they are difficult to update.

In O'Donoghue's network (1991a), as we shall see in Chapter Eleven, the basic unit was a vertical strip, which offers the ability to look upwards for more than one level in the parse tree. Although this project will not use O'Donoghue's networks, it will use (and extend) O'Donoghue's idea of a vertical strip.

Definite Clause Grammars are very similar to rewrite rules in the way that they work and have the same limitations described above.

⁴ As far as my research has shown, this has not been attempted elsewhere.

The Tree Adjoining Grammar's tree operations are of interest, not as a method of representing syntax, but due to the fact that they use tree joining operations which are similar to the operations that we use in the parser described in Part Four.

In the next section, I introduce the **corpus annotation method** that has been selected for this project. The method is one that can also be used to represent **syntactic relationships** and is therefore an alternative to the methods I have described in this section.

5.3 Summary: Representing the Corpus text and syntax structures

As we saw in Chapter Three, a number of different methods have been used to represent the texts and syntax of a parsed corpus. For example, the Penn Treebank (see Section 3.2.3) uses bracketed trees to represent a parse tree, and the Polytechnic of Wales (POW) Corpus uses a numerical tree notation. Typically, these early methods were specific to the corpus for which they are used.

We also saw in Chapter Three that mark up languages such as SGML and, more recently, XML are beginning to be used as an annotation method for some corpora mainly through the work of the Text Encoding Initiative (TEI).

As we shall see in the next two chapters, this project has been using mark up languages since its early days, and I shall introduce an extension of the use of mark up languages such that they can not only be used for a corpus annotation scheme, but also as a method for representing the syntactic relationships in the sentences of the corpus. The main advantages of doing this are that the representation is not restricted to a single level of the parse tree, and that it is able to provide information about the inter-relationships between different syntactic structures.

Before we can discuss the method in detail, it is necessary to provide some background information about mark up languages and Chapter Six will therefore start by providing this.

Having chosen the annotation method for the corpus, the next question requiring a decision is how the system can easily find, retrieve and update the sentences and syntactic relationships in the corpus in a fast and efficient manner. The answer proposed here is to store the corpus and its ancillary data in the tables of a Database Management System (DBMS), and the design of this is discussed in Chapter Seven.

Chapter Six

Marking up language texts

The use of mark up languages is the subject of this chapter, which is divided into two parts.

The first part, in Section 6.1, is concerned with how this project uses mark up languages for corpus annotation, and it introduces how the mark up scheme can be used to represent syntactic relationships as proposed in Chapter Five.

In Chapter Three, we saw that other corpus linguistics projects are beginning to use XML and, in the second part (Section 6.2), our selected methods are compared with the methods of these other projects.

6.1 The use of mark up languages in this project

In what follows, a minimal knowledge of mark up languages is assumed. For readers who need it, Appendix C provides the necessary background information together with the terms used and their definitions. Unfortunately, mark up languages use the terms **element**, **attribute**, **tag** and **parsing** in a different sense from the sense in which they are used in linguistics. I shall therefore use the phrase **mark up** before these terms in what follows.

This section starts by describing the requirements of a corpus annotation scheme for this project. Next it discusses the different approaches to mark up that were considered before the method selected was chosen. Then the way in which the chosen scheme meets the requirements that will be given in Section 6.1.1 is discussed.

The section finishes with an introduction to the way in which the mark up scheme can not only provide a corpus annotation, but also provide an index to the syntactic relationships that are within the corpus, and will be needed by the **corpus query tool** and by the **parser**.

6.1.1 The requirements for a corpus annotation scheme

The corpus annotation scheme for this project needs to be able to meet the following design criteria:

- (a) It can accommodate the concepts of Systemic Functional Grammar and be able to maintain (i) the distinction between items, elements and units, and (ii) the relationships of **componence**, **filling**, **co-ordination** and **exponence**,

- (b) It can be extended to include the representation of (i) intonation, (ii) punctuation and (iii) Participant Roles,
- (c) It is easy for a human to understand it and easy to render,¹
- (d) It can be edited quickly, using a standard editor,
- (e) Standard techniques for testing the validity of sentences can be used with it,
- (f) It can represent discontinuous structures,
- (g) It can mark ellipsis, questionable items and elements, and unfinished units.

Furthermore, as stated in Section 5.1 of Chapter Five, the method of storage of the corpus must be such that:

- (h) queries must be performed rapidly, because the performance will affect the speed of the parser, and
- (i) it must allow individual sentences to be extracted, modified and returned to the corpus, and also new sentences to be added as they are parsed.

6.1.2 The background of mark up languages, and their use in this project

At the start of this project in 1997, no other analysed corpus was available in a marked up form. The Standard Generalized Markup Language (SGML) (ISO 1986) became popular in certain niche markets for documentation (such as the aerospace industry (S1000D 2006), in the drug industry and in the automotive field).² The Text Encoding Initiative (TEI) began to use SGML as a neutral format to mark up the texts of corpora (see Chapter Three and Section 6.2.1).

Today, the situation is very different. The concepts of SGML were adopted by the World Wide Web Consortium (W3C) in the late 1990s, when the eXtensible Markup Language (XML) was created as a simplified subset of SGML (W3C 2004a).³ XML is now enjoying high popularity in the fields of documentation, database, and web-based business applications. The use of XML is also beginning to impact the field of computational and corpus linguistics (see Section 6.2 and Chapter Three).

Before XML, software tools that operated on SGML data were not freely available and were fairly expensive. In this project, therefore I had to implement some

¹ This requirement and the next is to satisfy the nature of our research, which was one in which a linguist had to be able to quickly understand and change sentences directly in the mark up format.

² The Standard Generalized Mark up Language (SGML) was invented by Charles Goldfarb of IBM and became an international standard in 1986 (ISO 1986).

³ XML is a cut-down version of SGML with some of SGML's more complex structures removed and some additional constraints added.

quite complex programs that manipulated marked up data. Now, there are many XML-aware software tools and, in particular, W3C has defined a Document Object Model (DOM) as a programming library that allows an application to navigate and change the mark up found in a marked-up document. Halfway through the project, I decided to switch from SGML to XML to take advantage of some of these tools and applications.

6.1.3 Other benefits of using mark up languages

A secondary reason for the decision to use XML is that there are a large number of supporting standards and software that XML brings with it which become available to our project, notably:

- (a) SGML or XML parsers (such as SGMLS, SGMLNORM by James Clark (www.jclark.com) or XERCES (<http://xerces.apache.org>) can be used to check that the input sentence is grammatical (see Section 6.1.5.3).
- (b) SGML/XML editors such as ArborText Epic, Adobe FrameMaker or oXygen can be used to produce sentences that conform to the rules of the grammar.⁴
- (c) One has access to an ever-increasing set of software tools (apart from those that are listed here) that are designed to work with XML.
- (d) As detailed above, the Document Object Model (DOM) is immediately available to programmers.
- (e) Corpora expressed in XML can easily be transformed using a stylesheet written using the XML Stylesheet Language (XSL) (W3C 2006).⁵ This could be used, for example to convert a corpus annotated to one model of syntax to another.
- (f) Parse trees can be rendered using XSL (or XSL formatting objects (XSL-FO) (W3C 2006))
- (g) Statistical information can be extracted using Xquery (W3C 2007).

This project was now at a point where it had an analysed corpus, and the benefits of converting it into SGML were realised. Before this could be done, however,

⁴ There are now a vast number of XML editors available; Epic and FrameMaker, having their roots in SGML, were among the first.

⁵ This could assist in, for example, translating a corpus annotated using the labels of one formalism into another. Note that although conversions using XSL are unlikely to be 100%, they could be used in conjunction with the DOM to produce more reliable results.

decisions had to be made about the mark up method that will be used, and this is discussed in the next section.

6.1.4 Towards a mark up scheme

6.1.4.1 Definitions

In the field of data (or document) analysis, there are different views of the same data that can be drawn. There is often a distinction made between **prescriptive** mark up and **descriptive** mark up (see, for example Day (1995)).⁶

6.1.4.1.1 Prescriptive mark up

A workshop manual, for example, gives instructions which tell the reader how to assemble, disassemble, clean, repair or inspect certain components. One view of this data could be that it is a book, which contains chapters, and the chapters contain sections, which contain paragraphs. At the start of each section, there is a table, which lists tools. A marked up document could, in this view, contain mark up elements **<chapter>**, **<section>**, **<table>** and so on. This is called **prescriptive** mark up.⁷

6.1.4.1.2 Descriptive mark up

A more semantic view of this information shows that the book contains procedures, and the procedures contain a list of tools required, the number of people required to perform the procedure, any safety related information. The body of the task is a number of steps of procedure; in a document marked-up in this way, we would expect to see meaningful generic identifiers for elements such as **<procedure>**, **<toolist>**, **<tool>**, **<partno>**, **<person>**, **<safety>** and **<step>**. This is called **descriptive** mark up as it describes the data that the document contains.⁸

In a descriptive view of a parsed sentence, there are mark up elements such as sentence **<Z>**, which contains Clauses **<C1>**. The Clause contains Subjects **<S>**,

⁶ Some authors use the term **content-specific** and **data-driven mark-up** as alternative to **descriptive mark-up**. The terms **specific mark up** and **generic mark up** are used by van Herwijnen (1994) respectively to mean the same thing as prescriptive and descriptive mark up.

⁷ Prescriptive mark-up methods often contain information about how the document should look when formatted (e.g. in terms of leading, fonts and emphasis).

⁸ Descriptive mark-up marked a revolution for the technical author as he was now able to concentrate on technical content and not page layout.

Operators <O> and Main Verbs <M> and so on. The disadvantage of descriptive methods is that they are typically only applicable to a single domain.⁹

6.1.4.1.3 Abstract mark up

I define the term **Abstract** mark up (Day 1993c) and it lies somewhere between **prescriptive** and **descriptive** mark up. It uses generic identifiers that have general names, and uses them with attributes to qualify the meaning. In theory, an abstract model can be applied to corpora for the representation of many different syntax formalisms. In this approach, one can expect to see generic identifiers such as <sentence>, <clause>, <phrase> and <word>, with, for example, attribute names such as **pos** (for part-of-speech) or **cat** (for category).¹⁰ The disadvantage of this approach is that an SGML or XML parser cannot be used on its own to ensure that the sentences comply with the rules of the grammar. For example, the ability to check that Subjects can only appear in a Clause, and the fact that a Subject can only occur once in a Clause by using standard SGML/XML parsing is lost. This means that the benefits that a descriptive DTD/Schema will bring are lost. It is not impossible to enforce these rules by using other software. Typically when abstract methods are used, a project defines the list of allowed values for the attributes and rules for co-occurrence of mark up elements with given attributes in a set of **business rules**. Proprietary software, called a business rules checker, is used to check that attribute values and elements have been correctly used.¹¹

6.1.4.2 An abstract mark up scheme for the Cardiff Grammar

A simplified version of an abstract Document Type Definition (DTD) that we created in the early days of the project is shown in Figure 6.1 together with an example sentence in Figure 6.2. For details of the syntax of a DTD, please refer to Appendix C.

⁹ In the example given here, mark up element names such as <S>, <C1> and <ngp> are applicable only to the Cardiff Grammar.

¹⁰ More general attributes such as **feature** can be used to store things that are unique to the grammar (for example, in SFG, this could be used for Participant Roles).

¹¹ Business rules checkers (Day 2006) check that elements and attributes contain correct data; for example, if an element contains a part number, that part number conforms to the structure of a part number and that it exists in some external database. When applied to SFG in an abstract approach, it could check, for example that the syntax tokens belong to the grammar and are valid in their context. Note that the point I make here is business rules checkers are useful for all types of mark-up but become essential for the abstract approach.

The DTD states that a mark up element called `<sentence>` must contain at least one `<unit>`, and that a `<unit>` must contain at least one `<elements>`.¹² The `<elements>` can contain either (a) one or more `<unit>`, (b) a single `<item>` or (c) nothing (to represent ellipsis). The mark up elements `<unit>` and `<elements>` are abstract because the class of unit, or the name of the element of structure are stored in the mark up attributes named `syntaxtoken`.

The advantage of using the abstract mark up scheme given above is that it can be used to mark up a wide range of different models of grammar. Further, if the grammar is modified, then the DTD (or Schema) is much less likely to have to be changed.¹³

```
<!DOCTYPE sentence PUBLIC "-//DTD/SENTENCE COMMUNAL VERSION 1/EN" [
<!ELEMENT sentence - - (unit+) >
<!ATTLIST sentence      sentid CDATA #IMPLIED>
<!ELEMENT unit          - - (elements+) >
<!ATTLIST unit          unitid ID #IMPLIED
                        syntaxtoken CDATA #REQUIRED>
<!ELEMENT elementsS    - - (unit+ | item)?>
<!ATTLIST elementsS    elemid CDATA #IMPLIED
                        syntaxtoken #REQUIRED
                        ellipted(RapidSpeech|Recoverable)
                        "RapidSpeech">
<!ELEMENT item          - - (#PCDATA)>
<!ATTLIST item          itemid ID #IMPLIED
                        intonationstart CDATA #IMPLIED
                        intonationend CDATA #IMPLIED>
]>
```

Figure 6.1: A simplified abstract DTD for the Cardiff Grammar (version 1)

¹² The generic identifier `elements` (element of structure) is used because `element` has a different meaning in DTD syntax.

¹³ An XML Schema is an alternative to a DTD (see Appendix C).

```

<!DOCTYPE sentence PUBLIC "-//DTD/SENTENCE COMMUNAL VERSION 1/EN" []>
<sentence sentid="10A...">
  <unit syntaxtoken="Cl">
    <elementS syntaxtoken="S">
      <unit syntaxtoken="ngp">
        <elementS syntaxtoken="dd">
          <item>The</item>
        </elementS>
        <elementS syntaxtoken="h">
          <item>seagulls</item>
        </elementS>
      </unit>
    </elementS>
    <elementS syntaxtoken="M">
      <item>ate</item>
    </elementS>
    <elementS syntaxtoken="C">
      <unit syntaxtoken="ngp">
        <elementS syntaxtoken="dd">
          <item>The</item>
        </elementS>
        <elementS syntaxtoken="h">
          <item>mackerel</item>
        </elementS>
      </unit>
    </elementS>
  </unit>
</sentence>

```

Figure 6.2: A sample sentence marked up according to the DTD in Figure 6.1

6.1.4.3 Evaluating the abstract model

The disadvantage with this model, as stated above, is that SGML/XML parsing is not enough to determine that the sentence conforms to the rules. The SGML/XML parser only complains if, say, a unit had no elements, but does not care if nominal group elements are used in a Clause.

A descriptive mark up model, where mark up elements have generic identifiers which reflect their meaning, overcomes these issues. It also comes with the advantage that the native mark up is easier to read and edit by a human. The disadvantages are

- (a) that even minor changes in the grammar model are highly likely to mean that the DTD/Schema has to be modified, and
- (b) that the model is specific to one grammar model.

Having considered the disadvantages, the ability to be able to check that sentences are grammatical is an extremely attractive one. This, coupled with the fact that a descriptive approach satisfies the requirement for a representation that is easy to understand and edit (see Section 6.1.1), is the reason why a descriptive approach was

selected for this project. The next section describes some of the experiments performed before I arrived at the chosen solution, which is described in Section 6.1.5.2.

6.1.5 Using a DTD to specify a rule-based grammar

The goal of the work presented in this section was to create a DTD, and use it to make sure that the sentences that we were taking from the POW Corpus (following their conversion into SGML) complied to the rules of syntax for the Cardiff Grammar. In the early days of the project, because we were using SGML, not having a DTD was not an option (because they are mandatory). When we switched to XML the need for a DTD was relaxed, as XML simply demands that documents are well-formed. Our decision, once XML was available, was not to use a DTD at all. The work that went towards creating a DTD (particularly from a parsed corpus) was fairly significant and was promising enough to be documented here, particularly as it is hoped to reinstate it for Phase Two of the project (see Chapter Nineteen).

Section 6.1.5.1 describes my work in creating a DTD manually, Section 6.1.5.3 describes how we converted the POW corpus in SGML and Section 6.1.6.4 my limited success in generating a DTD automatically from the parsed corpus.

6.1.5.1 Creating a DTD manually from a linguist's model of language

With my next experiments with SGML, I looked at actually specifying the syntactic relationships of the Cardiff Grammar using a DTD. Here my first experiments were with the syntax diagrams that appear in Appendix A of Fawcett (2000a). Although this was not a straightforward task (because of optionality and mutual exclusivity), it was accomplished with some success.

6.1.5.2 The chosen mark up scheme

Instead of using abstract generic identifiers (like sentence, unit, element and item), I decided to use a descriptive mark up and apply meaningful names to generic identifiers (like **Z**, **C1**, **S**, **dd**, **ngp**, **pgp** and **TEXT**). In addition to satisfying the requirements given in Section 6.1.1, there were other advantages in using a descriptive approach:

- (a) It made searching faster (instead of ‘find me all unit mark up elements that have a **syntaxlabel** attribute value "**ngp**", it was easier and faster to find all mark up elements with a generic identifier of **ngp**),¹⁴
- (b) When the SGML is stored in the database as described in Chapter Seven, it made SQL statements simpler to express and the execution time of the query faster,
- (c) The Cardiff Grammar uses Participant Roles (PRs) and conflations, and these apply only to certain elements and not others (e.g. PRs apply typically to subjects and complements and certainly not, for example, to an Operator). By treating a subject as its own mark up element, we were able to specify that it had different mark up attributes from say, an operator.

The nominal group was modelled using the SGML DTD content model in Figure 6.3. This states that it can optionally start with a linker (**Lnk**), followed by various types of determiner, with optional selectors (**v**), zero or more modifiers (**mo**) and then either a mandatory head (**h**), a proper name head (**h_n**) or a pronoun head (**h_p**),¹⁵ followed by zero or more qualifiers (**q**). As an example for one of the elements of structure of the nominal group, modifier can be filled by a **ngp**, or a **Cl**, or a **qlgp** or a **genclr** or be expounded by an **item**.¹⁶

Full details of this and the other Cardiff Grammar units expressed in the descriptive DTD, that was the result of this work, are given in Appendix C. In Figure 6.2 we saw the sample sentence marked up using an abstract mark up scheme. Figure 6.4 shows the same sentence marked up to our chosen descriptive mark up scheme.

¹⁴ This was using my own SGML search software, it is expected that similar figures will be found with XML's DOM.

¹⁵ the fact that a head is of a certain type could have been (but wasn't) modelled as attributes as in **<h type="pronoun">**.

¹⁶ At the time I created the DTD, I didn't realise that elements of structure may be filled by co-ordinated units of different classes, however, this is extremely rare.

```

<!ELEMENT ngp - - (Lnk?, (rd,v?)?, (pd,v?)?, sd|od), v?)?,
                  (td,v?)?, dd?, mo*, (h|h_n|h_p), q*)>
<!ELEMENT Lnk - - (item)>
<!ELEMENT rd - - (ngp+ | item)>
<!ELEMENT pd - - (ngp+ | item)>
<!ELEMENT sd - - (ngp+ | qlgp+ | item)>
<!ELEMENT od - - (ngp+ | item)>
<!ELEMENT v - - (item)>
<!ELEMENT td - - (qtgp+ | item)>
<!ELEMENT dd - - (qlgp+ | item)>
<!ELEMENT mo - - (ngp+ | Cl+ | qlgp+ | genclr | item)>
<!ELEMENT h|h_p|h_n - - (genclr | item)>
<!ELEMENT q - - (ngp+|Cl+|qlgp+|pgp+)

```

Figure 6.3: An SGML DTD content model for the nominal group

```

<!DOCTYPE Z PUBLIC "-//DTD/SENTENCE COMMUNAL VERSION 2/EN" []>
<Z>
  <Cl>
    <S PR="Agent">
      <ngp>
        <dd>
          <item>the</item>
        </dd>
        <h>
          <item>seagulls</item>
        </h>
      </ngp>
    </S>
    <M>
      <item>ate</item>
    </M>
    <C PR="affected">
      <ngp>
        <dd>
          <item>the</item>
        </dd>
        <h>
          <item>mackerel</item>
        </h>
      </ngp>
    </C>
  </Cl>
</Z>

```

Figure 6.4: An SGML instance of a sentence using the DTD in Figure 6.3

6.1.5.3 An SGML Parser as a natural language parser

Once the DTD was created, I automatically had a form of natural language parser. By creating sentences (similar to the one shown in Figure 6.4) I was able to check that the sentences conformed to the rules of the DTD. We could not, of course, take an input string of words and tag them automatically without being able to reference the DTD itself.

Now that we had a DTD, I could start the process of converting the POW Corpus into SGML. Programs were developed that took the POW numerical tree formats (see Figure 3.2 of Chapter Three) and converted them to create an SGML instance for every sentence. This proved to be a relatively straightforward programming task.¹⁷ We now had a DTD and a set of SGML instances for the sentences of the POW Corpus. Next, we attempted to parse the sentences against the DTD and found that a larger than expected number failed.¹⁸

Investigations showed that the failures were due to (a) ellipsis, specifically, missing heads of units (our DTD assumed that heads are mandatory),¹⁹ (b) transcription errors in the POW Corpus, and (c) sentences that described some rare linguistic constructs that we had not catered for in the DTD.

6.1.5.4 The automatically created DTD

To overcome these problems, I next attempted to create a DTD automatically from the mark up elements of our SGML sentences extracted from POW using a tool that I developed for industry called **createDTD**. This extracts SGML DTD content models from an SGML instance by investigating the mark up within it. The program, however, is not robust. It is limited to creating complete alternate DTD content models even where only one element differs in a different context.²⁰

The structure of a DTD that models a Systemic Functional Grammar that has been extracted from a spoken corpus is far more complex than a DTD that is used for the marking up of technical documentation and, therefore, **createDTD** failed to return a usable DTD. The main problem was the structure of the Clause. Because of optionality, the number of alternative content models returned by **createDTD** was large and these caused the SGML parser to exceed the maximum capacity limits that could be expressed in its SGML declaration.²¹ The method is worthy of further investigation

¹⁷ During the exercise, certain POW syntax tokens had to be renamed (e.g. & to Lnk) in order to comply with the rules of SGML and this was done before the work to convert the corpus into the later version of the Cardiff Grammar (see Chapter Nine).

¹⁸ Note that our original DTD differed from the one presented here and in Appendix C as it had symbols from POW and not the ones of the modern Cardiff Grammar.

¹⁹ This was bad judgement on my part because a great number of units in the corpus are missing their head elements, and as we shall see in Chapter Ten, is a reason why head-driven parsing approaches are also not suited to this project.

²⁰ **CreateDTD** was originally designed to do this, and allow a user to manually edit the DTD to improve the content models.

²¹ The SGML declaration is a file that defines various conventions (e.g. characters to use for mark up declaration open (MDO) and mark up declaration close (MDC) and limits such as maximum length of a generic identifier, the maximum depth of tags allowed etc.

after enhancing createDTD to produce more efficient models, and this will be done in Phase Two of this project (see Chapter Nineteen).

At this time in the project, XML became available and our restrictions on having to use a DTD were lifted. After converting the SGML into XML, we were able to use the Microsoft MSXML parser that is integrated with Internet Explorer (and available as a plug-in) to check that the sentences were well-formed; this did not, however, guarantee that the sentences were free from errors.

6.1.6 Satisfying the design criteria for Systemic Functional Grammar

Table 6.1 and the sections that follow show how the selected mark up scheme meets the requirements given in Section 6.1.

Criteria	Remarks
(a) It can be used to represent a corpus analysed according to the syntax of a Systemic Functional Grammar and be able to maintain both (i) the distinction between items, elements and units, and (ii) the relationships of componence, filling, co-ordination and exponence.	With the descriptive scheme meaningful mark up elements are used and hence the distinction between items, elements and units is maintained, as are the relationships.
(b) it can be extended to include the representation of intonation, punctuation and Participant Roles.	Participant Roles are handled using an attribute called PR on the Subject <code><S PR="Ag"></code> and Complement <code><C PR="Af"></code> elements as shown. Punctuation is handled by treating it as, for example, the ender <code><e></code> element. It is expected that intonation will be covered by using attributes.
(c) It is easy to understand by a human and easy to render	Any descriptive scheme is easy for a human to understand (and edit). Rendering is also straightforward.
(d) It must be editable (quickly) in a standard editor.	Sentences can be quickly edited using, for example notepad or an XML editor.
(e) It must be able to use standard techniques to test the validity of sentences.	Possible to check that sentences comply with the SFG rules for units and elements provided a DTD/Schema is created. A simple business rules checker has been adopted ahead of the DTD/Schema becoming available in Phase Two.
(f) It must be able to handle non-hierarchical (discontinuous structures).	See Section 6.1.6.1
(g) It must be able to mark ellipsis, questionable items and elements and unfinished units.	See Section 6.1.6.2 and 6.1.6.3.
(h) Queries must be performed rapidly, because the performance will affect the speed of the parser	See Chapter Seven.
(i) It must allow individual sentences to be extracted, modified and returned to the corpus, and also new sentences to be added.	See Chapter Seven.

Table 6.1: How the selected mark up scheme meets the project requirements given in Section 6.1

6.1.6.1 Handling discontinuous units

An important consideration in our decision to use mark up languages was the ability to handle non-hierarchical structures, which occur rarely in Systemic Functional Grammar when discontinuous units are encountered (see Section 4.3 of Chapter Four). We handled this phenomenon by using sequential identifiers (IDs) on the items. The technique, which is shown in Figure 6.5, allows the order of the items to be rebuilt in a formatted view of the parse tree through the repositioning of the sequential identifiers.

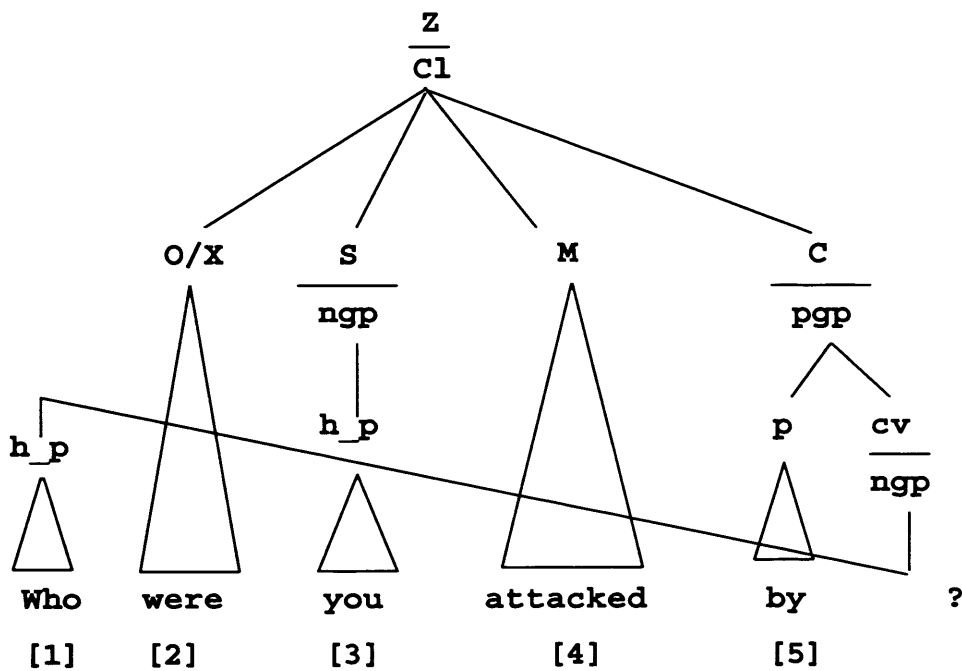


Figure 6.5: An example sentence showing a discontinuous unit

6.1.6.2 Handling ellipted elements

The meaning of ellipted elements is that they are either (a) recoverable from previous texts or (b) omitted by rapid speech (see Section 4.4 of Chapter Four). In the mark up scheme, I have introduced an attribute called **ellipted**, which can be applied to mark up elements that represent elements of structure, and can take the values "Recoverable" or "Rapid Speech" (e.g. <S ellipted = "Rapid Speech"/> represents a Subject (S) that is ellipted by rapid speech).

```

<Z>
  <C1>
    <OX>
      <ITEM id="2">were</DATA>
    </OX>
    <S>
      <ngp>
        <h_p>
          <DATA id="3">you</DATA>
        </h_p>
      </ngp>
    </S>
    <M>
      <DATA id="4">attacked</DATA>
    </M>
    <C>
      <pgp>
        <p>
          <DATA id="5">by</DATA>
        </p>
      <cv>
        <ngp>
          <h_p>
            <DATA id="1">Who</DATA>
          </h_p>
        </ngp>
      </cv>
    </pgp>
  </C>
</C1>
</Z>

```

Figure 6.6: A marked up version of the example sentence in Figure 6.5

6.1.6.3 Handling questionable (or unknown) units, elements and items

In the POW Corpus, there were occasions where the analyst is not certain about the decisions made during transcription; the most common causes were (a) unclear sounds on the recordings and (b) when the analyst was unsure of the analysis and wanted to indicate this. When either occurred, the analyst would use a question mark either against a syntax token or an item (e.g. *S?*, "*RAN?*") or by a question mark on its own.

To capture this, I used the following methods. When the syntax token was followed by a question mark, we used an attribute named **Questionable** with a value of "**Yes**". Therefore `<S Questionable="Yes">` represents a questionable Subject.

Where the element of structure was represented by a question mark on its own, we introduced a special element called `<ELEMQUERY/>`. When an item was only a question mark, we used `<ITEMQUERY/>`.

Furthermore, in some cases, the analyst could not decide which word had been uttered, the item would be followed by a question mark. When the linguist could not decide if one or another word was uttered, a slash would be used to separate the words that were thought to have been used. For example, "**BALL/BONE?**" shows that the analyst was not sure if the word was **ball** or **bone** and "**BRICK?**" shows that the analyst could not decide if the word was **brick**. In these cases, these words were left as they are, complete with the slashes and question marks.

6.1.7 Using the mark up scheme to represent syntactic relationships

In Chapter Five, I stated that there is a requirement for both: (a) a corpus annotation scheme and (b) a method of representing syntactic relationships. Furthermore, at the end of Chapter Five, I proposed that a mark up scheme will also be able to satisfy both requirements. The second requirement is met by storing the marked up corpus in the **corpus tables** of a native XML database, and through the provision of a **mark up index** (the **corpus index**). The exact methods of doing this will be described in Chapter Seven, when we describe the **corpus database**.

This concludes the description of our work with defining a mark up scheme for annotating the corpus, and more details of this work are provided in Appendix C. We next turn to the investigation of how other linguistics projects have used mark up languages in their work.

6.2 Other natural language projects that are use mark up languages

Since this project started in 1997, other projects have also realised the benefits of a mark up language annotation scheme. In this Section, I describe these projects and, where possible, compare their annotation schemes to our own.

6.2.1 The Text Encoding Initiative (TEI)

The information that I present in this Section is derived from the TEI website (www.tei-c.org).

The Text Encoding Initiative (TEI) started in 1987. Its aim was to develop guidelines for the encoding of machine-readable texts of interest to the humanities and social sciences. It created an SGML DTD and supporting documentation. There are now many projects that are conformant to the TEI specifications including the British National Corpus project (see Section 3.2.2 of Chapter Three).

Documents marked-up using the TEI standard consist of a header and a body. The header provides metadata about the text or corpus, which includes a description,

bibliographic information, details about the creator, encoding types used, details about the language and situation and the revision history. The concept of a header is not unlike the idea of an identification and status section in S1000D's data modules (S1000D 2006) or the idea behind the Dublin Core (W3C 1998).

The type of mark up used in the body of a TEI document depends on the project and its intended use. The TEI provides SGML abstract 'segment' elements for the marking up of parsed corpora (as shown in Table 6.2).

Element	Description	Remarks
<s>	Sentence-like division of a text	
<cl>	Clause	
<phr>	Phrase	
<w>	Word	has an attribute 'lemma' which is optional and identifies the spelling of the word's dictionary form.
<m>	Morpheme	
<c>	Character	

Table 6.2: The TEI mark up elements that are beneath a paragraph

All segment entries have the optional attributes `type` and `function`. The use of these attributes is a project decision. The `type` attribute value of phrase `<phr>` could be say 'noun' or 'verb' and `function` could be 'predicate' or 'subject' etc. In this way, the TEI has attempted to accommodate different grammatical models using an abstract mark up scheme. While it is easy to see that this model is designed to accommodate a Phrase Structure Grammar, the question must be asked if it could be used to encode any grammar? The answer to this question can only be 'yes' provided that the mark up model is used in a way in which it was not designed. For a systemic functional grammar, the `<phr>` element must be used to represent any syntax token (there is no concept of a phrase in SFG). An example sentence is shown in Figure 6.7. The reader should note that the use of elements in this way may be misinterpreted unless the business rules that governed their creation are understood.

```

<s>
  <cl>
    <phr type="S" function="element">
      <phr type="ngp" function="unit">
        <phr type="dd" function="element">
          <w>the</w>
        </phr>
        <phr type="h" function="element">
          <w>seagulls</w>
        </phr>
      </phr>
    </phr>
    <phr type="M" function="element">
      <w lemma="eat">ate</w>
    </phr>
    <phr type="C" function="element">
      <phr type="ngp" function="unit">
        <phr type="dd" function="element">
          <w>the</w>
        </phr>
        <phr type="h" function="element">
          <w>mackerel</w>
        </phr>
      </phr>
    </phr>
  </cl>
</s>

```

Figure 6.7: Using the TEI DTD to annotate the example sentence

Next, we turn to TEI's ability to mark discontinuous units. The TEI DTD provides an attribute **part** that can be used on any **<segment>** element to indicate whether or not the segment is fragmented by some other structural element. The values that **part** can take are **Y** (incomplete in some respect), **N** (complete), **I** (initial part of segment), **M** (medial part) and **F** (the final part). There is no standard way to indicate the parent or exact position of the discontinuous syntactic structure.

For marking up ellipted elements, one can assume that a **<phr>** element could be used with no child **<w>** element; however, there is no documented way of recording the type of an ellipsis.

The TEI goal is to provide a standard way of annotating corpora. This means that any text marked-up according to the TEI DTD/Schema is in a neutral format and will be able to be processed by any TEI compliant software application. It is therefore a good idea for any new project to encode its corpora using the TEI standard. Any scheme that is designed to be usable for any grammar has to be general and this

implies an abstract mark up scheme.²² To be able to satisfy the requirements of any grammar formalism, a more flexible approach is required. One way of achieving this in the TEI would be to introduce the concept of **specialisation**. Here, the mark up content models of elements can be modified by 'specialising' the base mark up elements in such a way that a project can define its own sub-structure beneath a **container** element. In doing this, a project can then benefit from using the TEI headers and be able to include parsed data according to their own annotation scheme.

Specialisation can be achieved in a number of ways. In SGML, marked sections allow content to be different depending on context. In XML, namespaces can be used to identify elements that belong to different schemas. Another approach has been adopted by IBM in the Darwin Information Architecture (DITA) (Priestley 2001). DITA allows core elements to be reused across documentation types and specialisation is achieved by supplying DTD modules that modify a mark up element's content for use in other contexts.²³

6.2.2 Other non-TEI projects that use mark up languages

McKelvie and Mikheev (1997) proposed the use of SGML indexing in their LT NSL project to provide fast access to SGML elements in a large corpus that is organised as one large SGML file.

Déjean (2000) used XML in his machine-learning system. Déjean produced a very simple DTD that was very similar to the TEI segment elements presented in Table 6.2 and hence his DTD is not suitable for SFG for the same reasons. Dejean's DTD cannot be used as it is for sentences that contain discontinuous units and ellipted elements.

In 2002, McKelvie, together with Carletta and Isard (Carletta, McKelvie and Isard 2002), recognised that linguistic annotations do not always fall into the hierarchical structures that can easily be described in an XML instance. They stated the need for overlapping hierarchies. Here, they use what they term **stand-off annotation**, where they apply an identifier to mark up elements and 'point' to them when they are being

²² While we see descriptive mark up elements in TEI's headers and body elements, the segment elements are abstract.

²³ There are also vast amounts of data already marked-up in SGML and XML. In the aerospace industry, for example, vast amounts of technical data are available encoded using the descriptive S1000D mark up scheme. To use this data as a TEI compliant corpus would mean that the descriptive elements already within the data would need to be removed. Specialisation techniques could be used to allow this data to be both encoded in TEI's scheme whilst retaining the semantics of its original mark up structure.

used either using an ID/IDREF mechanism or an XPOINTER. This method is similar to our own for indicating discontinuous units.

The MATE workbench was developed by Isard, McKelvie, Mengel and Möller (Isard et al 2003) to provide support for annotating corpora using XML and to provide a method of complex querying through linked files. While MATE does not specify a particular DTD /Schema structure, the examples in their work are of descriptive mark up. MATE also implements an XML database with a structure similar to ours.

TigerSearch (König, Lezius and Voormann 2003) (see also Section 3.4.4 of Chapter Three) uses an XML format (TigerXML) that maps to a directed acyclic graph for its corpus annotation scheme.²⁴ TigerXML has a corpus header encoded in XML, which contains the name of the corpus, the author, the date created, a description, details about the format and history. This is followed by a mandatory syntax definition data that contains information about the syntax labels used in the corpus and their meanings. This part of the TigerXML schema defines terminal and non-terminal nodes. Using this metadata, the Tiger project can be used for different grammar models.

Is TigerXML suitable for a Systemic Functional Grammar? The TigerXML schema is fairly flexible and is an abstract model as it stores syntax tokens within attributes. Like TEI, the TigerXML approach will not allow the distinction between elements of structure and units, however, in TigerXML this may be able to be recovered by using TigerXML's feature attributes. The application of this model to SFG becomes difficult when we consider the concept of terminal tokens versus non-terminal tokens. Unlike its PSG counterparts, in SFG, it is possible for the same element of structure to be either expounded by a lexical item or filled by another unit; as a consequence, the same syntax token may either be a 'terminal token' or a 'non-terminal' token. It appears that in TigerXML, a syntax token must either be one or the other but not both.

Because TigerXML works with ID/IDREF to point to edges and tokens, it is able to handle discontinuity in a similar way as our model (see Section 6.1.6.1). Provided the ID attribute contains a sequenced number, it can be used to restore the original structure of the parse tree. Ellipted elements would need to be handled as non-terminal elements and a feature instated to state the type of ellipsis; this method, while

²⁴ See also Mengel and Lezius (2000).

being feasible, is not ideal for SFG. A sample SFG sentence marked-up using the TigerXML annotation scheme is shown in Figure 6.8.

6.2.3 Comparing these mark up schemes

In this section we compare the TEI and TigerXML mark up schemes with our design criteria we listed in Section 6.1 and discuss their suitability for this project. Table 6.3 gives a list of advantages and disadvantages of each scheme.

6.2.4 What disadvantages are there with our chosen scheme?

Our chosen scheme is descriptive. This means that it can only be used for the Cardiff Grammar. This is a large disadvantage particularly in that we are not able to claim compatibility with the TEI standard and we will not be able to use our corpus in any TEI tools. Similarly, other corpora will not be expressible in our scheme.

For portability between corpus query tools, it is appealing to be able to express a corpus in TEI format. Because our corpus is in XML, transition into TEI is a reasonably trivial task provided work around methods are provided to encode the SFG concepts of elements, units and their relationships (as discussed above). However, a better method would be to allow TEI to include specialisation methods (as detailed in Section 6.2.1).

```

<corpus id="SFG1">
<head>
  <meta>
    <name>Sample sentence</name>
    <format>Cardiff Grammar</format>
  </meta>
</head>
<annotation>
  <feature name="pos" domain="NT">
    <value name="dd"/>
    <value name="h"/>
    <value name="M"/>
  </feature>
  <feature name="cat" domain="T">
    <value name="dd"/>
    <value name="h"/>
    <value name="M"/>
    <value name="ngp"/>
    <value name="Cl"/>
    <value name="S"/>
    <value name="C"/>
  </feature>
</annotation>
<s id="s1">
  <graph root="s1_504">
    <terminals>
      <t id="s1_1" word="the" pos="dd"/>
      <t id="s1_2" word="seagulls" pos="h"/>
      <t id="s1_3" word="ate" pos="M"/>
      <t id="s1_4" word="the" pos="dd"/>
      <t id="s1_5" word="mackerel" pos="h"/>
    </terminals>
    <nonterminals>
      <nt id="s1_504" cat="Z">
        <edge label="Cl" idref="S1_505"/>
      </nt>
      <nt id="s1_505" cat="Cl">
        <edge label="S" idref="S1_506"/>
        <edge label="M" idref="S1_3"/>
        <edge label="C" idref="s1_507"/>
      </nt>
      <nt id="S1_506" cat="S">
        <edge label="ngp" idref="S1_508"/>
      </nt>
      <nt id="S1_508" cat="ngp">
        <edge label="dd" idref="S1_1"/>
        <edge label="h" idref="s1_2"/>
      </nt>
      <nt id="S1_507" cat="C">
        <edge label="ngp" idref="S1_509"/>
      </nt>
      <nt id="s1_509" cat="ngp">
        <edge label="dd" idref="S1_4"/>
        <edge label="h" idref="S1_5"/>
      </nt>
    </nonterminals>
  </graph>
</s>
</corpus>

```

Figure 6.8: The TigerXML representation of the example sentence



Criteria	Remarks
(a) It can be used to represent a corpus analysed according to the syntax of a Systemic Functional Grammar and be able to maintain both (i) the distinction between items, elements and units, and (ii) the relationships of competence, filling, co-ordination and exponence.	The TEI and the TigerXML are abstract mark up schemes and so, in theory could be used for marking up a corpus with Cardiff Grammar syntax tokens. However, in both schemes there is no distinction between SFGs elements and units and therefore, unless some detailed business rules and / or SFG token renaming is performed, these relationships would be lost. ²⁵ Any workaround method employed to maintain the SFG criteria is very likely to affect the speed of queries.
(b) It can be extended to include the representation of intonation, punctuation and Participant Roles.	Both the TEI and TigerXML schemes provide no mechanism to record Participant Roles directly (as they were never designed for this purpose). To achieve it, there would need to be some renaming of SFG elements (e.g. Subject with an Agent participant role would have either a TEI type attribute or a TigerXML label attribute value of S_Ag . The methods in which the TEI and TigerXML records intonation requires further research.
(c) It is easy to understand by a human and easy to render	Abstract mark up schemes (such as TEI and TigerXML) are more difficult to understand by a human who is not used to the conventions.
(d) It must be editable (quickly) in a standard editor.	SGML/XML editor is required in particular for TigerXML because of its heavy use of ID/IDREF attributes.
(e) It must be able to use standard techniques to test the validity of sentences.	With an abstract approach it is necessary to use a business rules checker program to check that the sentences comply with the grammar. Standard tools such as DTDs and schemas will only check that a <phr> element is within a <cl> (in TEI) or that the <nt> element contains at least one <edge> (in TigerXML).
(f) It must be able to handle non-hierarchical (discontinuous structures).	This is possible in TEI, TigerXML (although the precise method of doing this in TEI needs further investigation).

Table 6.2: Comparing the TEI and TigerXML mark up schemes with the design criteria in Section 6.1.1

Criteria	Remarks
(g) It must be able to mark ellipsis, questionable items and elements and unfinished units.	Using the abstract TEI and TigerXML schemes, it is necessary to use methods for which the scheme is not designed to encode ellipsis, questionable items and unfinished units
(h) Queries must be performed rapidly, because the performance will affect the speed of the parser	Abstract mark up methods are always slower to query than descriptive methods. A query <i>"get me all mark up elements that have a generic identifier 'ngp'"</i> is quicker than one that asks <i>"get me all mark up elements that have a label attribute value of 'ngp' "</i> . This becomes an even bigger issue when we have to traverse the levels of a parse tree in the query (for example <i>"get me all ngps that have a dd expounded by 'the' "</i>). Queries are more difficult to express with an abstract mark up scheme.
(i) It must allow individual sentences to be extracted, modified and returned to the corpus, and also new sentences to be added.	This is easily achieved with TEI and TigerXML (and the scheme used in this project) as each sentence has a mark up element that represents it.

Table 6.2: Comparing the TEI and TigerXML mark up schemes with the design criteria in Section 6.1.1 (continued)

²⁵ Note that in TEI, the function attribute could be used for a purpose for which it was not intended to record that the TEI "phrase" contains an SFG element or an SFG unit.

6.3 Summary

Because mark up languages are well suited to annotating parsed corpora, we decided to use XML to represent sentences, query results and for the output of our parser.

One of the discoveries that we made was that it is possible to express a 'rule-based' grammar in a DTD or Schema and that it should be possible to reference these structures in a natural language parser. Such a method is ideal in this project because it would provide the ability to check sentences conform before they are added to the corpus. Unfortunately we were not able to use this directly in this project because of the complexity of the DTD produced from the naturally occurring texts in the corpus.

A feature of this work was deciding how to deal with cases in which the naturally occurring texts seem to break the 'rules' of what the linguist has modelled as 'the grammar'. One valuable side-effect of this research, therefore, has been that it has led Fawcett to improve and enrich the Cardiff Grammar by adding new elements of structure to it. In practice, it was difficult to distinguish between mistakes and rarely occurring linguistic phenomena that need to be accommodated in the model of syntax.

As described in Section 6.1.5.2, our mark up element generic identifiers are described using Cardiff Grammar tokens in a similar way to that shown in Figure 6.4 and Figure 6.6. The reasons behind this decision are:

- (a) we need to be able to edit the sentences quickly and using a representation understandable by a linguist in order to be able to create the new corpus as described in Chapter Nine, and
- (b) we need to be able to have corpus queries operate at an optimum speed and it is faster to find mark up elements that search attributes.

At this point, we will not use a DTD or Schema to check sentences as we shall rely on the XML being well-formed. The main reasons behind this choice are:

- (a) the difficulty in creating a DTD due to the nature of the naturally occurring texts in the corpus, and
- (b) avoiding having to maintain the DTD when a new structure is identified.

As detailed in Chapter Eighteen, we plan to produce a DTD in Phase Two by using a new version of **createDTD**.

The next chapter will include a discussion about storing our marked up data in an XML conforming relational database schema.

Chapter Seven

Defining the corpus database schema

As we saw in Chapter Six, mark up languages are ideally suited to annotating a corpus. In this chapter we investigate the methods that can be used to store this marked up data in such a way that the following three requirements are met:

- (a) both **sentences** and **syntactic structures** are quickly and easily accessible in a **corpus query tool** and to a **parser**,
- (b) it is possible to add new sentences, or to extract, modify and replace existing sentences in the corpus,
- (c) when sentences are modified or added, the information about the syntactic relationships is also updated.

Due to their ability to manage large numbers of records in commercial applications, our hypothesis is that these requirements can be easily met by using a relational **Database Management System (DBMS)** which has been equipped with suitable **indexes**.

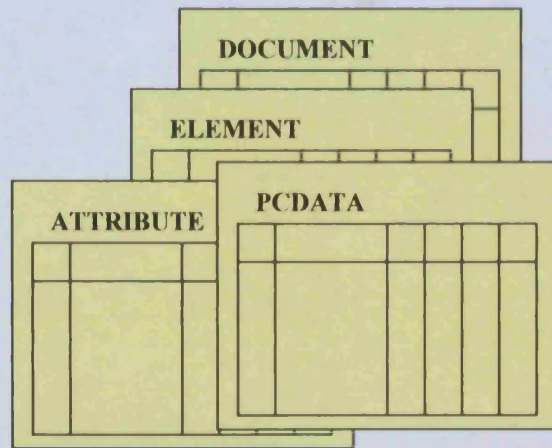
This chapter therefore starts in Section 7.1 with a survey of the field, in order to discover the methods that have been used by other projects to store marked up data in a database. It then goes on to describe how we solved the problem by using a **native XML** relational database.

Figure 7.1 shows a diagram of the database schema. This stores:

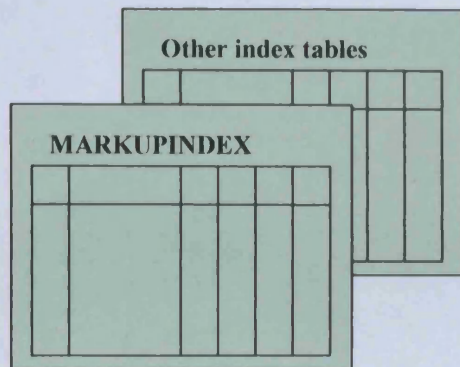
- (a) the corpus data - in the **corpus tables**,
- (a) indexes to the corpus tables - provided by the **corpus index tables**,
- (b) **probabilities tables** - used by the parser and updated from the corpus tables,
- (c) tables used to store the data structures created by the parser - the **parser working tables**.

Section 7.2 introduces the **corpus tables** and the **corpus index tables**. Chapter Fourteen will introduce the **probabilities tables** and the **parser working tables**. Full details of all tables and fields in the database schema are given in Appendix D and Appendix H.

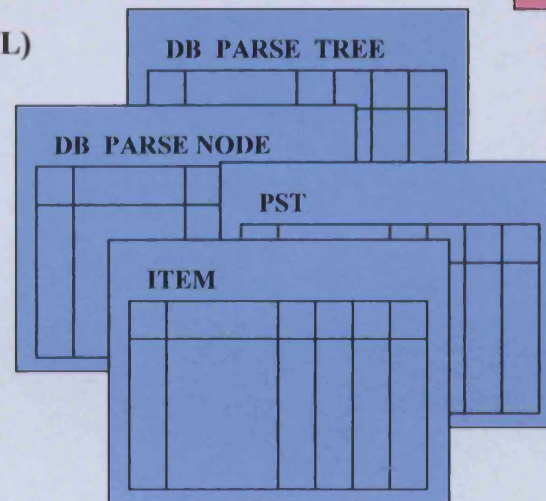
The Corpus Database (corpusDB)



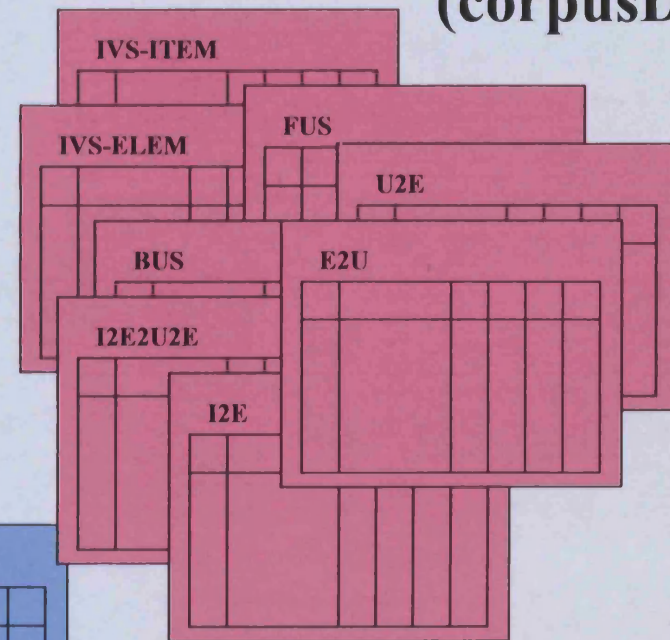
The corpus tables (native XML)
(see Section 7.2.1)



The corpus index tables
(see Section 7.2.2)



The parser working tables
(see Chapter Fourteen)



The probabilities tables
(see Chapter Fourteen)

Figure 7.1: The Corpus Database schema

7.1 Mark up and databases - surveying the fields

In this section we discuss the ways in which marked up data can be stored in a database management system, first in its applications in industry and then in its later applications in academia.

7.1.1 Applications in industry

There have been two main methods for storing marked up data in databases:

- (a) the data model approach, and
- (b) the native approach.

These are described in this section.

7.1.1.1 The data model approach

One approach to storing SGML in a database is to make the database closely model the data being stored. This is termed the **data model approach** and it was fairly common in the early days of SGML. With this approach, systems would extract the information from the SGML document and store it in tables that reflected the mark up structure of the Document Type Definition (DTD). For example, for a document marked up using a **descriptive mark up scheme** (see Section 6.1.4.2 of Chapter Six), and containing the procedural tasks of a maintenance manual, one would expect to have tables with similar names to the following: **TOOL**, **REQUIRED_PERSON**, **WARNING**, **CAUTION** and **STEP**. Things changed as implementers realised the limitations of this approach, which were: (a) changes in the DTD affected the database structure, and (b) the database was only able to hold data for one DTD. In order to overcome this, and allow multiple document types to be stored in the same database, **DTD independence** was a primary design goal.

7.1.1.2 The native approach

Native SGML databases began to appear.¹ These systems were designed to store SGML **instances** rather than the data-oriented tables, and were said to be **SGML aware**; their storage unit was an SGML document. There were three major approaches all claiming:

- (a) some degree of **DTD independence**, and
- (b) the ability to store native SGML data.

The first approach simply extracted important **metadata** from the SGML documents and placed it into dedicated database tables, and then stored either parts or the whole SGML file as text in a database binary object. Typically, an SGML search engine was able to retrieve data from the SGML data held in the database. The searches were slow and soon the implementers began extracting data that was often required for a search into dedicated database tables.

The second approach required processing of the DTD to define a database structure into which documents could be loaded. This resulted in database tables that would have one table for each SGML mark up element type. For example, in a DTD that had a mark-up element `<para>`, the system would create a **PARA** table after reading and processing the DTD. This approach worked well for simple data-oriented SGML applications but suffered when documents marked up to different DTD structures had to be stored in the same database or where the DTD was constantly being modified. A further disadvantage was that the approach resulted in a large number of database tables for SGML documents of fairly simple construction (a point also noted by Jagadish et al (2002), in their later work with a native XML database).

The third, and most successful approach, is the one used here. This demonstrates complete DTD independence and is today, perhaps the most common approach adopted by commercial off-the-shelf systems. They used database tables that described the parts of an SGML document (**DOCUMENT**, **ELEMENT**, **ATTRIBUTE** and **PCDATA**).² Here, a database table would store elements and be able to maintain parent, child and sibling relationships by assigning unique database object identifiers to them. This approach needed a method of handling **inline mark up elements**. These occur when a parent element can contain a mixture of data and sub-mark up elements. In the example that follows, `<xref>` is an empty inline mark-up element:
`<para>Use the Extractor <xref xrefid="SE0001"> to remove the
clamp.</para>`

With this approach, the database management system also needed to know which mark up elements were **empty**, or had optional **end tags**. This was either achieved by using knowledge extracted from the DTD, or by pre-processing the SGML and using a stack to determine any absent end-tags.³

¹ The origin of the term native is not known but first started to appear in the mid-1990s, and is thought to have derived from the term **native SGML files**.

² Other tables existed that are not used in this work for metadata and SGML entity handling etc.

³ We used this latter method in this project. The issue disappeared when we migrated to XML.

Native SGML database systems use two typical approaches to solve this problem. The first is to identify the inline mark up elements store them within the parent's PCDATA record, and the second is to introduce a special pseudo mark up element which is wrapped around the PCDATA spans. The main disadvantage of the former method is that inline mark up elements can not be queried like any non-inline mark up element and that some mark up elements can appear both inline and also non-inline. While the disadvantage of the latter method is that a special mark up wrapper element has to be defined (that is added to a document when it is loaded and stripped when a document is extracted from the database), the advantage is that inline mark up elements are treated like any other element.

```
<para><data>Use the Extractor </data><xref xrefid="SE0001"><data> to  
remove the clamp.</data></para>
```

Perhaps one of the first native SGML systems of this type was designed and implemented by myself in 1993 in the Rolls-Royce Common Source Database document management system (Day 1993c).

At the time, there was a disadvantage to the approach which was due to the processing time needed to break a large SGML document up into its constituent pieces when a document was loaded and to reconstruct it again when it is extracted.⁴ However, the benefits of the approach often outweighed the performance issues, and were (a) once in the database, data queries were very quick (although users needed knowledge of the DTD structure), and (b) the sharing of SGML fragments between different SGML documents became possible and these systems became known as **Content Management Systems (CMS)**.⁵

A similar method to my own was employed by a Scandinavian company called Corena in their Life*CDM system, which was designed and marketed from the mid 1990s (details can be found on www.corena.dk). The software allowed parts of documents to be shared in other documents and mark up elements in a document extracted from the database were given object IDs that reflected a unique key applied to it in the database. This enabled the detection of changes when the document was returned to the database and these could then be acted upon. Should a shared mark up

⁴ This performance issue has almost disappeared in modern systems due to quicker computers.

⁵ The first use of the term Content Management System is unknown but probably stemmed from standards organisations such as AECMA and CALS.

element be changed, the affects on its shared locations could be assessed. This approach worked well unless the object IDs were altered outside of the database.⁶

As SGML became more popular, a greater number of document management systems started to appear, and systems that stored mark up data in a similar way to the one presented here became common. With the introduction of XML, the number of commercially available CMS systems further increased and some systems based their technology on top of object oriented database management systems (some of these claimed better performance). However, many of these systems were not commercially successful because of the fact that large organisations demanded a relational approach in order to conform to their corporate data management strategies.⁷

7.1.2 Applications in research

Research applications that store mark-up documents in a database started to become more popular with the advent of XML. Many of these systems where, perhaps unsurprisingly, similar to the designs created for industry as described above. Some of these applications were in the field of computational linguistics.

TIMBER (Jagadish et al 2002) designed a native XML database that was based on a 'bulk algebra for manipulating trees'. Their main reason for adopting an XML database in the TIMBER project being that of performing rapid queries. Their approach mapped their documents as nodes, and they used nodes for mark up elements and data, but their attributes were grouped together into one node. The affect of doing this is that any attribute processing becomes a matter of string manipulation and is therefore slower than the normalised approach that we take in this project.

The MATE system (Isard et al 2003) implemented a similar method to ours except that they had the additional tables to store the DTD definitions. The MATE system was used for corpus annotation (see also Section 6.2.2 of Chapter Six).

The Tiger Project (Lezius and König 2000) used XML as a corpus annotation scheme and provided a corpus query Tool called TIGERSearch (see also Sections 3.2.8 and 3.4.4 of Chapter Three). However their methods for storing the corpus within the corpus database needs further investigation.

As the popularity of XML and the adoption of the Text Encoding Initiative (TEI) (see Section 6.2.1 of Chapter Six) increases, it is highly likely that the use of XML in

⁶ It further required a modification of an industry standard DTD for internal use; this was not popular with the system administrator who had to manage an Internal and External DTD.

⁷ Many of the non-relational systems developed in the 1990s are no longer commercially available.

corpus linguistics will become more popular. While projects like the BNC are already storing data in SGML and XML, others are beginning to look at it as an import and export mechanism to their systems. The ICE project (see Section 3.2.6 of Chapter Three) is looking at import and export to the TEI DTD / Schema (Sean Wallis, UCL, personal communication, February 2007). One can assume that many of these projects will have desires now or in the near future to store their data in a native XML database.

7.1.3 Choices made for this project

This section concludes with the decisions that I made for the storage of mark up data for this project. The survey of the field included applications in industry (where such databases have been used since the early 1990s) and the more recent academic applications (where attention was given to those applications connected with linguistics).

The conclusion of the investigation was that a native database of the type described in Section 7.1.1.2 is best suited to the requirements defined at start of this chapter. The native XML tables are therefore implemented in a relational database management system, and it offers the following advantages:⁸

- (a) **Rapid access speeds.** The speed of access to the data is of prime importance, particularly when used in the parsing process.⁹
- (b) **Data handling.** We need to be able to retrieve and process potentially, thousands of records at a time.¹⁰
- (c) **Querying.** A relational database system provides the Structured Query Language (SQL) and we saw this as a good way of expressing queries and retrieving data.
- (d) **Use of DBMS in parsing.** We wanted to store the parser's working data in the database and use it to investigate the structures that are created using a step-by-step incremental parse (see Chapter Sixteen).

Next, the implementation details of these native XML tables is introduced. The full details are given in Appendix D.

⁸ An alternative approach is offered by an object oriented database management system. However, the relational database management system gives a solution that has been proven by the author in industry applications.

⁹ The speed of access to the database was found to be quicker than our earlier attempts at indexing an SGML file and our later experiments with an XML Document Object Model (DOM).

7.2 The database schema: the corpus and the corpus index tables

In this section we introduce database schema design. Section 7.2.1 introduces the **corpus tables** which are implemented as native XML tables, and Section 7.2.2 introduces the **corpus index tables**.

7.2.1 The corpus tables

The native XML tables are at the core of the corpus database, they are called the **corpus tables** and are used to store the text and associated syntax of the corpus itself. They are also used to generate each of the other tables.

Chapter Six showed that marked up documents contain four central objects:

- (a) the **marked up document** itself,
- (b) the **mark up elements**,
- (c) the **mark up attributes**, and
- (d) **data** that belong to the mark up elements.

The chosen native XML approach described in Section 7.1.1.2 introduced four database tables that are used to store the mark up objects, these are called the **corpus tables** and they essentially form a native XML database.¹¹

- (a) the **DOCUMENT** table.¹²
- (b) the **ELEMENT** table
- (c) the **ATTRIBUTE** table
- (d) the **PCDATA** table.¹³

The use of these tables is best described by the use of an example, which is given in the next section. The full details of the tables are given in Appendix D.

¹⁰ The parser needs to retrieve potentially hundreds of records, while the corpus query tool may require fast access to potentially thousands of records.

¹¹ The native XML tables are able to store native SGML data. Further, with the addition of entity, comment, and processing instruction tables (which are not needed for this project), these native tables are able to store ANY mark up data.

¹² I decided to remain true to the SGML/XML environments by naming this table **DOCUMENT**; it could have been more aptly named **SENTENCE** for this work.

¹³ To be able to handle inline mark up elements whose parents can contain a mix of other mark up elements and text (see Section 7.1.1.1), we use a pseudo mark up element called **<data>** and these elements are also stored in the **ELEMENT** table.

Figure 7.2 shows a syntax tree of an example sentence with the following XML representation:

```

<Z docid="1" docname="6ABIHS#23" id="1">
  <Cl id="2">
    <S PR="Agent" id="3">
      <ngp id="4">
        <dd id="5">
          <data id="6">the</data>
        </dd>
        <h id="7">
          <data id="8">seagulls</data>
        </h>
      </ngp>
    </S>
    <M id="9">
      <data id="10">ate</data>
    </M>
    <C PR="Affected" id="11">
      <ngp id="12">
        <dd id="13">
          <data id="14">the</data>
        </dd>
        <h id="15">
          <data id="16">mackerel</data>
        </h>
      </ngp>
    </C>
  </Cl>
</Z>

```

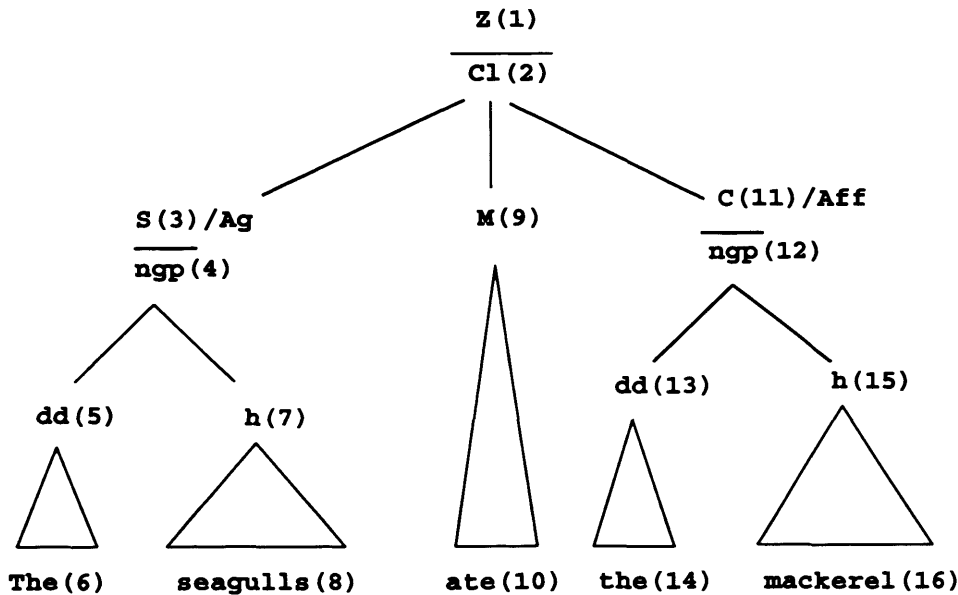


Figure 7.2: A sample sentence showing id attributes

Tables 7.1, 7.2, 7.3 and 7.4 show how this XML is stored in the corpus tables. The **DOCUMENT** table (Table 7.1) stores details of the sentence in terms of its cell

identifier (see Section 3.2.9 of Chapter Three) and gives it a unique identifier. The **ELEMENT** table (Table 7.2) contains details of the mark up elements that belong to the sentence, the field named **ELEMID** uniquely identifies the element within the sentence, and these are shown in brackets in Figure 7.2. The field named **GI** contains the generic identifier (which is the mark up element's name).

The hierarchy of the mark up data is maintained using the **PARENTID** field. A mark up element's child mark up elements are given a **PARENTID** value that is equal to the parent's **ELEMID** value. For example, the Clause (**C1**) in the example sentence has an **ELEMID** value **2**; its children: the Sentence (**S**), the Main Verb (**M**) and the Complement (**C**), all have their **PARENTID** fields set to **2** (i.e. the **ELEMID** of the Clause). The order of the mark up elements within the parent is maintained by the ascending order of the **ELEMID** fields within the parent mark up element.

The **ATTRIBUTE** table is used to store a mark up element's attributes and the example shown in Table 7.3 shows two attributes which represent the Participant Roles on the Subject (**ELEMID = 3**) and the Complement (**ELEMID = 11**).

The **PCDATA** table (Table 7.4) stores the text of the items in the sentence and has an **ELEMID** which identifies the mark up element to which the item belongs.

In this project, each mark up element in the **ELEMENT** table corresponds to (a) an element of structure, (b) a unit, or (c) a item. If the **ELEMENT** is an element of structure, the **PARENTID** will indicate the unit that contains the element of structure. If the **ELEMENT** is a unit, **PARENTID** will indicate the element of structure filled by the unit. If the **ELEMENT** is an item, **PARENTID** will indicate the element of structure that the item expounds.

The database schema for these tables, and a full description of the fields can be found in Appendix D.

DOCID	DOCNAME	TAG1	TAG2	TAG3	TAG4	TAG5	TAG6	TAG7..10
1	6ABIHS#23	6	A	B	I	HS	23	NULL

Table 7.1: Contents of the DOCUMENT table for the sample sentence

DOCID	ELEMID	GI	PARENTID
1	1	Z	-1
1	2	C1	1
1	3	S	2
1	4	ngp	3
1	5	dd	4
1	6	DATA	6
1	7	h	4
1	8	DATA	7
1	9	M	2
1	10	DATA	8
1	11	C	2
1	12	ngp	11
1	13	dd	12
1	14	DATA	13
1	15	h	14
1	16	DATA	15

Table 7.2: Contents of the ELEMENT table for the sample sentence

DOCID	ELEMID	ATTID	ATTNAME	ATTVAL
1	3	1	PR	Agent
1	11	1	PR	Affected

Table 7.3: Contents of the ATTRIBUTE table for the sample sentence¹⁴

DOCID	ELEMID	PCDATA
1	6	The
1	8	seagulls
1	10	ate
1	14	the
1	16	mackerel

Table 7.4: Contents of the PCDATA table for the sample sentence

This section described the **corpus tables**. These are used by the corpus query tool and for the automatic creation of the **probabilities tables**. The provision of the **corpus index tables** will speed up the process of finding and displaying sentences and syntactic relationships, and it is to these tables that we turn next.

7.2.2 The corpus index tables

The **corpus index tables** provide the rapid access to the mark up data in the **corpus tables**. Their main use is for finding the results of queries that are made from the **corpus query tool** (see Chapter Eight), and for the purpose of modifying the corpus (see Chapter Nine). The main index table (called **MARKUPINDEX**) has been designed for speed of access and therefore replicates some of the information in the

¹⁴ Note that the **id** and the **docname** attributes in the mark example of Figure 7.2 are not stored in the attribute table because these are the **elemid** and the **docname** fields of the element and document tables respectively.

ELEMENT table.¹⁵ It provides multi-directional access into the **ELEMENT** table using **ELEMID** as pointers. Each row of the main index table represents a mark up element (which I call the 'given mark up element' below) and includes the following information:

- (a) the given mark up element's **generic identifier (GI)**,
- (b) the given mark up element's **ELEMID** value,
- (c) a string of **ELEMIDs** and a string of **GIs** that represent the given mark up element's children,
- (d) a string of **ELEMIDs** and a string of **GIs** that represent the mark up elements that exist above the given mark up element,
- (e) a string of **ELEMIDs** and a string of **GIs** that represent the mark up elements that exist to the left of the given mark up element as its left siblings,
- (f) a string of **ELEMIDs** and a string of **GIs** that represent the mark up elements that exist to the right of the given mark up element as its right siblings.

Table 7.5 shows a portion of the main index table that shows the example sentence in Figure 7.2.

¹⁵ In database modelling terms, the index table contains redundant information. For example, the Generic Identifier (**GI**) can be gained by using the **ELEMID** field as a foreign key into the **ELEMENT** table. Although it can be argued that this approach represents a 'purer' data model, the speed of access would be slower as it would involve a query that needs a table join. There is a further argument that the string fields should be 'normalised' so that access to them does not involve string processing, however, an advantage of the chosen method is that it is also 'human readable' when the table is viewed as single table. Further, the speed of access of the string processing functions were more than adequate for the corpus query tool, and for the process of corpus modification work.

DOC ID	ELEM ID	CHILD IDS	GI	RIGHT_GIS	ABOVE_GIS	ABOVE IDS	LSIB GIS	RSIB GIS	LSIB IDS	RSIB IDS
1	1	2	Z	Cl						
1	2	3,9,11	Cl	S M C	Z	1				
1	3	4	S	ngp	Cl Z	2,1		M C		9,11
1	4	5,7	ngp	dd h	S Cl Z	3,2,1				
1	5	6	dd	DATA	ngp S Cl Z	4,3,2,1		h		7
1	6		DATA	"the"	dd ngp S	5,4,3,2				
					Cl Z	,1				
1	7	8	h	DATA	ngp Z Cl Z	4,3,2,1	dd		5	
1	8		DATA	"seagulls"	h ngp S Cl	6,4,3,2				
					Z	,1				
1	9	10	M	DATA	Cl Z	2,1	S	C	4	11
1	10		DATA	"ate"	M Cl Z	10,2,1				
1	11	12	C	ngp	Cl Z	2,1	S M		4,9	
1	12	13,15	ngp	dd h	C Cl Z	11,2,1				
1	13	14	dd	DATA	ngp C Cl Z	12,11,2		h		15
						,1				
1	14		DATA	"the"	dd ngp C	13,12,1				
					Cl Z	1,2,1				
1	15	16	h	DATA	ngp C Cl Z	12,11,2	dd		13	
						,1				
1	16		DATA	"mackerel"	h ngp C Cl	15,12,1				
					Z	1,2,1				

Table 7.5: Contents of the MARKUPINDEX table for the sample sentence (note some fields omitted for clarity)

In addition to the mark up fields, the table also contains linguistic fields for identifying:

- (a) if the element that is represented by the given element is ellipted,
- (b) if the item, element or unit represented by the given mark up element is initial in the sentence and initial in its parent (both taking into account ellipsis and ignoring it).

The corpus index tables include other tables that are used by the corpus query tool. The purpose of these are:

- (a) to hold information from the generation of the published reports,
- (b) for the recording of syntax tokens, their descriptions and their types,
- (c) for holding information that is of a temporary nature, for example, the results for a concordance, or a query about syntax.

In the summary of Chapter Five, it was stated that mark up languages can be used not only for corpus annotation, but for representing syntactic relationships. This is achieved using the MARKUPINDEX table. As Table 7.5 shows, this table contains

multi-level information. For example the nominal group (element 12), and the deictic determiner (element 13) show:

- (a) that the nominal group (**ngp**) contains a deictic determiner (**dd**) and a head (**h**) (by using the **GI** and **CHILDGI** fields) and this can be considered equivalent to the 'rewrite rule' notation given in Chapter Five.
- (b) above the deictic determiner (**dd**), the **ABOVEGIS** field shows that there are the following elements and units **dd ngp C C1 Z** and this can be considered equivalent to O'Donoghue's (1991a) vertical strips.¹⁶

In addition to these relationships that were described in Chapter Five, the **MARKUPINDEX** table also provides information about element co-occurrences in the **LEFTSIB** and **RIGHTSIB** fields.¹⁷ Furthermore, by connecting the identifiers of each mark up element in terms of their parents, children and siblings it is possible to answer complex queries that involve multiple levels of the syntax tree, and this will be discussed in the next chapter.

This now concludes our discussion on the **corpus tables** and the **corpus index tables**. The other sets of tables that complete the database schema, namely the **probabilities tables** and the **parser working tables** are introduced in Chapter Fourteen.¹⁸

7.3 Summary

This chapter has outlined the design of the corpus database. We saw, in Figure 7.1 and Section 7.2, that the corpus database is broken down into the following groups of tables:

- (a) the **corpus tables**, which store the text and syntax of the corpus itself,
- (b) the **corpus index tables**, which provide rapid access to the corpus data,
- (c) the **probabilistic tables**, which are queried by the parser,
- (d) the **parser working tables**, which store the parser's working data.

Chapter Six concluded that mark up languages are an ideal method for annotating a corpus. This chapter has shown that the combination of using mark up and a

¹⁶ The vertical strip can be extended to include the item by using the information in the entry that represents the item.

¹⁷ This is useful for answering questions about, for example, the elements that can come before a given element irrespective of what follows it.

¹⁸ The probabilities tables provide a further level of indexing above the **MARKUPINDEX** table that allow the parser to run at optimum speed.

database provides a rich method of representing the syntax relationships in a corpus. It has further shown that a marked up corpus can be stored in a relational database, and that index tables can be used to rapidly find and retrieve the information from the corpus that is stored in the corpus tables. By storing the mark up in the database, the final two requirements that were given in Section 6.1.1 are met. The first of these is the ability to quickly find and retrieve information in the corpus, and this is achieved through the use queries to the tables and indexes described in this chapter. The second requirement is the ability to add and extract sentences, to modify them and return them, or to be able to add new sentences.

The corpus database allows the sentences to be copied from the database into XML files and these can be edited in the **corpus editor**. The corpus query tool allows the user to return modified sentences back into the database after they are modified, and this action automatically updates the indexes. The parser that will be described in Part Four, can add new sentences to the database and this action also updates the tables and indexes automatically.

In the next chapter we will see one of the purposes of these tables when we describe the **corpus query tool**, which proved to be an invaluable aid for the development of the parser and for creating a new version of the corpus.

The remainder of the corpus database tables will be introduced in Chapter Fourteen. Full details of all tables can be found in Appendix D.

Chapter Eight

Enhancing the Interactive Corpus Query Facility

This chapter describes the **corpus query tool** - the enhanced version of the Interactive Corpus Query Facility (ICQF+). The original version was first developed in 1993 to provide COMMUNAL with a research tool which could be used to interrogate a parsed natural language corpus (Day 1993a).

The present project required an improved version of ICQF for the following purposes:

- (a) to provide us with a research tool to help us in the development of the parser itself (which is described in Part Four of this work), and
- (b) to enable us to create a corpus that is analysed according to the latest version of the Cardiff Grammar (see Chapter Nine).

The new version, which provides the ability to perform much more powerful queries, is described in this chapter. Section 8.1 provides a brief history of the original version of ICQF and Section 8.2 explains (a) why the new version was developed, and (b) how it is integrated with the corpus database described in Chapter Seven. Section 8.3 introduces the new version of the Corpus Query Language (CQL), and provides examples of the types of query that can be performed. Finally, Section 8.4 provides a very brief walkthrough of ICQF, using screenshots where appropriate, and describes how certain functions proved useful for particular aspects of this project.

8.1 History

The original version of ICQF, which was developed in 1993 as part of my MSc dissertation, enabled the user to perform **queries about items**, **queries about syntax** and limited **concordances**. Additionally, it was used to create several lists that provided COMMUNAL with valuable reference material about the syntax structures that appeared in the POW Corpus.

ICQF also played a vital role in two other projects:

- (a) it was one of the sources of probabilities for Weerasinghe's (1994) parser, and

- (b) it helped improve the description of English syntax in the Cardiff Grammar (Fawcett 2000a).

It was also provided to two other universities (Leeds and Sheffield) who wanted to interrogate the POW Corpus.

8.2 Towards ICQF+

As stated in the introduction to this chapter, a new version of the corpus query tool was required for this project in order to develop the algorithms and the probabilistic data that is used by the parser, and to enable us to create a new corpus that is annotated to the latest version of the Cardiff Grammar.

In developing the parser which is described in Part Four, we needed more powerful queries than those that could be provided by the original version. In particular, these involved

- (a) the ability to ask questions about syntax structures that occurred at any level in the syntax tree and,
- (b) the ability to extract information about vertical strips, i.e. queries about the elements and units above an element of structure or an item.

The original version did not allow sentences to be modified. This was essential for the task of updating the corpus to the latest version of the Cardiff Grammar. There were two types of change needed. The first type were **global changes** in which the linguist specified a syntax pattern that involved a mix of items, elements and units and their siblings; for these, the ability to express them in SQL and execute them as programs was required. The second type of change were those that were **manual** and could not be changed automatically. For these, the corpus query tool helped locate the sentences by using queries, and extracting them as XML so that they can be edited in a **sentence editor** and, afterwards, returned to the corpus.

In the new version, the basic functions of the original version were retained and substantially improved. In particular, the Corpus Query Language (CQL) is enhanced to support the new types of query required for this project (see Section 8.3).

Reports extracted from the corpus were described in Sections 3.4.1 and 3.4.2 of Chapter Three. These reports were available in the original version of ICQF, but were not part of the main program. In the new version the reports are improved and fully integrated. The report functions are described in Section 8.4.3.

Section 8.4 provides a walkthrough of some of the ICQF functions in order to demonstrate ICQF+'s new interface. Section 8.5 briefly describes the corpus modification programs that were used to modify the corpus.

8.3 Querying in ICQF+ - the Corpus Query Language

The Corpus Query language (CQL) as described in Day (1993a), has been considerably extended in ICQF+. The new features are defined to support the more complex queries that were required by this project.

CQL is a query language which allows the user to express corpus queries in a simple and compact format. It is sub-divided into two broad categories **queries about items** (see Section 8.3.1) and **queries about syntax** (see Section 8.3.2).

All queries are entered by typing them into the **Enter query** text box (item 9, Figure 8.1) on ICQF+'s **query and results** form. This is accessible from the **query menu**.

8.3.1 Queries about items

Queries about items allow the user to find occurrences of an item either together with the element it expounds, or irrespective of the element.

To find all uses of the lexical item irrespective of the element of structure that it expounds, the user simply enters the item. For example, to find all instances of the item **make**:

QUERY1: make

And to find all instances of an item when it expounds a particular element of structure, the **rewrite operator** is used:

QUERY2: M->"make"

The **asterisk wildcard** can also be used to specify parts of words. In the following example, any item that starts with the letters **rat** are returned (e.g. **rat, rats, rate, rates, ration, ..**)

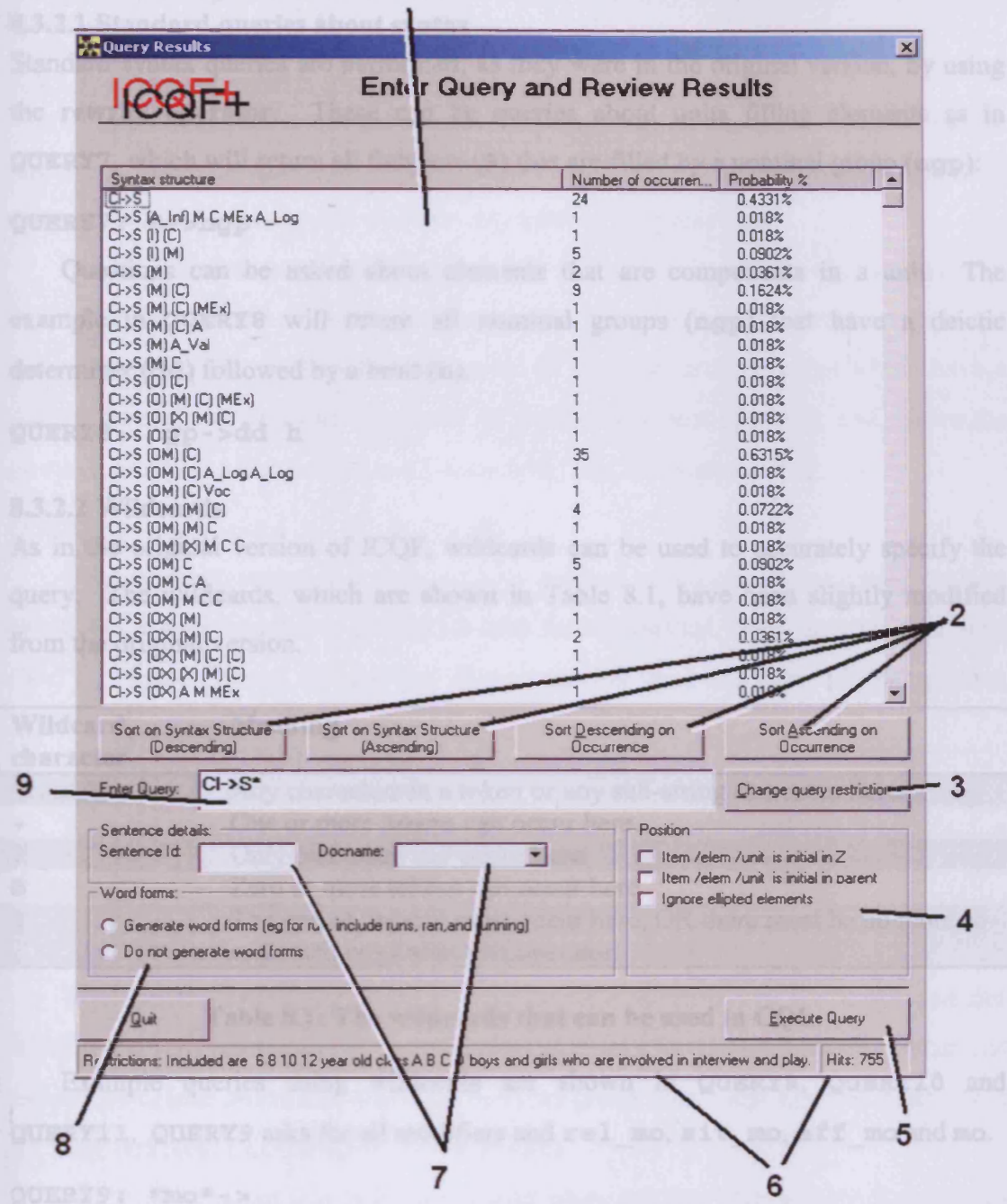
QUERY3: rat*

A new facility was added to ICQF+ to allow it to include different items in the same query; this was useful, as it allowed the linguist to retrieve sentences for similar words (for example, to see if they operate in the same way). The following queries demonstrate how this is done by using the **semicolon operator**.

QUERY4:make; makes

QUERY5:M->"make";M->"makes";M->"made";M->"making"
QUERY6:one;two;three;four;five;six;seven;eight;nine

Another useful function added to ICQF+ is the ability to automatically generate all the forms of a word. This capability uses algorithms that were originally developed for COMMUNAL's generator (Fawcett, Tucker and Young 1988). To use this function, the user presses the option button (Figure 8.1, Item 8) and enters, say, a verb stem. ICQF+ then searches for all forms of that verb. The method does rely, however, on the user entering a word **stem** and not a **form** (e.g. **run** and not **running**).



1. Results display: double click to see example sentences
2. Sort options
3. Change query restrictions (see Figure 8.2)
4. Restrict query to sentence initial, initial in parent and include / exclude ellipsis
5. Run the query
6. Status bar showing hits and restrictions
7. Find by docname (POW cell identifier) or by unique identifier
8. For item queries, generate all word forms
9. The Enter Query text box

Figure 8.1: ICQF+'s query and results form

8.3.2 Queries about syntax

8.3.2.1 Standard queries about syntax

Standard syntax queries are performed, as they were in the original version, by using the **rewrite operator**. These can be queries about units filling elements as in **QUERY7**, which will return all Subjects (**S**) that are filled by a nominal group (**ngp**):

QUERY7: S->ngp

Questions can be asked about elements that are components in a unit. The example in **QUERY8** will return all nominal groups (**ngp**) that have a deictic determiner (**dd**) followed by a head (**h**).

QUERY8: ngp->dd h

8.3.2.2 Wildcards

As in the original version of ICQF, wildcards can be used to accurately specify the query. The wildcards, which are shown in Table 8.1, have been slightly modified from the original version.

Wildcard character	Meaning
*	Any characters in a token or any sub-string can occur here.
+	One or more tokens can occur here.
?	Only one token can occur here.
@	Zero or more tokens can occur here
!	The end of the unit must occur here, OR there must be no more co-ordinated units after this operator.

Table 8.1: The wildcards that can be used in CQL

Example queries using wildcards are shown in **QUERY9**, **QUERY10** and **QUERY11**. **QUERY9** asks for all modifiers and **rel_mo**, **sit_mo**, **aff_mo** and **mo**.

QUERY9: *mo*->

QUERY10 asks for all single elements that can occur between a deictic determiner (**dd**) and a head (**h**) in a nominal group (**ngp**):

QUERY10: ngp->dd @ h

The end of unit operator (!) is important in the CQL. It indicates that the end of the unit must occur in this place. In **QUERY10**, for example, the results will include structures where there are other elements after the head (**h**). In **QUERY11**, the head (**h**) must be the last element in the nominal group (**ngp**).

QUERY11: `ngp->dd h !` [find all **ngps** that only have a **dd** and a **h** (i.e. no qualifiers)]

8.3.2.3 Advanced queries about syntax

The original version of ICQF had an **above** operator which was able to look for syntax structures which looked upwards in a parse tree. In ICQF+, this has significantly extended this to allow the user to query any level of the parse tree.

In CQL, if necessary, brackets are used to delimit levels, and the above operator **^** is used to indicate that the structure given in the brackets is above the structure to the left of the brackets. **QUERY12** will return all nominal groups (**ngp**) which have a deictic determiner (**dd**) and a head (**h**) as their component elements, and, above the nominal group, is a Subject (**S**) in a Clause (**C1**) that fills sentence (**Z**):

QUERY12: `ngp->dd h (^ S C1 Z)`

Queries about syntax that involve only vertical strips can be expressed without using the rewrite operator. **QUERY13** asks for all vertical strips that have an apex (**ax**) in a quality group (**qlgp**) that fills a modifier (**mo**) in a nominal group which fills a Subject (**S**):

QUERY13: `^ ax qlgp mo ngp S`

Only the asterisk wildcard can be used in **above queries**. **QUERY14** asks for all Clauses (**C1**) that are embedded within another Clause (**C1**) at any level:

QUERY14: `^ C1 * C1`

It is possible to include an item or an element to the left of the above operator when the rewrite operator is not used. **QUERY15** asks for all occurrences of **the** that expound a deictic determiner (**dd**) in a nominal group (**ngp**) that fills a Subject (**S**) in a Clause (**C1**) that fills the sentence element (**Z**).

This type of query was extremely useful when the **IVS-ITEM** table was created (see Chapter Fourteen). The **DATA** operator is used to signify that the query is for an item.

QUERY15:DATA(the)^dd ngp S C1 Z

The sibling operators **<** and **>** are used to find information about the elements that can occur before and after the given operator. **QUERY16** asks for the elements that can come after a Main Verb (**M**):

QUERY16: M>*

QUERY17 asks for all structures that can come before a Main Verb (**M**) and **QUERY18** asks for all occurrences of a Main Verb (**M**) that is followed by two complements (**C**):

QUERY17: *<M

QUERY18: M>C C

The **percent operator** allows the user to get probabilities about elements that can occur. **QUERY19** asks for the probabilities of all elements that can follow a Main Verb (**M**) in a Clause (**C1**) that has a Subject (**S**), and an Operator (**O**) before it. Only a single element will be returned together with a probability score. To base the query on a different number of elements, the 1 is replaced with the number required.

QUERY19: C1->S O M %1

The user is able to ask for the probability that a certain element follows the structure on the left by using the percent operator as shown in **QUERY20**.

QUERY20: C1->S %M

8.3.3 Finding Sentences by their DOCUMENT ID or POW CELL

By using the drop down list and text box shown in Figure 8.1, Item 7, the user can directly locate a sentence by its unique document identifier (**DOCID**) (as applied to it when it was loaded to the database) or by its FPD Cell identifier (see Chapter Three, Section 3.2.9 and Section 8.3.4.1). By selecting these, the matching sentence will be directly displayed in ICQF's sentence viewer (see Figure 8.3)

8.3.4 Restricting queries

Queries can be restricted such that the results returned will be from a subset of the entire corpus. There are three ways in which this can be done:

- (a) by restricting the query using parts of the FPD Corpus cell identifier,
- (b) by restricting the query using the fact that the results are initial within the sentence or unit,
- (c) by ignoring or including ellipsis.

These three types are described in the sections that follow.

8.3.4.1 Restricting queries by selecting parts of the sentence's cell identifier

This is a function that existed in the original version of ICQF and was found to be of great use to language students. The FPD cell identifier is an intelligent identifier and the parts of it are as follows:

- (a) **age** - the child's age: 6, 8, 10 or 12 years,
- (b) **social class** - the social class of the child: A,B, C or D,
- (c) **situation** - the type of the situation: I for Interview and P for Play,
- (d) **initials** - the initials of the child.
- (e) **sentence number** - the ordinal position of this sentence in this child's utterances

Hence, the cell identifier **10dgism#4** shows that the child is ten years old, belongs to social class D, is a girl with initials are I.S.M and this is the fourth utterance she has made.

During the conversion of the corpus into SGML and subsequent loading into the corpus tables, the cell identifier was extracted and its parts recorded in the **TAG** fields of the **corpus tables** and the **corpus index tables**. This provided the mechanisms which allowed ICQF+ to restrict the results of the query to only those sentences that matched the selected criteria. Figure 8.2 shows ICQF+'s **query restriction form**, which is selected by pressing the command button (Figure 8.1, Item 3), and the current restrictions are listed in the status bar of the query and results form (see Figure 8.1, Item 6).¹

8.3.4.2 Restricting queries by initial in sentence and unit

There are options that restrict the query only to results that are initial in their parent or initial in the sentence (Figure 8.1, Item 4). By selecting **initial-in-sentence**, the query is restricted to only those structures that occur in the initial position in the sentence (i.e. in the left-most vertical strip in a parse tree). **initial-in-parent** selects only those structures that are in the left-most vertical strip in their parent.²

The **INITIAL_*** fields in the **corpus index tables** allow this type of query to be performed (see Chapter Seven, Section 7.2.2 and Appendix D).

¹ It should be noted that the query restriction form shown in Figure 8.2 is specific to the FPD corpus, and if another corpus was used in ICQF+, a different form would be required.

² These types of query were developed especially for the research for the parser where we needed to know, for example, the vertical strips to include in our Initial Vertical Strip Item (**IVS-ITEM**) and Initial Vertical Strip Element Tables (**IVS-ELEM**) (see Chapter Fourteen).

8.3.4.3 Restricting queries by ignoring ellipsis

If **ignore ellippted elements** is selected (Figure 8.1, Item 4), then the results will not contain ellippted elements. That is, a search for all clauses that start with a Subject will not consider those clauses that have ellippted elements before the Subject.

The **ELLIPTED_*** fields in the **corpus index tables** allow this type of query to be performed (see Section 7.2.2 of Chapter Seven and Appendix D).

Once the results (if any) of the query are displayed in the list, the user is able to sort them using the buttons (Figure 8.1, Item 2). By double clicking the left mouse button on an entry in the list, all occurrences of whatever the match the value shown on that line of the list can be displayed in the sentence browser (Figure 8.1).

8.4.1.2 Navigation

A query line, as displayed in the list, can be selected (Figure 8.1, Item 1), can have one or more matching sentences displayed in the list (Figure 8.1, Item 3), can be viewed one by one in the sentence browser (Figure 8.1, Item 4), can be moved to the Previous-Sentence and Next-Sentence buttons (Figure 8.1, Item 5), can be highlighted using the matching sentence in the corpus (Figure 8.1, Item 6), and can be highlighted using the structure which matches the sentence (Figure 8.1, Item 7).

The user can view the sentence that is displayed in its context in the corpus, as shown in Figure 8.4, by selecting the Sentence Context tab (Figure 8.3, Item 8). Here, the current sentence is displayed together with the sentences before and after. By clicking on the sentence identifiers in this view, the selected sentence will become

1. Select the age(s) (6, 8, 10 and 12 years) of the children to be included in the query.
2. Select the sex of the child (male or female).
3. Select social class (A,B,C and D).
4. Select the situation (play and interview).
5. Apply options and close query restriction form.
6. Select all options for Age, Sex, Class and Situation.
7. Apply options and keep the query restriction form open.

Figure 8.2: ICQF+'s query restriction form

8.4 Using ICQF+

8.4.1 Retrieving the results of a query

The user enters a query by selecting **Queries** from ICQF+'s main menu. The **Query and Results form** is displayed (see Figure 8.1). After setting any necessary **Query**

Restrictions (see Section 8.3.4), the user enters the query in CQL format in the **Enter Query** text box (Figure 8.1, Item 9) and presses the **Execute Query** button (Figure 8.1, Item 5). ICQF+ processes the query and builds the results which are displayed in the **Results Display** (Figure 8.1, Item 1).

8.4.1.1 Displaying sentences

Once the results (if any) of the query are displayed in the list, the user is able to sort them using the buttons (Figure 8.1, Item 2). By double clicking the left-mouse button on an entry in the list, all occurrences of sentences that match the value shown on that line of the list can be displayed in ICQF+'s **Sentence Viewer** (Figure 8.3).

8.4.1.2 Navigation

A query line, as displayed in ICQF+'s **Query and Results form** (Figure 8.1), can have one or more matching sentences. The user is able to move between these sentences one by one in the sentence viewer by using the navigation buttons (Figure 8.3, Item 9). **Next-** and **Previous-match** will move through each matching sentence and **Next- and Previous-Sentence** will display the previous and next sentences next to the matching sentence in the corpus. The syntax structure of interest to the user is the structure which matches the query (in the **Sentence Viewer**, this is shown highlighted using black squares). Figure 8.3 Item 15 shows a match to the query for all nominal groups (**ngp**) that fill a Subject (**S**).

The user can view the sentence that is displayed in its context in the corpus, as shown in Figure 8.4, by selecting the **Sentence Context** tab (Figure 8.3, Item 8). Here, the current sentence is displayed together with the sentences before and after. By clicking on the sentence identifiers in this view, the selected sentence will become the current sentence and be displayed as a parse tree when the user moves back to the sentence view.

8.4.1.3 Editing and deleting sentences

Figure 8.3, Item 14 shows the XML, Save and Delete buttons, these are used to edit, save and delete the sentences in the corpus database respectively.³

When the XML button is pressed, a copy of the sentence is extracted from the native XML corpus tables as an XML file, and loaded into an editor. The user is able

³ The edit function copies the XML from the database and loads it into the sentence editor. The save function replaces the XML in the database with the edited XML. The delete function removes the XML from the database. All functions update the **corpus tables** and the **corpus index tables**.

to change the sentence as required, check that the XML is well formed, and then save the sentence back into the **corpus tables** by pressing the Save Button.

When the user presses the delete button, a confirmation message is displayed before the sentence is deleted.⁴ When a sentence is deleted, all records that relate to it are removed from the **corpus tables** and the **corpus index tables**. A similar process occurs when the sentences are edited and saved except that new entries are created in the tables that represent the saved XML file, and a date and time is recorded in the **DOCUMENT** table indicating when the sentence was last saved.⁵

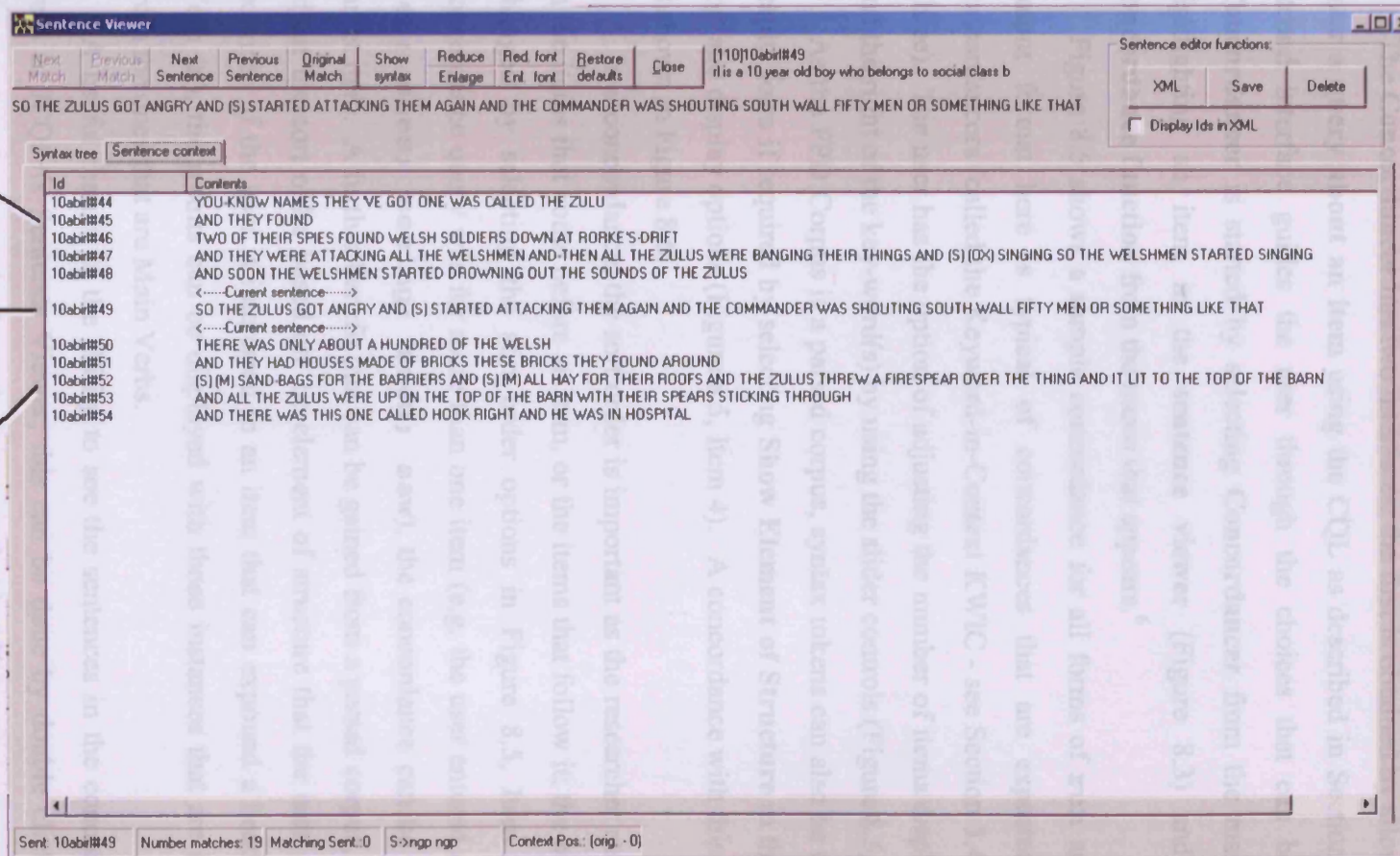
The sentence editor was used extensively in this project for the work connected with the modification of the corpus, which is reported in Chapter Nine.

⁴ On one or two occasions, during this modification work, it was identified that the sentence needed to be deleted.

⁵ Note that ICQF+ checks that the XML document is well-formed before it is allowed to be loaded back into the **corpus tables**.

1. Context indicator (shows how many sentences you are from the original query)
2. Query (in Query and Results form)
3. The number of this sentence is in the query (0 is the first)
4. The number of matches for the query
5. The Sentence (docname) of the sentence being viewed
6. The collapsible sentence view
7. Display the syntax tree
8. view the sentence in its context in the corpus (see Figure 8.4)
9. Navigation buttons
10. Show the match (as black squares) in the syntax tree
11. Enlarge/reduce sentence view
12. Unique id, sentence id and explanation
13. The words of the sentence
14. Sentence editor functions (XML - edit sentence as XML, Save Sentence in database or Delete it).
15. The syntax structure we are interested in is shown highlighted with black squares

Figure 8.3: ICQF+'s sentence viewer



1. Sentences after the current sentence
2. The current sentence
3. Sentences before the current sentence

Figure 8.4: ICQF+'s sentence viewer's view context form

8.4.2 The concordancer

ICQF's limited concordance tool has been significantly improved in ICQF+. The new concordancer was used extensively in the work to modify the corpus (see Chapter Nine), where it was used help identify criteria for changes based on items.

The **Concordancer** has two interfaces; the first is obtained by allowing the user to enter a **query about an item** using the CQL as described in Section 8.3.1, and the second interface guides the user through the choices that can be made. The **Concordancer** is started by selecting **Concordancer** from the main menu or by highlighting an item in the **sentence viewer** (Figure 8.3) and selecting the **concordance function** from the menu that appears.⁶

Figure 8.5 shows a sample concordance for all forms of **run** and **walk**. The output format here is typical of concordances that are expected from corpus concordancers (called the Keyword-in-Context KWIC - see Section 3.4.1.2 of Chapter Three). The user has the option of adjusting the number of items displayed to the left and the right of the key-word(s) by using the slider controls (Figure 8.5, Item 5).

As the FPD Corpus is a parsed corpus, syntax tokens can also be displayed in the results form if required by selecting **Show Element of Structure** in the concordancer sort and display options (Figure 8.5, Item 4). A concordance with this option selected is shown in Figure 8.6.

In a concordance, the sort order is important as the researcher may be interested in the items that come before an item, or the items that follow it; the sort order can be changed by selecting the sort order options in Figure 8.5, Item 4. When the concordance query was for more than one item (e.g. the user entered a query such as **see; sees; seeing; seen; saw**), the concordance can be sorted by each target item. A further benefit that can be gained from a parsed corpus, is the ability to adjust the sort order based on the element of structure that the item expounds. For example, if the user is interested in an item that can expound a head (**h**) or a Main Verb (**M**), the results can be displayed with those instances that are heads separated from the ones that are Main Verbs.

A useful feature is the ability to see the sentences in the concordance as parse trees in ICQF+'s Sentence Viewer; this can be done by double clicking the mouse button when the cursor is above the sentence that is of interest.

⁶ When the **Concordance** function is selected from the **Sentence Viewer**, ICQF+ automatically builds the concordance and loads it into the concordancer's results form.

Statistics for the lexical items selected for a concordance can be extracted from the corpus by selecting the **Statistics** button (Figure 8.5, Item 7); here, all occurrences of the target items are listed together with their frequency of occurrence for each element that it can expound. Sample statistics for the item **saw** are shown in Figure 8.7.

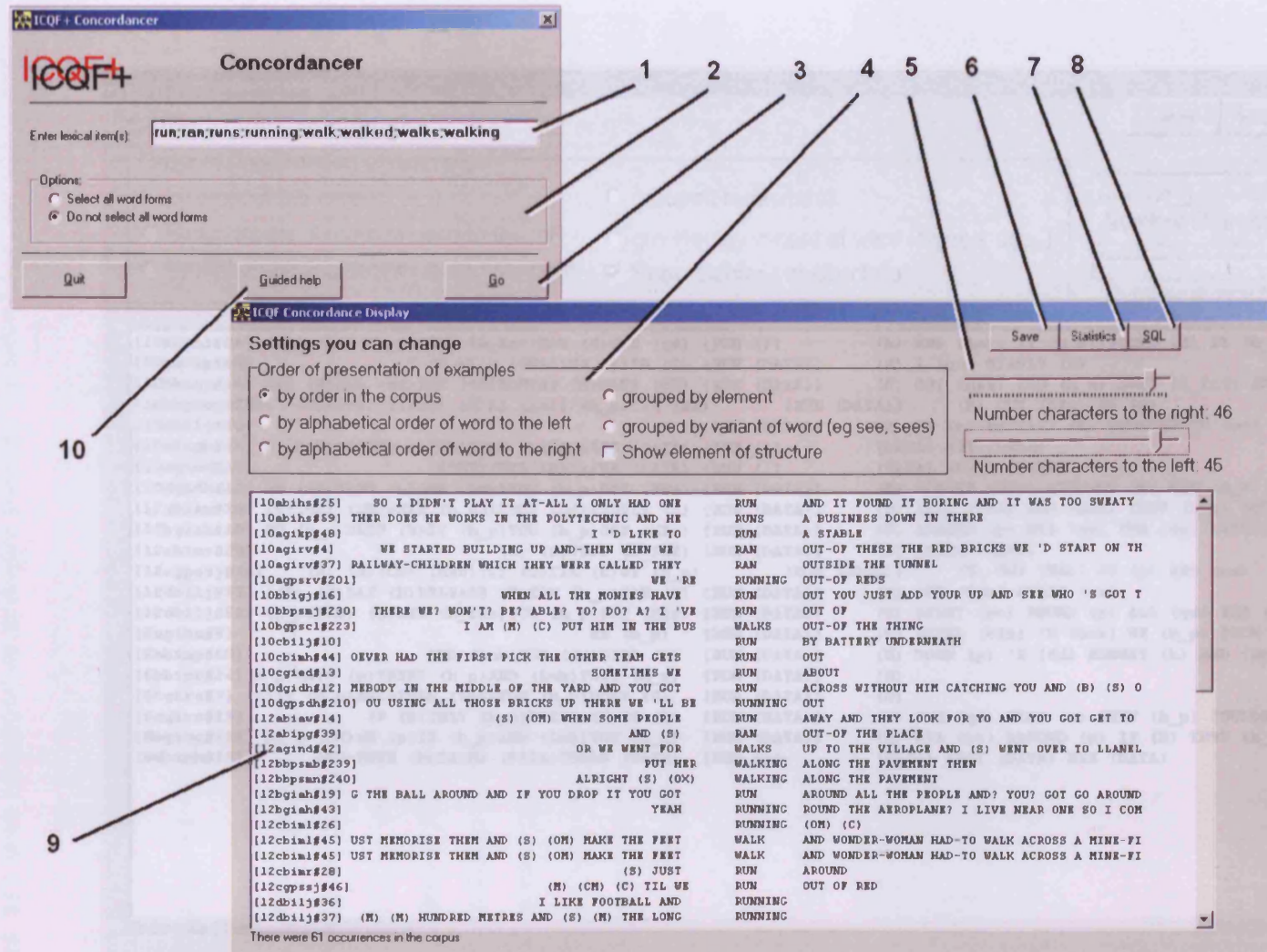


Figure 8.5: ICQF+'s concordancer

Figure 8.6: Concordancer showing elements of structure

1. Enter query here
2. Option to have ICQF+ generate all forms of the words
3. Build the concordance
4. Concordance sort and display options
5. Adjust the width of characters to the left and right of the target word
6. Save concordance as an XML report
7. Provide statistics (Fig. 8.7)
8. Show / adjust the SQL that was used in the query
9. Concordance display - double click the left mouse button to display the sentence in the sentence viewer (Fig. 8.3)
10. Launch guided help

ICQF Concordance Display

Settings you can change

Order of presentation of examples

- by order in the corpus
- by alphabetical order of word to the left
- by alphabetical order of word to the right
- grouped by element
- grouped by variant of word (eg see, sees)
- Show element of structure

Save Statistics SQL

Number characters to the right: 43

Number characters to the left: 42

```
[10abihs#25] -ALL (A)I (h_p)ONLY (A_Inf)HAD (M)ONE (qd) [RUN ( )] (h) AND (Lnk) IT (h_p) FOUND (M) IT (h_p)
[10agikp#48] I (h_p)'D (OM)LIKE (M)TO (I) [RUN (DATA)] (M) A (qd) STABLE (h)
[10bbigj#16] HEN (B)ALL (qd)THE (dd)MONIES (h)HAVE (OX) [RUN (DATA)] (M) OUT (MEx) YOU (h_p) JUST (A_Inf) ADD (
[10bbpsmj#230] (XEx)TO? (I)DO? (M)A? (qd)I (h_p)'VE (OX) [RUN (DATA)] (M) OUT (MEx) OF (p)
[10cbilj#10] [RUN (DATA)] (M) BY (p) BATTERY (h) UNDERNEATH (am)
[10cbimh#44] E (DATA)OTHER (DATA)TEAM (DATA)GETS (DATA) [RUN ( )] (DATA) OUT (DATA)
[10cgied#13] SOMETIMES (DATA)WE (DATA) [RUN ( )] (DATA) ABOUT (DATA)
[10dgidh#12] HE (dd)YARD (h)AND (Lnk)YOU (h_p)GOT (XEx) [RUN (DATA)] (M) ACROSS (MEx) WITHOUT (B) HIM (h_p) CAT
[12abiaw#14] (S)(OM) (OM)WHEN (h_wh)SOME (qd)PEOPLE (h) [RUN (DATA)] (M) AWAY (MEx) AND (Lnk) THEY (h_p) LOOK (
[12bgiah#19] OU (h_p)DROP (M)IT (h_p)YOU (h_p)GOT (XEx) [RUN (DATA)] (M) AROUND (p) ALL (qd) THE (dd) PEOPLE (h
[12chimr#28] (S) (S)JUST (A_Inf) [RUN (DATA)] (M) AROUND (MEx)
[12cypssj#46] (M) (M) (CM) (MEx) (C) (C)TIL (B)WE (h_p) [RUN (DATA)] (M) OUT (MEx) OF (p) RED (ax)
[12dbilj#53] ) (S) (S)SAY (M)RELEASE (M)YOU (h_p)CAN (O) [RUN (DATA)] (M) OUT (MEx) AGAIN (ax)
[12dbilj#60] (h_p)DROP (M)ONE (h_xcc)YOU (h_p)GOT (XEx) [RUN (DATA)] (M) RIGHT (pt) ROUND (p) ALL (qd) THE (dd)
[6agika#9] WE (h_p) [RUN (DATA)] (M) ROUND (MEx) 'N (Lnk) WE (h_p) PICK (M)
[6bbimp#59] AND (Lnk)THE (dd)BIRD (h) [RUN (DATA)] (M) DOWN (p) 'E (dd) RUNWAY (h) AND (Lnk)
[6bbipc#24] h_p)ONTO (p)THEM? (h_p)AND (Lnk)THEY (h_p) [RUN (DATA)] (M)
[6cgirs#7] (p)IT (h_p)AND-THEN (Lnk)YOU (h_p)MUST (O) [RUN (DATA)] (M)
[6cgirs#29] IF (B)THEY (h_p)TRIED (M)TO (I) [RUN (DATA)] (M) ONE (qd) SIDE (h) THEY (h_p) COULDN'T
[8bgibc#20] _p)'S (OM)ON (p)IT (h_p)AND (Lnk)YOU (h_p) [RUN (DATA)] (M) ALL (pt) AROUND (p) IF (B) THEY (h_p)
[8dbimh#59] AND-THEN (DATA)HE (DATA)COULD (DATA) [RUN ( )] (DATA) AWAY (DATA) SEE (DATA)
```

There were 21 occurrences in the corpus

Figure 8.6: Concordance showing elements of structure

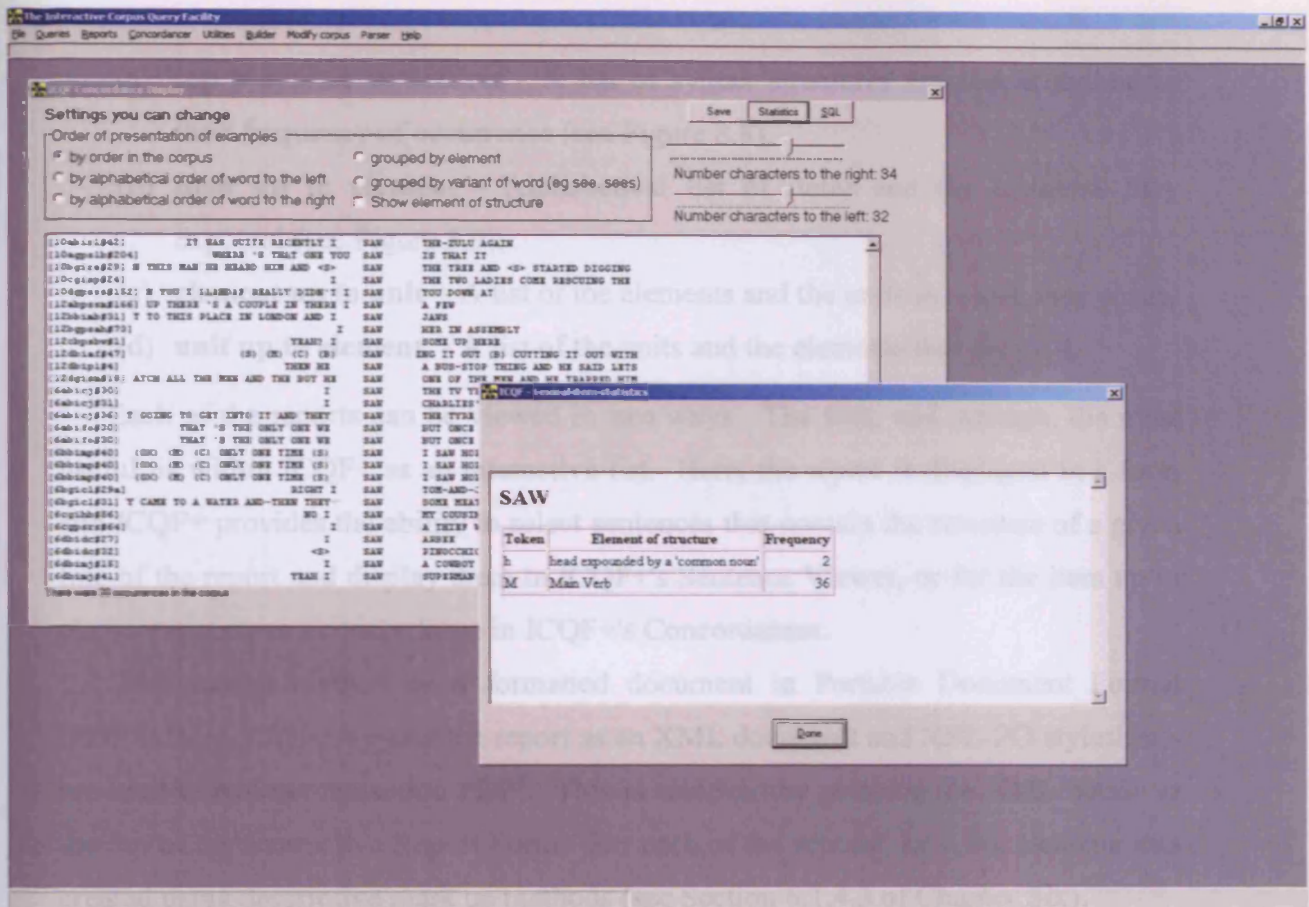


Figure 8.7: Statistics for the key-word in ICQF+'s concordancer

8.4.3 The reports

The original ICQF included utilities to extract lists of syntax structures and lists of items from the corpus. These were printed and used as valuable reference documents in the COMMUNAL Project. Their functions and formats are shown in Sections 3.4.1 and 3.4.2 of Chapter Three. In this present work, the reports are automatically updated as new sentences are parsed, changed in the **Sentence Editor**, or through one of the modification programs that are described in Chapter Nine; in this way, the reports, like the corpus database tables, dynamically change as the corpus changes.

In their electronic form, they were of particular use in this project to assist in the requirements for building the parser's probabilities tables (see Part Four, Chapter Fourteen).

The reports are accessed from ICQF+'s reports menu. There are the following report types.⁷

⁷ The XSL-FO stylesheets exist outside of ICQF+ and can therefore be changed to alter the format of the report. For the conversion from XML into PDF, faced the XSLFO2PDF project.

⁷ Reports (b), (c) and (d) are essentially formatted versions of the probabilistic tables **I2E**, **E2U** and **U2E** respectively (see Part Four, Chapter Fourteen).

- (a) **top n syntax structures** - A list of syntax structures ordered according to their frequency of occurrence (see Figure 8.8),
- (b) **item up to element** - Alphabetical list of items and the elements they expound (see Figure 8.9),
- (c) **element up to unit** - A list of the elements and the units in which they occur,
- (d) **unit up to element** - A list of the units and the elements that they fill.

Each of the reports can be viewed in two ways. The first, and perhaps, the most useful, is within ICQF+ as an interactive list. Here, the report is displayed in a form and ICQF+ provides the ability to select sentences that contain the structure of a given line of the report and display them in ICQF+'s Sentence Viewer, or for the item up to element report, as a concordance in ICQF+'s Concordancer.

The second method as a formatted document in Portable Document Format (PDF). Here, ICQF+ exports the report as an XML document and XSL-FO stylesheets are used to convert these into PDF⁸. This is achieved by pressing the XML button at the top of the Interactive Report Form. For each of the reports, an XML Schema was created using descriptive mark up methods (see Section 6.1.4.3 of Chapter Six).

Figure 8.8 shows an interactive report for the top n syntax structures. It shows the sequence of forms that are used to build all reports. Figure 8.8, Item 1 demonstrates the **select report type form** and Item 2, the form where the value of n is entered. Item 3 shows the **main report viewer**. This is a generic form that is used for all types of report; the column titles and other labels are populated according to the report type selected. One very useful feature is the ability to display examples of a structure or item in the Sentence Viewer; this is done by selecting a row in the grid by double mouse click. It is possible to reorder the results in the list by using the **sort options** (Figure 8.8, Item 5). To find a particular value in any column, the **find options** are used (Figure 8.8, Item 6); this is very useful when the list is not sorted alphabetically. To export the list as an XML document from where it is also converted into PDF, the XML button is used (Figure 8.8, Item 7).

⁸ The XSL-FO stylesheets exist outside of ICQF+ and can therefore be changed to alter the format of the report. For the conversion from XML into PDF, I used the Apache FOP processor.

The screenshot shows the ICQF+ Report Builder interface. The main window displays the 'Top 2000 syntax structure report' with a table of results. The report window is titled 'ICQF+ Report builder' and shows the following data:

Position	Syntax structure	Frequency	Probability
0	ngp->h_p	10957	6.0713%
1	S->ngp	8882	4.1109%
2	Z->Cl	8009	3.7068%
3	C->ngp	7012	3.2464%
4	cv->ngp	2737	1.2668%
5	pgp->p cv	2679	1.2399%
6	h_p->'I'	2645	1.2242%
7	ngp->dd h	2308	1.0682%
8	Cl->F	2267	1.0492%
9	dd->'THE'	2165	1.002%
10	qlgp->ax	1959	0.9067%
11	S->	1898	0.8785%
12	qd->'A'	1881	0.8706%
13	ngp->qd h	1775	0.8215%
14	Z->Cl Cl	1763	0.816%
15	Lnk->'AND'	1658	0.7674%
16	C->pgp	1512	0.6998%
17	h_p->'IT'	1509	0.6984%

Sort options:

- Sort by position (ascending)
- Sort by position (descending)
- Sort by alphabetical order of syntax structure (ascending)
- Sort by alphabetical order of syntax structure (descending)
- Sort by frequency (ascending)
- Sort by frequency (descending)
- Sort by probability (ascending)
- Sort by probability (descending)

Find options:

- Find position
- Find syntax structure
- Find frequency
- Find probability

Buttons: Cancel, Next, Rebuild list, Go, XML.

1. The select report type form
Enter the value of n for top n
2. The report view
3. Values column, double click to display example sentences in the 4. sentence viewer
5. Sort options
6. Find values in the list
7. Export XML and hence convert into PDF

Figure 8.8: ICQF+'s Report Builder showing the top n syntax structures

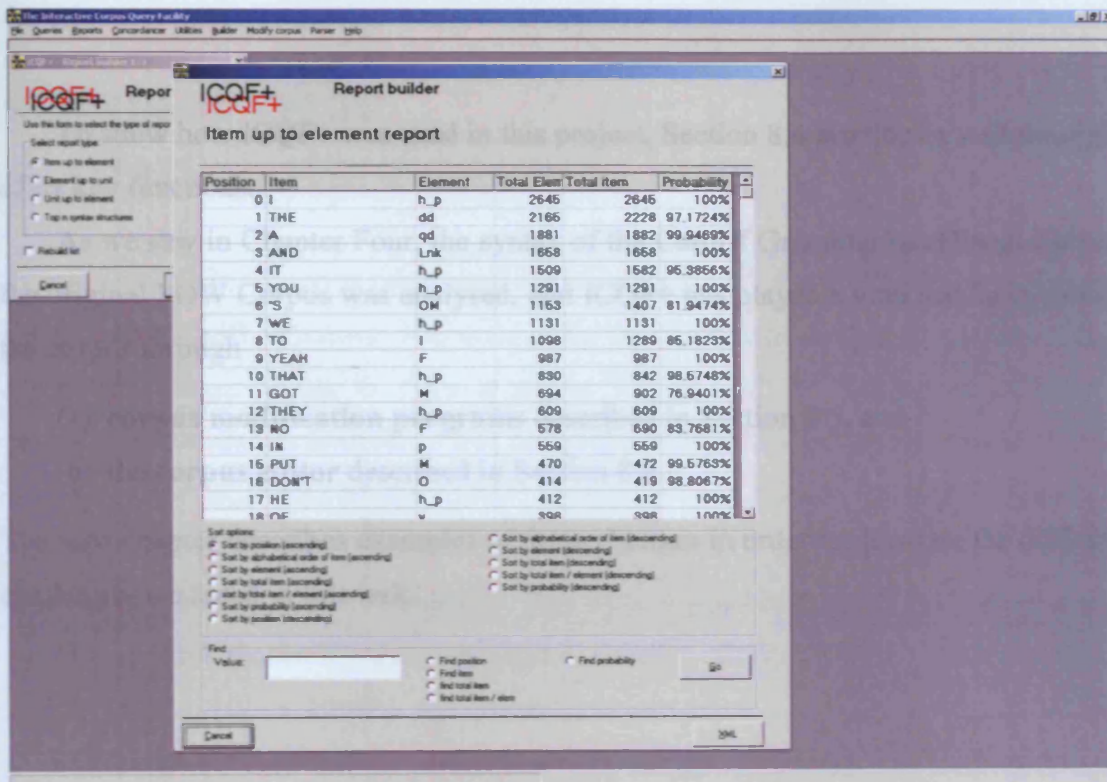


Figure 8.9: An item-up-to-element report

8.5 Corpus modification programs

ICQF+ includes the ability for the programmer to use ICQF+'s functions and routines to perform **global changes** to the corpus. Typically, these are created as program functions and are included on the ICQF+ **modify corpus menu**. The changes involved using either SQL or the query routines to extract a set of matching records from the **corpus tables**. These records could then be traversed and changed as appropriate and the **corpus index tables** updated. This work is described in Chapter Nine.

8.6 Summary

This chapter has described the corpus query tool: the Interactive Corpus Query Facility (ICQF).

After a brief history of ICQF in Section 8.1, Section 8.2 explained the rationale behind creating a new version, i.e. ICQF+, and explained its importance to this project. In particular it assisted in building the **probabilities tables** used by the parser and 'translating' the corpus analysis into the latest version of the Cardiff Grammar. The Corpus Query Language in its revised form was described with examples in Section 8.3.

To show how ICQF+ was used in this project, Section 8.4 provided a walkthrough of its key functions.

As we saw in Chapter Four, the syntax of the Cardiff Grammar has changed since the original POW Corpus was analysed, and ICQF+ has played a vital role in updating the corpus through

- (a) **corpus modification programs** described in Section 8.5, and
- (b) the **corpus editor** described in Section 8.4.

The next chapter describes examples of these changes in order to illustrate the difficult challenges we faced in this task.

Chapter Nine

Updating the corpus

When Fawcett and Perkins developed the Polytechnic of Wales (POW) Corpus in the 1980s, the Cardiff Grammar was still under development. Indeed, it was to a large extent the requirements to have a syntax that would be capable of describing adequately the data in the corpus that led to the changes to Halliday's original model, and so to the existence of the Cardiff Grammar as a distinct 'dialect' of Systemic Functional Grammar (SFG). The syntax used in the POW Corpus therefore contains a number of significant differences from the version described in Fawcett (2000a), and in other recent works by Fawcett (2000b, 2000c, 2007a, 2007b), Tucker (2006a, 2006b), Neale (2002a, 2002b), and others.

As the project to build a corpus-based parser got underway, we realised that, in order to implement the best possible parser, we needed to create a new version of the original corpus that would be analysed in terms of the latest version of the Cardiff Grammar. As no alternative parsed corpus was available, we decided to take the Fawcett and Perkin's POW Corpus and convert the syntax of that corpus into the syntax of the latest version of the Cardiff Grammar.

The changes were defined and made with the linguist and the computer scientist working closely together, but essentially with the linguist defining the types of change that had to be made and the computer scientist working out how this could be done and implementing the changes. At times, there was no firm distinction between the two fields of expertise, with each member suggesting changes and checking solutions in the other's field.

The result of this major piece of work is a corpus called the Fawcett-Perkins-Day (FPD) Corpus, which is annotated in terms of the latest version of the Cardiff Grammar. This chapter describes how we created the FPD Corpus. Section 9.1 describes why the changes were needed; Section 9.2 outlines the overall strategy for change, and Section 9.3 describes how the major changes were performed.

9.1 Why the changes are needed

The corpus was modified so that its syntax is represented in terms of the latest version of the Cardiff Grammar for the following reasons:

- (a) to provide the best possible model for parsing, and
- (b) to make the corpus and the tools that use it compatible with the other components of COMMUNAL (see Section 2.5 of Chapter Two).

The advantages of updating the corpus so that it is compatible with the other COMMUNAL components are that the parser (described in Part Four) is able to communicate with the other components of the overall model, since they are using the same version of the grammar. Further, the **corpus query tool ICQF+** (see Chapter Eight) is of greater use to the COMMUNAL Project.

In what way then, is the latest version of the Cardiff Grammar better suited to parsing than the earlier version used in the POW Corpus? This is best explained by outlining the major differences, and highlighting the cases where the change helps the parsing process.

The major difference between the POW version of the grammar and the current grammar is the recognition that the data that were originally analysed as cases of the quantity-quality group (**qqgp**) should instead be treated as two distinct units - the quantity group (**qtgp**) and the quality group (**qlgp**). This approach was used in the current computer implementation of the Cardiff Grammar in the COMMUNAL Project in the 1990s, and the elements of the two units of the **groups** are described in Fawcett (2000a:164), and in Part One, Chapter Four. Tucker (1989) gives the fullest description yet published of the quality group. The change of the **qqgp** into two separate units was particularly hard to implement, since it required us to discover ways of identifying which of the two units the old groups should be assigned to. However, this work was rewarded as the decomposition of the quantity-quality group into the two separate units, helps the parsing process because the two units typically serve different functions, and this fact can be used to predict the element of structure that the unit fills.

The second type of change was in the element which certain items expound. For example, the items **this**, **that**, **these** and **those** were originally analysed as a deictic determiner (**dd**), even when they appear on their own in a nominal group, but now, they are treated as pronoun heads (**h_p**). Similarly, **which** and **what**, when not

followed by a noun, were originally analysed as a wh-deictic determiner (**DDWH**), and now they are wh-heads (**h_wh**). And some units, which were formerly analysed as quantity-quality groups, are now treated as a nominal groups (**ngp**).

Thirdly, as the result of the analysis of more and more data by Fawcett and Tucker, the introduction of a small number of new syntactic elements has occurred. For example, the temperer (**t**), in the 1980 model is now sub-divided into the degree temperer (**dt**), the emphasising temperer (**et**) and the adjunctival temperer (**at**). As these temperers serve different functions in their units, this assists the parsing process in helping determine the function of the item.

Another change is that the different types of determiners are now identified by their functional type first (e.g. a quantifying determiner is now represented by a **qd** in preference to its earlier form of **dq**).

Finally, Fawcett's notation uses lowercase letters to identify the elements of structure in the nominal, prepositional, quality, and quantity groups, and in the units themselves, whereas initial capitals are used for the elements of the Clause. Due to restrictions in the computer system that Fawcett and Perkins were using for creating the POW Corpus in the 1980s, all syntax tokens are expressed in uppercase letters. Therefore the final change was a mapping between the original syntax tokens and the latest syntax tokens.

9.2 The overall strategy

All changes were made in two distinct stages. Stage One involved changing syntax structures within sentences but, where possible, keeping the old Cardiff Grammar syntax tokens.¹ This stage is described below in Section 9.3.

Stage Two was global and changed all POW syntax tokens into the current Cardiff Grammar syntax tokens and is described in Section 9.4.

9.3 Stage one: how the changes were made

I will now explain the methods used to solve the problem of how to make such changes in a corpus of approximately 70,000 words.

¹ Some changes needed the introduction of syntax tokens that were not in the original version (e.g. **qtgp**), these were added but with uppercase letters (**QTGP**).

9.3.1 Details of the change process

The challenge was to construct versatile and powerful queries that would be able to return a relevant set of sentences, for either:

- (a) manual review and editing, or
- (b) to be the subject of a global automatic transformation.

Using the characteristics of the words of the text and associated syntax structures, we were able to extract XML sentences from the **corpus tables** (see Chapter Seven) based on the nodes within their parse trees (in the format described in Chapter Six). We found sentences that contained specific combinations of items, elements and units, based on any combination of:

- (a) any sibling nodes that appear to the left or right,
- (b) any elements and units that appear above or below the node, or
- (c) any structures elsewhere in the same parse tree.

These variables were combined to form very complex retrieval criteria.

9.3.1.1 Automatic changes

Automatic changes were used where this was possible without introducing errors. Some changes were relatively simple and involved a change of syntax token, while other changes demanded the extraction and rebuilding of sub-trees in terms of parents and siblings (as described above). Following any large set of changes or periodically, the corpus database was checked for XML errors.

The order of the changes performed in Stage One was important because changes made in an earlier change were assumed to be in place when later changes were made.

9.3.1.2 Manual changes

Manual changes were generally assisted by ICQF+'s **corpus editor**, which was specially designed for this purpose. Here, following a query, we were able to view any sentences that matched our requirements in ICQF+'s **sentence viewer** (see Figure 8.3 of Chapter Eight). We viewed them either one by one, or in groups, and selected them, if required, for editing in the corpus editor. After we had changed the sentence, an **XML parse** was performed on it, before it was returned back into the corpus

database.² As sentences were returned to the database, ICQF+ rebuilt the **corpus index tables**.

9.3.1.3 Errors discovered in the POW Corpus

In the course of this work, we fairly frequently found errors in the syntactic analyses made by the original team of analysts. The corpus editor also proved useful for correcting these mistakes, which included spelling mistakes, incorrect syntax labels, and incorrect analyses.³

In rare cases, the transcription of the spoken text was unclear, so that, in rare cases, sentences had to be deleted. ICQF+ was provided with a 'delete function', which removed any such sentence from the corpus.⁴

In the remainder of this section we show examples of these automatic changes that are in the form of transformation diagrams. Space does not permit me creating diagrams for all changes made in this way, therefore Appendix F gives a complete list of the automatic changes and serves as a record of the work performed. It does not, however, contain details of changes that were made manually as described in Section 9.3.1.2.

9.3.2 Stage One: example changes

In the transformation diagrams that follow, the exclamation mark (!) represents the end of a unit or the start of a unit. The vertical bar (|) is the Boolean OR operator. The asterisk wildcard operator means zero or more elements in a unit or any one or more unit in a co-ordinated relationship.

This section serves a secondary purpose: to demonstrate how the corpus index table can be used to rapidly find records in the corpus tables based on ancestors and siblings at any level in the syntax tree, and also to show how the XML parse trees can be modified. To serve this purpose, example queries and pseudo code is given, and the reader may want to refer back to Chapter Seven for a description of the database tables and their fields.

² The XML parse ensured that the XML document was structurally well-formed.

³ As will be seen in Chapter Seventeen of Part Four, errors in the corpus cause problems in the parsing process.

⁴ Only two sentences were deleted in this exercise.

Rather than give an exhaustive list here, I have given selected examples of each type of change. The change numbers represent the number given to the change during the modification task, and correspond to the change numbers in Appendix F.

9.3.2.1 A simple change of a mark up element

Change 0: **this, that, these and those**

This change involved finding every nominal group (NGP), or unfinished nominal group (NGPUN), that has a deictic determiner (DD) as its last element in the unit, where the DD is expounded by the items **this, that, these** or **those**, and change them to a pronoun head (HP). The reason for this change is that these are now analysed as pronoun heads in the new Cardiff grammar, for linguistic reasons that we will not go into here. Figure 9.1 shows a transformation diagram that represents this change.

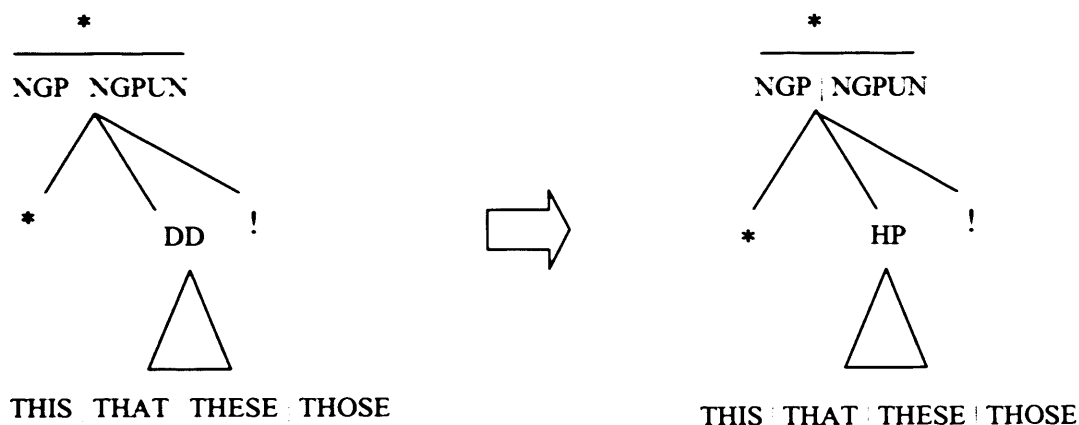


Figure 9.1: The transformation diagram for Change 0

The query used to find the matching structures involved finding all instances of **this, that, these** and **those** in the main corpus index table where the parent mark up element has a generic identifier DD, and there are no right siblings. This query can be expressed in SQL as follows. The SQL is similar to the queries used for all the examples shown here, unless otherwise stated:

```
SELECT * FROM MARKUPINDEX
WHERE GI = 'DD' AND
(CHILDGIS="THIS" OR CHILDGIS="THAT" OR CHILDGIS="THESE" OR
CHILDGIS="THOSE") AND
RSIBGIS = '';
```

Once found, the records could then be traversed and modified. The modification algorithm used is given below:

1. Get the **ELEMENT** record from the **corpus tables** that corresponds to the **ELEMID** and **DOCID** of the **corpus index** entry (this represents the **DD** element).
2. Modify the **ELEMENT** record so that its generic identifier field (**GI**) is set to **HP** (thus changing it from **DD** to **HP**).
3. Update the **corpus index records** for this sentence (so that the change is recognised in subsequent queries).

9.3.2.2 A simple change of a mark up element and its parent

Change 3: here, now, then and when

This change involved finding all instances of the above words where it expounds an apex (**AX**), which is the only element in its unit. The quantity-quality group (**QQGP**) was replaced by a nominal group (**NGP**) and the apex (**AX**) by a pronoun head (**HP**). Figure 9.2 shows a transformation diagram that represents this change.

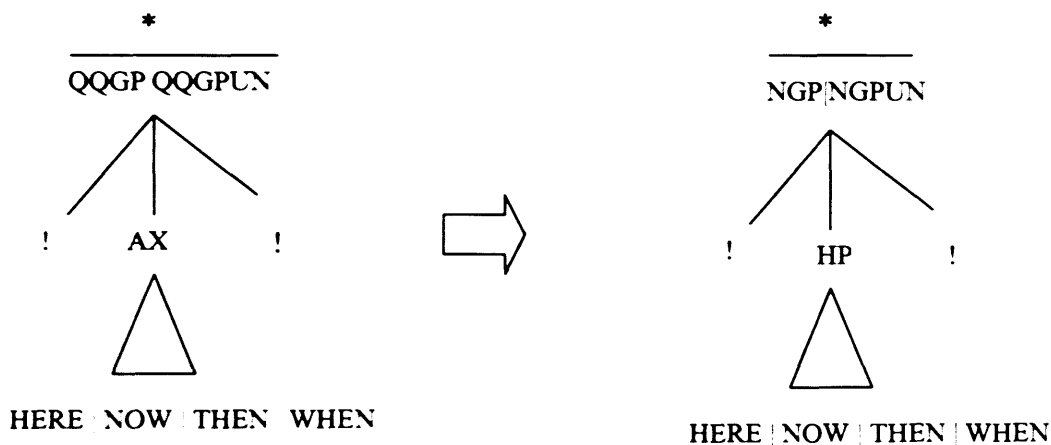


Figure 9.2: The transformation diagram for Change 3

This query involved finding all instances of **here**, **now**, **then** or **when** in the corpus index where the parent mark up element has a generic identifier **AX**, and there are no right siblings or left siblings (i.e. the **RSIBIDS** and **LSIBIDS** fields are empty), and then modifying these elements.⁵

⁵ Note that the element **AX** is always in a **QQGP** and there is no need to specify the unit in the query.

Again, once found, the records could then be traversed and modified. The modification algorithm was:

1. Get the **ELEMENT** record from the **corpus tables** that corresponds to the **ELEMID** and **DOCID** of the corpus index entry (this represents the apex (**AX**)).
2. Modify the **ELEMENT** record so that its generic identifier field (**GI**) is set to **HP** (thus changing the syntax token from **AX** to **HP**).
3. Get the **ELEMENT** record from the **corpus tables** that corresponds to the parent element of the **ELEMENT** just modified (this will be the unit **QQGP** or **QQGPUN** and it will be identified by the **PARID** of the corpus index record, or the **PARENTID** of the **ELEMENT** record).
4. If the **GI** field of the **ELEMENT** record is **QQGP**, change it to **NGP** (thus changing the parent to a nominal group).
5. If the field **GI** field of the **ELEMENT** record is **QQGPUN**, change it to **NGPUN**.
6. Update the **corpus index records** for this sentence, so that the changes will be reflected in subsequent queries.

9.3.2.3 A complex example with mark up element insertion

Change 31: Item **Let 's** to expound two elements

This change was more complex than the last two examples, and was designed to cover the new Cardiff Grammar element called **Let**. It involved finding all instances of the item **let's** that expound a Subject (**S**), adding a new element to the left of the Subject (**S**) for the **Let** element (**L**), and directly expounding it by the item **let**. Finally changing the Subject (**S**) so that it fills a nominal group (**NGP**) which has a pronoun head expounded by **'s**. Figure 9.3 shows a transformation diagram that represents this change.

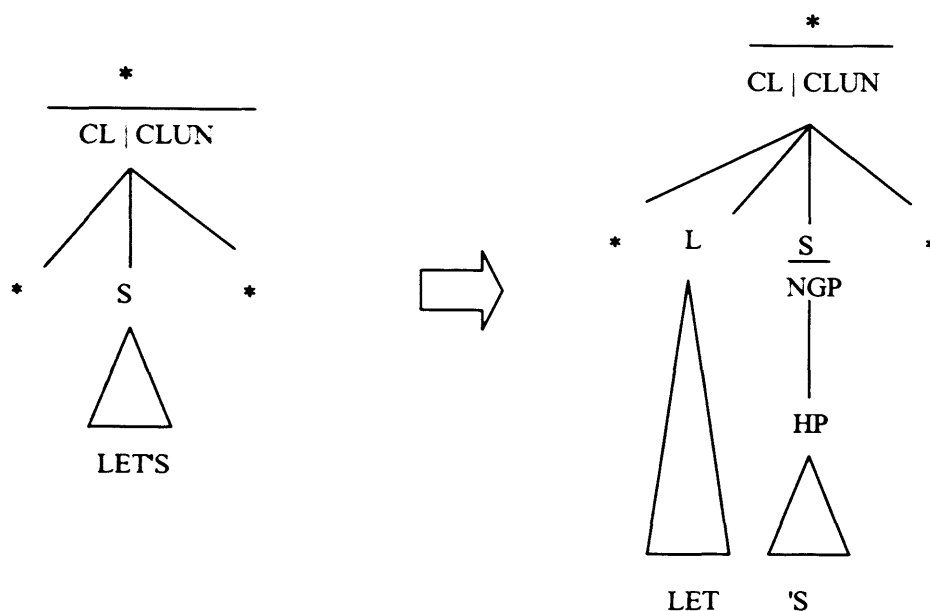


Figure 9.3: The transformation diagram for Change 31

The corpus index records that result from a query for all Subject elements that are expounded by **let's** were traversed, and the following modification algorithm was used. To help the reader, Figure 9.4 shows the parse tree following each step of the algorithm:

1. Create a new record in the **ELEMENT** table for the Let element (**L**), and:
 - set its **GI** field to the value **L** (for the **Let** element).
 - set the **DOCID** field to the value of the **DOCID** field in the corpus index record (so that it appears in the same sentence).
 - set the **PARENTID** field of the new **ELEMENT** to the value given in the **PARID** field of the corpus index record (so that it has the same parent as the Subject record).
 - give the **ELEMID** field of the new record, a value of 0.5 below that of the **LHSID** of the corpus index record (so that it appears before the Subject record in the Clause).⁶
2. Create a new record in the **PCDATA** table for the item **LET**, and:
 - set its **DOCID** to the **DOCID** value of the corpus index record,
 - set its **ELEMID** to the value given to the **ELEMENT** record in Step 1 (so that its parent is the new **LET** element).

⁶ The **ELEMID** field value represents a unique value for the **ELEMENT** within the **DOCUMENT**, and provide the order in which the **ELEMENT** resides within its parent. Therefore a value of 0.5 places the new element in the correct position.

- set the **PCDATA** field to the value **LET**.
3. Delete the **ELEMENT** record that represents the old item **Let ' s** (as given in the **CHILDIDS** field of the corpus index record), and its associated **PCDATA** record as we are going to create a new record in the next step.
 4. Create a new **ELEMENT** record for a nominal group that will fill the Subject, and
 - set its **GI** field to the value **NGP**,
 - set the **DOCID** field to the value of the **DOCID** field in the corpus index record, and its **ELEMID** to the next available **ELEMID** in the **DOCUMENT**,
 - set the **PARENTID** field of the new **ELEMENT** to the value given in the **ELEMENT** record that represents the Subject (given by the **ELEMID** field of the corpus index record).
 5. Create a new **ELEMENT** record for the pronoun head, and
 - set the **GI** field to the value **HP**
 - set the **DOCID** field to the value of the **DOCID** field in the corpus index record,
 - set the **PARENTID** to the **ELEMID** of the record created in Step 4.
 6. Create a new **PCDATA** record for the item **' S**, and
 - set the **DOCID** field to the value of the **DOCID** field in the corpus index record, and the **ELEMID** to the value of the **ELEMID** created in Step 5 so that it belongs to the **HP** element.
 7. Update the corpus index tables for this sentence.

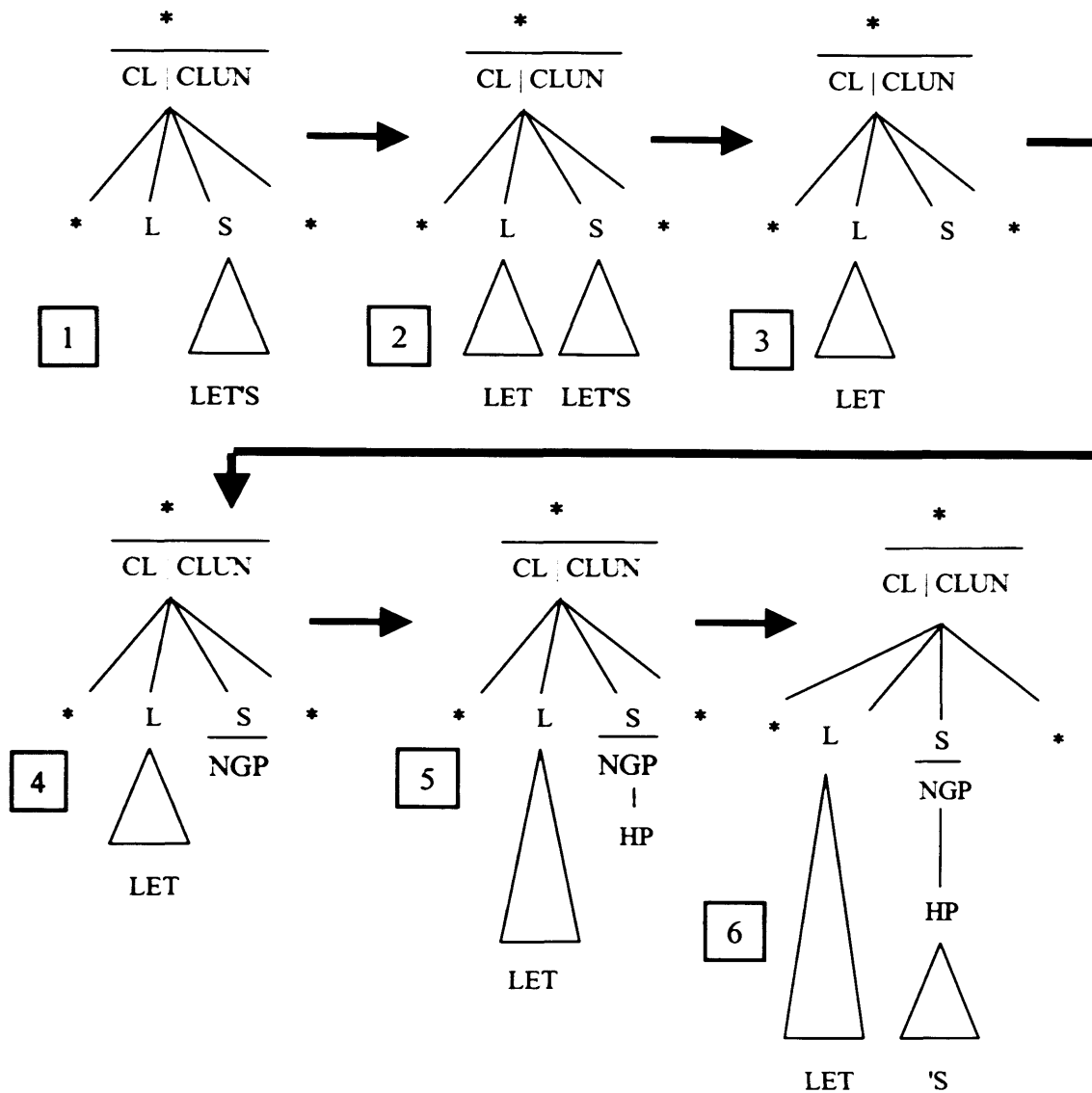


Figure 9.4: Steps involved in changing the syntax structure (the numbers represent steps in the algorithm)

9.3.2.4 Example showing a more complex query

This final example, for which the actual modification algorithm is similar to the ones shown in the earlier examples, includes finding all cases of an item that is followed by another item, and then modifying matching sentences.

Change 8: have, had, has, 'ave and 'ad, followed by to

The query involves finding all instances of the items **have**, **had**, **has**, **'ave**, and **'ad**, where it expounds the modal operator (OMO, OM) in a Clause (CL) or an unfinished Clause (CLUN) and the modal operator (OMO) is followed by the infinitive element (I) expounded by **to**. The modal operator (OMO, OM) is changed into an

operator (O).⁷ In the latest version of the Cardiff Grammar, Modal Operators are simply recorded as an instance of an Operator (O). Figure 9.5 shows a transformation diagram that represents this change.

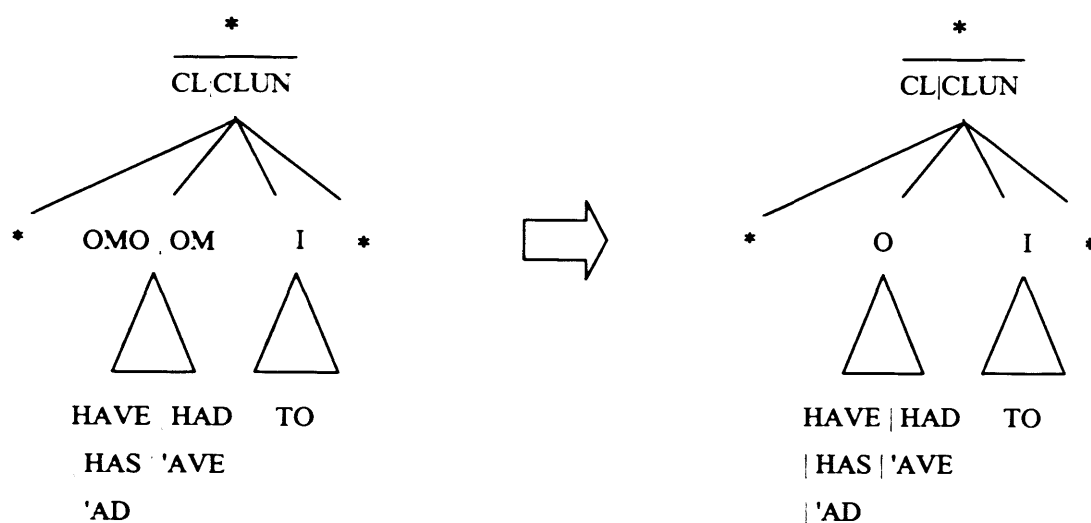


Figure 9.5: Transformation diagram for Change 8

The query to find these records could have been expressed in a number of ways. For example, the **DOCUMENT** table contains a concatenation of the data in the **PCDATA** fields in the order they appear, and the query could be expressed in SQL as

```
SELECT * FROM DOCUMENT
WHERE PCDATA LIKE '* HAVE TO *';
```

In practice, it was simpler to use two nested queries, the first getting elements that have records that match one of the items expounding the elements required, and the second retrieving any records that have a right sibling containing the item **to** and expounding an Infinitive Element (**I**). The remainder of the processing then became similar to the examples given above.

This section gave a description of the global changes that were made to the POW Corpus in order to create the FPD Corpus. Having finished these automatic changes, which are fully detailed in Appendix F, we now had a new corpus which was in an intermediate state in which most of the syntax tokens were those used in the version of the Cardiff Grammar used in the POW Corpus. The next stage was to convert the syntax tokens into those given in the latest version of the grammar, and it is to this that we turn next.

⁷ There was an error identified in the POW Corpus. The syntax token OM was being used for Operator conflated with the Main Verb and for a Modal Operator.

9.4 Stage Two

Stage Two involved changing the syntax tokens from the tokens given in the POW version of the Cardiff Grammar to the tokens given in the latest version of the grammar (as given in Fawcett 2000a). In most cases, this was a fairly simple mapping of the old token into its new mixed case version. In a few cases, it was necessary to use the context of the syntax token, where the old syntax token was mapped to a new one depending upon its parent syntax token.

The processing required for Stage Two was straightforward. A database table (see Appendix G), called **TOKENCHANGE** was established with three fields:

- (a) **OLDTOKEN** - the token as it is found in the POW Corpus version of the Cardiff Grammar.
- (b) **CONTEXT** - a token that is the parent of this token, which must match the parent of any token that is a candidate for the change specified by this record. This field is empty if the change applies to all instances of **OLDTOKEN**.
- (c) **NEWTOKEN** - the token as it is given in the latest version of the Cardiff Grammar.

The **OLDTOKEN** fields were automatically populated with the unique tokens found in the **GI** field of the mark up **ELEMENT** table. The mappings to the new tokens were then manually populated, in some cases after querying ICQF+ to see the uses of the particular token. Once complete, the **ELEMENT** table was traversed, and the changes were made by referring to the **TOKENCHANGE** table for the old to new token mappings. Following this change, the **corpus index tables**, and the **probabilities tables** (used by the parser) were regenerated.

9.5 Summary

This chapter has shown how the POW Corpus, which is analysed and annotated in an older version of the Cardiff Grammar, was modified to create the Fawcett-Perkins-Day Corpus, which is analysed and annotated to the current version of the Cardiff Grammar.

The reasons for the change were to give the parser the best possible model for parsing, and to make the corpus compatible with the other COMMUNAL components.

The modification process was divided into two Stages. Section 9.3 showed that modifying the corpus was not a simple matter of changing the syntax tokens. It

involved querying the corpus and identifying examples in terms of their ancestors and siblings in matching partial syntax trees, and then modifying their syntax structures. This demonstrated one practical use of the **corpus index tables** (see Chapter Seven). Full details of the changes made in Stage One are given in Appendix F.

As we saw in Section 9.4, the Second Stage involved mapping syntax tokens from the old version of the grammar to those equivalents in the current version of the Cardiff Grammar, the changes sometimes being conditional on the context of the token. A complete list of these mappings is given in Appendix G.

The result of this work is a new corpus and supporting tables, which is better suited to parsing. The new corpus contains fewer errors and mis-analyses, and is compatible with the other COMMUNAL components. It is a corpus which now presents a better platform for other researchers in language.

This work concludes Part Two of this thesis. We move next to Part Three, which describes the most recent precursors of the present parser, including the prototype parsers created for this project.

PART THREE

Parsing Natural Language: Some Relevant Antecedents

This part of the thesis takes us on from the corpus database to the process of parsing. More specifically, it provides surveys of earlier work that is relevant to the present project, so giving the necessary background for Part Four, in which the Corpus-Consulting Probabilistic XML Parser itself is discussed.¹

Chapter Ten provides a selective history of parsing, describing (a) the parsing concepts that have emerged in this field, and (b) the types of parser that have been developed through half a century of work. Our purpose is to evaluate the main concepts in terms of their ability to be useful in the construction of a parser that is capable of handling:

- (a) the full range of syntactic phenomena found in naturally occurring texts, and
- (b) the representation of these phenomena in systemic functional syntax (as described in Fawcett (2000a) and in Chapter Four).

Chapter Eleven describes earlier attempts at building parsers that use Systemic Functional Grammars. It begins with Winograd's ground-breaking SHRDLU (Winograd 1972), then it provides overviews of the work of Kasper (1988), O'Donnell (1993, 1994), and the early work of the Leeds COMMUNAL Team. It then concludes with more detailed accounts of the parsers of O'Donoghue (1991a), Weerasinghe (1994) and Souter (1996), all of which were earlier attempts to build a parser within the framework of the COMMUNAL project.

Finally, Chapter Twelve prepares the way for Part Four (which describes the new parser) by (a) outlining the main features of two previous prototype parsers that were developed as part of the current project, and (b) comparing one of these parsers (a chart parser) with the approaches of Weerasinghe (1994) and Souter (1996).

¹ In the remainder of this thesis, the term XML is not included in the name of the parser. It is referred to simply as the Corpus-Consulting Probabilistic Parser (CCPP).

Chapter Ten

Concepts used in parsing: a selective history

This chapter provides some necessary background information from the field of natural language parsing. Section 10.1 describes the goals of parsing, then Section 10.2 introduces some of the concepts used when classifying parsing techniques, and shows how these are relevant to the Corpus-Consulting Probabilistic Parser (CCPP) that will be described in Part Four. Section 10.3 then describes some of the more common parsing algorithms together with other relevant works, and identifies those from which ideas are drawn for the CCPP.

Researchers have used various techniques in an attempt to improve both the efficiency and accuracy of their algorithms, and the most common of these are described in Section 10.4. Section 10.5 introduces the parsing technique that is most relevant to this parser: **probabilistic parsing**. These approaches use statistics to decide the most likely paths and to order the final analyses when more than one has been produced. Probabilistic methods are typically used in conjunction with **corpus-based approaches** and these are described in Section 10.6.

Because of the functional richness of its syntax, Systemic Functional Grammar (SFG) presents a particular set of challenges to researchers when they attempt to use the algorithms given in Section 10.3 for parsing. These are outlined in the final section of this chapter, Section 10.7, in preparation for the next chapter.

10.1 The goals of parsing

The goal of a natural language parser is to take as its input a string of words that together constitute one sentence, and to produce a syntactic representation of it. This is typically in the form of a tree diagram, with varying levels of richness in the annotations on the **nodes** of the tree. If the sentence is ambiguous, the parser should provide two or more such representations. In a natural language understanding system, the semantic interpreter and the other higher level components then use these representations as their inputs, as discussed in Section 2.3.1 of Chapter Two.

Figure 10.1 shows a parsed output in an XML format that was given in Chapter Six. This has been produced from the input string shown, and Figure 10.2 shows a

rendered parse tree derived from that parse output, which reflects the conventions used in linguistics (e.g. as in Chapter Four and in Fawcett (2000a)).

Input string: The seagulls ate the mackerel

Output data:

```
<!DOCTYPE Z public="-//DTD//Cardiff Grammar 20060801//EN" []>
<Z>
  <C1>
    <S>
      <ngp>
        <dd>The</dd>
        <h>seagulls</h>
      </ngp>
    </S>
    <M>ate</M>
    <C>
      <dd>the</dd>
      <h>mackerel</h>
    </C>
  </C1>
</Z>
```

Figure 10.1: XML parse tree for a sample sentence

Formatted output:

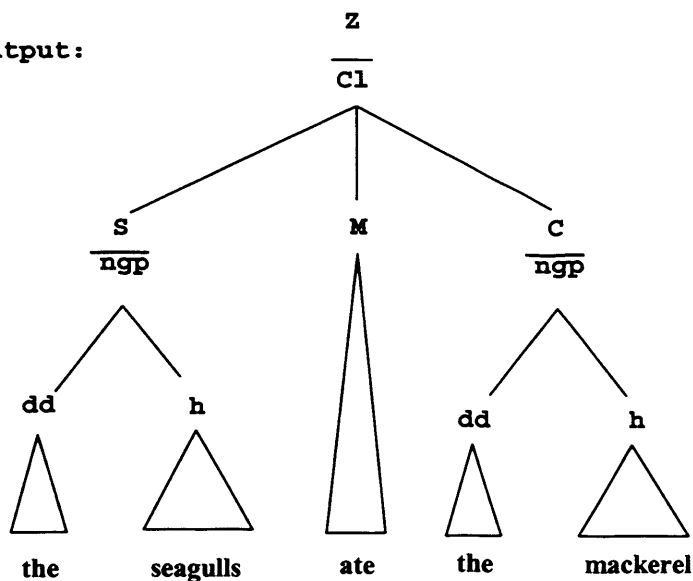


Figure 10.2: Formatted parse tree for a sample sentence

These two representations express primarily the same information, and are therefore representational variants of each other.

10.2 Some early classifications used in parsing

This section provides some common terms used to classify parsing algorithms. Section 10.2.1, 10.2.2, 10.2.3, 10.2.4 and 10.2.5 provide details of these terms and Section 10.2.6 shows, where appropriate, how they can be used to classify the parser described in Part Four of this work.

10.2.1 Mode of operation

There are three basic terms given to the way a parser works vertically with a syntax tree, these are

- (a) **top-down**,
- (b) **bottom-up** and
- (c) **mixed-mode**.

These are described in the sections that follow.¹

10.2.1.1 Top-down

A parser is said to operate in a **top-down** manner if it starts with the hypothesis that the input string forms a sentence and where the **root** (which is always shown paradoxically at the top of the diagram) is the symbol Σ (in Fawcett 2000a) and **Z** in the present work. It then works 'downwards' towards its items at the bottom of the diagram, identifying its syntactic constituents.

As it works down from the sentence, it may decide next that the input has one Clause (**C1**) which starts with a Subject (**S**) (as in Figure 10.2). Then, it may hypothesise that the Subject is filled by a nominal group (**ngp**) and that the nominal group starts with a deictic determiner (**dd**) (which it is able to test by looking at the input string). If this proves to be true, the parser can create structures for the sentence, the clause, the Subject, the nominal group and the deictic determiner. It may then hypothesise that the Subject comprises one nominal group and that it must have a head (**h**), and, if it then finds one in the input string, it can build the structure and complete the Subject. Notice that, so far, I have said nothing about what it does if its hypothesis is not satisfied; this is covered during our discussion on parsing algorithms (see Section 10.3).

¹ There is also another classification: **left-to-right** or **right-to-left** which describes the direction a parser operates on the items in the input string. Right-to-left approaches are extremely rare.

10.2.1.2 Bottom-up

A parser is said to be a **bottom-up** parser if it starts with the items themselves, and then works upwards towards the root. For example, it may examine the input string and find a deictic determiner. It will then seek rules, which tell it that the deictic determiner can be the first element in a nominal group. Then, perhaps after further breadth-wise analysis (see Section 10.2.2.1), determine that the nominal group is a Subject in a Clause and that the Clause is filling a Sentence (**Z**).

10.2.1.3 Mixed-mode

Mixed-mode parsing employs both top-down and bottom-up approaches. For example, a bottom-up **chart parser** (see Section 10.3.1.5) may use top-down **filtering** to avoid creating analyses (specifically, **edges**) that are not permitted by the rules of the model of syntax being used. One well-known early parser of this type (no pun intended) was the Earley parser (Earley 1970), and Weerasinghe (1994) provides a good example of a Systemic Functional Grammar (SFG) parser that used this approach. Earley used top-down, breadth-first methods with bottom-up recognition, while Weerasinghe used a bottom-up breadth first approach with top-down filtering.

10.2.2 Search strategies

10.2.2.1 Breadth-first versus depth-first

A **breadth-first** parser is one that works through the input string of words typically from left to right, assigning syntax tokens to each word before building the higher components. A **depth-first** approach will build the structure vertically (i.e. up to the sentence node or down to the word node) before it moves to the next word in the input string. The direction is from left to right in a language such as English which is written in this way, but the breadth-first approach could be used from right to left if that was appropriate.

10.2.2.2 Best-first and beam search (n-best)

The **best-first** search is a more efficient form of search, because it starts by exploring the most likely analyses first. These are selected by using the knowledge that, for example, following one particular path is likely to be more fruitful than another. When more than one analysis is followed in order of likelihood, this is called the **n-best search**. The value of **n**, which represents the number of analyses taken forward, is often called the **beam width**, and hence some implementers refer to the method as **beam-search**. Researchers have investigated the effects of changing the beam width

upon parsing accuracy and efficiency (see, for example, Collins (1999) and Henderson (2003a)).

Charniak and Johnson (2005) and Collins (1999), developed best-first, **coarse-to-fine** approach. Coarse-to-fine approaches, which used **discriminative ranking** for identifying the most likely parse trees, generate a large set of parse trees and revisit them afterwards, pruning them and discarding those of low probability.

10.2.3 Deterministic and non-deterministic parsing

Until the publication of Marcus' account of his parser PARSIFAL (Marcus 1980), virtually all parsers were **non-deterministic**. A deterministic parser is a parsing algorithm that is committed to the decisions that it has made, and only under very special circumstances, changes its mind. As Sampson (1983) points out, deterministic parsers are designed to avoid making false hypotheses, and therefore, they do not constantly backtrack from false hypotheses as non-deterministic parsers do, nor do they clone themselves into parallel processing systems when the grammar offers more than one choice (Sampson 1983).

Sampson (1983) favours deterministic parsing, arguing that determinism is a feature of the human parsing system, and that a human, when reading a text does not constantly have to re-read parts until he or she is sure of the meaning. However, there are certain 'garden-path' sentences that 'lead the reader up the garden path', and that catch out humans. Examples are:

The horse raced past the barn fell.

The cast iron their own clothes.

John gave the new cat food to Mary in a sandwich.

A fully non-deterministic parser, is one that takes ALL possible choices when they are presented and takes each of them to their final conclusion, and most early non-deterministic approaches are prepared to discard the work that it has done and backtrack and start again.² In contrast, a deterministic parser only backtracks when a human would (i.e. on encountering a 'garden path sentence').³

In order to enable a deterministic parser to function with accuracy, and to prevent backtracking, Marcus (1980) argues that it needs help through some sort of **look-ahead** ability. With this, the parser is allowed to look ahead for a small number of

² Some non-deterministic approaches keep previous analyses and reuse them when they are forced to backtrack. Other non-deterministic approaches, such as **chart parsers** (see Section 10.3.1.5), do not backtrack and generate all possible analyses, even those that are impossible.

³ Some argue that any parser that backtracks is non-deterministic (e.g. Sampson 1983).

words in the input string to help it make a deterministic decision about which route is the correct one.

This raises the question "what should a truly deterministic parser do when it encounters structural ambiguity, such as in the classic example sentence **John saw the man with the telescope ?**" Here, **John** can be interpreted as using the telescope as a visual aid to see **the man** (in which case, **with the telescope** is an Adjunct in the Clause), or **the man** could have a telescope in his possession (in which case it is a qualifier in a nominal group). Here, no amount of look ahead will tell the parser which is the correct attachment, and it can only get this information from the text or from its 'knowledge of the world' (if it has the ability to consult these, which it normally does not have). Sampson (1983:96) argues that if a deterministic parser produced more than one output, it is no longer completely deterministic. He quotes Marcus as he states that the deterministic parser should create only one output, and then supplement it with a flag which states that it is potentially ambiguous, so leaving the decision up to the semantic interpreter or higher component in the understanding system.

It can be argued, therefore, that one disadvantage of a deterministic parser is that it will normally return only one parse tree, and cannot cope well with structural ambiguity.

Next, I turn to the question about how deterministic parsers handle what appears to be an ungrammatical sentence. Some non-deterministic approaches keep partial parse trees and use them again when they are needed; for example, a chart parser (see Section 10.3.1.5) does not normally destroy partial analyses that it has built, and some backtracking parsers keep partial structures in case they are needed in a future analysis. Therefore, I argue that after the non-deterministic parser has done its best but failed, it could present some partial structures to the higher components. In contrast, deterministic parsers cannot handle ungrammatical sentences as well, as it does not build any partial structures except those to the left of the point at which it failed. Being able to handle ungrammatical sentences is a questionable advantage with a **non-deterministic parser**; but one could argue that many ungrammatical structures that have been created as text could still be interpreted by a higher-level understanding component. We should also note that humans frequently utter sentences that most parsers would treat as ungrammatical, because they only operate with the standard dialect of the language. Examples are **he never saw I**

(English West Country local dialect), and **me and him will be there**, which is common in the younger generation, and represents a change in the language that may become standard English at some time in the future.

10.2.4 Backtracking

All the early parsers were constructed on the assumption that backtracking was of interest as part of parsing. Some non-deterministic parsers use this mechanism when they find that they have made a mistaken analysis, and they wind back to the point in the parse that is known to be correct. As stated above, deterministic parsers **never** backtrack, and they make sure that they have made the correct decision before proceeding.

Some non-deterministic parsers (for example, those based on ATNs) employ backtracking techniques, whereas others (such as **chart parsers**) avoid backtracking as they follow all possible routes to their conclusion. Both techniques are extremely wasteful, and it was this that led Marcus to introduce the concept of a **non-deterministic parser**.

There is a difference between **computational backtracking** and **linguistically motivated backtracking** used in a parser. Typically, the former moves back a stage and takes the next path that has not been followed. The latter moves back to a point in processing that is recognised (using linguistic knowledge) as being an alternative to the failed attachment.

In its implementation in Phase One (see Chapter Thirteen), the CCP uses computational backtracking. For example, when all of the **n**-best joins fail to succeed, the parser moves back and takes the next **n** most likely joins. In Phase Two, the parser will use linguistically motivated backtracking (see Chapter Eighteen and Appendix L).

10.2.5 Incremental parsing (on-line parsing)

It is accepted by some psycholinguists that the human parser operates from left-to-right in a piece-meal fashion and builds up structures as it goes and modifies its analysis as more lexical items are recognised, as they become available. An incremental parsing process acts in the same way (see for example, Costa et. al (2003) who refer to the work of Marslen-Wilson (1973)). An incremental parser is able to present partial structures as it goes to the higher levels (for example a semantic interpreter) as it reaches a new lexical item in a left-to-right sequence. Incremental

approaches have been taken for chart parsing - for example Wirén (1989) and Weerasinghe (1994).

The term incremental parsing is also used by others to describe a slightly different approach which is more akin to shallow parsing and chunking. For example, Yang (1994) describes an incremental parser that is able to reuse parts of previously parsed trees. Wirén (1994) describes an incremental chart parser that reuses what it has done before given a small (incremental) change in the input sentence.

10.2.6 Classifying the parser described in Part Four

In this section, I detail how these terms can be used to classify the parser that is described in Part Four of the present work - the **Corpus-Consulting Probabilistic Parser (CCPP)**. As will be seen in Part Four, the CCPP algorithm constructs trees called **built structures** (which represent the parse so far) and attaches new partial trees, called **candidate structures**, to the built structures as the parse progresses.

As the CCPP starts with the items and works upward towards the sentence node, it is essentially a **bottom-up** parser. It also uses a **depth-first** approach, as it works vertically building elements up to units and units up to elements before attaching the structure to a built structure, which includes the sentence node before it moves to the next item in the input string.

The CCPP can be classified as **semi-deterministic**. Although it has been designed to take the most likely path first, and it is highly likely that this is the correct choice, it is not committed to that path in the same way as a deterministic parser is. There are two parts of the design which are not deterministic, these are (a) the use of **n-best** techniques and (b) the use of **backtracking**.

The CCPP allows **n** most likely trees to be taken forward to the next stage of the parse. Although the CCPP is not deterministic when values of **n** that are greater than 1 are used, this feature does not mean that it generates every possible analysis in the same way that a non-deterministic **chart parser** does (see Section 10.3.1.5). The CCPP also has backtracking, and according to the definitions, a deterministic parser is not allowed to backtrack. The provision of these two non-deterministic features provide the CCPP with the tools that it needs to be able to work with the complex model of syntax that is derived from the naturally occurring texts of a parsed corpus.

The CCPP is an **incremental parser** since it moves from left to right making analyses about the items that it finds, and builds candidate structures for attachment to the built structures. As will be seen in Chapter Sixteen, the incremental nature of the

CCPP has been exploited to assist in the development of the algorithm through the adoption of the **step-by-step parse**.

10.3 Parsing algorithms

This section will look at the algorithms used for parsing. Introductions to some of the most common algorithms and their significance to the field are given in Section 10.3.1. Some less common, but relevant approaches are given in Section 10.3.2. Section 10.3.3 provides details of how some of these concepts have been used in the present work.

10.3.1 Some common algorithms

10.3.1.1 Augmented Transition Networks (ATNs) and Recursive Transition Networks (RTNs)

Before the appearance of Marcus's parser in 1980, natural language parsing was dominated by parsers that used transition networks (see Section 5.2.2 of Chapter Five). The most influential was the Recursive Transition Network (RTN) developed by Woods (1970), named LUNAR (because it was used with a semantic interpreter to describe rocks brought back from the moon).

RTN parsers typically operate in a top-down, depth first manner, following the order of the arcs in the transition network. But a breadth-first approach is also common (see Johnson (1983) for a useful summary of the concept of ATNs).

10.3.1.2 Truly deterministic parsing

Marcus (1980) claims that PARSIFAL is a truly deterministic parser. It operates with three main data structures: the **active node stack**, the **buffer**, and the **unconsumed text**.

The active node stack contains those nodes that have been parsed so far, for which 'child nodes' are now sought (i.e. constituents). The sentence node is looking for a full-stop at the end of the input string).

The buffer contains nodes that are semi-analysed and are seeking parent nodes, and this buffer provides Marcus's parser with its look-ahead mechanism.

Elements in the buffer may be other individual words, or more complete structures which are looking for a place to which they may be attached. As attachments are decided, the appropriate buffer structures are transferred into the active node stack, and new words are consumed from the text into the buffer. Marcus limits the buffer to no more than three word look-ahead.

As Sampson (1983) points out that Swartout (1978) casts doubt on Marcus's claim that PARSIFAL is truly deterministic on the grounds that the buffer gives the parser some degree of non-determinism.

Another noteworthy deterministic parser was PARAGRAM (Charniak 1983). It was adapted from PARSIFAL and was able to handle ungrammatical input.

10.3.1.3 Shift-reduce parsing

Typically, **shift-reduce parsers** operate with a **stack** which maintains the current state of the parse. Normally, they operate left to right and bottom-up; Allen (1987:172) describes them as being bottom-up with top-down prediction, and therefore they can be considered to be mixed-mode. They can be implemented so that they are deterministic (Allen 1987:166-176). Shift-reduce parsers continue to enjoy popularity in the field.

As a word in the sentence is encountered, it is **shifted** onto the stack. When the top of the stack contains a set of tokens that can be rewritten by a rule, the top tokens are 'popped' from the stack, so that the stack is **reduced**, and the new left-hand side token is shifted onto the stack. Figure 10.3 describes this process in terms of a sentence analysed in terms of the Cardiff Grammar.

Examples of shift reduce parsers can be found that operate with rewrite rules, while others operate with state transition networks. Allen (1987:186) cites the pioneering work of Shieber (1984). A good introduction to the concept can be found in Winograd (1983).

Parse Stack	Input String	Remarks
()	The seagulls ate the mackerel	Stack initialised
(dd)	seagulls ate the mackerel	"the" as dd shifted to stack
(dd, h)	ate the mackerel	"seagulls" as h shifted to stack
(ngp)	ate the mackerel	dd and h reduced as ngp
(S)	ate the mackerel	ngp reduced as S
(S, M)	the mackerel	"ate" shifted as M
(S, M, dd)	mackerel	"the" shifted as dd
(S, M, dd, h)		"mackerel" shifted as h
(S, M, ngp)		dd h reduced by ngp
(S, M, C)		ngp reduced by C
(Cl)		S M C reduced by Cl
(Z)		Cl reduced as Z

Figure 10.3: Parsing a sentence with a shift-reduce parsing algorithm

Ratnaparkhi (1999) worked on Collins ideas, he developed a shift-reduce parser that used a **maximum entropy model**; his parser, like Collins, used the **head-driven**

parsing approach (see Section 10.4.2). Henderson (2003a), also building on the work of Collins (1996, 1999), created a shift-reduce probabilistic parser that used a **left-corner algorithm** (the parameters of which were calculated using neural networks). He implemented two versions; one with a discriminative model and the other a generative model. He reported significant efficiency gains when he used look-ahead in his discriminative model, but higher accuracy when he used a generative model that made predictions as it reached words and built up sub-structures.

10.3.1.4 Definite Clause Grammar (DCG) parsers

Definite clause grammars provide parsers that are dependant on the use of PROLOG. Examples include those described by Colmerauer (1978), Warren and Pereira, (1982), and Pereira and Shieber (1987). A more recent example is described in Sprech et al (1995), who implemented a DCG parser for Hebrew texts.

The disadvantages of DCGs arise from the fact that they are programmed in a declarative language (i.e. PROLOG). Much of the way the parser works is under the control of the programming language, and it may be difficult to implement new features, such as specialised searches. They are also limited to a top-down, breadth-first type of search, unless the programming language's search strategy is changed.

10.3.1.5 Chart parsers

There are many parsing systems that have a **chart** as their core data structure. Early examples include Kay (1989), Magerman and Marcus (1991b), Stolcke (1993), and Weerasinghe (1994), and a more recent example is described by Hall (2005). Chart parsing continues to be a popular approach.

A chart parser is non-deterministic, since it follows all possible hypotheses to their conclusion, so that they never need to backtrack. They can be either top-down or bottom-up, breadth-first or depth-first, with bottom-up breadth-first being the most popular.

Chart parsers have been developed to operate with a 'rewrite rules' version of Systemic Functional Grammar (SFG), e.g. Souter (1996), and Weerasinghe (1994), as we shall see in the next chapter. Other examples use a transition network, e.g. Hall (2005), and, in the present work, a database formalism (see Chapter Twelve).

Collins (1996, 1999) used a simple bottom-up chart parser that extended the standard **bigram** algorithms to calculate probabilities that were based on the dependencies between the 'head' words in the parse tree. In his work, he claims a

parsing speed of 200 sentences per minute when a **beam search** strategy is introduced. He also found that although increasing the **beam size** improved the accuracy of the parser, its speed reduced.

Let us look briefly at how this popular type of parser works. As Figure 10.4 shows, the basic chart has **nodes** and **edges** (sometimes called **arcs**). The input string is divided into its items, and a node is placed at the start and end of the input string and at the gap between each of these items (represented by the numbers in Figure 10.4). An edge, which starts at one node and finishes at a later node (shown by arrows in Figure 10.4), represents an analysis of the syntax structure for the **span** of the edge. An edge that starts at the first node and ends at the last node, and that has the sentence token dominating the edge, represents an analysis of the complete input sentence.

An edge can be **active**, which means that it is not yet complete and is seeking further elements of structure, or **inactive** which means that it is complete, and it has been incorporated into the higher analysis. A dot notation is typically used in the literature to separate the part of the rule that has been found from the part that is still being sought. Thus, the rule **ngp->dd . h** represents a **ngp** that has found a deictic determiner (**dd**), and is looking for a head (**h**). In a typical implementation, an edge is a five tuple **{start, end, token, found, toFind}**, where:

- (a) **start** represents the start position of the edge,
- (b) **end** represents the end position of the edge,
- (c) **token** is the syntax token that represents the syntactic category of the edge,
- (d) **found** is a list of tokens that have been found so far (i.e. tokens to the left of the dot),
- (e) **toFind** represents the tokens that the edge is still looking for to complete the edge (i.e. tokens to the right of the dot).

The chart parsing algorithm has the great advantage that it never throws away a partial analysis. This means that both successful and non-successful analyses are recorded in the chart, and that the parser never has to backtrack and re-analyse what it has analysed before.

Typical implementations have the notion of a **completed edge**, this is an edge that has become inactive, since it is not looking for any further elements (i.e. its **toFind** list is empty).

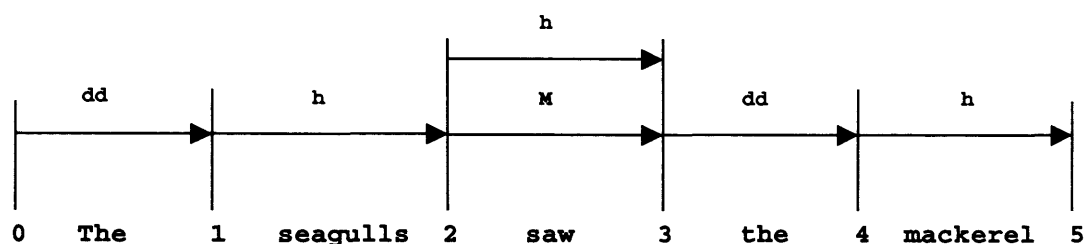


Figure 10.4: The chart after the initialisation stage

The process starts with **chart initialisation**. This adds an edge for each element of structure that an item can expound, as in Figure 10.4. Typically, the parser does this first for all items (i.e. operating breadth-first). Weerasinghe's parser (1994), implemented an incremental approach where items are added to the chart as they are accepted. In Figure 10.4, which shows the state of the chart after initialisation, the item *saw* is ambiguous, and therefore has two edges in the chart.

Following initialisation, the algorithm applies the **fundamental rule**. This states that an active edge that is seeking an element is to be combined with other edges that have that element on their left-hand side. The **bottom-up** rule then ensures that each edge that is **inactive** (i.e. complete, and seeking no further elements) will create a new edge for each rule in the syntax list that has the token represented by the edge, as the first token in the elements on the right-hand side of the rule.

Typically, when applied to a Systemic Functional Grammar, the fundamental rule is typically used to add elements to the elements that have been found for a unit. Further, the application of the bottom-up rule is problematical in any full model of syntax, such as the Cardiff Grammar, since it is not easy in such a grammar to decide that a unit is closed. This is either because there are optional elements that may not occur after the head of the unit (e.g. a qualifier in a nominal group, or an Adjunct in a clause), or because the head itself is missing from the unit (as in the samples cited in Section 10.4.2). In the experimental work carried out before deciding on the form of the final parser, I implemented a chart parser that introduced the concept of 'potentially closed edges'. These were allowed to act as inactive edges, and so trigger the bottom-up rule, while the lower edge still remains active, and can therefore accept further elements if they are found (see Chapter Twelve).

Typically, chart parsers employ an **agenda** which may take the form of a queue or a stack; the former provides a breadth first search and the latter a depth first search. Figure 10.5 shows a complete parse and typical edges for the example sentence.

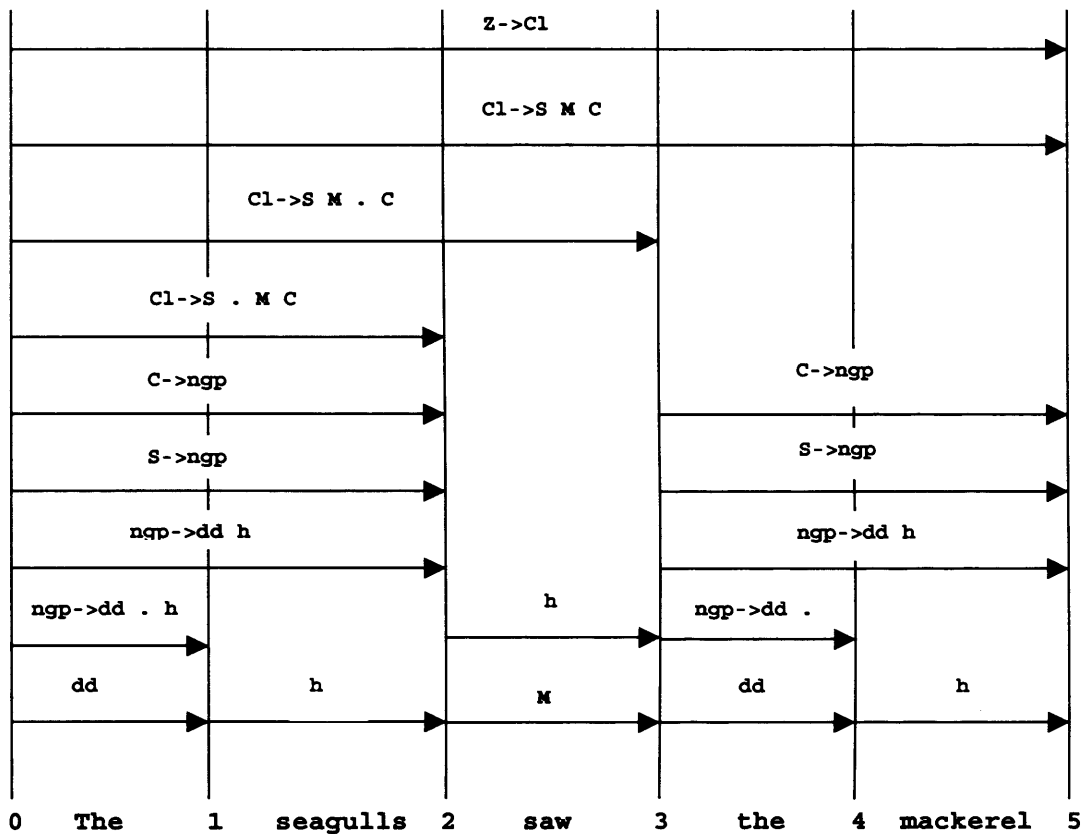


Figure 10.5: A simplified view of the chart after parsing the example sentence

10.3.1.6 CYK parsers

The CYK algorithm (so called because this type of parser was developed independently by these different researchers: Cocke, Younger and Kasami), is very similar to the chart parsing algorithm. The main difference is that it requires the grammar to be in what is termed Chomsky Normal Form (CNF), whereas a chart parser can be used with different formalisms. For a description of this algorithm, see Grune and Jacobs (1990). Klein and Manning (2003) produced a CYK parser that showed better than expected results when using unlexicalised grammars (where no head word annotates the phrasal nodes).

This now concludes our survey of the common algorithms that have been used in parsing. In the present work, in the development of the CCPP, a chart parser was developed, and this work is described in Chapter Twelve. One of our main goals in the development of the parser, as will be described in Part Four, was to closely model the way the human parser is thought to work. That is, working from left-to right in the sentence, and making decisions about the syntax structures as it goes, by drawing on linguistic knowledge.

The number of syntax 'rules' extracted from the naturally occurring texts of a parsed corpus is very large. As will be seen in the next chapter, and in Chapter Twelve, the 'brute-force' approach that is used by the chart parsing algorithm, and the related CYK algorithm, is only suitable for a corpus based approach provided some additional techniques are applied in order to reduce the number of analyses. It can be argued that these approaches, although they have had their successes, do not model the human parsing process, and represent a more mathematical approach to the problem.

As has been suggested by Marcus, Sampson and others (see Section 10.2.3 and 10.3.1.2), deterministic parsing does follow the human parsing process, and the human only backtracks when a garden path sentence has been encountered. Therefore, the CCPP will be designed to display a degree of determinism, while drawing upon some of the features that have seen success in other approaches (such as the beam search). In the next section, I describe some of the methods that others have used, normally in combination with the algorithms described in Section 10.3.

10.3.2 Other relative parsing algorithms

10.3.2.1 Vertical strips parsing

The Vertical Strip Parser developed by O'Donoghue (1991a), was implemented within the COMMUNAL Project. He extracted a set 'vertical strips' from the POW Corpus, and created a network of strips. The network indicated which strips can follow others and essentially formed the rules of his 'grammar'. His parser operated by joining one vertical strip to another where they had matching units and elements, using the network to find strips, and then join them together.

A vertical strip being defined as a set of nodes from the sentence node downwards to an item (but not including the item). Below is a set of vertical strips extracted from the sample sentence in Figure 10.1 and Figure 10.2. O'Donoghue's parser, which was designed for SFG, is discussed fully in Section 11.5 of Chapter Eleven.

```
the          dd ngp S C1 Z
seagulls     h ngp S C1 Z
ate          M C1 Z
the          dd ngp C C1 Z
mackerel     h ngp C C1 Z
```

10.3.3 Concepts used in the current work

As described in Section 10.2.6, like Marcus, one of the primary goals of this project was to design a parser that has deterministic qualities. Although we do not use look

ahead, we have recognised the need for it for multiple word items, however, we will not then be using it for the same purpose as Marcus did.

In the development of the CCPP, the use of chart parsers was investigated and a prototype was developed. The purpose of this work, which is reported in Chapter Twelve, was to determine if the chart-based algorithm is suitable for a corpus-based approach building on the work of Weerasinghe (1994) and Souter (1996).

The CCPP in its final form, is quite unlike any other parsing algorithm described here. However, it uses a **state transition** to move between the stages of the parse based on success or failure. This is used in the same way as a workflow system and the states have nothing in common with an ATN.

As will be described in the next chapter, Weerasinghe found that the number of edges generated even for a simple sentence caused a problem to the efficiency and accuracy of the parse. Souter found that these problems were exacerbated when the model of syntax was extracted from a corpus. Weerasinghe, Souter and many others have employed techniques in order to overcome these limitations. The next section describes these techniques and their limitations and successes in order to assess their suitability for this present work.

Although the CCPP uses the concept of O'Donoghue's vertical strips extensively, it does not use his parsing algorithm.

Algorithm	Applications	Advantages	Disadvantages
Augmented Transition Networks / Recursive Transition Networks (ATN/RTN)	non-SFG: - Woods (1970,1980) - Bobrow (1978)		- Grammars are modelled as networks and these can be difficult to express for complex naturally occurring grammars.
Charts	non-SFG: many examples eg - Kay (1989) - Magerman and Marcus (1991b) - Hall (2005) SFG: - Weerasinghe (1994) - Souter (1996)	- Finds all possible parses without the need to backtrack - Edges can apply to more than one analysis - no need to regenerate	- Brute-force approach which generates every possible edge (gives a great many impossible edges). - Slow for complex grammars. - Researchers forced to implement methods of reducing the number of impossible edges generated.
Shift-reduce	non-SFG: - Shieber (1984) - Abney (1991) - Ratnaparkhi (1999) - Collins (1999) - Henderson (2003a) SFG: - Winograd (1983)	- Good at handling ambiguity because decisions of which rule to apply can be delayed until further on in the parse (Allen 1987)	
Definite Clause Grammar (DCG)	non-SFG: - Colmerauer (1978) - Warren and Pereira (1982) - Pereira and Shieber (1987) - Sprecht et al (1995)	- Easy to implement in PROLOG facts	- Difficult to extend method. - Search order defined by the programming language (although can be modified with effort). - Grammar generated from a corpus will result in thousands of Horn Clauses.
Deterministic approach	non-SFG: - Marcus (1980) - Charniak (1983)	- Gets it right first time - Does not over generate	- Has to backtrack to get alternative parses. - depending on method, the same partial analysis may have to be generated for alternative parses.
Vertical Strips	SFG: O'Donoghue (1991a)	- High success rate using a corpus-generated grammar (approx. 79%)	- Success rate can never be 100% (O'Donoghue 1991a). - Needs methods to generalise for rare structures that have not occurred in the corpus.

Table 10.1: A Comparison of parsing algorithms

10.4 Other techniques

Two critical success factors with which to measure parsing algorithms are **efficiency** and **accuracy**. Efficiency is measured in terms of (a) the length of time taken to find a solution, and (b) how many incorrect structures have been created before the correct solution is found. Accuracy is the ability of the parser to find the correct parse.

This section provides details of some of the techniques that have been used in conjunction with the algorithms described in Section 10.3 in order to improve the **efficiency** or **accuracy** of the parsing process. I describe the advantages and disadvantages of these approaches in the context of the present work.

10.4.1 Left-corner parsing

Left-corner parsing, as used for example by Rozenkrantz and Lewis (1970), is an attempt to improve the efficiency of the parser by introducing a new node into the parse tree only when it has fully analysed the left-most node. The remaining nodes for the same sub-tree then being parsed in left-to-right order. See Henderson (2003a) for a recent attempt at left-corner parsing.

10.4.2 Head-driven parsing

Head-driven parsing provides an alternative to the simple left-to-right type of breadth-first parsing. In this approach, the parser looks first for the words that are likely to be functioning as **head** of a syntax unit first. Kay (1989) states that the parser should start with the main verb, since this gives information about the types of subjects and objects that are likely to be found. After finding the subjects and objects, Kay suggests that the parser should go looking for the head words in them. Weerasinghe's parser (described in Section 11.6 of Chapter Eleven) is of this type. Problems arise, however, when the head is not present, as in examples such as:

S1: Give me five

where, in the Cardiff Grammar, **five** is a quantifying determiner, because it can be followed by **of them**. Another example is given below where the word **people** is ellipped:

S2: The very rich (people)

10.4.3 Shallow parsing

Shallow parsing is the name given to the technique of finding the main constituents of a sentence before processing the complete parse. For example, when applied to a

Systemic Functional Grammar, we would find units first, and then attempt to detect the relationships between them using local evidence in the vicinity of the construct.

Abney (1991) called his shallow parsing method **parsing by chunks** and argued that the model is similar to the way a human will break down a sentence in order to understand it. Abney's method has two distinct stages. The first stage is performed by the **chunker** which, in SFG terms, is responsible for identifying the main units. The second stage is responsible for joining the chunks, and is performed by the **attacher**. In Abney's modular approach, the job of the chunker, he argues, is simplified as it does not have to worry about complex joins, which are the responsibility of the attacher.

Abney's chunks always must contain a **head word** and **function words** that he argues always match a fixed template, and it is therefore a variation of a head-driven parsing. We have seen in our arguments above on head-driven approaches that the method suffers as heads are not always present in units.

10.4.4 Pre-tagging

In a similar approach to shallow parsing, the use of part-of-speech tagging as a pre-stage to parsing is becoming more popular. This method was used by the International Corpus of English Project (see Section 3.2.6 of Chapter Three), and by Souter (1996), in his Systemic Functional Grammar chart parser (See Section 11.7 of Chapter Eleven). Pre-tagging uses a part of speech tagger (see Section 3.3 of Chapter Three) to identify the elements that items expound before starting the parsing process.

10.4.5 Word lattice parsing and neural networks

Hall (2005) argues that word-lattices provide a compact representation for a set of word strings and their scores. Hall, transformed word-lattices into finite state machines for use in his best-first, bottom-up chart parser.

Neural networks have been used in conjunction with probabilistic methods within parsing algorithms. These comprise Simple Synchrony Networks (SSNs) which return a score that can be used to determine the most promising path.

Henderson (2000), working at the University of Exeter, used Neural Networks to return a probabilistic score that represented the most likely path. He used the parsed SUSANNE Corpus (see Sampson (1995) and Section 3.2.4 of Chapter Three) and the Lancaster-Leeds Treebank (see Garside et al (1987)) for the part-of-speech tags. He found that the SSN parser outperformed other parsers that used a probabilistic context-free grammar.

10.4.6 The relevance of these techniques to the present work

This section has looked at the techniques that have been used to improve the efficiency or accuracy of the parsing process. Next, I turn to the suitability of these methods in the present work.

Weerasinghe successfully used a head driven approach in an attempt to reduce the number of edges in his chart. However, as discussed in Section 10.4.2, often the head is not present, particularly in spoken texts. Shallow parsing works in a similar way to the head driven approach, and both methods are not suitable for parsing naturally occurring texts and hence are not used in the CCPP.

The CCPP arguably, operates in a similar way to the left-corner approach as it builds up from an item to an element, and then to a unit before attaching it to the built structure. Pre-tagging is not needed in the CCPP as the information about the elements that particular items expound is extracted from a parsed corpus.

The CCPP does extensively use probabilities in order to decide which analysis is the correct one and to only follow those that are less likely when the chosen analysis fails to reach a conclusion. The next section looks at probabilistic approaches and how they have been used in parsing.

10.5 Probabilistic and statistical approaches

In the early 1990s, the combination of statistical grammars, and the availability of large corpora, opened up a new and exciting branch of computational linguistics called probabilistic parsing.

Probabilistic methods can be used in conjunction with any of the common parsing techniques described in Section 10.3 by augmentation. For example:

- (a) in a DCG parser, a Horn clause can be expressed with an associated probability that is used when the fact is used,
- (b) The rules used by a chart parser can have a probability that can be used to calculate the probability of the edges in the chart,
- (c) The paths through a transition network can have a probability associated to the arc that represents a transition from one state into another.

One technique for modelling probabilities is the use of the **Markov model**. Researchers in parsing borrowed techniques from the field of speech recognition. They began by using the simple conditional models provided by **bigrams** (the co-occurrence of two items), and some moved on to **trigrams** (the co-occurrence of three

items) with greater success. Weerasinghe (1994;71) reports that further transitions (over three) have proven less successful.

Stolcke and Omohundro (1994) used a best-first Hidden-Markov model approach and Guyon and Pereira (1995) researched the effect of variable length Markov models to predict the next character in sequence in a sentence. They reported trade-offs in accuracy and speed depending on the length of the n-gram being used.

As reported in the next chapter, Weerasinghe's parser (1994) was a probabilistic chart parser for Systemic Functional Grammar, and other researchers such as Collins (1996) implemented probabilistic parsers that used Phrase Structure Grammars. The techniques used in many implementations were similar, and they have been also applied to Shift-Reduce parsers, by for example, Ratnaparkhi (1999) and Henderson (2003a). The Leeds COMMUNAL team applied probabilistic methods to various parsing techniques. Probabilistic parsing continues to enjoy popularity and success today.

There has to be a source from which the probabilities are derived. Some early parsers used statistics that are created manually based on intuition. Most of the more promising approaches, however, use probabilities that have been extracted from a parsed corpus. As the number of parsed corpora increases, it is expected that the number of parsers using this approach will also increase. It is to **corpus based parsing** that we turn next.

10.6 Corpus-based approaches

In contrast with parsers that use formal syntax rules which have been developed by a linguist, corpus-based parsing methods rely on past linguistic events (i.e. texts) that are stored in a corpus. Different names have been used for the approach. The term **data-oriented parsing** is frequently used to describe corpus-based approaches. Corpus-based approaches typically use probabilities.

History-based parsing is a probabilistic corpus-based approach that uses mechanisms for taking advantage of contextual information from anywhere in the discourse history. The method, which was first used in the IBM Research Center by Black et al (1993), uses decision trees to enable it to look at the other sentences that occur in the context of the sentence being parsed. This allows the parser to make a decision about which analysis is the correct one when there is more than one.

Souter (1996) extracted his probabilistic data from a parsed corpus and his work was probably one of the first examples of a parser that worked with the syntax information extracted from a parsed corpus (see Section 11.7 of Chapter Eleven for an analysis of Souter's parser).

The disadvantage of approaches in which syntax 'rules' are automatically extracted from a corpus, is that the model of syntax is only as good as the corpus that created it. In a small corpus, some rarely occurring constructs may not be present because they have not occurred yet. Further, rare constructs that have occurred may be given undue weighting, as the sample size is small. Therefore, I argue that the larger the corpus is, the more reliable the 'grammar' that is extracted will be.

The work to be presented in Part Four is a corpus-based approach. However, it is different to Souter's in a number of ways. The most important, from an innovation perspective, is that its probabilistic data comes from a **dynamic corpus**, i.e. a corpus that is expanded as sentences are parsed and added to the corpus, and so changing the probabilities (see Section 1.1 of Chapter One).

10.7 Applying the algorithms to Systemic Functional Grammar

Note that all algorithms when applied to a Systemic Functional Grammar (SFG) will need to be modified to handle the relationships between items, elements and units (see Chapter Four). For example, when a chart parsing algorithm is used for SFG, either of the following methods can be used:

- (a) edges always represent units (**ngp->dd mo h** but not **mo->ngp**),
- (b) edges represent either a unit or an element (**ngp->dd mo h** and **mo->ngp**),
- (c) syntactic rules are rewritten such that elements and units are merged (**S_ngp->dd mo h**).

In the first method, the units are handled directly by the fundamental rule. When a unit is complete, the rule will create a new unit edge for the parent unit of each element of structure that the unit can fill. This approach was used by Weerasinghe (Weerasinghe, 1994).⁴ In order to use the method, changes to the standard algorithm have to be made to handle co-ordination. The second method requires no change to the standard chart parsing algorithm but will mean that the syntactic 'rules' will be larger and the number of edges in the chart will be significantly greater when

compared to the first method. The third method, which was one of the methods explored by Souter (1996), also demonstrates the advantages of the first but can use the standard algorithm without modification. However, there will be more syntactic 'rules' for the third method.

Similar approaches will be needed for other parsing approaches. For example, the Horn Clauses in a DCG parser can follow methods 1, 2 or 3, although it may be a slightly more complex algorithm for method 1. In a shift-reduce parser, all three methods can be implemented; for method 1, the unit can be popped automatically from the stack when it is complete however, like the chart parser, non-standard methods would be needed for co-ordination.

10.8 Summary

This chapter started in Section 10.1 by identifying the goal of any parser - to turn a sentence from a string of words into a tree diagram annotated with syntax labels.

Section 10.2 looked at the ways parsing algorithms have been classified, in terms of the way they operate, and the search strategies they use. These terms were then used to classify the parser that is described in Part Four of this work.

Next, in Section 10.3, the more common parsing algorithms were introduced, and any concepts from these that the parser described in Part Four draws on were identified.

Some researchers have produced parsers that use additional features in order to increase the efficiency and accuracy of their parsing algorithms, and these have improved their results. The most common of these approaches were identified in Section 10.4. They include those that use head-driven methods, but these, which despite their success, cannot be used for parsing naturally occurring texts as the heads are not always present.

As discussed in Section 10.5, probabilistic approaches have been used with promising results; however, as Souter (1996) discovered, problems arise when a parser uses (a) probabilistic 'rules' that have been extracted from a parsed corpus, and (b) traditional algorithms such as chart parsing. The root of the problem is the large number of 'rules' extracted. These corpus-based approaches were discussed in Section 10.6.

⁴ He could do this because of his head-driven approach (see Section 11.4 of Chapter Eleven).

Systemic Functional Grammar (SFG) presents a significantly different set of challenges to those presented by Phrase Structure Grammars, and these differences mean that the algorithms given in Section 10.3 work less efficiently for SFG. Section 10.7 outlined these differences in preparation for Chapter Eleven (where attempts at parsing SFG are described), and for Chapter Twelve (which describes early prototype parsers developed in the present project).

Chapter Eleven

Earlier parsers that use Systemic Functional Grammar

This chapter examines previous attempts to build parsers that use Systemic Functional Grammar (SFG). The early parsers that used the Cardiff version of SFG were developed at Leeds University by Atwell, Souter and O'Donoghue (see Sections 11.2 and 11.5 below), and two later parsers were constructed by Weerasinghe, at Cardiff University in 1994 (Section 11.6), and by Souter, at Leeds University in 1996 (Section 11.7). These will be discussed in the later sections of this chapter, but we shall begin with a brief survey of earlier parsers that used SFG. The first of these was developed at the end of the 1960s by Winograd at the Massachusetts Institute of Technology (MIT), after studying SFG under Halliday in London (Winograd 1972), and later attempts include those of Kasper (1988) and O'Donnell (1994).

11.1 Winograd's SHRDLU

Winograd's (1972) system was not merely a parser, but a complete (if limited) language understanding system, and his work was widely recognised as a landmark achievement in parsing.¹ His system allowed a human to communicate, using a restricted form of English, with a virtual robot. The human could ask it to manipulate building blocks of different shapes and colours within its virtual world, and then to answer questions about what it had been doing. Winograd's system was the first to demonstrate that the type of 'flat tree' structure that is in the output from an SFG natural language generator can also be used as an output from a natural language parser. However, the impact of his system on the field of natural language understanding was due less to his use of SFG syntax and more to the fact that that it contained small versions of all the main components of a complete natural language processing system. Not only did it parse the input, but it also developed a semantic interpretation of it, then reasoned about it, and, if appropriate, it responded by generating natural sounding dialogue.

¹ An even earlier attempt at parsing SFG was that of Parker-Rhodes in 1962-63. He worked with Halliday on the NUDE project (see O'Donnell and Bateman 2005:346), and he developed a parser for a grammar based on Halliday's Scale and Category Grammar.

Winograd (1972:22) classifies his parser as a **top-down, left-to-right** parser. His approach was different from that of others, as its knowledge of syntax was encoded within the procedures of an interpreted program. A special programming language, called PROGRAMMAR, was specifically designed for writing his 'grammars'. Another interesting part of his design (1972:23) was the fact that it was an interleaved approach with the parsing process being 'unified' with the semantic interpreter, as the semantic interpreter was used to 'guide the parsing'. He also implemented a backtracking mechanism within PROGRAMMAR that did not, like other parsers of the time, blindly go back and try alternative paths, but he allowed his program to decide specifically what should be tried next.

Weerasinghe (1994:44) rightly points out that Winograd's model of syntax is difficult to modify, because it is encoded within the lines of a programming language and one has to take care not to interrupt the complex interrelationships between the procedures of the program. The problem, which is similar to the problems associated with transition networks (see Section 5.2.2.2 of Chapter Five), increases when the syntax model is extracted from the naturally occurring texts of a corpus, because the interrelationships then become even more complex.

11.2 Earlier work at Leeds University

The earliest attempts at parsing what was later to be called the Cardiff Grammar, were carried out by the Leeds Team of the COMMUNAL project in the late 1980s. Atwell et al (1988a and 1988b) attempted to create a Definite Clause Grammar parser (see Section 5.2.5 of Chapter Five, and Section 10.3.1.4 of Chapter Ten), which was based on patterns of elements in units ('rules') that were extracted from the POW Corpus. There were many such patterns, each of which were functionally equivalent to a rewrite rule in a Phrase Structure Grammar (e.g. **CL -> S O M C**). However, success was limited due to memory limitations of the machines of the time.

Atwell et al (1988b) experimented with a number of parsers, the most promising of which was called the Realistic Annealing Parser (RAP). It used **simulated annealing**, which is a general-purpose stochastic optimisation technique for finding a solution in a very large search space (Atwell et al 1988b:75). The following explanation of the process is taken from Souter (1996:67). Simulated annealing works by starting with a parse tree that contains the sentence node, and nodes for each item. It then proposes changes, in the form of partial trees, and evaluates it using

probabilistic methods. Changes that represent an improvement are accepted as local changes, and changes that do not represent an improvement are accepted (initially) as global changes. As the parse progresses, fewer and fewer worsening changes are accepted.

Souter (1996:68) reported that the parser was very slow and unreliable, particularly on longer sentences. O'Donoghue later did a number of further experiments with the parser, but abandoned it in favour of his Vertical Strips Parser (VSP). The VSP, which has had a direct influence on the data structures that are used in the parser to be presented in Part Four, is described in Section 11.5.

11.3 Kasper's NIGEL-based parser

Kasper experimented with the NIGEL grammar (see Section 2.3.2 of Chapter Two) to see if it could be applied to parsing. As O'Donnell and Bateman point out (2005:351), Kasper (1988) used a **bottom-up chart parser** to produce a representation of syntax in terms of a simple PSG like formalism, which then became the input to the application of the realisation statements (operating in reverse), the product of this being a representation in terms of the features in the system network. This second stage of the process was therefore the equivalent of O'Donoghue's (1991b) semantic interpreter (as described in Section 2.3.1 of Chapter Two). It is therefore only the preparatory parser that is in any way comparable to the function of a parser, as the term is understood here.²

Although Kasper's system is noteworthy as a first attempt at parsing that used a full-Hallidayan grammar, O'Donnell and Bateman (2005:352) rightly argue that the importance of the work is lessened by its dependence upon Phrase Structure Grammar.

11.4 O'Donnell's work

O'Donnell and Bateman (2005) also report that O'Donnell and Whitney repeated Kasper's work, using a knowledge representation system called LOOM.

O'Donnell (1993, 1994, 1996) then produced a chart parser that did not need a PSG backbone that operated within the Workbench for Analysis and Generation (WAG) system (see also Section 2.3.2 of Chapter Two). He argues that the system networks are top-down, and a bottom-up approach is needed for parsing. Therefore he automatically extracted a parsing grammar from the system networks and realisation

rules of the NIGEL grammar (O'Donnell 1994:124). However, he found that parsing with the full NIGEL grammar was very slow and he extracted a smaller parsing grammar that was suited to the types of texts he was parsing.

O'Donnell (O'Donnell 2005:1), working in a Belgian informatics company (Language and Computing) with van Der Vloet, Coppens and Van Mol, developed a Systemic Dependency Grammar (e.g. as in Hudson 1976). This was used in a commercially available probabilistic parser that predicted the most likely parse when more than one was produced. O'Donnell (2005) builds on his earlier work in the development of the UAM parser.³

11.5 O'Donoghue's Vertical Strips Parser

O'Donoghue has made two major contributions to the COMMUNAL Project. First, as we have already noted in Section 2.3.1 of Chapter Two, he developed COMMUNAL's semantic interpreter (O'Donoghue 1991b). His second contribution was the concept of using **vertical strips** in a parser (O'Donoghue, 1991a). This was a parser that operated in an innovative way. He extracted a set of **vertical strips** from a corpus called the Prototype Grammar 1.5 Corpus (PG1.5), together with their frequencies and associated probabilities of occurrence.⁴ A vertical strip is defined as a complete set of nodes from the sentence node downwards to an item (but not including the item).

The crucial difference between this approach, and traditional approaches, is that in a vertical strips parser, the key generalisations relate to vertical slices through a parse tree rather than horizontal ones (as in a PSG). These simply specify when strips can follow any given strip, but, of course, there are a great many of them.

11.5.1 The data structures

As reported by O'Donoghue (1991a:1), the algorithm works in a similar way to the CLAWS word tagger (Leech et al 1994), but with the important difference that, instead of assigning one or more parts-of-speech, it assigns one or more vertical strips.

The vertical strips are stored in a **network**, together with the strip co-occurrence rules. These give the details of the strips that can follow a given strip with its associated probability. In addition to the strips, he also extracted a lexicon from the

² Each Phrase Structure rule also contained information for a mapping into a systemic structure tree. This approach is referred to as parsing with a 'context-free backbone' (O'Donnell and Bateman (2005:352)).

³ So-called because it is developed at the Universidad Autonoma de Madrid.

⁴ The PG1.5 Corpus is an automatically generated (and so artificial) corpus, and contains 100,000 sentences from an early version of the COMMUNAL GENESYS generator.

corpus, which maps items to the elements of structure that each can expound, so making it possible to complete each vertical strip down to the items themselves.

The network provides for each possible strip, together with pointers to other strips that can co-occur with it, and with the probability that each may follow the given strip. A vertical strip network (taken from O'Donoghue (1991a)) is shown in Figure 11.1.

11.5.2 The algorithm

The parsing process starts at the leftmost item in the input string. The first step is to retrieve all possible elements that the item can expound from the lexicon. Then, for each of these, it matches the element with the leaf node of a vertical strip that is available in the vertical strip network, and follows the start strip which is indicated by #. The parser moves to the next item in the input string, and identifies any strips which: (a) are 'pointed to' from the last matched strip(s), and (b) which match the element(s) of structure that the new item can expound. The process continues until there are no more items in the input string. A successful parse is one in which the last matched strip is a 'legal' end strip (i.e. one that points to the strip indicated by a \$).

However, this description presents an over complicated picture, because there may be more than one path through the strips in the network. In the second stage of the parsing process, O'Donoghue determines the most likely parse using the probabilities assigned to the pointers between the matched strips in the network, as shown in Figure 11.1. Table 11.1 demonstrates O'Donoghue's approach, as applied to the sample sentence shown in Figure 10.1 of Chapter Ten.

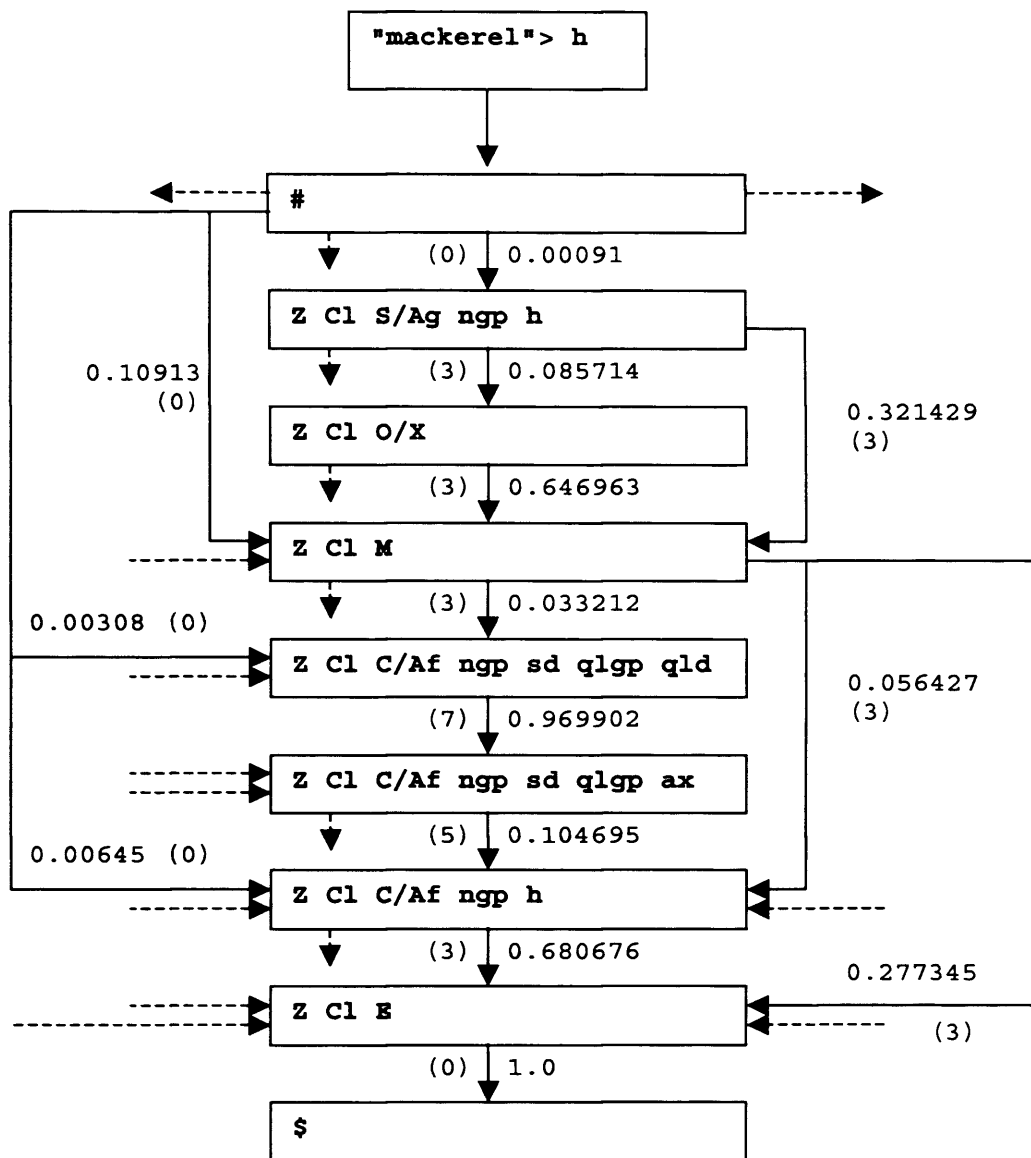


Figure 11.1: A vertical strip network (from O'Donoghue 1991a)

11.5.3 Evaluation

O'Donoghue's results were encouraging. He tested it on 1000 further sentences generated by the PG1.5 Generator, and found that, for 98.8% of these, the parser identified a set of potential syntax trees. He reports that the number of trees found varied from 0 to 58, with an average of 6.6 trees per parse. For the 1000 sentences, 98.4% returned a set of parse trees which included the correct parse, and 0.4% of cases the 'correct' parse was returned as the most probable. O'Donoghue recognised that the performance of the Vertical Strips Parser would need to be improved if it was operate with a network that had been extracted from a non-artificial corpus (because the network would be larger and more complex). In this situation, he states that the average number of parse trees generated per sentence would rise (O'Donoghue

1991a:16), and the effect of this would be that his parser would be less efficient (in terms of speed and the number of structures generated). It would also be less accurate, because a greater number of parse trees would be generated. However, one would expect that the percentage of sentences that contained the correct parse tree would remain constant.

O'Donoghue's network representation is similar in many ways to the augmented transition networks that were used, for example, by Woods (1970), as we saw in Section 5.2.2 of Chapter Five. The first major difference is that, a node in the network does not simply represent a syntax node. Instead, it represents a vertical strip. Secondly, O'Donoghue attaches **probabilities** to the arcs (pointers) that show a transition from one partial parse tree to another.

Predictably, O'Donoghue's vertical strip network is very complex, given that it was constructed automatically from a corpus. He reports that it contained 1624 strips, and 6358 transitions (O'Donoghue 1991a). Like other ATN-type parsers, a network aiming at covering anything approaching the full range of strips that would be represented in naturally-occurring texts, becomes increasingly difficult to manage. Manual changes to the automatically extracted grammar may be tried, in an attempt to improve the parser so that it can still arrive at a successful parse for sentences that contain strips that have not (yet) occurred in the corpus. For a small corpus, we can expect this to happen quite regularly. It may be difficult to implement augmentations to overcome these coverage problems within O'Donoghue's networks.

As we will see in Section 14.1 of Chapter Fourteen, the concept of a vertical strip (which we owe to O'Donoghue), plays a critical role in the parser presented in Part Four.

Step	Strips found	Input sentence	Path followed	Remarks
1		The seagulls ate the mackerel		
2	dd ngp S/Ag Cl Z	seagulls ate the mackerel	(dd ngp S/Ag Cl Z)	the is retrieved from the lexicon as a dd . The parser gets all strips matching [dd *] that follow the start strip [#].
3	h ngp S Cl Z	ate the mackerel	(dd ngp S/Ag Cl Z)+(h ngp S Cl Z)	seagulls is recognised as a h , and all strips matching [h *] are found that follow any strip identified in Step 2.
4	M Cl Z	the mackerel	(dd ngp S/Ag Cl Z)+(h ngp S Cl Z) + (M Cl Z)	ate is retrieved from the lexicon as an M . All strips are found that match [M *] which follow any strip identified in Step 3.
5	dd ngp C/Aff Cl Z	mackerel	(dd ngp S/Ag Cl Z)+(h ngp S Cl Z) + (M Cl Z) + (dd ngp C/Aff Cl Z)	the is recognised as a dd , and all strips are found that have a dd at the leaf, and follow any strips identified in Step 4.
6	h ngp C/Aff Cl Z		(dd ngp S/Ag Cl Z)+(h ngp S Cl Z) + (M Cl Z) + (dd ngp C Cl Z) + (h ngp C/Aff Cl Z)	mackerel is recognised as a h , and strips are found that match [dd *] and that follow any strips identified in Step 5.
7				The strips are merged to give the final parse tree. The probabilities associated with the path through the network are used to calculate a tree probability that indicates the most likely parse.

Table 11.1: Simplified vertical strip parse of the sample sentence

11.6 Weerasinghe's Probabilistic On-line Parser (POP)

11.6.1 The algorithm

In the early 1990s Weerasinghe developed a probabilistic chart parser for Systemic Functional Grammar, which is described in Weerasinghe (1994). His parser was developed in PROLOG, and used a model of syntax which had been extracted from GENESYS (e.g. Fawcett 1990) and enhanced by a linguist. The probabilities, which he attached to his 'rules', were extracted from the POW Corpus, using the first version of the Interactive Corpus Query Facility (ICQF) (see Chapter Eight).

A unique method adopted by Weerasinghe was the **on-line** approach that parses each word as it is typed on the keyboard. In this way, he is able to set the parser working as soon as the user presses the space key (or a punctuation character which signifies the end of the word) in the input string.

In order to be able to use it for a Systemic Functional Grammar, Weerasinghe adapts the three ways in which the standard chart-parsing algorithm adds edges to the chart (see Section 10.3.1.5 of Chapter Ten).

First, he uses **Chart initialisation** to create the initial edges in the chart for each item in the input string by determining the elements of structure that a given item can expound, and then creates an edge that represents the unit to which the element belongs. Using this approach, Weerasinghe avoids the need for having edges that represent elements and items (e.g. **h**, "**man**"), and therefore reduces the number of edges in the chart.

Second, using the **fundamental rule**, he joins two edges together based on the fact that the unit represented by the first edge is seeking the element that is represented by the second edge (which has been completed).

Third, he uses the **bottom-up rule** to determine the elements which completed units can fill.

For details of the elements that can expound items and the associated probabilities, Weerasinghe (1994:79) uses what he terms the **multi-source lexicon**. For information about syntactic 'rules', the elements that belong to units, and elements that can follow other elements in a unit, Weerasinghe (1994:79) uses his **multi-source syntax list**. Further information about both of these lists is given in Section 12.1.3.1 of Chapter Twelve).

11.6.2 The probabilistic scoring of edges

As each edge is added to the chart, it is given an associated probability, and the probability assigned depends on how the edge was created. For edges that are created during **chart initialisation**, the probability represents the probability that an item expounds the given element, and this is derived directly from the multiple sources Weerasinghe uses for the creation of his **multi-source lexicon** (see Section 12.1.3.1 of Chapter Twelve). Weerasinghe (1994:105) then scores each such edge by combining three probabilities:

- (a) the item expounding probability of immediately preceding element,
- (b) the element co-occurrence probability for the element, and the element that precedes it,
- (c) the element filling probability for the current edge.

Weerasinghe (1994:105) uses the product of these probabilities. If, for example, he is parsing the nominal group **the mackerel**, his score for the edge that represents the unit will be the product of the probabilities of :

- (a) the element **dd** expounding item **the**,
- (b) that **h** directly follows a **dd** in its unit, and
- (c) the element **h** is expounded by **mackerel**.

An interesting feature of this approach is that he applies a greater weighting to the element co-occurrence rules than he does to the exponence rules - a matter that we have considered carefully when developing our own model.

The simplest and most common approach to the scoring of edges which have been co-joined as the result of the **fundamental rule**, is to take the product of the scores of the two edges. Based on the work of Magerman and Marcus (1991b), Weerasinghe (1994:106), takes the 'weighted geometric mean' of the probabilities of the two edges being co-joined, and the element co-occurrence probability - using Magerman and Marcus argument that this is a better heuristic for estimating the probabilities involved. Weerasinghe gives twice the weight to the element co-occurrence probability, than he does to the probability scores of the two edges - on the basis that he argues that this is the single most important reason why the edges are being joined.

When scoring new edges that have been produced using the **bottom-up rule**, Weerasinghe (1994:106) takes the product of:

- (a) the probability that the preceding element fills the unit below it,
- (b) the probability of the current unit filling the element being considered, and
- (c) the co-occurrence probability of the two elements occurring sequentially in the unit.

11.6.3 Modifications to the standard chart parsing algorithm

In Systemic Functional Grammar syntax, because of the relationships between items, elements, and units, and because of the fact that the parse tree is richly annotated with semantically labelled syntax tokens, it produces many more potential analyses when compared to a Phrase Structure Grammar. Weerasinghe found that the standard chart parsing algorithm had to be modified to improve the accuracy of his results and the efficiency of his parser. These methods either (a) reduced the size of his syntax list, or (b) reduced the number of edges in the chart. To achieve this he used the following methods, which are further explained in Section 12.1.3.3 of Chapter Twelve.

First, he implemented a **top-down filtering** process in which edges are only added to the chart provided that they 'match' the edges that are already in the chart. Using this method, Weerasinghe avoided having to add edges for elements that were not being sought by edges that are already to the left of them and that could not start a new unit.

Second, he implemented a **head-driven approach** based on the work of Kay (1989). This only adds an edge to the chart once its main element has been found. The method is unsuitable for parsing naturally occurring texts for the reasons given in Section 10.4.2 of Chapter Ten.

Third, he marked **optionality** and **mutual exclusivity** into the 'rules' of his syntax list. These specified (a) the optional places in which an element can occur, and (b) if an element has appeared in one place, then it cannot also appear in another. Using this, he managed to considerably reduce the size of the list, and thus he reduced the number of 'rules' that the parser searches through to find acceptable matches, and this in turn reduced the number of edges in the chart.

11.6.4 Evaluation

In his conclusions Weerasinghe (1994:127) reported that he found a significant improvement over the standard chart parsing algorithm by applying the techniques described above. Using his parser for a typical sentence length of 4-9 words, he reported:

- (a) a reduction of approximately 80% in the number of edges in the chart, and
- (b) significantly quicker parse times (with some timings presented in seconds rather than tens of seconds).

Thus Weerasinghe proved that a chart parsing approach could be adapted for a Systemic Functional Grammar that uses probabilities about syntax.

While there is much to learn from Weerasinghe's parser, there is one serious problem with his head-driven approach. This is particularly noticeable when parsing spoken texts. The problem is, as shown by an examination of the FPD Corpus, that a unit does not always contain a head. There is nothing unusual about this phenomenon, which is nevertheless one that is simply not provided for by definition in head-driven parsers. As an example, ICQF+ reports that 665 (or about 3%) nominal groups are without a head and, which is of even greater concern, 3479 (20%) Clauses are without a Main Verb. Specifically, 197 of these contain Operators (O) and Subjects (S), and are filling a 'confirmation-seeker' such as **will they?**, **could I?**, and **wouldn't you?**; 193 are Clauses that are in single- or double-word sentences where ellipsis has occurred; 158 are exclamations like **god!**, **hey!**, **drat!**, and **oh-dear**; 84 are Frames (Fr) like **right!** and **now-then**. Such sentences are not amenable to the head-driven approach, and it follows, therefore that the head-driven approach is not suitable for parsing unrestricted English, and should be abandoned.

11.7 Souter's corpus-trained parser

11.7.1 The algorithm

Souter (1996) created a somewhat different type of probabilistic chart parser for Systemic Functional Grammar from Weerasinghe's POP (see Section 11.6). It was based on Pocock and Atwell's (1993)⁵ parser, and it used a set of syntactic probabilities that he extracted from the POW Corpus and a lexicon supplemented by words derived from the CELEX database (although, in a later version, he replaced the CELEX look-up with pre-parse tagging).⁶

Souter's chart data structure was the one used in the standard chart parsing algorithm (as described in Section 10.3.1.5 of Chapter Ten), as it contained edges with the following fields:

⁵ Pocock and Atwell's parser was derived from that of Gazdar and Mellish (1989).

⁶ The Centre for Lexical Information (CELEX) is based at the University of Nijmegen and have produced a large lexical database for English, German and Dutch.

- (a) a start position,
- (b) an end position,
- (c) a syntax token that the edge represents,
- (d) a list of syntax tokens found,
- (e) a list of syntax tokens that are to be found before the edge can be considered complete.

Additionally his edge data structure contained probabilistic scores.

One of the principal findings of this project was that he discovered severe limitations in this approach, which was due to the sheer quantity of syntactic 'rules' that can occur in a naturally occurring text. This is shown in the FPD Corpus, which contains 8617 distinct 'rules'.⁷ Souter then made a number of attempts at improving the efficiency and accuracy of his parser as described in Section 11.7.3, each of which usefully increases our understanding of how a probability-based parser should operate.

11.7.2 The probabilistic scoring of edges

For **chart initialisation**, Souter followed Pocock and Atwell (1993) by using the **lexical probability** of an item. This is the product of the probability based on the frequency of the item in a corpus, and the probability that it expounds a particular element of structure.

When combining edges using the **fundamental rule**, he applied a probabilistic function, which took the product of two combined edges together. In his final parser, (as will be reported next), he modified the algorithm to include vertical relationships given by trigrams, and he included the probabilistic score of the trigram by multiplying it with the score for the two edges.

11.7.3 Modifications to the standard chart parsing algorithm

Souter's early experimental parsers were unacceptably slow. In order to make it quicker, he had to find ways: (a) to reduce the search space. Like Weerasinghe, he did this in two ways: by reducing the number of 'rules' in his syntax list, and (b) by reducing the number of edges in the chart.

⁷ These included rules for exponents, components and filling.

To amend his syntax list, first he combined similar 'rules' by marking certain elements as optional (for example, the 'rules' in Figure 11.2 were combined (Souter 1996:77)).⁸

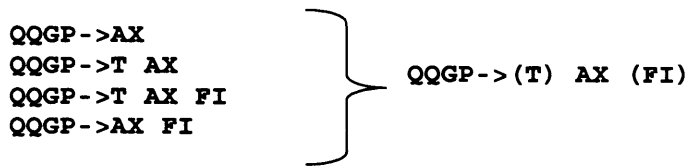


Figure 11.2: Souter's merging of rules by marking optionality

His second amendment (Souter 1996; 81) was to incorporate rules with 'co-ordinating children'. When he extracted his syntactic representation from the POW Corpus, he used a method of representing the 'rules' similar to the one shown in Table 5.1 of Chapter Five. This meant that for the co-ordinated Clauses (C1) shown in Figure 11.3, he had a 'rule' that represented the co-ordination as Z->C1 C1 C1.

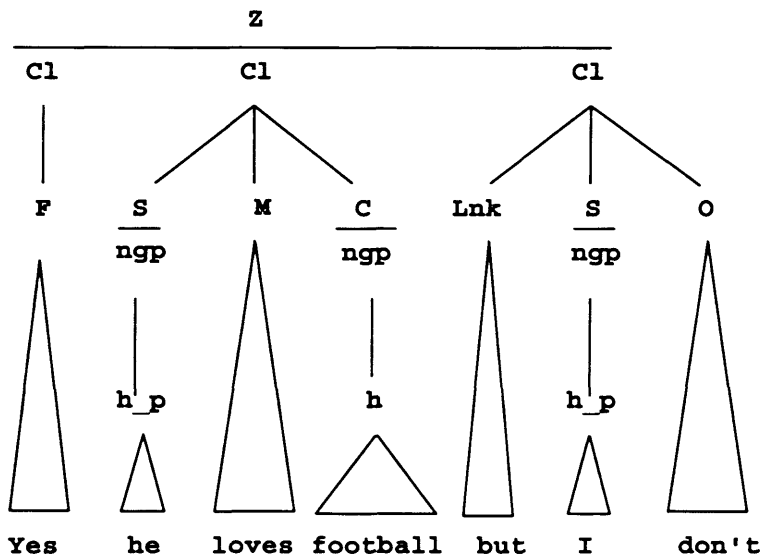


Figure 11.3: An example sentence from the FPD Corpus showing a co-ordinated clause

In the FPD Corpus the largest number of co-ordinated Clauses is ten, and this would be represented by the 'rule': Z->C1 C1 C1 C1 C1 C1 C1 C1 C1 C1. Souter (1996:81) substantially reduced the number of rules in his syntax list by

⁸ As we shall see in Chapter Twelve, Weerasinghe (1994) also had this approach and also marked mutual exclusivity which indicated that, for example, that an operator could appear in more than one position in a clause.

representing all co-ordinated units by one rule $Z \rightarrow C1$, and the probability of the number of co-ordinated units at the start of the rule in the form: $p1, p2, p3, p4, p5, p6, p7, p8, p9, p10, Z \rightarrow C1$, where $p1-p10$ represents the probability of a single unit filling the element through to ten units filling the element. The problem associated with this approach is that, in rare cases, it is not always the same class of unit that fills an element in a co-ordinated relationship, although there are no cases of these in the FPD Corpus.⁹

In attempts to further speed up his parser, Souter implemented further measures which reduce the number of edges in the chart. These include: (a) introducing a **maximum tree depth**, (b) using a **vertical trigram model**, (c) having a **stopping function** which terminated the parse after a 'stopping condition' was met, and (d) by implementing **pre-parse tagging**.

By implementing the **maximum tree depth function**, Souter (1996:111) was able to prevent the addition of edges that represented partial parse trees that were unreasonably deep. To determine depths that were unreasonable, Souter compared the depth of the partial parse tree that is represented by the proposed edge, with partial parse trees in the corpus, which spanned the same number of items.

Souter (1996:111) used a **vertical trigram approach** which introduces a vertical dependency into his algorithm. Although the approach increased the size of his syntax list, he reported considerable improvements using the method. It is applied to his equivalent of the standard chart parser's **bottom-up rule**. When a new edge is proposed, the element that is represented by the proposed edge together with the unit that is represented by the inactive edge (that 'proposes' the new edge) and each of the unit's child elements are collected into **trigrams**. For example, if the proposed edge represented a Subject (**S**), and the proposing edge a nominal group (**ngp**) that contained a deictic determiner (**dd**) and a head (**h**), this will result in two trigrams (**S ngp dd** and **S ngp h**). If any of the vertical trigrams do not appear in the list of trigrams that were extracted from the corpus, Souter rejects the proposed edge, otherwise the edge is accepted and will be assigned a probability which includes the trigram's probability in its creation.

⁹ While it is not stated in Souter (1996), I presume that he counted the maximum number of co-ordinated units for any unit, and if (for a given unit), there were no cases of co-ordination at a particular number, the value of p would be zero.

The third method that was used by Souter (1996:112) to modify the standard algorithm was the implementation of his **stopping conditions**. These were:

- (a) limits set on the number of complete parse trees that could be generated before the parser stops. Souter used a limit of 6.
- (b) the score of the first found solution is recorded and no new edges are allowed with a weight of 1.5 times that of the first solution,
- (c) when no solutions are found, Souter stops the parser when the chart reaches a certain size.

Finally, Souter (1994:125) used the Brill tagger for **pre-parse tagging** (see Section 10.4.4 of Chapter Ten). By training the Brill tagger using the POW Corpus, and modifying the **chart initialisation** by incorporating the **tagger**, he ensured that only one element was assigned to each item. Although this approach relied on the accuracy of the Brill tagger, it effectively reduced the number of edges in the chart.

11.7.4 Evaluation

Souter discovered that these modifications (particularly the trigram approach) significantly improved his parser. Before these modifications his parse times were recorded in days, and he found himself having to stop the parser manually before it had finished. Following the implementation of these methods, most of his results were measured in hours, and this represented a significant improvement.

Souter found that when 'rules' are extracted from an analysed corpus, many occur just once, and a relatively smaller number of 'rules' occur many times. This characteristic of language was identified by Zipf. Zipf's law states that the frequency of any word is inversely proportional to its rank in the frequency chart (Zipf 1935). This claim has been supported in my work (Day 1993a) and Souter's, and was found to apply to the syntactic 'rules' as well as to words.

One of the most significant of the findings of Souter's work is that he demonstrated that when a syntactic representation is extracted from a corpus of naturally occurring texts, the size of the representation severely hampers the performance of the chart parsing algorithm. Like others before him therefore, he had to find ways of reducing the size of his syntactic representation, and the number of edges that are produced in the chart. However, despite the various modifications that he introduced, his parser remained unacceptably slow. It is clear therefore, that 'tweaking' the chart parser approach is not sufficient to solve the problems that face the

researcher who is seeking to build a parser that (a) handles unrestricted natural text, and (b) analyses it in terms of a Systemic Functional Grammar syntax.

11.8 Summary

This Chapter has looked critically at the attempts at parsing Systemic Functional Grammar by Winograd (1972), O'Donoghue (1991a), Weerasinghe (1994), and Souter (1996) and has also reviewed the work of Kasper (1988) and O'Donnell (1994, 2005).

Winograd (1972) was the first to develop a parser for Systemic Functional Grammar. This was one of the components in his ground-breaking work of building a complete natural language understanding and generation system, which involved the movement of blocks in a small virtual world. Winograd's 'rules' however, are embedded into an **interpreted program**. Although it probably would not be impossible to achieve, a 'program' that represents syntax automatically extracted from the texts of a naturally occurring corpus would be extremely complex and difficult to update, as care would be needed to ensure that the complex interrelationships are not broken.

O'Donoghue's Vertical Strip Parser (1991a), which came before Weerasinghe's and Souter's work, used a network of vertical strips extracted from a pseudo-corpus. Like Winograd's programs, O'Donoghue's vertical strip networks would also be difficult to update, but it was nevertheless the most innovative method described here. He showed that an alternative approach to chart parsing may lead to greater success.

Weerasinghe's work (1994) demonstrated that the standard chart parsing algorithm needs to be augmented in order to work successfully with a Systemic Functional Grammar; but his head-driven approach is not in fact suitable for parsing the full range of natural language texts. This is because not all units have a 'head' - especially in a spoken corpus.

Souter (1996) shows that the standard chart parsing algorithm is given a much greater challenge when faced with the abundance of 'rules' that are generated from a real corpus. He too had to develop special enhancements in order to make his parser a viable research tool.

Both Souter and O'Donoghue's work was based on real-life corpora, from which their syntactic representations and associated probabilities were extracted automatically. The implications of this are that both the coverage and the size of the corpus are very significant factors in the degree of success that a parser may attain. In

a small corpus, one will find that some words and rules that one would expect to occur are not there - because they have not been encountered in the small sample. Other words and syntactic co-occurrences may be given undue weighting because they have occurred, whereas they would have had a much lower frequency in a larger corpus (we encountered this phenomenon with the FPD Corpus of children's spoken language). Even with large corpora it remains true that one cannot argue that if a word or a syntax pattern does not appear in the corpus, that word or syntax pattern is ungrammatical. Indeed, this is a shortcoming of many corpus-based approaches to parsing, which fail because they cannot analyse particular structures that do not appear in their corpus. I would nevertheless argue that the corpus approach is far better than an approach based on an 'armchair' grammar (as is often the case with the parsers described in Chapter Ten). The aims of the parser to be introduced in Part Four are to be able (a) to operate with very large corpora, and (b) to provide for known syntax structures that have not appeared (so far) in the corpus. These are the primary reasons for adopting the **dynamic corpus** approach used in this work.

The next chapter looks at the prototype parsers that were developed as part of the process of developing the Corpus-Consulting Probabilistic Parser that will be described in Part Four.

Chapter Twelve

Early attempts at parsing in this project

The purpose of this final chapter of Part Three is to set the scene for Part Four, in which I describe the parser that is the main product of this project: **the Corpus-Consulting Probabilistic Parser (CCPP)**. This chapter describes two early parsers that were developed in the early stages of this project, and compares the first of them with two of the earlier Systemic Functional Grammar approaches that were introduced in Chapter Eleven.

The main concept to be introduced in this chapter is that of a **database-oriented parser**. This is a new type of parser, which uses a **corpus database** to store:

- (a) its **working data**, and
- (b) to gain information to be used in the parse (i.e. a model of the probabilities of the relationships between syntactic categories).

This data is stored in the **parser working tables** and in the **probabilities tables** respectively.¹

Section 12.1 describes and discusses the design of the first of the two parsers (a chart parser), and it then compares it with the two earlier SFG chart parsers that were described in Chapter Eleven: Weerasinghe's (1994) **Probabilistic Online Parser (POP)**, and Souter's (1996) **corpus-trained parser**. The aims of this work were (a) to test the new **database-oriented** approach, and (b) to re-evaluate the suitability of a corpus-based chart parsing algorithm in the parsing of natural language.²

The second parser is described and evaluated in Section 12.2. It is named the **Star Parser**, because of the 'shape' of its main data structure. It was a first attempt to address the problems experienced with the chart parser, and it strongly influenced the final parser that will be described in Part Four.

12.1 Experiments with a chart parser

The main aims of this initial work in developing a chart parser were to answer these two questions:

¹ The full description of the tables is given in Chapter Seven, Chapter Fourteen, Appendix D and Appendix H.

² We shall refer to the first parser as 'the chart parser'.

- (a) Given the advances in computer technology since the work of Souter (1996), is the chart parsing approach now more efficient (in terms of speed), than it was in the early 1990s?
- (b) Is the database-driven, corpus database approach suitable for operation in a parsing environment?

12.1.1 Background

The parser described here, like Weerasinghe (1994) and Souter (1996), uses a **modified chart parsing algorithm** of the type described in Section 10.3.1.5 of Chapter Ten. Both Weerasinghe and Souter discovered that the efficiency of their chart parsing approaches suffered from the over-generation of edges in the chart, and both implemented various methods to attempt to reduce the number of edges. I was faced with this same problem, and I was greatly helped by being able to draw on the work of these two earlier researchers.

Although the intention in this present project was always to implement a parser that follows the human parsing process more closely than that of a chart parser, it was nonetheless thought useful to establish whether a chart parsing approach was now viable, using current computers.³ It also served as a test bed for developing and testing the viability of the database-oriented approach (i.e. the use of the **probabilities tables** and the **parser working tables**).

12.1.2 Implementing a chart parser

This section describes the prototype chart parser, its data structures, and how it uses the probabilities tables. I describe it in some detail, because many of its characteristics are incorporated in the parser to be described in Part Four.

12.1.2.1 The chart data structure

The chart is implemented in a database table where an **edge** is represented by a record that has the following structure:

```
CHART(startPos, endPos, token, rhsTokens, edgeType,  
potentiallyClosed, probability, history)
```

³ The chart parsing work described in this chapter was concluded in 2003, and since Souter's work finished in 1996, the speed of computers has significantly increased.

```

Example edge (record in the database table):
startPos = 2
endPos = 4
token = "ngp"
rhsTokens = "dd h"
edgeType="UNIT"
potentiallyClosed = true
probability = 0.58
history = "<ngp><dd>the</dd><h>point</h></ngp>"

```

Each edge has a start position (**startPos**), and an end position (**endPos**), which respectively represent the start and end of the edge in terms of its ordinal position in the sentence. In the following example, an edge that represents a nominal group that contains **the point** has a **startPos 2** and an **endPos 4**. An edge that represents the whole sentence has **startPos 0** and **endPos 4**.

0 What 1 's 2 the 3 point? 4

The field named **token** gives the syntax token that represents the unit or element on the left-hand side of the 'rule' that produced the edge.

For an edge that represents an element or unit, the field named **rhsTokens** gives the syntax tokens that have been found by the parser so far for this edge. For a **unit edge**, this will be the element(s) of structure that are components of the unit given in the **token** field. For an edge that represents an element, i.e. an **element edge**, this is one or more units that are co-ordinated. An **item edge** will have an item in the **rhsTokens** field.

The field named **edgeType** contains a label that identifies the type of the syntax token that is identified in **token** field, and is simply one of the values **ITEM**, **ELEMENT** or **UNIT**.⁴ An element edge and unit edges have an element of structure, or a unit expressed in the **token** field respectively. Item edges also have an element of structure expressed in the **token** field.

The field named **potentiallyClosed** was a concept introduced in an attempt to reduce the number of edges. It is set to the value of either **true** or **false**. This represents a major difference between the algorithm presented here, and the standard chart parser described in Section 10.3.1.5 of Chapter Ten, and is a concept that is

⁴ It appears that the edges of Weerasinghe's parser always represented units and not elements (Weerasinghe 1994:98). This had the temporary beneficial effect of reducing the number of edges that were generated in the chart, but such an approach could not handle co-ordination as easily as the method used here (as described in Chapter Fifteen).

retained in the final parser. Consider, for example, a **unit edge** such as a nominal group (**ngp**). If the last element in a unit can finish its unit (e.g. a head (**h**) in a **ngp**), this field is set to the value **true**, even though further elements (i.e. one or more qualifiers (**q**)) may follow it. Unit edges that have the **potentiallyClosed** field set to the value **true** remain active and may therefore either be extended by further elements (using the **fundamental rule**), or be used to create a new element edge (using the **bottom-up rule**).

This proved to be an effective way of reducing the number of edges in the chart. Using the dot notation (see Section 10.3.1.5), the standard chart parsing algorithm adds the edges shown below to the chart. The first edge is **inactive** (since it is complete), and the parser will create a number of edges that represent the elements that a nominal group can fill. The other rules remain active (as they are still seeking elements).

```
ngp->dd h .           (a nominal group that is complete)
ngp->dd h . q         (a nominal group that is seeking a qualifier)
ngp->dd h . q q       (a nominal group that is seeking two qualifiers)
```

Typically, the standard chart parsing algorithm would have a **toFind** field for each edge. For example, the value of the field for the second edge in the sequence above would contain the value **q**. In the approach used by the chart parser reported here, however, this field is not needed. Instead, the fact that the first edge is labelled as being 'potentially closed', means that the unit, and so the edge, can be extended - but only of course, if a qualifier is found. The information about which elements can follow the last element is given by the **Forward Unit Structure (FUS)** query (which will be described in Section 12.1.2.3). In the example above, this method reduces the number of edges in the chart that represent the nominal group from three to one. When this technique was applied to the 'rules' extracted from the POW Corpus, the reduction in the number of edges is very substantial. The reason being, that many units are lacking their head elements (as described in Section 11.6.4 of Chapter Eleven, and Section 10.4.2 of Chapter Ten). For a nominal group that has only found a deictic determiner, the number of edges is reduced by 3097. This is because there are 3097 nominal groups in the corpus that start with a deictic determiner. There may, however, be other elements (i.e. other determiners, modifiers, a head, and qualifiers), but these are considered 'optional', because a deictic determiner may be the only element in the unit, and hence 'potentially closes' it.

We turn now to the field named **Probability**. This contains the **probabilistic score** associated with the edge, as returned by the **edge scoring function**. The method used to calculate the score depends on the type of edge, and is shown through the use of examples in Table 12.1.

Edge type	Description
Item	e.g. h -> saw . The probability that saw expounds a head (h).
Element	e.g. C -> ngp . The probability that a Complement (C) is filled by a nominal group (ngp)
Unit	e.g. C1 -> S O M C . The probability that a Complement (C) follows a Subject (S), Operator (O) and Main Verb (M) in a Clause.

Table 12.1: Calculation of an edge's probability

The final field is named **history**. This contains XML data that represents the history of the parse in terms of the items, elements and units from the edges that were involved in the edge's construction. The format of the XML is identical to that used to annotate the corpus itself, and is of the form given in Section 6.1.5.2 of Chapter Six. An edge representing a complete parse will contain an XML structure that represents the syntax analysis of the whole sentence.

12.1.2.2 The chart parsing algorithm

The algorithm used by the chart parser described here did not vary greatly from the standard chart parsing algorithm which is presented in Section 10.3.1.5 of Chapter Ten. The main differences were changes introduced to handle:

- (a) potentially closed edges,
- (b) interfaces with the corpus database for the syntactic relationships.

The first difference involved the way completed edges are handled. As described above, A **potentially closed edge** is one for which an element that has the potential for being the final element of the unit has been found. On encountering such an edge, the parser considers it to be a completed edge, and generates additional higher edges using the **bottom-up rule**. The edge, however, remains **active** and can have extra elements that may be found later extending its **rhsTokens** field using the **fundamental rule**. When a potentially closed edge is extended in this way, the parse histories and end positions of all edges that are descendants of the potentially closed edge have to be updated. The second difference involves the interface with the corpus database, and will be discussed in the next section.

12.1.2.3 Corpus queries

A significant difference between, on the one hand, the approach of the chart parser developed for this project, and on the other: (a) the standard chart parsing algorithm, and (b) those of Weerasinghe, Souter and others, was the implementation of different types of **corpus query** used in construction of edges. These methods of building structures were to develop later with some improvements, into the **probabilities tables** that are used in the Corpus-Consulting Probabilistic Parser (these being described in Chapter Fourteen). The probabilities tables used in the chart parser are:

- (a) Item up to element (**I2E**),
- (b) Forward Unit Structure (**FUS**),
- (c) Unit up to Element (**U2E**).

I2E queries are designed to use the **I2E probabilities table**, and are used in the **chart initialisation** to create **item edges** that represent items and the elements that they expound. For example, an **I2E** query for the item **saw** will return two records, one giving the probability that the item expounds the head (**h**), as in **he used the saw**, and the other the probability that it expounds the Main Verb (**M**) of a Clause, as in **he saw him**.

FUS queries are designed to use the **FUS** table, and are used to extend unit edges (those that have the **type** value of **UNIT**), when a new element edge is the subject of the **fundamental rule**. It identifies the fact that the element that is the subject of a given edge is a legal 'next element' in a unit (such that it has the elements expressed in the edge's **rhsTokens** field), and in so doing, the **FUS** query identifies that the two edges can be combined. The **FUS** query also provides the probability that the element is the next element in the unit that is represented by the edge. The structure of a row in the **FUS** table is given in Table 12.2, Table 12.3 shows example **FUS** entries.

Field	Description
UNIT	the unit that contains the elements.
ELEMENTS	the elements that have been found so far.
NEXTELEMENT	the element that can occur after the elements in the ELEMENTS field.
PROBABILITY	the probability that the element given in the NEXTLEMENT field follows the elements given in the ELEMENTS field, considering all elements that can follow them.

Table 12.2: The structure of the **FUS** probabilities table

UNIT	ELEMENTS	NEXT ELEMENT	PROBAB- ILITY	Description
ngp	! dd	h	0.7882	Approx. 78% chance that a head (h) follows the deictic determiner (dd).
ngp	! dd	mo	0.0586	Approx. 6% chance that a modifier (mo) follows the deictic determiner (dd).
ngp	! dd	h_rcc	0.0299	Approx. 3% chance that a head that is recoverable from cultural classification (h_rcc) follows the deictic determiner (dd).
ngp	! dd	rel_mo	0.0289	Approx. 3% chance that a relative modifier (rel_mo) follows the deictic determiner (dd).
ngp	! dd	!	0.0289	Approx 3% chance that the deictic determiner finishes the nominal group.
ngp	! dd	th_mo	0.0244	Approx 2.5% chance that a thing modifier (th_mo) follows the deictic determiner (dd).

Table 12.3: An excerpt from the FUS table showing the elements that may follow in a nominal group that starts with a deictic determiner (note that the end of the unit, and the start of a unit are indicated by !)

The U2E queries operate on the U2E probabilities table. They are used as soon as a **unit edge** becomes potentially closed. The parser asks for a list of elements that the unit represented by the potentially closed edge can fill, and for each, creates a new complete **element edge** that represents the element filled. This in turn causes the parser to 'look for' all edges that are 'looking for' the new element as being next within their unit. Figure 12.1 shows a chart for a partial parse, and indicates how the different types of query were used to create its edges.

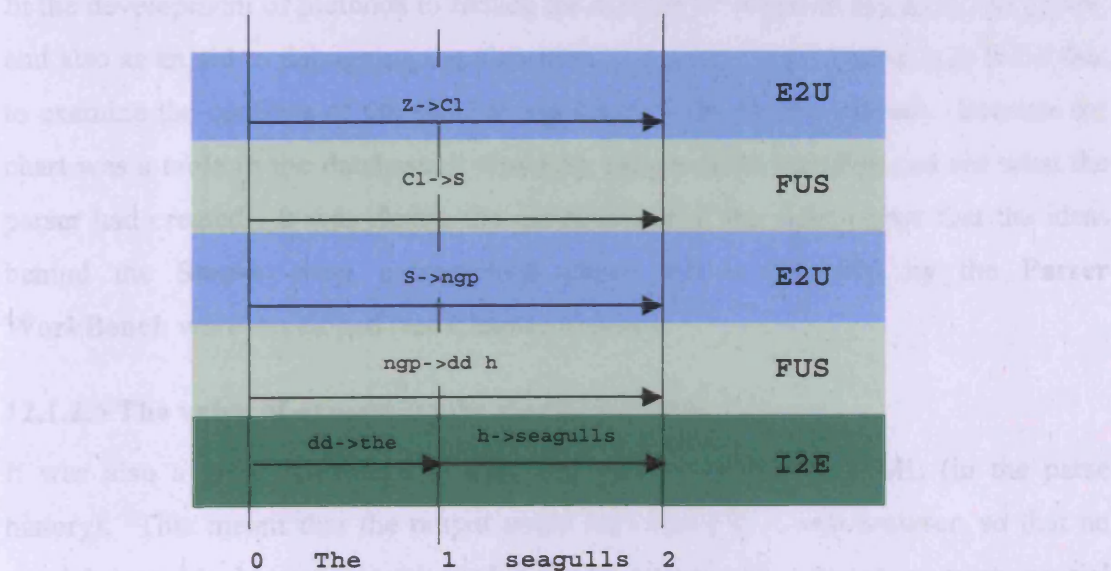


Figure 12.1: The chart after a partial parse (showing queries used to establish edges)

12.1.2.4 The probabilistic scoring of edges

When edges are added to the chart, they are assigned a probability. In the parser described here, the **item edges** that are assigned in the chart initialisation are given a probability value that is equal to the value returned from the **I2E** query that created the edge. Element edges are the result of identifying a **potentially closed edge**, and are assigned the probability that is the product of the probability that is returned from the **U2E** query and the probability assigned to the potentially closed edge.⁵ Edges that are extended by joining them with other edges, are assigned the probability that is the product of three factors:

- (a) the probability that the given element follows the structure (given in the **rhsTokens** field, and as specified by the **FUS** query),
- (b) the probability of the edge that caused the creation of the new edge, and
- (c) the probability that is already assigned to the edge that is being extended.

Notice then, the strong role played by various types of probabilities in this parser - concepts that will be found in the final parser, though in a different form, since it is not a chart parser.

12.1.2.5 The value of the database-oriented approach

In the development of methods to reduce the number of edges produced by the parser, and also as an aid to debugging the algorithm, it proved to be an advantage to be able to examine the contents of the chart at any stage of the parsing process. Because the chart was a table in the database, it was very easy to open the table and see what the parser had created. It was during the development of the chart parser that the ideas behind the **Step-by-Step incremental parse** that is provided by the **Parser WorkBench** were developed (see Chapter Sixteen).

12.1.2.6 The value of expressing the results in XML

It was also a great advantage to have the parse expressed in XML (in the parse history). This meant that the output could be viewed in a web browser, so that no special tree graphing program needed to be developed. An even more crucial advantage was that, since the output from the parser was expressed in the same form as the sentences stored in the corpus tables, it would be a simple matter to update the

corpus tables with the sentences that are parsed, thus contributing to the achievement of the goal of implementing a **dynamic corpus** (see Section 1.1 of Chapter One).

12.1.3 A comparison with the work of Weerasinghe (1994) and Souter (1996)

In this section, the chart parser implemented here is compared to the approaches used by Weerasinghe (1994) and Souter (1996), which were described in Chapter Eleven.

12.1.3.1 The model of syntax and the lexicon

One of the main differences between the work described here and that of Weerasinghe (1994), is in the way that the model of syntax and its associated probabilistic data are stored and used. The chart parser described here used the **probabilities tables** and queries described in Section 12.1.2.3. In contrast, Weerasinghe used what he termed the **multi-source lexicon**, and **multi-source syntax** lists for his lexicon and syntax lists (Weerasinghe 1994:81). His lexicon was extracted from the POW Corpus (following some mappings in the syntax labels to a newer version of the Cardiff Grammar). The syntax lists were derived from four sources:

- (a) the POW Corpus using an early version of the **Interactive Corpus Query Facility (ICQF)** (as described in Chapter Eight),
- (b) the **ARK Corpus**, which was automatically generated by COMMUNAL's sentence generator **GENESYS** (see Section 2.5 of Chapter Two),
- (c) a crucial book by West (1965), which was an invaluable source of probabilistic data that was available before computer corpora existed, and
- (d) the suggestions of experienced linguists associated with the project (namely, Fawcett and Tucker).

Using this approach, Weerasinghe had the luxury of being able to relegate (or even ignore) any rarely occurring syntax rules. He did this by adjusting the probabilities, or by simply not including them in his syntax lists. Therefore, statistics that were due to any peculiarities attributed to the nature of the corpus, or to mistakes in the original analysis, could also be adjusted or discarded.

⁵ Note that the probability of the new edge has to be recalculated if the edge(s) that created it are extended.

Although in Souter's earlier parser, his lexicon was supplemented by using the CELEX database (Souter 1996:88), his approach used a syntax model that was extracted only from the POW Corpus.⁶

12.1.3.2 The chart data structure

I now turn to the question: how does the chart structure used here compare with those of Weerasinghe (1994) and Souter (1996)? The three approaches are all different to the standard 5-tuple method used in the standard chart parser algorithm (described in Section 10.1.3.5 of Chapter Ten). This is essentially due to two reasons: (a) the ability to include a probabilistic weight to the edge, and (b) to implement methods of reducing the number of edges in the chart.

The data structure presented in Section 12.1.2.1 is very similar to the one used by Weerasinghe (1994:98), the biggest difference being that (a) Weerasinghe's edges represent only units, and (b) that he does not construct edges that represent elements or items.⁷ His approach reduces both the number of edges in the chart, and also the number of 'rules' in the syntax list. For a high proportion of most texts, this method is well-founded, in that element edges (as described above), represent filling and except when co-ordination occurs, these edges can be considered redundant. This means, however, that Weerasinghe would need to implement non-standard approaches for handling co-ordination. In addition, Weerasinghe has additional fields for semantic and syntactic features, which are not used in the chart parser described here, or in the final parser described in Part Four. Finally, he also has applied a similar construct to identify edges that are potentially closed, and hence also did not require a **toFind** field in his edge data structure.

Souter's chart data structure was closer to the standard chart parser structure. He, however, adds two extra fields, the first, **weight** contains his probabilistic score for the edge, and the second, **depth** which provides an additional weighting factor for his vertical trigrams method (see Section 12.1.3.3 and Section 11.7.3 of Chapter Eleven).

⁶ Souter's final parser used pre-parse tagging using the Brill Tagger which had been trained on the POW Corpus (see Section 11.7.3 of Chapter Eleven).

⁷ Weerasinghe uses the names **Begin**, **End**, **Elemseq**, **Unit**, **Parse**, and **Prob** as his equivalents to **startPos**, **endPos**, **rhsTokens**, **token**, **history** and **probability** respectively.

12.1.3.3 Methods used to improve the efficiency of the parser

The three parsers compared here all suffered from problems of efficiency. These were due to the following reasons: (a) the large number of spurious edges generated in the chart, and (b) the large number of syntax rules in the syntax list (or database table). Each of the three researchers adopted methods to attempt to reduce the number of edges and 'rules'.

Although Weerasinghe's parse times were measured in seconds, he also felt the need to implement measures to improve the speed of his parser. The most prominent of these methods was his adoption of the **head-driven approach** - an approach associated with quasi-Chomskyan models of syntax (as discussed in Section 2.2.1 of Chapter Two). Here, however, we take the view that this approach is not adequate to handle the full complexity of natural language texts, especially when we take account of the need to be able to parse spoken texts (for the reasons given in Section 10.4.2 of Chapter Ten). He also used **top-down filtering** in which the parser only applies edges if they are syntactically acceptable, depending on the edges already in the chart, and thus introduces 'context-sensitivity' to the process of adding of edges.

His third method of improving the speed of his parser was to reduce the number of entries in his multi-source syntax list. He did this through the description of **optionality** and **mutual exclusivity** in his 'rules'. An example of this is given in the 'rule' below. Round brackets indicate optionality, and state that the element may or may not occur. Angle brackets indicate that, if the element occurred in a previous angle-bracketed position, it cannot also occur in the second. In this way, Weerasinghe was able to reduce the number of rules in his list.⁸ Souter (1996:77) adopted a similar approach to the 'optionality' of elements, and so also, he too reduced the number of entries in his syntax list by this means.

C1->& B (A) <O> S <O> M C (A)

The methods for avoiding the over generating of edges used here was fundamentally different from those used by Souter and Weerasinghe, in the following two ways.

First, the records stored in the Forward Unit Structure (**FUS**) table have a different form to the equivalent 'rules' used in Weerasinghe's and Souter's parsers.

⁸ Weerasinghe's notation is equivalent to the following mark up element declaration in the following syntax from a mark up DTD extract: <!ELEMENT C1 - - ((Lnk, B, A? O, S | Lnk, B, A?, S, O), M, C, A?)> (see Chapter Six and Appendix C)

Weerasinghe and Souter essentially used the same type of rules as those described in Section 5.2.1 of Chapter Five. Each of their 'rules' has a defined left-hand side (consisting of a unit or an element) and a collection of tokens on the right-hand side (as in Weerasinghe's rule for the clause above). The format of the **FUS** table (see Table 12.2), as we have seen, is rather different. It consists of a unit and a string of syntax tokens (given in the field **ELEMENTS**) that matched with the tokens that have been found so far for a given edge. The field **NEXTTOKEN** then gives a single syntax token that may occur next (i.e. after the syntax tokens in the field **ELEMENTS**), irrespective of the elements that follow it. Coupled with the **potentially closed edge** method, this provides quite an effective way of reducing the number of edges in the chart. However, it increases the number of entries in the database equivalent of the syntax list which represent units by 45% compared with the number of syntax rules in the FPD Corpus.⁹

This increase does not affect the performance of the parser because of the database indexes, and the more efficient storage and retrieval presented by the database approach. The **database-oriented approach**, then, is the second difference between the approaches of Weerasinghe and Souter and my own. The number of entries in the syntax list, although still an issue, it is considerably reduced. The reason is that they are stored more efficiently, so that access times are more acceptable than they would be if they were stored using the methods of Weerasinghe and Souter. The number of such 'rules' was indeed recognised as an issue by Souter (1996:120). Because his list was derived from a corpus, it was significantly larger than Weerasinghe's, and he reduced his access times to them by applying an 'indexing program' (we shall not discuss this here; see Souter 1996:95).

Souter (1996:111) also created a list of partial vertical strips in the form of **vertical trigrams**. This list contained all three-level partial vertical strips that had occurred in the corpus, and when a new edge was proposed, he checked its conformance against the list before he added it.

Souter (1996:72) also experimented with reducing the number of syntax 'rules' by merging the rules for filling with those of competence, such that a Subject (**S**) filled by a nominal group (**ngp**) that has a deictic determiner and a head (**h**) becomes

⁹ The number of rules of the type used by Weerasinghe and Souter in the FPD Corpus is 3498, this is compared with 6273 entries in the **FUS** table.

S_{ngp}->**dd h** (as reported in Section 5.2.1.2 of Chapter Five). By doing this he reduced the number of rules by nearly a half. However, he lost the difference between primary and relative clauses - as he recognised. He also adopted a way of collapsing co-ordination by implementing a method of expressing the probability of the number of co-ordinated units for a given rule (see Section 11.7.3 of Chapter Eleven).

12.1.3.4 Probabilistic scoring

Weerasinghe's syntax 'rules' were stored separately from their probabilistic scores. This is in contrast with the method used in the **FUS**, which gives the probability that the next element **E** will occur, given that elements **E₁, ..., E_n** have already occurred in a given unit **U**, such that the probability of **E** is given by the frequency of **E** divided by the total frequency of units that have the same structure. On the other hand, Weerasinghe (1994:80) used **element co-occurrence** rules. These give the probability of element **E2** following element **E1** in unit **U**, irrespective of the elements that have already occurred. The fact that the new chart parser described here does take account of **all** the preceding elements in the unit, gives the present parser a major advantage over that of Weerasinghe.

For similar reasons, the new chart parser is an advance on Souter's method (1996:78) which simply uses the frequency of a given 'rule' divided by the total frequency of 'rules' that have the same left-hand side syntax token.

Weerasinghe's approach (1996:105) for the scoring of edges is fully described in Section 11.6.2 of Chapter Eleven. He uses different methods depending on how the edge was created. He makes use of element co-occurrence (as described above) when calculating the score to assign to the new edge. In particular, he doubles the weight of the element co-occurrence score when he uses the **fundamental rule**, and like Magerman and Marcus (1991b), he uses a geometric mean, instead of the more commonly used product to score the new edges. Element co-occurrence also features in scoring new edges in Weerasinghe's **bottom-up rule**, and in his **chart initialisation**. His application of element co-occurrence in chart initialisation is unique to his parser.

In contrast with Weerasinghe's in some respects unique approach, Souter (1996:101), in his final parser, and the chart parser described here, both use the product of the edges that combined to form a new edge, which is commonly employed by probabilistic chart parsers. Souter's final parser also applied the product of the

edge's score with a weight for his vertical trigrams. Souter's approach is fully described in Section 11.7.2 of Chapter Eleven.

12.1.3.5 Comparing the programming methods

The programming method used here differs from Weerasinghe's and Souter's approaches. The major differences being:

- (a) the use of a procedural language (first C++ and Java, and then Visual BASIC), instead of a declarative one (Weerasinghe and Souter both used PROLOG),
- (b) the use of a database to store the parser's working data (the chart).

In early versions of the chart parser described here, I first used the C programming language, and this was followed by a Java version, but both of these were abandoned in favour of a Visual BASIC (VB) version. This was due to the availability of a more straightforward interface to a relational database (for integration with the Corpus Database) which VB provided. Additionally, VB was used for ICQF+ (see Chapter Eight), and it offered a rapid prototyping environment, which included the user interface development.

The advantages of using a procedural language were both (a) it was easier to integrate with the database for the queries and the storing of the parser's working data, and (b) I was able to have full control over the search strategies and computational methods employed.

12.1.4 Evaluating the chart parser reported here

Any chart parsing approach that uses rules extracted from the naturally occurring texts of a corpus is bound to face the major problems that Souter and Weerasinghe experienced before me (i.e. that of the over-generation of edges in the chart). The prototype parser described here, although it uses a different approach, it suffers in the same way. The FUS table contained over 6,200 syntax 'rules' (in normalised form).¹⁰ Like Souter and Weerasinghe, therefore, I found myself continuously inventing and implementing new methods in order to try to reduce the number of edges. The most successful of these were (a) the implementation of the **potentially closed edges** (as

¹⁰ For example, a rule of the form $C1 \rightarrow S \ O \ M$ followed by C is counted once irrespective of what can follow the C .

described in Section 12.1.2.1) and the implementation of an **n-best** approach (see Section 10.2.2.2 of Chapter Ten).

The **n-best** approach involves only constructing **n** edges for the most likely analyses, where **n** is a specific value (e.g. 5 or 10) that is set before the parse. I found that this cuts out some unlikely edges, and so, made the parse quicker. However, as there was no mechanism for backtracking, and the correct parse was often missed when it was less frequent and therefore not within the **n-best**, and the parser was therefore only able to handle simple sentences.

In its pre-**n-best** form, I found that a great number of edges existed that could not form part of the final analysis, and that in many parses, thousands of spurious edges were therefore produced. I also discovered that the number of edges significantly increased when the lengths of sentences parsed increased. On a 2.4GHz Pentium 4 machine with 512Mb memory, short sentences (of three or four words) were taking minutes to parse, and sentences of five to eight words took considerably longer. Performance was approximately proportional to the square of the sentence length. This was clearly far from satisfactory.

My non **n-best** approach produced a parser that was not as quick as Weerasinghe's, but and this can be attributed to the greater number of 'rules' in the **FUS** table (see Section 12.1.2.3) when compared to Weerasinghe's **multi-source syntax list** (see Section 12.1.3.1). The parse times were, however, quicker than those recorded by Souter, but this was probably mainly due to changes in speed of computer hardware since Souter's conclusion in 1996.

The main conclusions from the work of building a chart parser were

- (a) that an alternative approach was needed, and
- (b) that a non-deterministic chart based method was far from ideal for parsing SFG syntax when dealing with sentences such as those found in large corpora of naturally occurring texts, and
- (c) that no methods of reducing the number of spurious edges proposed so far are sufficient to increase the efficiency of the parser significantly.

Work on the prototype chart parser described here was very useful, in that it satisfied its aims (as stated at the start of the chapter). First, I have shown that a database-oriented approach as proposed in Chapter Seven, can indeed operate successfully in a parsing environment. Second, although technology has advanced

since the work of Souter, and the speed of the chart parsing algorithm is now more or less acceptable for short sentences, it is clearly worth exploring an alternative approach, which may yield more efficient results.

It is to the exploration of this alternative approach that we now turn. The next section describes the start of this work, i.e. the development of a second prototype parser which significantly extended the use of the probabilities tables. This is the Star Parser.

12.2 Beyond a chart parser

After implementing and evaluating the chart parser described above - and so concluding that a better method is required. I started experimenting with a number of different ideas, some of which would ultimately lead to an implementation in the Corpus-Consulting Probabilistic Parser (CCPP). The main new idea was that of the data structures that gives the Star Parser its name, and it is this which is described in this section.

12.2.1 The Star Parser

The Star Parser is so called because of the shape of its data structure. While it did not lead to the implementation of a fully functional system, some of the ideas behind it developed into the concepts used in the CCPP and it therefore deserves a brief description here.

12.2.1.1 The Star data structure

The Star data structure has the form of a four-pointed star, as shown in Figure 12.2 and Figure 12.3. At the core of the star is a **unit**. Each Star has a **parse history**, **left predictions**, **right predictions**, and **above predictions**.

The **parse history**, which like the rest of the model, is expressed in XML, contains the syntactic structures that were used to form the star, including the analysis of the item directly below it up to that point. The **left predictions** are an ordered string of elements of structure that can appear before the left-most element (e.g. a **Lnk**) that is a child of the unit at the core of the star (e.g. a **C1**). The **right predictions** are the elements that may follow the right-most element that is a child of the unit at the core of the star (e.g. a **C** in a **C1**). The above predictions are the elements of structure that the unit can fill.

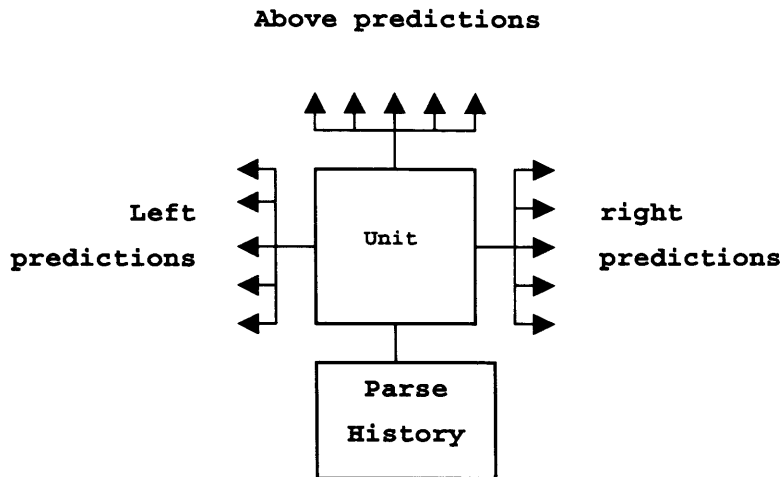


Figure 12.2: The Star data structure

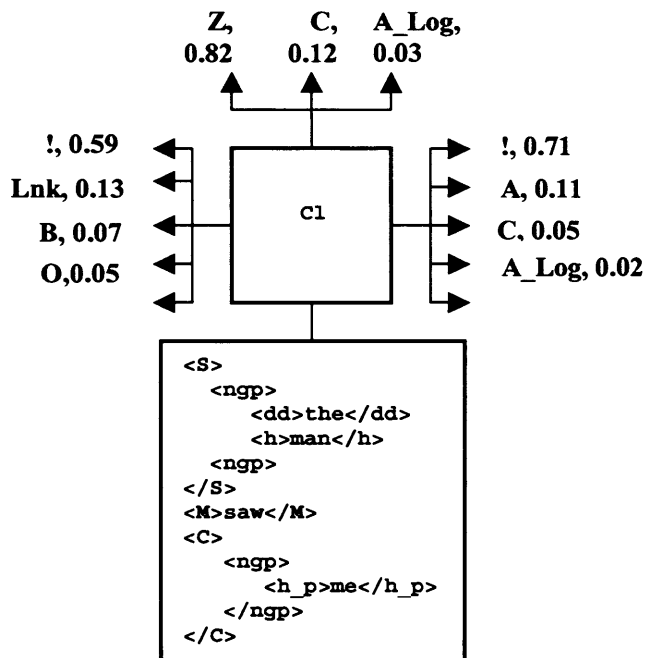


Figure 12.3: A Star data structure representing a Clause (Cl)

12.2.1.2 The Star parsing algorithm

The Star Parser starts by taking the first item from the input string and requesting an **item-up-to-element** query (**I2E**). A Star is created and initialised for each element that the item can expand. An **element-up-to-unit** query (**E2U**) provides the unit that will be stored at the core of the star.¹¹ A query about the **forward unit structure** (**FUS**) provides the appropriate right predictions. A novel feature was the introduction

¹¹ Normally this returns one unit. Few elements (for example, linker (**Lnk**)) can be in more than one unit. In which case, a separate star is created for each.

of a second type of unit structure query, in order to give the backward predictions as to the elements that might precede the element under focus. This was the **backward unit structure (BUS)** query, which operated on the **BUS probabilities table**, and has the structure given in Table 12.4. A 'unit-up-to-element' (**U2E**) query provides the above element predictions.

Field	Description
UNIT	The unit in which the element exists.
THISELEM	The element that is the subject of the query.
PREVELEM	The element that can precede the element that is the subject of the query.
PROBABILITY	The probability that the element in PREVELEM can precede the element given in THISELEM .

Table 12.4: The structure of the BUS probabilities table

The next stage of the Star parse process involves growing the Star up to the sentence node **Z**, so creating a set of new stars as it grows. References to the Stars that were involved in creating the new Star are maintained. The process of growing involves performing a series of **element-up-to-unit (E2U)** and **unit-up-to-element (U2E)** queries until the **above predictions** contain a sentence element. As the parse progresses, the parse history changes, and this is recorded in the parse history field.

Next, the parser moves to the next item in the input sentence and creates a new Star for it. It then performs a join operation for all stars that are to the left of the new Star, that predict that the left-most element in the new star follows the right-most element of the left Star. After the join, the Star's probability is calculated as the product of the predictions.

When a Star cannot be joined it is grown so that it contains a new element and unit before a join attempt is made on the new Stars.

12.2.1.3 Evaluating the Star Parser

The Star parser was a prototype. While working on it, we realised its limitations. It was then abandoned to allow work on the CCPP to start. The problems with the Star approach were similar to those experienced by the chart parser - it was slow and inefficient with a great number of stars being created for analyses that could not exist.

However, work on the Star Parser provided useful material to take forward into the Corpus-Consulting Probabilistic Parser. It gave the basic methods for joining data structures (based on forward and backward predictions) and above all, it provided the

early versions of the **probabilities tables** that we use for parsing, as described in Chapter Fourteen.

12.3 Summary: Towards the Corpus-Consulting Probabilistic Parser

Section 12.1 discussed the development of a chart parser. It is different from that of other chart parser algorithms in that it is **database-oriented**, i.e. it (a) stores its **working data** in the tables of the database, and (b) receives its information about the model of syntax and associated probabilities from other tables in the database (the **probabilities tables**). The database-oriented approach worked very well, particularly in the development of the parser algorithm, and the work also showed that XML is a good means of annotation, not only for the annotation of the Corpus (see Chapters Six and Seven) but also for representing the output of the parser.

Due to the fact that the syntactic relationships used were derived from the naturally occurring texts of a corpus, the parse times were slower than those reported by Weerasinghe (1994). Although improvements on Souter's (1996) parse times were made, they were considered to be mainly due to improvements in computer technology. I concluded that the chart parsing algorithm, even with modifications to reduce spurious edges, does not provide the best model for the requirements of this project. This conclusion was only reached after making many different attempts to reduce the number of edges in the chart, and so overcome the problem. The most promising of these new concepts were (a) the adoption of the concept of a **potentially closed edge**, and (b) the use of the **n-best** approach. However, the n-best approach inherently demands the support of backtracking to pick the next n-best edges to take forward - a solution that goes somewhat against the nature of a chart parser. Unlike Souter and Weerasinghe before me, I did not attempt to reduce the number of entries in the syntax list, and if we had not by that point started working on the CCPP parser (to be described in Part Four), this might have been the next logical step to take, in attempting to turn the prototype into a working model that would return acceptable results.

Section 12.2 has documented the first steps towards a new parsing algorithm by describing the Star Parser. This algorithm, which demanded the implementation of further probabilities tables, was really the birth of the Corpus-Consulting Probabilistic Parser (CCPP), which is the main output from this research.

This now concludes Part Three. The scene is now set for the chapters of Part Four, in which the new parser is introduced, discussed in detail, tested and evaluated.

PART FOUR

The New Parser

This part of the thesis describes the new parser - the Corpus Consulting Probabilistic Parser (CCPP).

Chapter Thirteen introduces the parser, and gives the background to its development.

Chapter Fourteen describes:

- (a) the model of syntax and the probabilistic data that the parser uses, and how it is organised in the **probabilities tables**,
- (b) the parser data structures and how they are organised in the **parser working tables**, and
- (c) the **functions** (and hence, the operations) that are used with the data structures.

Chapter Fifteen gives full details of the parsing algorithm, and shows how it uses the probabilities tables and the parser working tables.

Chapter Sixteen describes the parser's operating environment: the **parser workbench**, and describes how it was used in the development of the parser and in so doing it illustrates the parser at work on a simple sentence.

Chapter Seventeen describes the ways in which the parser was tested, and evaluates it against the other models presented in Chapter Twelve.

Chapter Thirteen

Introducing the Corpus-Consulting Probabilistic Parser

This chapter provides an introductory overview of the **Corpus-Consulting Probabilistic Parser**, and of the way it interacts with its complementary components within the overall system.

As I pointed out in Section 1.3 of Chapter One, some of the development of the parser has been the subject of joint research that reflects equally both: (a) the essential linguistic concepts that are needed, for which Professor Robin Fawcett has primary responsibility, and the computational concepts that complement these, for which I have primary responsibility. While the work on the corpus database (presented in Part Two) was overwhelmingly my own, many of the concepts introduced here are a genuine blend of the two sources of insight, and in many cases it would be invidious to identify one or the other of the two members of the team as the person responsible, since the solution finally adopted was the result of extended discussion and experimentation.

As we saw in Section 10.1 of Chapter Ten, the goal of this parser is to transform a string of items (roughly 'words') into a **richly annotated tree diagram**, and Section 13.1 expands on the implications of this aim in the context of this project.

In Chapter Twelve, we noted the lessons learned from the earlier SFG parsers developed both by Weerasinghe (1994) and Souter (1996), and within the present project. It described these early attempts, and the problems associated with them when they were used in a **corpus-based** chart parsing approach with SFG syntax. The conclusion of that work was that an alternative approach to parsing complex texts in SFG terms should be sought. The approach proposed here is introduced in Section 13.2 and outlined in Section 13.6.

The work reported in Chapter Twelve also confirmed that it is possible to integrate a **corpus-consulting database-oriented** approach with the parsing process. Here, the parser consults a database that contains knowledge of relationships between syntactic categories, together with their probabilities. The **probabilities tables** are also used to store the **working data** of the parser, and this has been particularly useful in the development of the parsing algorithm. It gives us the ability to stop the parse at

any point and so examine the structures that it has created. It is this work that led to the development of the environment in which the parser described in this part of the thesis operates: the **Parser WorkBench** (see Chapter Sixteen).

13.1 Aims

As stated in Section 10.1 of Chapter Ten, the major aim of any natural language parser is to receive an **input string** containing **items** (words), and to convert this string into a syntax tree diagram. The aim of this parser is no different, except that since the output is expressed in SFG syntax, the tree diagrams are more richly annotated than those in a typical PSG-based parser (such as those in the Penn Treebank).

Figure 13.1 shows the Corpus-Consulting Probabilistic Parser in the context of this project. It shows the input to the parser, and how the parser interacts with the corpus database for two purposes:

- (a) to retrieve information from its knowledge of syntactic probabilities through the use of **corpus queries**, and
- (b) to store its **working data**.

The richly annotated syntax tree, which is in XML format, uses the same method of annotation as that used for the corpus itself in the **corpus database**. The output is one or more syntactic analyses of the structure of the sentence. This may be in the XML format shown on the left of Figure 13.1 (see also Section 6.1.5.2 of Chapter Six), or (after the application of formatting) in the standard representation used in linguistics, as in the analysis shown on the right in Figure 13.1.

The primary aim of creating a parse tree in XML format is to be able to add any new successfully parsed sentences to the **corpus database**, and so to update the **probabilities tables**. In doing this, the probabilities associated with the **units**, **elements** and **items** are changed, and this has the effect of changing the parser's knowledge of syntax. It is in this way that the corpus is said to be **dynamic**, i.e. it is changing in principle, with each new sentence that is parsed. The provision for this in the present model makes it possible to satisfy the goal stated in Section 1.1 of Chapter One, i.e. that the parser will model the fact that language changes with time, and so it is, in this sense, **dynamic**.

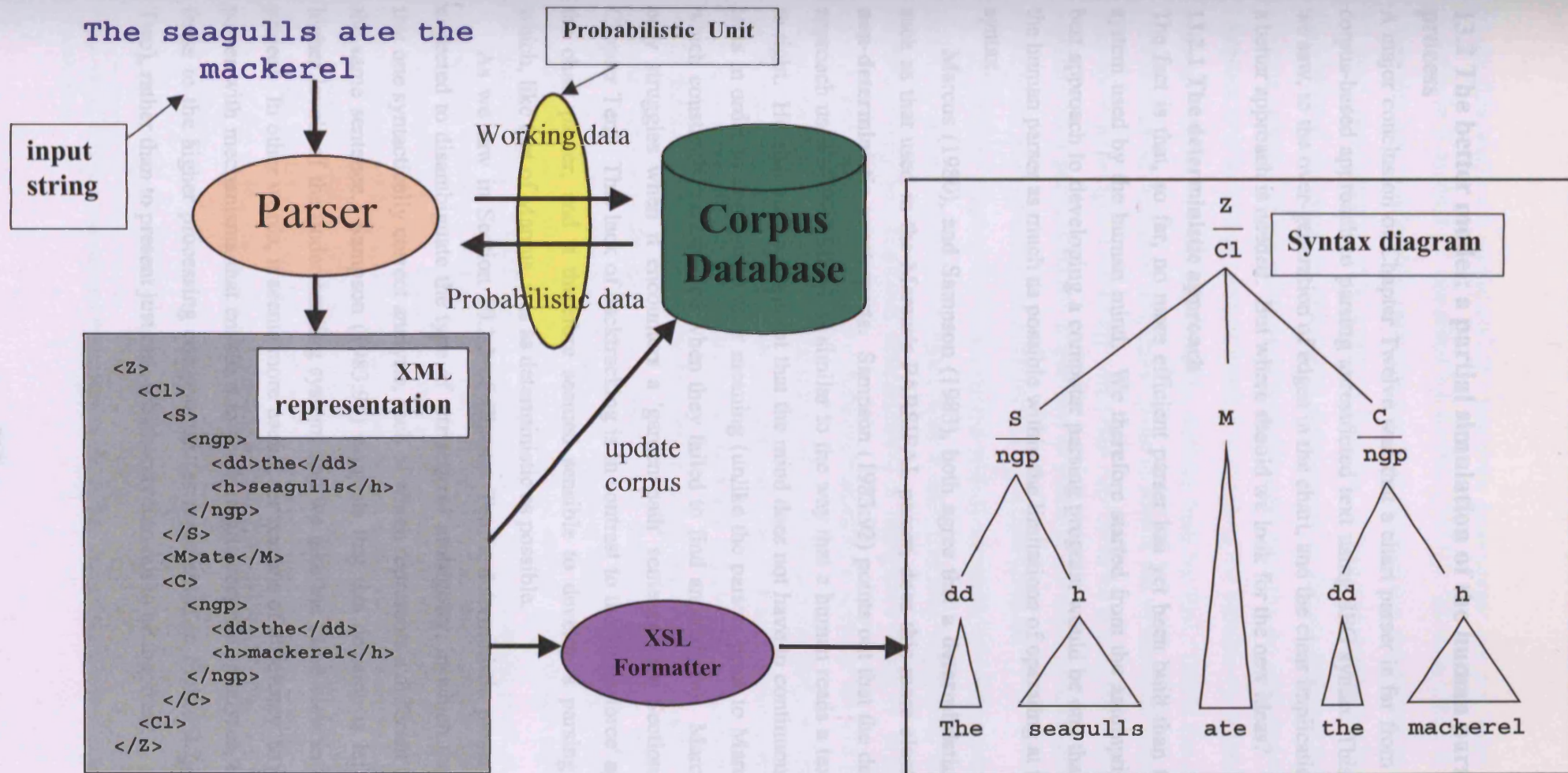


Figure 13.1: The corpus-consulting probabilistic parser: inputs and outputs

13.2 The better model: a partial simulation of the human parsing process

A major conclusion of Chapter Twelve was that a chart parser is far from ideal for a corpus-based approach to parsing unrestricted text using SFG syntax. This is due, as we saw, to the over-generation of edges in the chart, and the clear implication was that a better approach is needed. But where should we look for the new ideas?

13.2.1 The deterministic approach

The fact is that, so far, no more efficient parser has yet been built than the parsing system used by the human mind. We therefore started from the assumption that the best approach to developing a computer parsing program would be one that simulates the human parser as much as possible within the limitations of operating at the level of syntax.

Marcus (1980), and Sampson (1983), both agree that a **deterministic** approach, such as that used in the Marcus's PARSIFAL parser, does this more closely than its **non-deterministic** counterparts. Sampson (1983:92) points out that the deterministic approach used in PARSIFAL is similar to the way that a human reads a text from left to right. He also makes the point that the mind does not have to continuously re-read texts in order to understand their meaning (unlike the parsers prior to Marcus's work, which constantly backtracked when they failed to find an analysis). Marcus's parser only struggles when it encounters a 'garden path' sentence (see Section 10.2.3 of Chapter Ten). This lack of backtracking is in contrast to the 'brute-force' approach of the chart parser, and it therefore seemed sensible to develop a parsing algorithm which, like that of Marcus, was as deterministic as possible.

As we saw in Section 10.2.3 of Chapter Ten, a deterministic parser cannot be expected to disambiguate the type of **structural ambiguity**, in which there is more than one syntactically correct analysis, each of which represents a different meaning of the same sentence. Sampson (1983:96) suggests that this decision is left up to the higher levels of the understanding system, and we take the same view in the present project. In other words, it seems more useful, for reasons of efficiency, to provide the parser with mechanisms that enable it to produce all acceptable analyses, and present those to the higher processing components (as represented in Figure 2.2 of Chapter Two), rather than to present just one, which may turn out to be incorrect.

In this work, the parser and the **semantic interpreter** are not 'interleaved' (for example in a manner as described by Ritchie (1983)). In such cases, the semantic interpreter can, in principle, assist the parser in determining the correct analysis. But we have not followed this route to building a superior parser because: (a) the resolution of the problem frequently only occurs at the stage when the incoming message is added to the addressee's **belief system**, and (b) the goal that we have set ourselves is to build a parser that handles as much of the problem as it can at the level of form. The second reason, of course, follows from the first. This is not to deny that there is some advantage to be gained by the judicious use of semantic features (as was used by Weerasinghe (1994:57)). But this can never be a complete solution to the problem. So it seems sensible to perform as much of the work of analysis as possible at one level, before passing on more than one possible analysis.

In Chapter Twelve, we saw that the **n-best** approach can bring some success. However, if this method is coupled with a **backtracking algorithm**, the resulting parser can only be considered deterministic, in Sampson's terms, if **n** is set to the value 1, and it will only succeed if the correct solution is found at the first attempt (i.e. without the need to backtrack) (Sampson 1983). Although the goal for the new parser is that it should be able to find the correct parse at the first attempt whenever possible, there is an argument for allowing it to go back and discover more analyses. This can be for the following reasons. The first reason is that the parser is working at the level of syntax and does not have the knowledge to disambiguate when more than one analysis of a given sentence is legal (i.e. when structural ambiguity occurs). The second reason is to allow the parser to be able to analyse garden-path sentences (see Section 10.2.3 of Chapter Ten), i.e. to be able to go back to the point at which the wrong path was taken, and to take an alternative route. **Computational backtracking** is an expensive process. By implementing the **n-best** approach, and by allowing **n** to be a value greater than 1, allows the parser to take **n** analyses forward to the next stage of the parsing process without the need to backtrack. The value of **n** represents the size of the parser's search space at any point in the parsing algorithm.¹ During the testing of the parser, we will experiment with different values of **n** to determine an optimal value.

¹ The value of **n** has been termed the '**beam-width**' in the work of others (see Section 10.2.2.2 of Chapter Ten).

13.2.2 The corpus-based approach

One problem in developing a parser that models the human parsing process is that we cannot be absolutely certain how the human parser works. It seems likely, however, that both the **performer** and the **understander** of a text operate, unconsciously, with a knowledge of the possible syntactic relations between items, elements and units within sentences (interpreted as 'rules'), and of the probabilities of such relationships (as summarised in Appendix B of Fawcett, 2000a). We can assume that the humans gain this knowledge through their experience of what they have heard or read (and hence parsed) before, and that this knowledge changes dynamically as new texts are encountered. Similar knowledge can be represented in a syntactically analysed computer corpus from which a parsing system can extract its predictions. Such an approach is used here, hence the term **Corpus-Consulting** in the name of the parser. And since the corpus that is consulted resides in a database along with the parser's working data, the approach used here can be further characterised as **database-oriented**.

13.2.3 The probabilistic approach

Knowledge of the relationship and its probability between an item and an element, and between an element and a unit, is an important tool in the process of deciding that a given sentence is grammatical. Our knowledge of probabilities is an integral part of what we know about these relationships; hence the term **probabilistic** in the name of the parser.

The **probabilities tables** reside in the **corpus database** (see Chapter Seven). These are updated as new sentences are parsed, and they provide the parser with the data that in other approaches is typically presented as the separate component: the **lexicon**, as well as its knowledge of syntactic relations and their associated probabilities.

13.2.4 A comparison with the traditional ways of classifying a parser

In Sections 10.2.1 and 10.2.2 of Chapter Ten, we noted ways in which parsers have typically been classified, and in Section 10.2.6, we noted those classifications that best described the parser that is the subject of this work. In these terms then, the parser is **probabilistic**, **corpus-based**, **bottom-up**, **left-to-right**, **depth-first**, **incremental**, and, in principle, **deterministic**. It also uses a form of **best-first**, **beam search** in its **n-best** approach.

In terms of the concepts foregrounded in the last few sections, however, the most important characteristics of the new parser are that it is **deterministic** (in principle), **n-best**, **database-oriented** and **corpus-based** and **probabilistic**. The difference between these two lists of features is one indication of the degree of innovation reached at the present stage of the evolution, pioneered by Marcus, O'Donoghue, Weerasinghe, Souter and others, of probabilistic corpus-based parsers.

13.3 The overall method of research

As we saw in Chapter One, the method of research employed in the development of the parser demonstrated teamwork between a linguist and computer scientist who are working closely together. This proved to be an excellent research environment, and the project could not have been succeeded without it. The parser has access to linguistic knowledge of the way in which certain items affect the syntax that occurs around them, and it is precisely this that differentiates this work from the more mathematical, and formal approaches used by other parsers, where the concepts are strongly influenced by formal language theory. We have built the linguistic knowledge into the parser in two ways: most obviously, it is embedded into the **probabilities tables** (which will be described in Chapter Fourteen), but it is also present within the **parsing algorithms**, e.g. in the criteria for 'joining', and 'coordination' (as we shall see in Chapter Fifteen).

Linguistic knowledge is also required for Phase Two of the project, where it features in the new mechanisms that are currently under development for **linguistically motivated backtracking** (see Chapter Eighteen and Appendix L), and for Version Two of the probabilities tables, which provide a wider coverage of language for the parser to use (see Chapter Fourteen and Appendix I). I describe the project's implementation phases in the next section.

13.4 The relationship of the implemented research reported here and the work of future phases of the project

The parser is being implemented in a number of distinct phases. Phase One implements the algorithm and methods, which are described in Chapters Fourteen and Fifteen and demonstrated in Chapter Sixteen.

Phase Two covers improvements based on our findings after testing and evaluation. It includes the following:

- (a) the implementation of the Version Two **probabilities tables**,

- (b) the detection of multi-word items,
- (c) the treatment of punctuation, and
- (d) the implementation of **linguistically motivated backtracking**.

Much of the preparatory work for each of these has been completed; what is needed now is more time for the cycle of implementing, testing, improving and re-testing.

Phase Three will provide a parser that can handle

- (a) discontinuous units,
- (b) the recognition of **participant roles**, and so complements from the Main Verb (see Section 4.2.1.3 of Chapter Four),
- (c) additional sub-types of modifiers and Adjuncts - and most importantly,
- (d) the functions that have been added to provide the dynamic expansion of the database of knowledge of syntactic probabilities by constant updating, will be fully tested.

At the time of writing, the project is at the start of Phase Two. Further details of the improvements for Phase Two (and beyond) are given in Chapter Eighteen. As backtracking is an important concept in the implementation of the parsing algorithm, we start the discussion in the next section.

13.5 Backtracking

When we consider backtracking, we need to distinguish clearly between what we will term **computational backtracking** and **linguistically motivated backtracking**. We will take computational backtracking first. When a parse fails, a computational approach simply takes the parser back to the last path that has not been tried, and if that fails, back again to the path before that has not been tried, and so it is therefore computationally inefficient. In a deterministic approach, backtracking should occur relatively rarely, and ideally only when the parser is being led up the garden path, or when alternative analyses are sought. When the need to backtrack occurs, it is sensible to implement a **linguistically motivated approach** in which the parser knows the best point at which to go back based on **linguistic reasoning**.

The algorithm implemented in Phase One adopts a computational backtracking approach of the type described above. We now come to the parsing process itself.

13.6 Introducing the new parsing process

The **Corpus-Consulting Probabilistic Parser** operates in the **Parser WorkBench**, it is this that enables the researcher to run and test the parser. When being operated in the **Parser WorkBench**, the user can request that the parser simply performs the parse and present the results, or request an **incremental step-by-step parse**. Here the results can be reviewed and changed, or the parameters can be modified at the end of each stage of the process in order to test and evaluate changes. The **Parser WorkBench** is fully described in Chapter Sixteen.

The basic data structure of the parser is a tree, and there are two types. The **built structure** represents the items and syntactic structures that have been parsed so far, as the parser moves from left-to-right, and the **candidate structure** represents the item and syntactic structures that the parser is attempting to join to the built structure.

The parser operates through a cycle of seven stages, each responsible for a part of the algorithm, and is implemented as a **State Transition Machine (STM)**. This provides a **workflow**, which is a collection of states, actions and paths, through which a **work-unit** passes. The basic work-units are single trees or **tree pairs** that consist of a **built structure** and a **candidate structure**. The **Parser State Table (PST)**, which is one of the **parser working tables**, records the path of these work units through the states, and is used for backtracking. These data structures are described in Chapter Fourteen, and the **parser workflow** is described in Chapter Fifteen.

The unit nodes of the parse trees contain **forward** and **backward predictions**, which are used to identify potential **joins** between a **built structure** and a **candidate structure**. The predictions are obtained by requesting **forward unit structure** and **backward unit structure** corpus queries. Other queries to the corpus are used to 'grow' trees vertically, and these include queries about the various elements an item may expound, the units in which an element can be a component, and the elements that a unit can fill. These corpus queries are addressed to the **probabilities tables**, and they are described in Chapter Fourteen.

Each state that the parser enters implements a Stage (or a part of a Stage) of the algorithm. The algorithm is described in Chapter Fifteen, and a demonstration of how a sample sentence produces trees (which move through these states) is given by a walkthrough of the algorithm in Chapter Sixteen and in Appendix K. Briefly, the stages are as follows:

- (a) **Stage 1** is responsible for the **initial item** in the input string, and builds the initial **built structure** trees. It builds these trees all the way up to the Sentence Node (**Z**), and, in doing this, it operates differently for the first item than it does for all the other items in the sentence (see Section 15.2.2 of Chapter Fifteen).
- (b) **Stage 2** is used for the second and all subsequent items, and it builds up from the item to either the first or the second unit vertically.
- (c) **Stage 3** attempts to join the **candidate structures** built in Stage Two to the existing **built structures**, and consists of two sub-stages. The first nominates pairs of trees for a join, and the second performs the **n-best** of them (where **n** is one of the **configurable parameters**). If there are no 'good' joins, the parser moves to Stage 4.
- (d) **Stage 4** is therefore invoked if the attempted joins in Stage 3, or Stage 5 have not succeeded. It grows the candidate structures by a further element and unit, and then presents them to Stage 3 again.
- (e) **Stage 5** is reached after a certain number of attempts at growing and joining (Stages 3 and 4) have failed, or the candidate item is a linker. It is concerned with **co-ordination** of units. Like Stage 3, it has two sub-stages that are responsible for the nomination, and the joining of these co-ordinated trees.
- (f) **Stage 6** provides for backtracking. It is only reached if the previous stages fail. It is responsible for directing the parser back to the appropriate point, in order to attempt a new parse. However, the use of the **n-best** approach reduces the number of occasions in which this is required to a minimum.
- (g) **Stage 7** is reached when the parse succeeds. If the parser is operating in **step-by-step mode** (see Chapter Sixteen), it asks the user if further parses are required. Otherwise: (a) it returns the parser to the appropriate state in which it can parse another sentence (if the required number of analyses has been reached), or (b) it backtracks to the previous **move** that has not been followed, and looks for the next analysis for the given sentence.

Finally, when a successful parse is identified, the user can check it in the **Parser WorkBench's Sentence Viewer**, and then add it to the **corpus tables** in the corpus database. The effect of this is to automatically update the corpus (so that the new sentences are available when responding to corpus queries from the **corpus query tool**), and also the **probabilities tables** (so that the changed probabilities will affect future parses). It is in this way then, that the corpus is **dynamic**.

13.7 Summary

This chapter has introduced the **Corpus-Consulting Probabilistic Parser**. It began by restating and clarifying the aims of the parser, before discussing better methods of parsing that overcome the problems experienced by the earlier researchers that were described in Chapter Twelve. The new approach will more closely model the human parsing process by using **determinism** and **incremental parsing**.

The way in which this project is implemented in phases was discussed, and the deliverables from each stage were outlined.

This chapter provided an overview of the new parsing algorithm, and the next few chapters will provide the necessary detail.

Chapter Fourteen

The parser's probabilities tables, working tables and data structures

This chapter provides a brief overview of: (a) the **probabilities tables** and the **queries** by which the parser accesses them, (b) the **parser working tables**, and (c) the parser's **data structures**. Full information about these tables can be found in Appendix H.

Section 14.1 describes some of the different types of **probabilities tables** and their queries. Section 14.2 introduces Version Two of the probabilities tables, and Section 14.3 introduces the data structures used by the parser and the parser working tables. Information about the Version Two tables can be found in Appendix I.

The way in which the parser uses the probabilities tables and the data structures will be explained in the next chapter.

Both the probabilities tables and the parser working tables are stored in the **corpus database**, and it is this that gives the parser its unique **database-oriented approach**. In effect then, the chapter completes the work of describing the corpus database schema that was begun in Chapter Seven.

14.1 The probabilities tables and their queries: Version One

This section describes Version One of the probabilities tables and the queries performed on them. There are two groups of queries, **vertical queries** and **horizontal queries**.

Vertical queries have this name because they ask about the relationship upward between an item, element or unit and the syntactic category (or categories) above it in the tree. This relationship may reach up to the 'root' of the tree (i.e. the Sentence node(**Z**)).

The vertical queries return **vertical strips**, or parts of vertical strips. As an example of a vertical strip, consider Figure 14.1, in which the first vertical strip consists of **"The" dd ngp S C1 Z**.

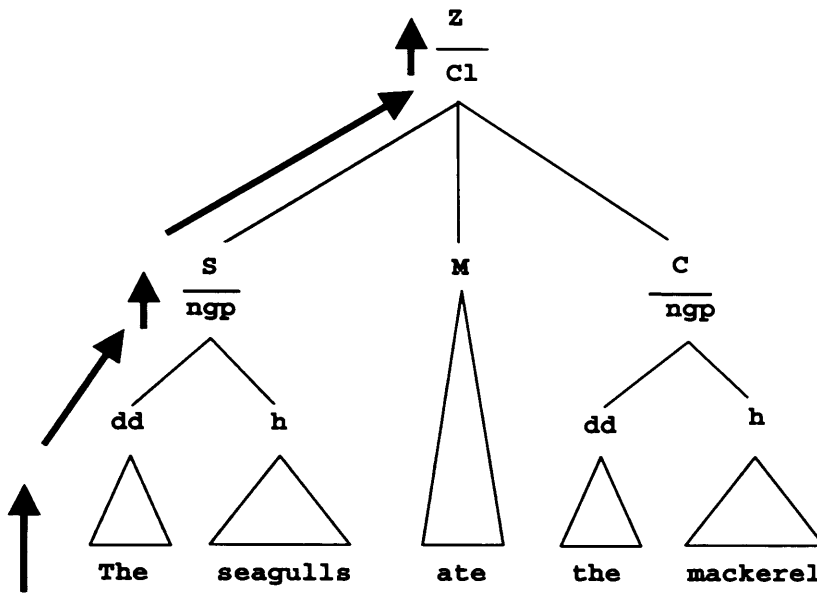


Figure 14.1: A sample sentence and a set of initial vertical strips extracted from it

The types of vertical queries used by the parser are:

- (a) Initial vertical strip based on Item (**IVS-ITEM**),
- (b) Initial vertical strip based on element of structure (**IVS-ELEM**),
- (c) Item-up-to-Element (**I2E**),
- (d) Item-up-to-Element-up-to-Unit (**I2E2U2E**),
- (e) Element-up-to-Unit (**U2E**),
- (f) Unit-up-to-Element (**E2U**).

As an example of the results of an **IVS-ITEM** query, see Table 14.1.

ITEM	VERTSTRIP	PROBABILITY
that	h_p ngp S C1 Z	0.3228
that	dd ngp S C1 Z	0.3199
that	h_p ngp C C1 Z	0.2796
that	dd ngp C C1 Z	0.2771
that	h_p ngp cv pgp C C1 Z	0.0547

Table 14.1: An excerpt from the **IVS-ITEM** query for the item "that"

The table shows that there is a 32.28% probability of **that** being a pronoun head (**h_p**) in a nominal group (**ngp**) that fills the Subject (**S**) in the primary clause (**C1**) and a similar probability that it may be a deictic determiner (**dd**) in a Subject - and lower probabilities that it may occur in a Complement (**C**).

See Appendix H for a full account of all types of vertical tables and their associated queries.

Horizontal queries return **elements** (or groups of elements) together with an associated probability. They operate using **horizontal slices** (or partial horizontal slices) at any level of the tree which represents the elements in the structure of a **unit**.

There are two horizontal queries called **Unit Structure Queries**:

- (a) Forward Unit Structure (**FUS**),
- (b) Backward Unit Structure (**BUS**).

In contrast with the tables and their queries that have been described so far, unit structure queries ask about the horizontal relationships in the parse tree. A Forward Unit Structure (**FUS**) query elicits a set of elements that may follow a given string of elements that has already been established, and their probabilities. A Backward Unit Structure (**BUS**) query elicits the set of elements that can precede a given element in the same unit and the probabilities of each. See Table 14.2 for an example of the results returned for a **FUS** query.¹

UNIT	ELEMENTS	NEXTELEM	PROBABILITY
Cl	! S O	M	0.7643
Cl	! S O	!	0.1134
Cl	! S O	X	0.0539
Cl	! S O	A_Log	0.0121
Cl	! S O	A_Inf	0.0087
Cl	! S O	A	0.0066
Cl	! S O	I	0.0055

Table 14.2: Unit structure query for the next element in a Clause (Cl) that has found a Subject (S) and an Operator (O)

This concludes our brief description of the probabilities tables and the types of query that are used to extract data from them. See Appendix H for a more detailed account of all the types of vertical and horizontal tables and their queries. In Chapter Fifteen we will see how the parser uses these tables in the parsing algorithm.

14.2 The development of the Version Two tables

In constructing the tables introduced in this chapter, we noted several places in which the probabilities were skewed by the fact that the corpus was one that contained only children's spoken texts. This led us to develop a second version of the tables so that

¹ Note that in the unit structure queries, the start and the end of the unit are treated (for computational purposes) as if they were elements in the unit - each being represented by an exclamation mark (!).

there are now two versions of the probabilities tables, called **Version One** and **Version Two**. The two versions are stored in database tables that have the same structures, the difference being the different data in each version. Appendix I provides a full account of the reasons why we decided to develop Version Two and how we accomplished this demanding task.

The result of this work is that the parser now has access to a database of approximately a 16,000 Main Verb forms, 200,000 nouns, 16,000 adjective forms, and 13,000 manner adverb forms, and a score of other elements such as prepositions, Main Verb extensions and so on.

Next, we turn to a very different set of tables, and the algorithms that are used to populate them. These are the tables that store the parser's working data including the data structures.

14.3 The parser data structures

The way in which the parser is **database-oriented** is that its working data are stored in database tables. This section describes these tables and the functions that operate on them.

The basic data structure in the parser is a **database tree**. There are two types of tree. Trees that represent what the parser has built so far are called **built-structures**. As the parser moves from left to right through the words of a sentence, the parsing of each new item results in the building of one or more **candidate structures**.

Each tree is represented as a record in the database table called **DB_PARSE_TREE**, and full details of this and other parser working tables can be found in Appendix H. For a full description of how the parser uses these trees, see Chapter Fifteen.

14.4 Summary

This chapter has outlined, very briefly, the nature of the **probabilities tables** and the **parser working tables**, Appendix H provides a much fuller specification of these. Appendix D provides details of all tables and their fields. Appendix I provides details how the Version Two tables were constructed, and the reasons why they were created.

We are now in a position to examine the parsing algorithm itself, which is the subject of the next chapter.

Chapter Fifteen

The parsing algorithm

Chapter Fourteen (supported by Appendix H) has described the database used in this approach to parsing, so showing why it should be characterised as **database-oriented**. We have seen that the probabilistic data is held in the **probabilities tables**, and how **queries** can be performed to extract the data to be used by the parser. Chapter Fourteen also defined the **parser's data structures**, since these are also stored in the database, using the **parser's working tables**. The purpose of this present chapter then, is to describe the parsing algorithm itself, and in doing so it will show how the concepts described in Chapter Fourteen are used in parsing a sentence.

Section 15.1 provides definitions of the basic concepts introduced in Chapter Thirteen and now to be presented as part of the working model.

One important concept that is introduced here for the first time (in Section 15.1.4) is the idea that a parser can be implemented as a **state transition machine**. This is a virtual device that provides the parser with its **workflow model** through which trees, and pairs of trees pass during the parser's analysis of a sentence.

As we saw in Chapter Thirteen, the parsing algorithm is implemented in modules called 'Stages'. Section 15.2 describes these stages in greater detail.

Finally, Section 15.3 introduces briefly the parser's operating environment - the **Parser WorkBench**, which is described in detail in Chapter Sixteen.

15.1 The basic concepts defined and illustrated

15.1.1 Candidate structure trees and built structure trees

Recall from Chapter Thirteen that a **built structure** is a parse tree that contains information about what has been parsed so far. Hence it comprises of one **item** node (for each item that has been parsed) and a number of **element** and **unit** nodes. Please now inspect Figure 15.1, which provides a visual representation of this concept. It shows a built structure tree that represents how far the parser has got after it has processed two items (**the** and **seagulls**). Note that it identifies the rightmost strip by stars.

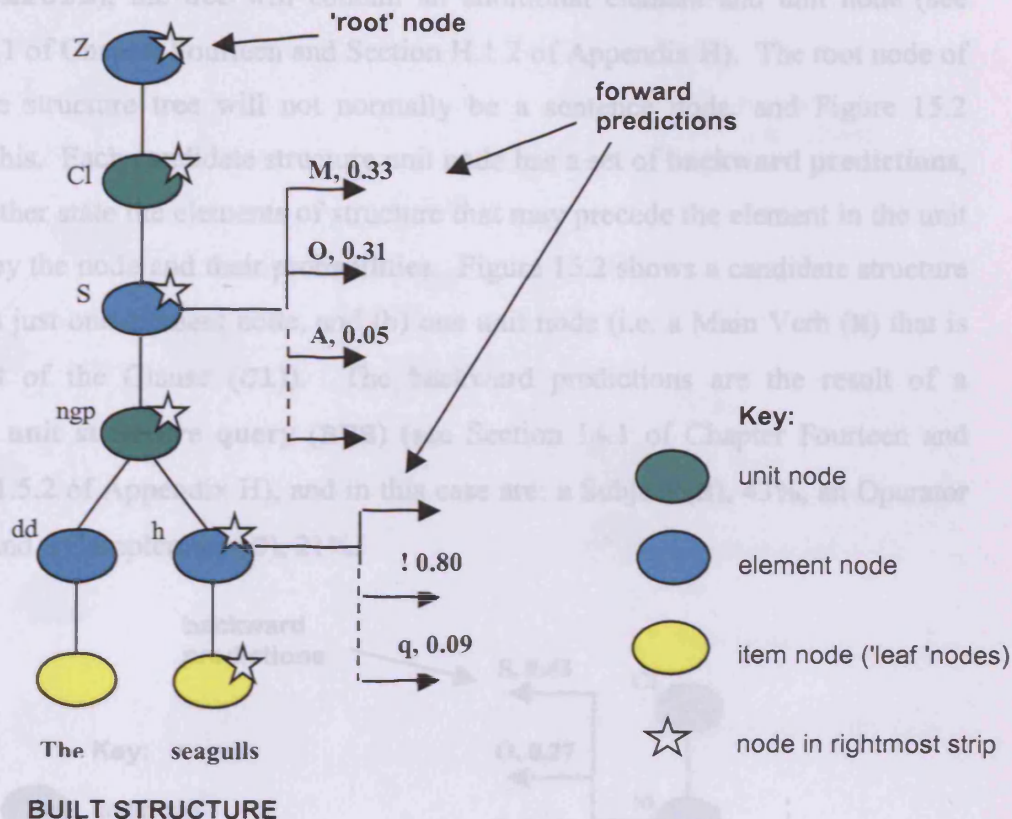


Figure 15.1: A built structure

The topmost node (the 'root' node) is the sentence (**Z**) node. It is the **unit** nodes (i.e. the Clause (**C1**) and the nominal group (**ngp**)), that function as the reference point for the **forward predictions** about the element that may follow the last element parsed so far in each unit. These predictions are the product of a **forward unit structure query (FUS)** (as we saw in Section 14.1 of Chapter Fourteen and H.1.6.1 of Appendix H), and are based on all the elements in the unit that have been parsed so far (i.e. the Subject (**S**) in the Clause (**C1**) in Figure 15.1, and the deictic determiner (**dd**) and the head (**h**) for the nominal group (**ngp**)). However it is only the elements in the **rightmost strip (RMS)** of the **built structure** that are involved in an attempt to join a **candidate structure** to the **built structure**.

A **candidate structure** is a **partial vertical strip** that the parser is about to attempt to join to the built structure (or to more than one, if alternative built structures have been generated). Technically it is a tree, but it is one with unary branching. At the stage of the parsing process when a new candidate structure is being created, it will typically consist of an **item node**, and an **element node** and a **unit node**. However, for items that are the subject of an **item-up-to-element-up-to-unit-up-to-element**

query (I2E2U2E), the tree will contain an additional element and unit node (see Section 14.1 of Chapter Fourteen and Section H.1.2 of Appendix H). The root node of a candidate structure tree will not normally be a sentence node, and Figure 15.2 illustrates this. Each candidate structure unit node has a set of **backward predictions**, which together state the elements of structure that may precede the element in the unit described by the node and their probabilities. Figure 15.2 shows a candidate structure that (a) has just one element node, and (b) one unit node (i.e. a Main Verb (M) that is an element of the Clause (C1)). The backward predictions are the result of a **backward unit structure query (BUS)** (see Section 14.1 of Chapter Fourteen and Section H.1.5.2 of Appendix H), and in this case are: a Subject (S), 43%, an Operator (O), 27%, and a Complement (C), 21%.

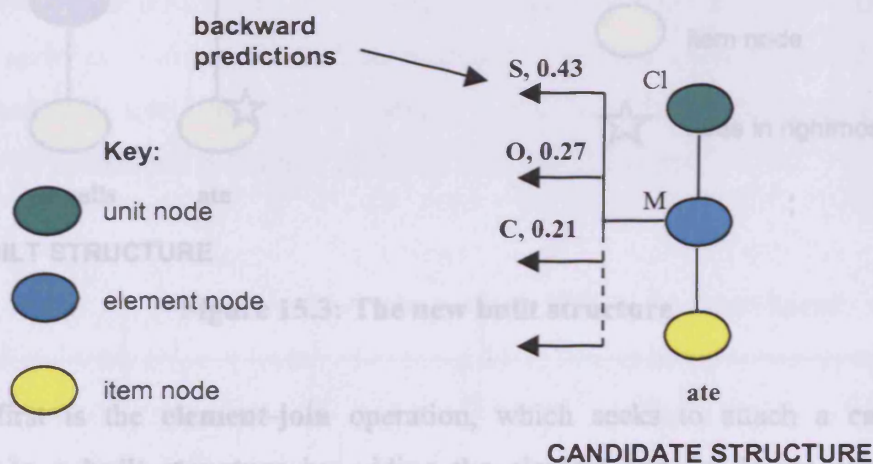


Figure 15.2: A candidate structure

15.1.2 Tree operations and tree pairs

The goal of the parser is to join the **candidate structure(s)** appropriately to the **built structure(s)**, hence tree joining operations are performed on **tree pairs**. The parser will first attempt to merge a unit node (or element node) from one with the same class of unit (or element) in the other. As the preceding words imply, there are two types of **join operation**.

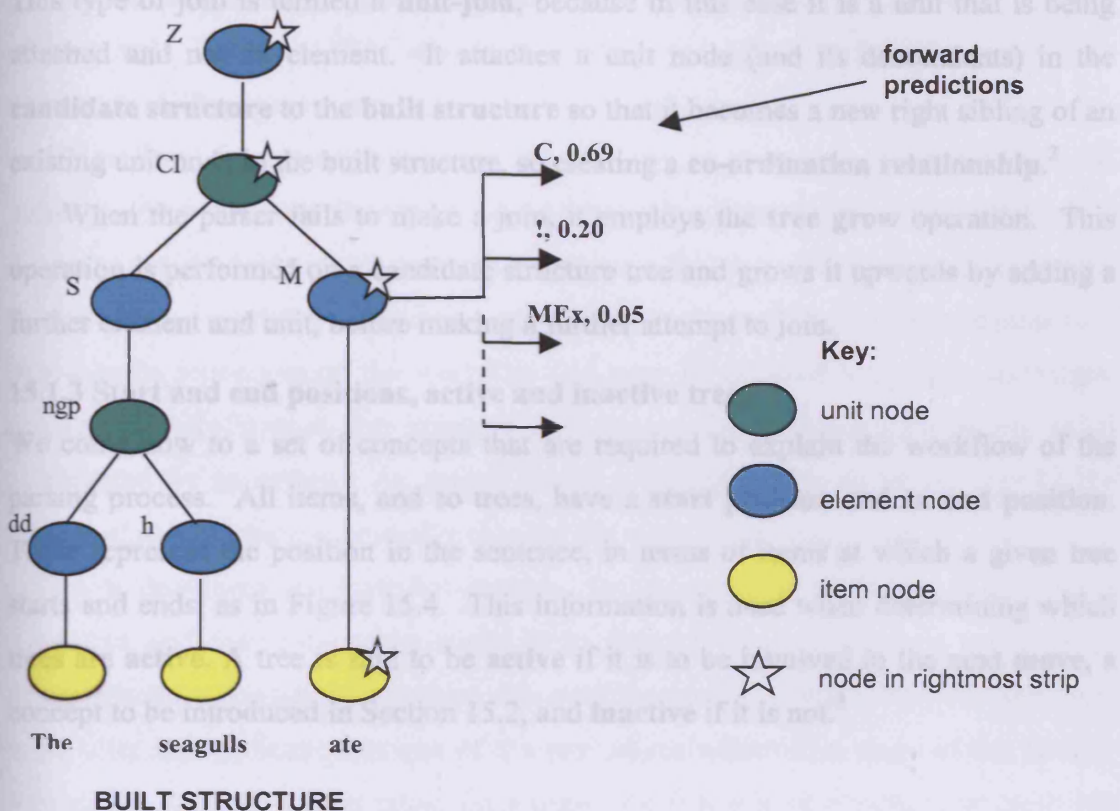


Figure 15.3: The new built structure

The first is the **element-join** operation, which seeks to attach a **candidate structure** to a **built structure** by adding the element and its descendants in the candidate structure to the built structure. Here the units must match, and there must be a match in the backward and forward predictions. The result of the join is that the unit in the built structure is extended by the additional element (and it is copied into a new **active** built structure). Figure 15.3 shows the **element-join** between the built structure shown in Figure 15.1, and the candidate structure in Figure 15.2. Following the join, the forward predictions are recalculated so that they include the new element. The new forward predictions for the elements of the Clause (**Cl**) are created on the basis that the new Clause structure now contains a Subject (**S**) and a Main Verb (**M**), and the Clause is now predicting that the Main Verb is most likely to be followed by a Complement (**C**) (69%), although it also can be the last element in the Clause (!) (20%).¹

¹ Recall from Chapter Fourteen that the ! means start or end of the unit.

The second and far less frequent type of join is designed to handle **co-ordination**. This type of join is termed a **unit-join**, because in this case it is a unit that is being attached and not an element. It attaches a unit node (and its descendants) in the **candidate structure** to the **built structure** so that it becomes a new right sibling of an existing unit node in the built structure, so creating a **co-ordination relationship**.²

When the parser fails to make a join, it employs the **tree grow** operation. This operation is performed on a candidate structure tree and grows it upwards by adding a further element and unit, before making a further attempt to join.

15.1.3 Start and end positions, active and inactive trees

We come now to a set of concepts that are required to explain the workflow of the parsing process. All items, and so trees, have a **start position** and an **end position**. These represent the position in the sentence, in terms of items at which a given tree starts and ends, as in Figure 15.4. This information is used when determining which trees are **active**. A tree is said to be **active** if it is to be involved in the next **move**, a concept to be introduced in Section 15.2, and **inactive** if it is not.³

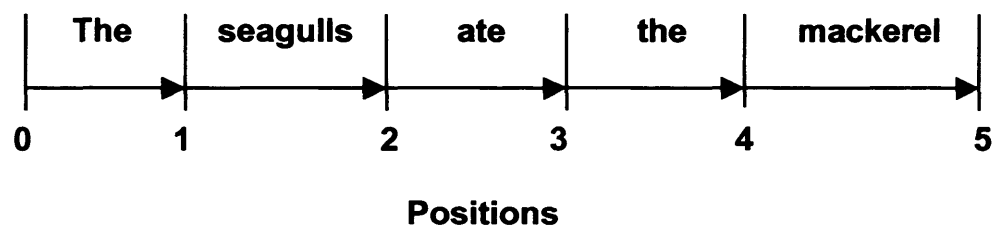


Figure 15.4: Positions

15.1.4 The workflow of the parsing process

The overall 'flow' of the work performed by the parser is implemented as a state transition machine (STM).⁴ STMs are very common in the field of computer science and have many different uses. They are used, for example, for modelling real-time systems which have to respond to different events which cause the system to move to a different state (Sommerville 1992:244). Although applications of similar principles can be found in computational linguistics (e.g. in the form of recursive and augmented

² Note that when there are elements and units above the matched unit in the candidate structure that they must match the elements and units above in the built structure.

³ Thus a tree may be **active** or **inactive** in a sense similar to that in which these terms are used to the edges in a chart parser.

⁴ The terms **finite state automata**, **finite state machines**, and **transition networks** (among others) are often used to describe similar methods. The term **state transition diagram** (STD) is often used to describe the type of diagram shown in Figure 15.5.

transition (RTN/ATN) networks as described in Section 5.2.2 of Chapter Five), the purpose of these is significantly different since an ATN or RTN is often said to model a 'grammar'. Here, the STM is used to model a **workflow**, and the 'grammar' is a completely different concept.

Workflow management systems are growing in popularity in commerce and industry where they are frequently used to represent business processes. Workflow diagrams are similar to state transition diagrams in that they define **states** (represented as ellipses on a **workflow diagram**) and **actions** (represented by arcs on the diagram). **Work-units** move around the workflow and may be grouped into **work packages**. The movement of a work-unit from one state into another is called a **transition**, and it is triggered by an **action**. An action, which is performed by an **actor**, may be the subject of **conditions** and normally changes the work-unit in some way.

In a document management system, for example, a work-unit would be a document, and one of the actors would be an author.⁵ The parser uses a workflow of essentially the same type. The states represent the condition that a tree, or a tree pair, is in, after the application of one of the procedures within in a stage of the parsing algorithm. A **work-unit** is either (a) a parse tree (i.e. a built structure, or candidate structure), or (b) a tree-pair (as defined in Section 15.1.2). A **parser move** from one state to another is the equivalent of an **action**, and the **actor** is the parser itself. Figure 15.5 shows the workflow model for the parser. The filled circles show the start of the flow and the end of the flow. Ellipses show states, and the arrows show the paths that the work-units follow as they **transition** from one state into another. In reading the rest of this chapter, it will be useful to refer frequently to Figure 15.5 in order to understand the relations between the various stages, and what each stage seeks to accomplish.

In a workflow management system, a history is normally maintained which records dates, times, and the actions that have moved a work-unit from one state to another. In the document management system, for example, the workflow history provides details of when the document was approved or rejected. In the parser workflow the equivalent is the **parser state table** (PST) (for which see Section

⁵ The other actors would be for example, editor, reviewer, illustrator etc., and the states of the workflow may be new, work-in-progress, ready-for-approval, approved, and rejected. The document moves between the states new (a requirement for a document is recognised), work-in-progress (the document is being authored), ready for approval (authoring has finished and the document is sent for approval), approved (the document has been approved) and rejected (the document has been rejected, and has to be moved back to work-in-progress).

15.1.5). This records trees, or tree-pairs, the states they were in, and the states to which they moved following the execution of a stage of the parsing algorithm.

Only one **move** can happen at any one time, but multiple work-units can be handled as a **work package** which represents the **n-best** set of tree pairs that is being processed during the transition from one state to another. The size of the package is determined by the value of **n** in the appropriate **n-best** parser parameter (see Section 13.2.1 of Chapter Thirteen, and Section 16.3 of Chapter Sixteen).

The other innovation introduced in the parser's workflow is the concept of **backtracking**. This occurs when the parser 'jumps' back to a previous state in the workflow. This concept simply would not be there in a business process workflow. **Backtracking** is covered in Section 15.2.7.

A **parser move**, then, is defined as the movement of the parser from one state in the workflow to another. The work-unit is a **tree** or a **tree pair**, and the workflow history is stored in the **parser state table**. The actor in the workflow, as we have seen, is the parsing algorithm itself.

The workflow, then, is a core concept in modelling the parsing process. Table 15.1 summarises stages and states of the process.

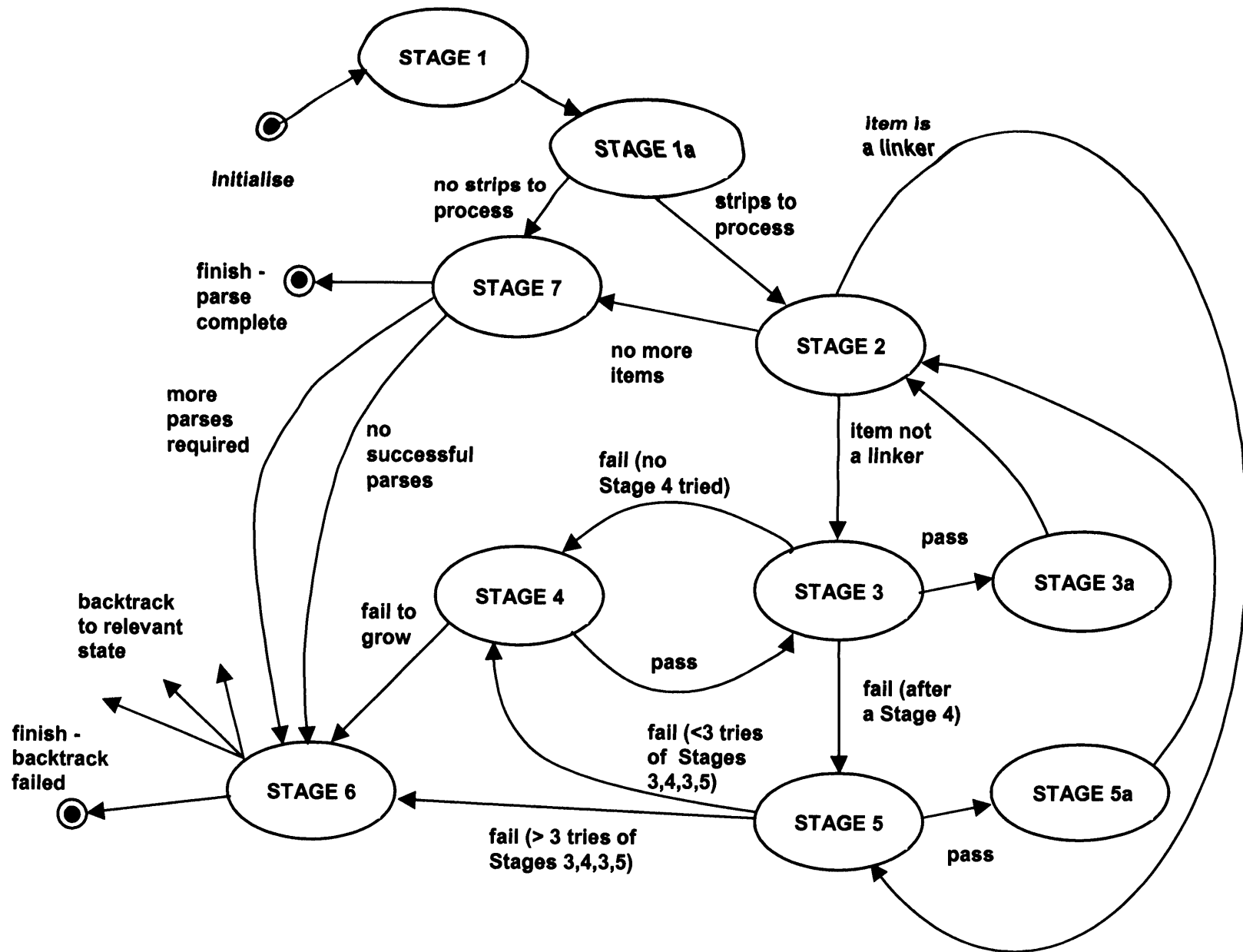


Figure 15.5: The workflow of the parsing process

Stage	Purpose
Stage 1	Create an ordered list of the initial vertical strips that can occur for the first lexical item.
Stage 1a	Take the n most likely initial vertical strips from Stage 1 and express each as a built structure.
Stage 2	Take the next item from the input and create candidate structures for it.
Stage 3	Calculate a joining score for each possible join that can be made between any active built structures and candidate structures.
Stage 3a	Take the n most likely possible joins (i.e. those over a certain joining score) from Stage 3 and create new built structures for them.
Stage 4	Grow the active candidate structures by one element and one unit.
Stage 5	Calculate a co-ordination joining score for any active built structure and candidate structure.
Stage 5a	Take the n most likely possible joins (i.e. those over a certain joining score) from Stage 5 and create new built structures for them.
Stage 6	Backtrack (see Section 15.2.7).
Stage 7	Determine if the parse has succeeded, and whether more parses are required.

Table 15.1: The stages of the parser

15.1.5 The parser state table

The **parser state table** (PST) is used to record the **current state** of the parse, and the trees, or tree pairs involved in it. It is used (a) to record and order the most likely analyses that will be considered next, (b) to identify the next state for any given tree or tree pair, and (c) as a tool to enable backtracking to be performed. It includes the following fields:

- (a) **BSTREEID** and **CSTREEID** are the identifiers (IDs) of the built and candidate structures,
- (b) **BSNODEID** and **CSNODEID** are the IDs of the nodes that can be joined (for example),
- (c) **NEXTSTATE** tells the parser where it goes next with the given tree or tree-pair,
- (d) **SCORE** which, depending on the state of the parser, contains the score of a join, grow, or the score of a given tree.

The contents of the fields of a PST record depend on the state of the parser when it was created. Taking Stage 3 as an example, the parser creates:

- (a) a set of PST records which represent each possible join together with its score,
- (b) a set of PST records which represent a join that has a score which is beneath the join threshold value (see Section 15.2.4),
- (c) a set of PST records which represent the tree pairs that cannot be joined.

For (a), the **NEXTSTATE** field is set to **STAGE 3a**. For (b) and (c) it is set to **STAGE 4**. If there are records of type (a), the parser will move to Stage 3a and make **n** joins (or all joins, if less than **n** score above or equal to the **join threshold value**), if there are none, the parser moves to Stage 4 and grows the **candidate structures**. Any structures joined or grown will have the **FOLLOWED** field set to true in the original PST record.

Table 15.2 shows the contents of the PST record for each state of the parsing algorithm.⁶

⁶ Two fields are omitted from Table 15.2. These are (a) **CYCLES** which records how many times the tree-pair has traversed the 3-4-3-5 cycle (if it is above the threshold, the **NEXTSTATE** is set to **STAGE 6** for backtracking), and (b) **STAGE4TRIED** which indicates if the given candidate structure has already been grown. If it has, then the parser will move to a Stage 5 rather than a further Stage 4.

Stage	BSTREE ID	CSTREE ID	BSNODE ID	CSNODE ID	NEXTSTATE	SCORE	Remarks
Stage 1	not used	not used	not used	not used	Stage 1a	not used	Creation of initial vertical strips in the DB_IVS table.
Stage 1a	Id of the created BS tree	not used	not used	not used	Stage 2	Probability of the BS tree	Stage 1a creates the next n built structures from the DB_IVS table.
Stage 2	not used	Id of the new CS tree, if there was an item to process, otherwise not used	not used	not used	Stage 3 if there was an item to process, otherwise Stage 7	Probability of the CS tree	Stage 2 creates a candidate structure for the next item.
Stage 3	Id of BS tree	Id of CS tree	Id of the BS node which can be joined to the CS node	Id of the CS node that can be joined to the BS node	Stage 3a	Joining score associated with the join of the trees at the nodes indicated	Stage 3 identifies the trees that can be joined at the given nodes.
Stage 3a	Id of the new BS tree when join succeeds. Not used if join fails.	Not used if join succeeds. Id of the CS Tree if fails	not used	not used	Stage 2 (if the join succeeds), Stage 4 (if failed and STAGE4TRIED is false) Stage 5 (if failed and STAGE4TRIED is true)	Probability of the new BS tree	Stage 3a performs the n best joins and creates new BS trees.

Table 15.2: The use of the parser state table for each part of the parsing algorithm (Sheet 1 of 2)

Stage	BSTREE ID	CSTREE ID	BSNODE ID	CSNODE ID	NEXTSTATE	SCORE	Remarks
Stage 4	Not used	Id of CS tree	not used	not used	Stage 3 if grow succeeds otherwise Stage 6 (backtrack)	Probability of CS tree after grow operation	Stage 4 grows the CS tree by an element and unit
Stage 5	Id of BS tree	Id of CS tree	Id of BS node that may be joined	Id of CS node that may be joined	Stage 5a if there are any joins. Stage 4 if it fails and CYCLES of 3,4,3,5 is less than the limit, otherwise Stage 6 (backtrack)	Co-ordination joining score	Stage 5 identifies potential co-ordination joins between trees
Stage 5a	Id of BS tree	not used	not used	not used	Stage 2	Probability of BS tree	Stage 5a joins the trees by co-ordinating units and the PST records the new trees
Stage 6	No PST record because Stage 6 is backtracking and it includes a decision on the state into which the parser will move.						
Stage 7	No PST record because Stage 7 identifies that the parse is complete, and responds to a request to find more parses.						

Table 15.2: The use of the parser state table for each part of the parsing algorithm (Sheet 2 of 2)

As the parser moves from state to state, built structures and candidate structures are made **active** and **inactive** (see Section 15.1.3). When the parser backtracks, it will re-activate any inactive trees that were active in the move to which the parser backtracks (it gets the information about which trees to re-activate from the parser state table).

15.1.6 The **n**-best approach

As the parser moves between states, it takes with it, as its work-package, the **n** most likely trees (or tree-pairs). For developing and fine-tuning the parser, these values can be set by the user in the **Parser WorkBench** (see Chapter Sixteen).

As we saw in Section 10.2.2.2 of Chapter Ten and Section 12.1.4 of Chapter Twelve, the **n**-best approach is similar to the **beam search** or **best-first** algorithms that have been successfully used in other parsers (e.g. Collins 1996, 1999). However, there is a difference between these other approaches and the one used here in that the algorithm applies a threshold to the **n** value. This specifies that only the best **n** values over **x%** should be followed, and this means that in some cases, there will be fewer than **n** routes taken into the next stage. The value of **x%** is termed the **join threshold value**.

Initially, we took the position that a join should always be preferred to growing a candidate structure upwards, even when the value of joining score was low. But testing showed that this should not always be the case, since some very unlikely joins were being made when, in reality, the candidate structure should have been grown. We introduced the refinement of first taking the **n**-best joins over **x%**, and then performing the tree growing operations, and following them through, before going back to consider the joins under **x%**.

We also implemented the concept that a threshold value of **x%** can be varied during the parse. The tests showed that, with some configurations of the joining score formula's parameters, the score decreases as more items are parsed, and hence the value of **x** depends on the position in the sentence of the parse. Furthermore, the value of **x** may be decreased when the parser backtracks, as explained above, so that joins with a lower score are accepted after backtracking - provided that they are above (or equal to) the new lower value of **x**. These values (and the other configurable parameters) are set in the **Parser WorkBench** (see Chapter Sixteen).

15.1.7 Node and level probabilities

When the candidate structure or built structure consists of a single vertical strip, the **node probability** is the probability that has been established through the query that generated the node. For example:

- (a) if the node is an **element node**, the node probability will be determined by the probability derived from either (i) the item-up-to-element query (**I2E**), (ii) the item-up-to-element-up-to-unit-up-to-element query (**I2E2U2E**) or (ii) the unit-up-to-element query (**U2E**) that generated it.
- (b) If the node is a **unit node**, the node probability will be the result of the element-up-to-unit (**E2U**) query that created it.

The **level probability** of a node is defined as the **product** of the **node probability** and the child node's **level probability**. The level probability of the root node, therefore, is the probability associated with the tree.⁷

When the built structure contains more than a single strip, the node probability of nodes that have more than one child is even more complex. Since the new built structure is the result of a join operation, the node probability becomes the product of:

- (a) the node probability of previous element in the unit,
- (b) the joining score, and
- (c) the node probability of the node that has been gained from the candidate structure tree.

As you can see, the new parser introduces quite a number of concepts that are new to the field of parsing. We turn now to consider the complete picture of the parsing process in terms of its stages - and, as we do this, we will see the concepts in use.

15.2 A full specification of the algorithm

15.2.1 Stage 0: Initialising the parser

Summary: This stage simply initialises the parser.

Stage 0 is not shown in the workflow given in Figure 15.5. It is merely a 'housekeeping' stage in that its only responsibility is to prepare the parser for use. It sets variables, reads parameters and clears the parser working tables ready for the parse. It is performed once for every new sentence.

⁷ The level probability of the root node is the probability of the tree to which the node belongs and is stored as the tree probability in the **DB_PARSE_TREE** table (see Chapter Fourteen and Appendix H).

15.2.2 Stage 1: Parsing the initial item

Summary: Stage 1 is broken down into two stages, Stage 1 and Stage 1a. Stage 1 gets the initial item from the input string and performs an initial vertical strip (IVS) query, and creates a list of all possible IVSs for the given item. Stage 1a takes the n most likely of these and creates built structure trees for them.

Stage 1 only applies to the first item in the input sentence. This item has to be treated differently from all the other items because there are no preceding items, so that there are no built structures that could provide **forward predictions** to assist the parser in determining the correct analysis. Nor, of course, are there any **backward predictions**. In addition, some items have different probabilities when they occur in the sentence initial position. It was for these reasons that we introduced the **initial vertical strip (IVS) tables** and their queries (as described in Section 14.1 of Chapter Fourteen and Section H.1.1 of Appendix H).

Using the **IVS** tables greatly enhances the efficiency of the parser. In the early versions of the parser we employed techniques similar to those that will be described below for use with the second and subsequent items, in order to build the structures above the initial item up to a sentence node. But we found that this approach generated an enormous number of unwanted built structures that were either syntactically impossible or highly unlikely, and these slowed down the parser.⁸

Please consult Figure 15.5. Stage 1, which is reached directly from the initialisation (Stage 0), and is only performed once in the entire parse, even if backtracking occurs. It reads the first item from the input string and performs an **item-up-to-element (I2E)** query for it. The **I2E** query will return one **I2E** record for each element that the item can expound, together with its associated probability, and it indicates whether:

- (a) an **initial-vertical-strip-item (IVS-ITEM)** query is needed for this item, and
- (b) if an **item-up-to-element-up-to-unit-up-to-element (I2E2U2E)** query is required for this item-element pair, in which case one is performed.

If an **IVS-ITEM** query is required, the parser requests one for either:

⁸ For example the parser would generate a built structure for the qualifier of a nominal group, which is virtually impossible in the sentence initial position.

- (a) all of the **item-element pairs** from the **I2E** query, or
- (b) all of those **I2E** entries that lead onto an **I2E2U2E** query, in order to obtain an **item-element-unit-element quadruple** (e.g. **I-h_p-ngp-S**).

It then builds a working **initial vertical strip table** (the **DB-IVS** table - described in Section 14.1 of Chapter Fourteen and Section H.1.2 of Appendix H), which contains all possible initial vertical strips, together with an associated score.⁹

Next the parser requests an **IVS-ELEM** query both for the item-element pairs that are not identified as **IVS-ITEMS**, and for those that are.¹⁰ The parser adds any new vertical strips that to the parser's **DB-IVS** table. It may, of course, find vertical strips with the same structure as that in the table generated by the **IVS-ITEM** query, and if so, the duplicate **IVS-ELEM** strips are ignored.¹¹ At the end of Stage 1, the parser will have a complete ordered set of vertical strips, including both common and uncommon ones, and then it moves to Stage 1a.

In **Stage 1a**, the parser takes the **n** best initial vertical strips from the **DB-IVS** table and generates built structure trees for them. At this point, therefore, the parser will have the **n** best built structures ready to take forward to the next stage. In Chapter Seventeen we will see how we came to determine the optimal value of this and the other configurable parameters. As it builds these trees, **node** and **level probabilities** (see Section 15.1.7) are calculated using the results of **I2E**, **I2E2U2E**, **E2U** and **U2E** queries, together with the probability that the element can be first within its unit, by using **unit structure (US)** queries (see Section 14.1 of Chapter Fourteen and Section H.1.5.1 of Appendix H).

Any added built structure trees are set to **active** (see Section 15.1.3), and the parser uses a **forward unit structure (FUS)** query to populate the **forward predictions** with the elements of structure that can follow the last element in each unit in the built structure's **rightmost strip**.

Stage 1a is a **backtrackable** state, i.e. it is a state to which the parser may backtrack. If it does, it will take the next **n**-best initial vertical strips from **DB-IVS** table, and process them. Should there be no more strips to process, the parser moves to Stage 7 and registers a success or failure.

⁹ It does not generate built structures at this point as this is computationally expensive and would cause a significant delay in the parse time.

¹⁰ The **IVS-ITEM** items also have to have an **IVS-ELEM** query to cover the less frequent uses of the item.

15.2.3 Stage 2: Building a candidate structure

Summary: Stage 2 grows any item (other than the initial item) in the sentence upwards by one element and one unit - and in some cases by an additional element and unit - to create a candidate structure.

Stage 2 involves creating a set of candidate structures for the next item in the input string. It starts by retrieving the item and requesting an **I2E** query for it (see Section 14.1 of Chapter Fourteen and Section H.1.2 of Appendix H).

The **I2E** records indicate which item-element pairs need an **I2E2U2E** query. For non-**I2E2U2E** item-elements, candidate structures that contain the item and element are created and probabilities are assigned. **I2E2U2E** queries are requested for item-elements that need it, and candidate structures are created for each item-element-unit-element quadruple.

To complete Stage 2, a **unit-up-to-element (U2E) query** is requested and each candidate structure is grown to add the appropriate its unit (see Section 14.1 of Chapter Fourteen and Section H.1.4 of Appendix H). Where an element (such as a linker) can belong to more than one unit, the candidate structure tree is copied before growing it, so that there will be one candidate structure tree for each unit of which the top element can be a component of.

At this point, the parser requests a **backward unit structure (BUS) query**, in order to find the elements that can occur before each element in the candidate structure (see Section 14.1 of Chapter Fourteen and Section H.1.5.2 of Appendix H). These probabilities are stored in the **BACKWARDPREDICTIONS** table (see Section H.2.2 of Appendix H).

¹¹ Any **IVS-ITEM** generated vertical strips will therefore override any **IVS-ELEM** generated ones.

15.2.4 Stage 3: Attempting to join a candidate structure to a built structure (joining two sibling elements in a unit)

Summary: Stage 3 is broken down into two sub-stages, Stage 3 and Stage 3a. Stage 3 creates a list of potential joins between the active candidate structures (created in Stage 2) and the active built structures, and assigns a joining score to them. Stage 3a takes the n most likely of these joins that are equal to, or greater than, the join threshold value, and makes them.

Stage 3 is the key stage. It is at this point that the parser attempts to join candidate structures to built structures. It is divided into two sub-stages. The first (Stage 3) is responsible for calculating joining scores, and the second (Stage 3a) uses the scores to create new built structures that represent the joins.¹²

To calculate the joining score, the parser starts by using a SQL query to select all unit nodes in all active candidate structures. It then traverses the nodes, and for each it requests a further query for all matching unit nodes in the right-most strips of the active built structures.

In the rest of this description of Stage 3, the diagrams in Figures 15.6 and 15.7 will be at least as important in explaining the concepts as the verbal description in the text.

Please see Figure 15.6. A match is found if:

- (a) the **forward predictions** from the unit in the **built structure** indicate that the element in the candidate structure can **follow** the last element found in the built structure unit, and
- (b) the **backward predictions** from the **candidate structure** unit indicate that the last element in the built structure unit can **precede** the first element in the candidate structure's unit.

¹² It is at the start of Stage 3 that the **CYCLE** field of all tree-pairs that may be involved in a join is incremented. This means that if the tree-pair now has a **CYCLE** field value above the threshold (3), and if after a failure of a subsequent co-ordination join, the parser will backtrack rather than try another Stage 3 (see Figure 15.5).

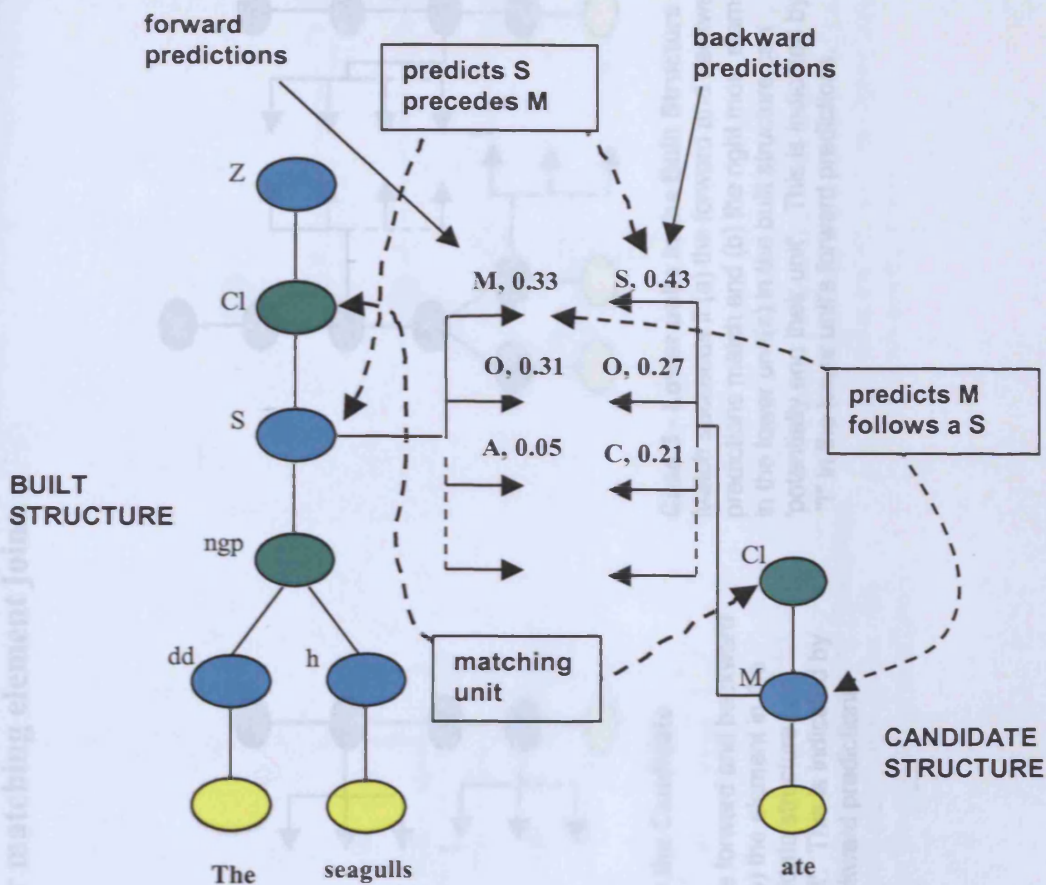


Figure 15.6: Identifying potential joins with forward and backward predictions (the dotted lines indicate the successful match criteria)

There are five cases in which a match is successful, and these are shown in Figure 15.7. The simplest case (Case 1 in Figure 15.7) is when there are no lower units in the built structure or in the candidate structure, and the match succeeds if the forward and backward predictions match with respect to their elements as described above.

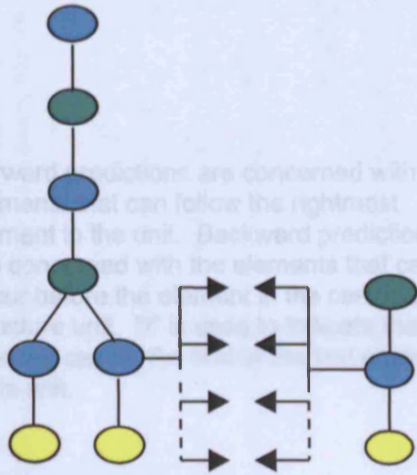
The second case is when there are lower units in the candidate structure, and therefore the match can only succeed if the left-most element in all lower candidate structure units can 'potentially start' the unit (signified by the ! in Case 2, Figure 15.7).

The third case is when there are lower units in the built structure. Here, the match can only succeed if the right-most element in all lower built structure units can **potentially close** the unit (signified by ! in Case 3, Figure 15.7).

The fourth case is when there are lower units in both the built structure and in the candidate structure, in which case both the right-most elements in all lower built structure units must **potentially close** their units, and left-most elements in all lower candidate structure units must **potentially open** their units (see Case 4, Figure 15.7).

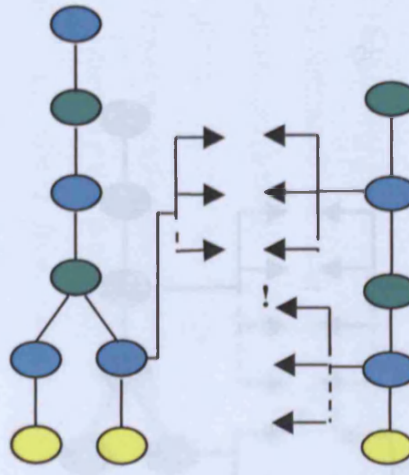
The five cases for matching element joins

249



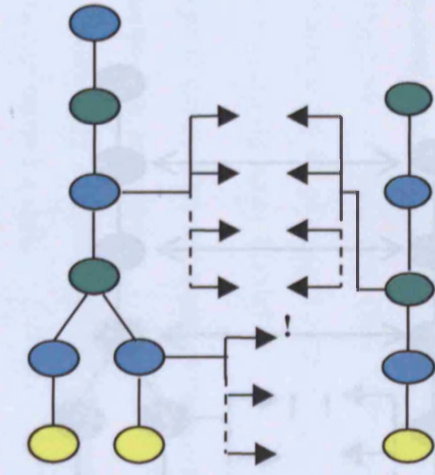
Case 1 - No lower units

Match succeeds if the element in the candidate structure is in the built structure's forward predictions and the rightmost element in the built structure is in the candidate structure's backward predictions.



Case 2 - Lower units in the Candidate Structure

Match succeeds if (a) the forward and backward predictions match and (b) the element in the lower unit(s) of the candidate structure can 'potentially start their unit'. This is indicated by "!" in the lower unit's backward predictions.



Case 3 - Lower units in the Built Structure

Match succeeds if (a) the forward and backward predictions match and (b) the right most elements in the lower unit(s) in the built structure can 'potentially end their unit'. This is indicated by "!" in the lower unit's forward predictions.

KEY:

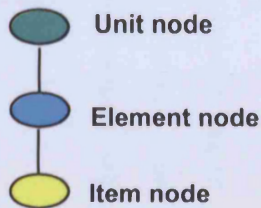
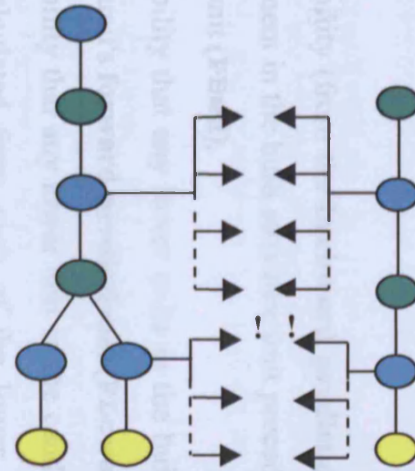
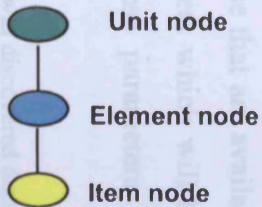


Figure 15.7: Element join cases and actions (Sheet 1 of 2)

The five cases for matching element joins (continued)

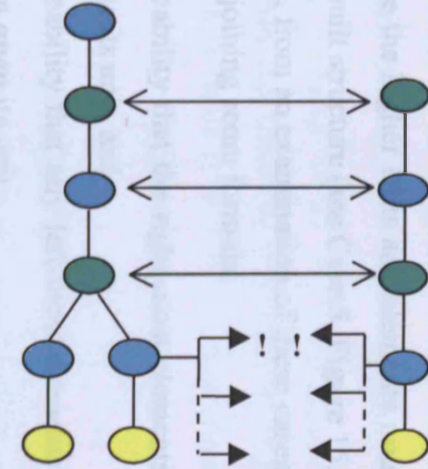
Forward predictions are concerned with the elements that can follow the rightmost element in the unit. Backward predictions are concerned with the elements that can occur before the element in the candidate structure unit. "!" is used to indicate that the element can be the first or the last element in its unit.

KEY:



Case 4 - Lower units in the Built Structure and Candidate Structure

Match succeeds if the forward and backward predictions match, and the lower units are closed (in the built structure) and started (in the candidate structure).



Case 5 - There are higher units in the Candidate and Built Structures

Match succeeds if the forward and backward predictions match, and the higher units and elements are the same.

Figure 15.7: Element join cases and actions (Sheet 2 of 2)

The final case is when the candidate structure has higher units above the matching unit. In this case the higher units and elements must also match the higher units and elements in the built structure (see Case 5, Figure 15.7).

You can see, from an examination of these cases, that it is important to include the following in the joining score formula:

- (a) the probability that the right-most elements in any lower built structure unit can **close** its unit, and
- (b) the probability that any left-most elements in any lower candidate structure units can **open** its unit.

The joining score is calculated by using a **joining strength formula**, which accepts the following parameters:

- (a) the probability (from the **forward predictions**) that the given element in the candidate structure follows the last element in found in the built structure unit (**PFwd**),
- (b) the probability (from the **backward predictions**) that the last (i.e. the right-most) element in the built structure unit precedes the element in the candidate structure unit (**PBwd**),
- (c) the probability that any lower units in the built structure are closed (each of the lower unit's **forward predictions**) (**PLowerBSClosed**)¹³,
- (d) the probability that any lower unit in the candidate structure can start its unit (this is calculated from each of the lower unit's **backward predictions**) (**PLowerCSOpen**),
- (e) the level and node probabilities in the matching nodes of the built and candidate structures (**PBSLevel**, **PBSNode**, **PCSLevel** and **PCNode**)
- (f) the tree probabilities of both the built and candidate structures (**PBSTree** and **PCSTree**).

This set of parameters is the complete set of those that are available in the Parser WorkBench, where it is possible for the user to select which will be involved (see Section 15.3). The various configurations of these parameters with which we

¹³ The probability that lower units are closed was a concept that was discovered during the development of the chart parser (see Chapter Twelve). It is calculated by taking the product of the probabilities of each of the lower units forward (or backward) predictions containing the end or start of unit (!). If there are no lower units, a value of 1 is returned.

experimented during the development of the parser are compared and evaluated in Chapter Seventeen. One of the formulae using some of the parameters is as follows:¹⁴

$$JS = ((P_{Fwd} + P_{Bwd}) / 2) * ((P_{LowerBSClosed} + P_{LowerCSClosed}) / 2)$$

The parser then uses the score to identify those joins that have a value that is equal to the join threshold value or is above it, their details are added to the **parser state table** (see Section 15.1.5) so that the tree-pairs are available in the next part of the parsing algorithm - Stage 3a.

Some candidate structures may be unable to be joined to any built structures. The reason will either be because no join was possible, or because a join was possible but its score was below the threshold value. In such cases they are marked in the parser state table with a **NEXTSTATE** value of '**Stage 4**', and the effect is that the candidate trees will be grown before they are considered for a further join at a later stage of the parsing algorithm.

Stage 3a provides for those cases in which there are trees that can be joined and that have a join score of over **x%**. In Stage 3a, the parser takes the **n**-best of them (or all of them, if there are less than **n** equal to, or above the **join threshold value**) from the parser state table and joins them, so creating up to a maximum of **n** new built structures.¹⁵

When the trees are joined, the new built structures are made **active**, and the **node and level probabilities** for the nodes in the new structure are calculated (see Section 15.1.7). To provide a further aid to understanding this complex procedure, Figure 15.8 gives a 'walkthrough' of stages 1 to 3 and shows how the probabilities are calculated.

¹⁴ We also used the built structure score and candidate structure score to rank the new built structures (see Chapter Seventeen).

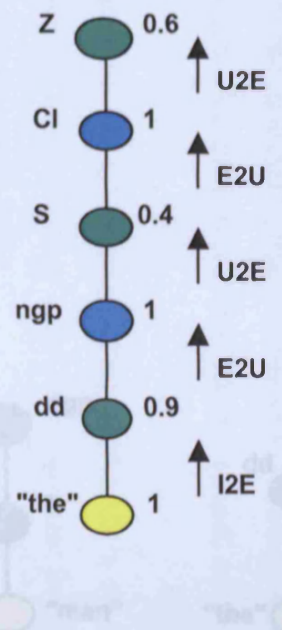
¹⁵ If there are more than **n** possible joins, the remainder will be taken if the parser backtracks to the current move.

5 Stage 3 determines that a join is possible and assigns it a score (this determines the order in which the

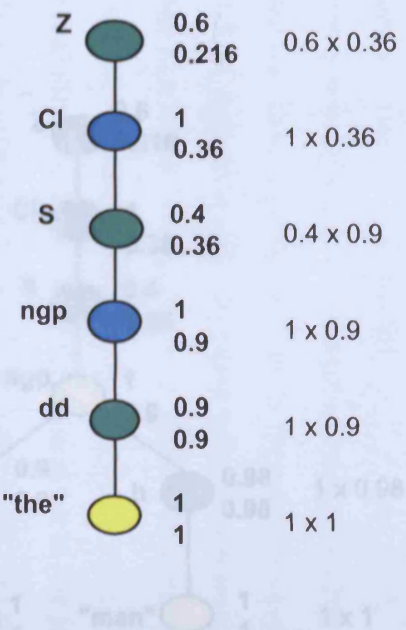
1. The parser has created this BS from an IVS



2. It calculates the node probabilities...



3. ...and the level probabilities



4. Stage 2 creates a number CS trees, this is one of them:

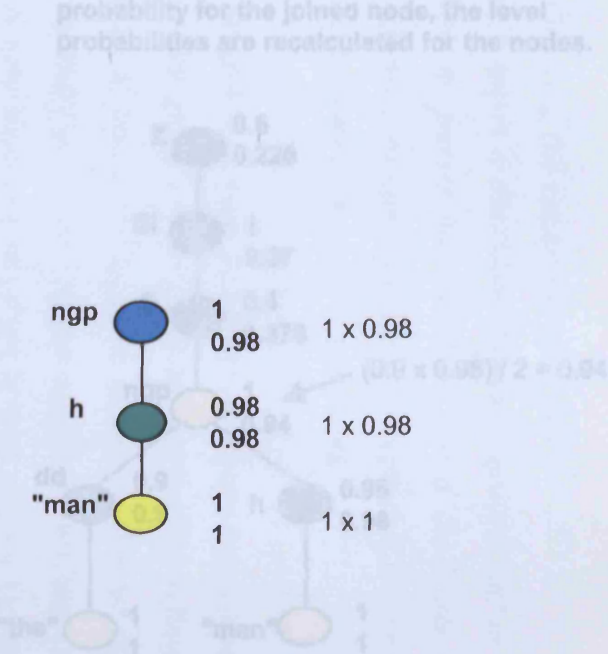
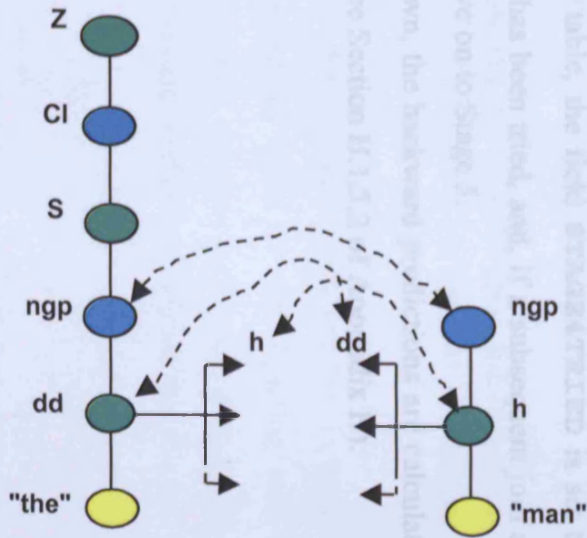
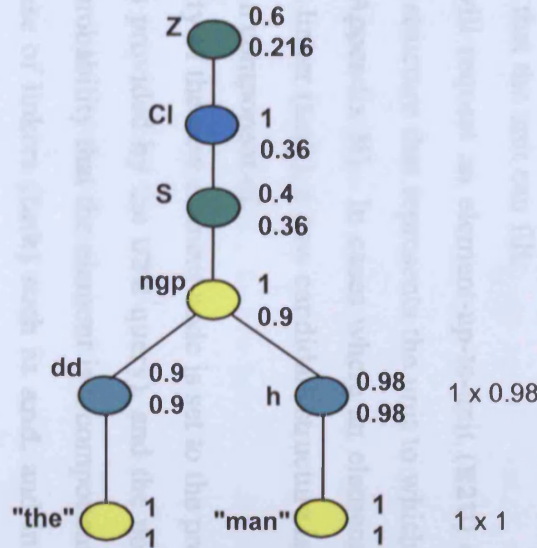


Figure 15.8: A walkthrough of Stages 1 to 3 (showing how the probabilities are calculated) (Sheet 1 of 2)

5. Stage 3 determines that a join is possible and assigns it a score (this determines the order in which the joins will be made)



6. Stage 3a makes the join, and.....



7. ...recalculates the node and level probability by taking the mean of (a) the BS level probability (for the ngp node) and (b) the level probability of the CS root node. After assigning the new level probability for the joined node, the level probabilities are recalculated for the nodes.

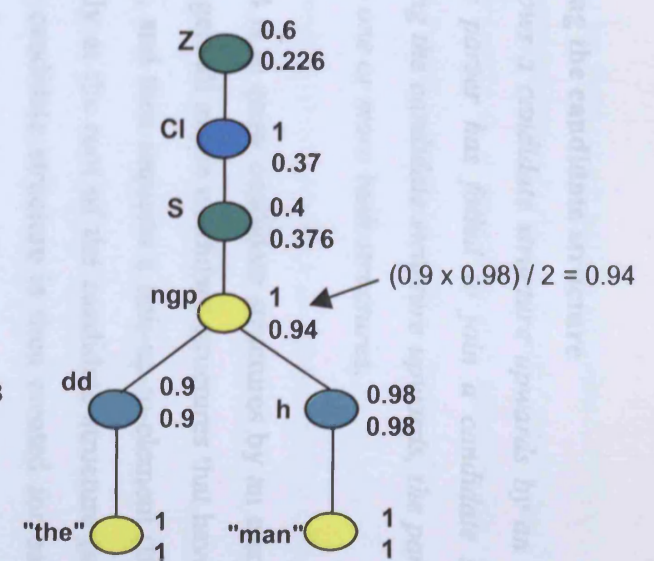


Figure 15.8: A walkthrough of Stages 1 to 3 (showing how the probabilities are calculated) (Sheet 2 of 2)

15.2.5 Stage 4: Growing the candidate structure

Summary: Stage 4 grows a candidate structure upwards by an element and a unit, and is used when the parser has failed to join a candidate structure to a built structure. After growing the candidate structure upwards, the parser then attempts to join the new element to one or more built structures.

The purpose of Stage 4 is to grow candidate structures by an element and a unit. To do this, the parser first gets all active candidate structures that have their **NEXTSTATE** field set to '**STAGE 4**', and then requests a unit-up-to-element (**U2E**) query based on the unit that is currently at the root of the candidate structure (see Section H.1.4 of Appendix H). A new candidate structure is then created for each element that the results of the query say that the unit can fill.

Next, the parser will request an element-up-to-unit (**E2U**) query, and a node is added to the candidate structure that represents the unit to which the element belongs (see Section H.1.3 of Appendix H). In cases where an element can belong to more than one unit, such as a linker (**Lnk**), a new candidate structure is created for each unit that the element can be a component of.

The node probability of the new element node is set to the probability that the unit can fill the element (as provided by the **U2E** query), and the node probability of the unit node is set to the probability that the element is a component of that unit. This is 100%, except in the case of linkers (**Lnk**) such as **and**, and inferers (**inf**) such as **only**.

In the parser state table, the field **STAGE4TRIED** is set to the value **true** to indicate that a Stage 4 has been tried, and, if a subsequent join attempt also fails, this forces the parser to move on to Stage 5.

After trees are grown, the backward predictions are calculated for the new nodes using the **BUS** query (see Section H.1.5.2 of Appendix H).

15.2.6 Stage 5: Joining by the co-ordination of units

Summary: Stage 5 comprises two sub-stages, Stage 5 and Stage 5a. Stage 5 identifies potential joins between units in a co-ordination relationship, and assigns them a joining score. Stage 5a makes the n most likely of these joins.

15.2.6.1 An overview

Stage 5 is concerned with a second and less frequent type of joining. This is the joining of two co-ordinated units that fill the same element, as in **Ivy and her friend**. Any successful joins will be **unit-joins** since they attach a unit from one of the candidate structures to a unit in one of the built structures. Stage 5 is divided into two sub-stages, as in earlier stages. Stage 5 calculates the co-ordination joining score, and Stage 5a takes the top n of these potential joins and makes them.

Because only a small proportion of joins are the result of co-ordinated units (below 0.5% in the FPD Corpus), the parser works more efficiently if, after growing the candidate structure in Stage 4, it first attempts an **element-join** as in Stage 3, before moving to Stage 5 to attempt a join based on co-ordination. The algorithm, therefore, includes a check after performing Stage 3 to see if a Stage 4 'grow' operation has been performed before for this particular tree-pair (i.e. the built structure and the candidate structure).

Please now consult Figure 15.5. It shows that the parser moves to Stage 5 only after a failure of a Stage 3 'join' that was attempted after a previous Stage 4 'grow'. If a sequence of **element-join** (Stage 3), **grow** (Stage 4), a further **element-join** and then a **co-ordinated unit-join** (Stage 5) has failed three times, that the parser will move to Stage 6 for backtracking.¹⁶

15.2.6.2 Stage 5: Calculating the co-ordination joining score

The Stage 5 algorithm includes tests for success or failure. The algorithm is as follows:

- (a) For each unit in the candidate structure that is both active and has a first element that potentially starts the unit (starting with the most likely one):
retrieve all active built structure unit nodes that are (i) in the rightmost strip and (ii) whose forward predictions indicate that the unit is potentially closed.

¹⁶ The number of cycles before a backtrack can be set in the **Parser WorkBench**.

- (b) For each unit returned, the parser applies the following tests, and sets the **bLinker**, **bUnitSame** and **bPunct** variables accordingly (note that these Boolean variables are set to **false** before the tests are performed).

The **linker test**: Set **bLinker** to **true** if the element in the candidate structure below the unit being considered for a unit-join is a linker (**Lnk**).

The **same unit test**: Set **bUnitSame** to **true** if the unit being considered in the built structure is the same as the unit being considered in the candidate structure (e.g. both are nominal groups).

The **comma or semi-colon test**: Set **bPunct** to **true** if the last element in the built structure unit that is being considered is either a comma or a semi-colon.

After the tests:

- (a) if **bLinker** is **true** and **bUnitSame** is **true** (irrespective of the value of **bPunct**), set the join probability to 90%.¹⁷ An example of a join of this category is two clauses, the second of which starts with a linker (e.g. **my brother and his friend** or **Robin and Margaret**).
- (b) if **bLinker** is **true** and **bUnitSame** and **bPunct** are both **false**, set the join probability to 0.1%. This covers, for example, a quality group and a prepositional group in a co-ordinated relationship (which occurs rarely e.g. **very slowly and with great care** where the two units fill a manner Adjunct).
- (c) if **bLinker** and **bPunct** are **true** and **bUnitSame** is **false**, set the joining probability to 0.1%. An example would be a quality group (**qlgp**) which has a comma as the last element followed by a prepositional group which starts with a linker (e.g. **very slowly, and with great care**).
- (d) if **bUnitSame** and **bPunct** are **true**, but **bLinker** is **false**, set the joining probability to 50%. This covers, for example, two nominal groups separated by a comma.
- (e) if **bUnitSame** is **true** and **bLinker** and **bPunct** are both **false**, set the joining probability to 0.1%. This covers, for example, two nominal

¹⁷ The threshold percentages can be set in the parser workbench.

groups that are not separated by a comma, where there is no linker (e.g. the first two nominal groups in **my brother his friend and I**).

- (f) if **bPunct** is **true** and **bUnitSame** and **bLinker** are both **false**, set the joining probability to **0.1%**. This covers, for example, a quality group that ends with a comma co-ordinated with a prepositional group (e.g. **very slowly, with great care**).¹⁸

The assigned joining probabilities are stored in the PST, and are marked as needing to go to **STAGE 5a** (i.e. the **NEXTSTATE** value). Any impossible tree-pair joins are also marked as such in the PST, and for these tree-pairs the value of the **NEXTSTATE** field will be set to '**STAGE 4**', if the value of the **CYCLES** field in the PST record for the Candidate Structure is less than 3 or '**STAGE 6**' if the **CYCLES** field is 3 or more.

15.2.6.3 Stage 5a: Making the co-ordination Joins

For any tree-pairs that can be joined in a co-ordination relationship, the parser will create new built structures for the top **n** potential joins. New PST records will show that the joins have been made and that their **NEXTSTATE** is '**Stage 2**'. The PST records that represent the joins that have been made will have their **FOLLOWED** field set to **true**, and any remaining potential joins are still available in case a backtracking operation should be needed from a later move.

15.2.6.4 The join cycle

The movement of the parser through all five of Stages 3, 4, 3, 4 and 5 is defined as a **join cycle**. The number of iterations for a tree-pair through the join cycle is recorded in the Parser State Table (PST). This is used to determine whether, after a Stage 5 has failed, the parser moves to Stage 6 or back to Stage 4. A further field in the PST states for each tree pair whether a Stage 4 has been attempted, and this will guide the parser to Stage 5 or Stage 4 depending on the field's value.¹⁹

If Stage 5 succeeds, the parser moves on to a further application of Stage 2 and so takes the next item from the input sentence.

¹⁸ More testing is needed to check that these initial 'linguist's estimates' of what the joining score should be are reliable over the full range of types of co-ordination.

¹⁹ Note that a move from Stage 3 to Stage 5 will reset the **STAGE4TRIED** variable for the pair in the PST.

15.2.7 Stage 6: Backtracking

Summary: Stage 6 is concerned with going back to an earlier point in the parsing process after the parser finds that it has made a mistake, or when the parser has finished, and either (a) the user has asked for more analyses, or (b) less than the required number of analyses have been found (as indicated by Stage 7).

Stage 6 provides currently, only for what we will call **computationally motivated backtracking**. It enables the parser to go back and follow any routes through the parser's workflow that have not yet been followed. It is forced to backtrack under the following conditions:

- (a) a Stage 4 has failed to grow by an element and unit,
- (b) Stage 5b has failed to join after three iterations of the join-grow-join-coordination cycle, or
- (c) in Stage 7, either the parse has processed the last item and failed to find a successful parse OR the user has asked for more than one analysis to be created.

Backtracking is performed by moving the parser back to the state it was in for an earlier move, and by then taking the next **n**-best routes to a conclusion (or possibly failure). To do this, it copies the next **n**-best records from the PST that are marked as not having been followed into a new move in the PST. After de-activating all trees, the parser reinstates any trees that were active for that move. The records in the earlier move are then marked as being followed.

Backtracking can lead back to the following states:

- (a) Stage 3a: to take the next **n**-best **element-joins** that have not yet been made,
- (b) Stage 5a: to take the next **n**-best **unit-joins** that have not yet been made,
- (c) Stage 1a: to take the next **n**-best **initial vertical strips** into built structures.

The parser backtracks to the most recent move first, irrespective of the state in the workflow that it represents.

The method of backtracking used here is an application of a solution to the **Hamilton Path Problem** in graph theory (Ore 1960). This is used in classic computer science backtracking problems such as the 'Knight's Tour of the Chessboard' and the 'Travelling Salesman Problem'. These algorithms and others are widely described in the literature (e.g. Wirth 1976).

As such, the algorithm as described here for Stage 6 is therefore not **linguistically-motivated**. In other words, it backtracks blindly to each of the previous moves, one at a time, until it finds a parser state record that contains possibilities for taking one or more alternative paths (i.e. one that has its **FOLLOWED** field set to **false**).

A more intelligent approach to the problem of a failed parse is to use linguistic knowledge of known 'attachment problems' of the type found in garden-path sentences. Such **linguistically-motivated backtracking** enables the parser to go directly to the most likely backtrack point which is determined using linguistic knowledge, rather than just taking a previously 'not followed' route. For a description of how this type of backtracking will work see Chapter Eighteen and Appendix I.

Finally, however, we should note that a mistaken parse that is NOT syntactically unacceptable will be occasionally produced, e.g. **when she cut the bread with a carving knife** is analysed with **with a carving knife** as a qualifier to **the bread**. Ultimately, it is only our beliefs about the World that can tell us that 'a food' such as 'bread' is extremely unlikely to possess a knife (though many linguists (e.g. Chomsky 1965) have explained the possibility of modelling such knowledge as part of semantics). In such cases, both the two possible parses should be passed to the semantic interpreter, and so in turn to the belief system.

15.2.8 Stage 7: Determine if more analyses are required

***Summary:** Stage 7 is used when the parser reaches the end of the sentence and there are no more items to parse. It performs any final processing that is required to determine the most likely scores for complete trees before it seeks further analyses if they are required (by invoking backtracking).*

Stage 7 may be reached at any point after the parser has reached Stage 2 and it finds that there are no more items in the input string. There is a parser parameter that can be set in the Parser WorkBench that indicates the maximum number of analyses required before the parser stops. If the number of 'complete' trees is equal to this value when Stage 7 is reached, then the parser stops. If the parameter is not set in the Parser WorkBench, the parser will stop at Stage 7 and display the complete trees in the Parser

WorkBench's Overview facility, and then asks the user if more are required. If the parser continues at Stage 7, it moves to Stage 6 to perform backtracking.

15.3 The Parser WorkBench

The **Parser WorkBench**, as will be clear from the occasional references to it in the above descriptions, is the development environment in which the parser runs. The user can change certain parameters in the algorithm, and then study the intermediate results that the parser produces at each step of an incremental parse. Certain data can also be changed mid-parse to allow the user, for example, to force the parser down a different route. The parser workbench is fully described in Chapter Sixteen.

15.4 Summary

This chapter has described the parsing algorithm in detail, and we have seen how the parser uses the **probabilities tables** and the **parser working tables** that are described in Chapter Fourteen and Appendix H. I have described the way in which the algorithm operates in the parser's **State Transition Machine**, and how the trees (or tree-pairs) flow around the states of the STM, in the parser's **workflow model**.

We have also looked in much more detail at the different stages through which the parser goes in the context of the parser's workflow. Chapter Sixteen describes the Parser WorkBench, and it will further illustrate the operation of the parser by providing a walkthrough. More detailed examples of walkthroughs are given in Appendix K.

Chapter Sixteen

The Parser WorkBench

This chapter describes the **Parser WorkBench**. This is the environment in which the Corpus Consulting Probabilistic Parser is implemented, and it forms the interface with the user - i.e. a researcher developing the parser, or an observer who wishes to study it. I include a 'walkthrough' of the parsing of a simple sentence, so complementing Chapter Fifteen. The more detailed walkthroughs in Appendix K provide further examples of how the parser works.

In Chapter Fifteen we saw that the parser is implemented as a state transition model which comprises seven distinct states. The Parser WorkBench allows the user to either (a) work through those states and examine the results at each point (using a 'step-by-step incremental parse') or (b) run the parse in full.

The Parser WorkBench also allows the user to adjust the **configurable parameters** before entering a sentence and initiating the parse and it allows the user to review the results and, provided that it is done in a controlled way, to modify them at the end of each stage.

The chapter starts by discussing the incremental properties of the parser, and then it discusses the advantages of the Parser WorkBench as a research tool. Next it gives a detailed account of the parameters that can be configured for each parse. In order to give the reader an idea of the look and feel of the environment, I provide a simple walkthrough with screenshots.

16.1 Step-by-step incremental parsing

Recall from Section 10.2.5 of Chapter Ten that there are two different definitions of the term **incremental parsing**. The first (as found in Wirén 1989 and Weerasinghe 1994) defines a piece-meal approach, in which the parser builds up structures based on what it has seen from the input so far, operating in a left-to-right fashion. The second definition (e.g. Yang 1994) relates to the reuse of structures that have already been built.

The parser described here certainly satisfies the criteria for the first definition, since it takes a strictly incremental step-by-step approach. It accepts input from the user and analyses it, item by item, building each item upwards and joining parse trees as it proceeds. In some ways, it also satisfies the criteria for the second definition,

although somewhat more loosely. It does this in two ways: (a) when the parser backtracks, it reuses structures that have already been built, and (b) the Parser WorkBench allows the user to save a parse (or a part of a parse) and to reload it, modify it, and reuse it in a later operation.¹

16.2 The value of the step-by-step incremental approach

In this section I discuss the way in which the incremental facility in the Parser WorkBench provides a valuable research tool for developing both the parsing algorithm and the probabilities tables that support it.

16.2.1 How it works

The Parser WorkBench allows the user to stop at the end of each stage of the parsing algorithm, and examine (and change) the database tables that form the working data structures of the parser and change the user configurable parameters.

16.2.2 Its use in the development of the new parser

Whenever the parser makes an unexpected decision, the user is able to step back and determine why it had made the choice it did. Normally, this is done to: (a) identify possible bugs in the program, or (b) identify the cause of an unexpected analysis which is due to the data. The latter caused us to review the contents of the indexes and the corpus itself. Where the problems were due to the coverage of the corpus, we were able to identify the ways in which we would need to improve the probabilities tables (i.e. for the Version Two tables described in Section 14.2 of Chapter Fourteen and Appendix I). Some of the results of the testing resulted in changes in the algorithm itself, and any changes we made could themselves be also tested in this way by using the incremental facility. Therefore, the Parser WorkBench was central in the scientific development of the parser.

16.2.3 Speeding-up research

This section covers the ways in which the Parser WorkBench allowed faster research.

16.2.3.1 Changing a sentence in mid-parse

During our research we often needed to parse similar sentences to test alternative attachments, e.g. to the different probable attachments of a prepositional group. Using

¹ However, the parser does not fully fit the 'data reuse' concept of the second definition of incremental parsing (e.g. Yang 1994), because it only reuses what has already been constructed in a truly left-to-right fashion, with no gaps. That is, it can reuse information as it steps backwards through the state

the **save parse** options, we were able to save what the parser had already done and modify the parts of the input sentence that had not yet been processed.² This had the beneficial effect of speeding up our work, as we did not have to wait for the parser to perform again tasks that it had already carried out for the similar sentences. Consider the following sentences:

S1: I saw the old man with the telescope

S2: I saw the old man yesterday

S3: I saw the old man and his wife at the football match

In each case, we were able to save the parse as soon as it had finished processing **man**, and we were able to change the input sentence for each different sentence after the item **man**.

16.2.3.2 Altering probabilities and joining scores

One of the most useful facilities provided in the Parser WorkBench is that it enables us to change probabilities and joining scores at the end of each step. This was particularly useful on those occasions when we realised that the results returned for a query were not as expected, due to a misanalysis in the corpus or the occurrence of rare structures, so giving undue weight to a particular result. In these cases, we were able to alter the path that the parser took by adjusting the scores, without the immediate need to make a change to the probabilities tables. Furthermore, we can add or delete records in the parser's working tables and, provided that we do this in a controlled manner such that it does not upset the normal parsing process, this becomes a very useful tool for speeding-up research.

We might want, for example, to have a rare construct considered before its probability suggests that it should in a normal parse run and thereby avoid having to wait for the parser to reach the required construct in the normal order of processing. Sometimes these rare constructs may be only reached after lengthy backtracking operations. By deleting PST entries or by modifying probabilistic scores, we can see the results of our tests much sooner.

transition model, but it does not reuse information to the right of the current position of the parser when it has backtracked.

² Note that the save function described in this section does not save the sentence to the corpus database. This is handled by the Commit function (see Section 16.5).

16.2.3.3 Altering the configurable parameters in mid-parse

By changing the configurable parameters in mid-parse, we are able to see the effects on particular constructs. For example, by changing the value of **n** for the **n**-best trees to take forward to the next stage, we have enabled the parser to consider a construct without the need for backtracking. However, there are some parameters which we would not wish to change in mid-parse, one example being the configuration of the parameters for the algorithm used for calculating a joining score.

16.2.4 Answering 'what-if' questions

By changing the scores or adding new records to the PST, or by changing the probabilities assigned to built and candidate structure trees, we are able to obtain answers to **what-if?** questions before deciding to make a proposed change in the probabilities tables permanent. This could be the result of, say, an unexpected analysis, or a join to an incorrect unit, and we might want to see the effects of preferring the alternative.

16.3 The configurable parameters

In this section, we define the parameters that can be changed at the start of the parse or during it. Note that the user is able to save the values of the parameters so that they can be used for subsequent parses, or to restore them from defaults or from a previously saved set (see Figure 16.1, Item 5). The parameters that can be changed are defined below. The use of these parameters and their optimum configurations and values are described in Chapter Seventeen. Figure 16.1 shows the user interface that is used to change these.

16.3.1 The number of trees to create in Stage 1 (Figure 16.1, Item 1)

This value controls the maximum number of trees that the parser creates in Stage 1. Note that more will be created, should a backtrack event to Stage 1 be needed.

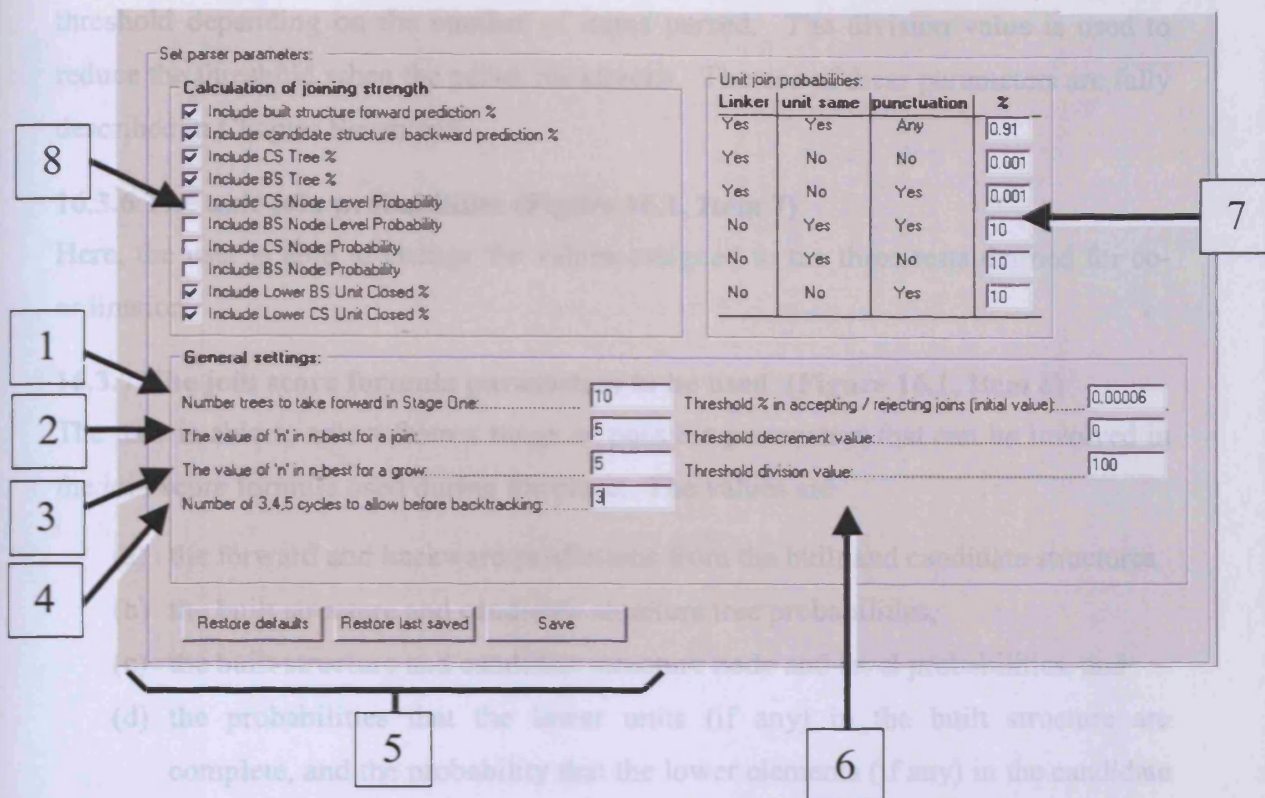


Figure 16.1: The user interface for changing the parser's configurable parameters

16.3.2 The value of n for best- n joins (Figure 16.1, Item 2)

The value of n can be changed here. This controls the number of the 'most likely' tree-pairs to take forward to the next stage after Stage 3a.

16.3.3 The value of n for best- n to grow (Figure 16.1, Item 3)

The value of n for the number of candidate structures to grow in Stage 4 following a failed join attempt can be set here.

16.3.4 The number of cycles of Stages 3, 4, 3 and 5 to allow before backtracking

The number of iterations of Stages 3, 4, 3 and 5 that are allowed before the parser is forced to backtrack can be set here.

16.3.5 The threshold probability for accepting / rejecting join attempts (Figure 16.1, Item 6)

This is the lowest probability ($x\%$) for which the parser will stop considering potential joins to take forward to the next stage from Stage 3. If less than the n value joins over $x\%$ are found, then only those joins over $x\%$ will be taken forward. The initial value sets the value before backtracking occurs. The decrement value is used to reduce the

threshold depending on the number of items parsed. The division value is used to reduce the threshold when the parser backtracks. The use of these parameters are fully described in Chapter Seventeen.

16.3.6 The unit-join probabilities (Figure 16.1, Item 7)

Here, the user is able to change the values assigned to the three tests defined for coordination.

16.3.7 The join score formula parameters to be used (Figure 16.1, Item 8)

The user is able to select from a range of possible parameters that can be involved in the join score formula used during the parse. The values are

- (a) the forward and backward predictions from the built and candidate structures,
- (b) the built structure and candidate structure tree probabilities,
- (c) the built structure and candidate structure node and level probabilities, and
- (d) the probabilities that the lower units (if any) in the built structure are complete, and the probability that the lower elements (if any) in the candidate structure can open the units.

These values are discussed in Chapter Seventeen and Section 15.2.4 of Chapter Fifteen.

16.4 The user interface

16.4.1 The configurable parameters

The main user interface for the Parser WorkBench is shown in Figure 16.2, in which the tab showing the configurable parameters is displayed. Descriptions of the functions indicated in Figure 16.2 are shown in Table 16.1.

Corpus Consulting Probabilistic Parser

The Corpus Consulting Probabilistic Parser

Input sentence:
I saw the man

Navigation: PST, Overview, Items, IVS, IZE results, Join candidates, XML Parse trees, **Parser parameters**, TreeView

Set parser parameters:

Calculation of joining strength

- Include built structure forward prediction %
- Include candidate structure backward prediction %
- Include CS Tree %
- Include BS Tree %
- Include CS Node Level Probability
- Include BS Node Level Probability
- Include CS Node Probability
- Include BS Node Probability
- Include Lower BS Unit Closed %
- Include Lower CS Unit Closed %

General settings:

Number trees to take forward in Stage One: Threshold % in accepting / rejecting joins (initial value):

The value of 'n' in n-best for a join: Threshold decrement value:

The value of 'n' in n-best for a grow: Threshold division value:

Number of 3,4,5 cycles to allow before backtracking:

Buttons: Restore defaults, Restore last saved, Save

Unit join probabilities:

Linker	unit same	punctuation	%
Yes	Yes	Any	0.91
Yes	No	No	0.001
Yes	No	Yes	0.001
No	Yes	Yes	10
No	Yes	No	10
No	No	Yes	10

Functions:

- Stage zero: Start a new parse
- Stage one: Stage one grows a sentence initial item up to the sentence node
- Stage two: Stage two grows the next item in the sentence
- Stage three: Join a non-initial element to the built structures
- Stage four: Growing a non initial unit
- Stage five: Joining a non-initial unit to the built structures (coordination)
- Stage six: Backtrack!

Buttons: Parse rest, <-Go back

What am i doing now?
I am waiting to start a new parse. Please enter a sentence and press the Parse button, or do a walkthrough parse by pressing the Stage Zero button.

Buttons: Save, Load, Reload saved parse, Commit

Annotations: 1-12

Figure 16.2: The main Parser WorkBench form with parser parameters shown

Item	Description
1	The 'Step' buttons (red / green show the current state). The user presses these at the end of each step after examining results and making changes if they are required.
2	Parse the complete sentence from the current state to the end of the sentence.
3	Go back a step button. This moves the parser back a step in the workflow.
4	The user feedback panel.
5	The Commit button, which is used to save the most likely or chosen parses to the database.
6	Save a parse at current state or load a parse at previous state.
7	Co-ordination score parameters.
8	The Save and Restore parameters button, which is used to save the parameter settings so that they can be used in subsequent parses.
9	General user configuration parameters.
10	The user configuration parameters for the join score formula.
11	The tabs that can be selected, each has a different purpose.
12	The sentence that the user has entered.

Table 16.1: Legend to Figure 16.2

16.4.2 Starting a step-by step incremental parse

In the Parser WorkBench, buttons are provided to start each stage of the parse. They are made available according to the current state of the parser. At any one time only one button is available, and this is shown by the presence of a small green icon to its left. The unavailable buttons are shown with red icons to their left. Note that, at any stage, the button labelled **Parse rest** (which equals 'the rest of the parse') (Figure 16.2, Item 2) can be pressed to complete the parse from that point, if it is pressed when Stage 0 is highlighted, the parser will perform the complete parse in non-step-by-step mode.

16.4.2.1 Stage 0

To start a parse, the Stage 0 button must be green (Figure 16.2, Item 1). When the button is pressed, the parser initialises itself and user is asked for a sentence to parse. In this example, we use the simple sentence **the man saw me**.

Next the user is given the choice of selecting an incremental parse, if it is not selected, the parser will perform the parse in non-step-by-step mode. Here, we will assume that a step-by-step parse is required. After the user enters the sentence, the parser populates the **ITEM** table and moves to Stage 1, and the Stage One button is available. Figure 16.3 shows the **ITEM** table displayed in the Parser WorkBench.

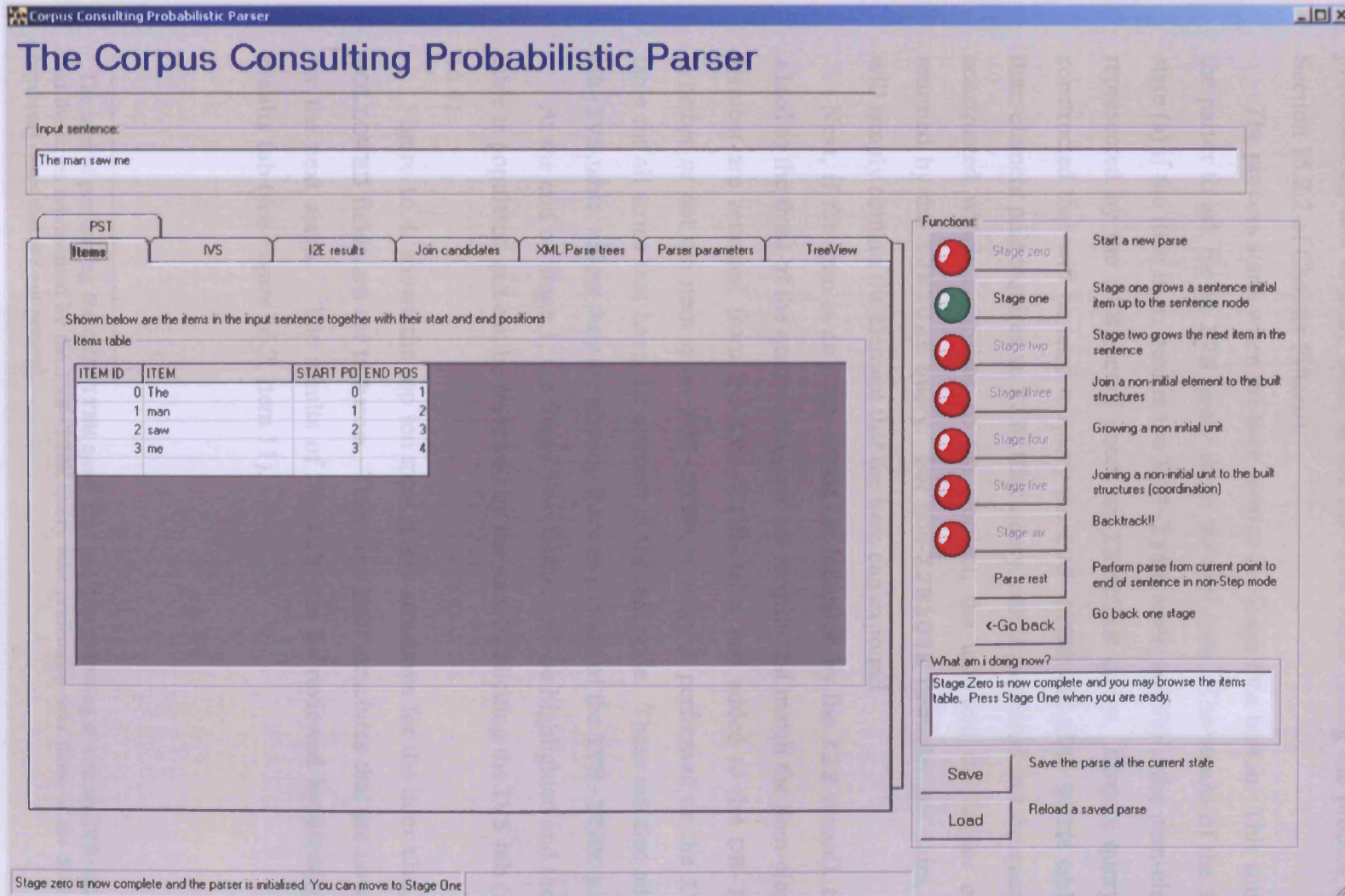


Figure 16.3: The user is displaying the item table

16.4.2.2 Stage 1

The purpose of Stage 1 is to create a list of initial vertical strips and their associated probabilities, and to store them in the **DB-IVS** table (using the process detailed in Section 15.2.2 of Chapter Fifteen).

The process starts when the user presses the Stage One button. This action causes the parser to ask for an **I2E** query about the first item. The results of the **I2E** query state (a) if the item is an item in the **IVS-ITEM** table, and (b) if the item-element pair represented by the **I2E** record needs an **I2E2U2E** query. First, a query string is constructed that will be used to query the **IVS-ITEM** and **IVS-ELEM** tables. If the item-element pair requires an **I2E2U2E** query, one is requested and the query string is constructed which comprises of the element, the unit and the higher element as returned by the **I2E2U2E** query. For non-**I2E2U2E** item-element pairs, the string will simply contain the element that the item can expound.

Next, if the item is an **IVS-ITEM** (as indicated by the **I2E** record), the item is added to the front of the query string and all records that match the item-element-unit-element are returned from the **IVS-ITEM** table and added to the **DB-IVS** table. Whether or not the item is an **IVS-ITEM**, a query is performed on the **IVS-ELEM** table for all strips that have the element at the leaf node. These are also added to the **DB-IVS** table, unless they are already there as a result of the **IVS-ITEM** query.³

At the end of Stage 1, the Stage Two button will be highlighted and the **DB-IVS** table is populated and can be reviewed by the user by clicking the **IVS tab** (see Figure 16.4).

Figure 16.4, shows the top ten most likely structures for the item **the**, and their **FOLLOWED** fields are set to **true**. These are built structures that are now available for the next stage. The results of the **I2E** can be reviewed by pressing the **I2E results tab** (see Figure 16.2, Item 11).

³ The idea of performing the **IVS-ITEM** query first and then following it with an **IVS-ELEM** query is that the strips represented by the **IVS-ITEM** query take preference over those of the **IVS-ELEM** and represent cases that are not general.

The Corpus Consulting Probabilistic Parser

Parsed: The

Input sentence:

The man saw me

PST

Items **IVS** I2E results Join candidates XML Parse trees Parser parameters TreeView

Shown below are the results of the IVS Element and IVS item queries performed by the parser

DB-IVS table

ITEM	VERTSTRIP	FOLLOWED	PROBABILITY
The	dd ngp S CIZ	True	0.125116021617302
The	dd ngp C CIZ	True	2.63637535954602E-02
The	dd ngp Voc CIZ	True	4.69749787745129E-03
The	dd ngp A CIZ	True	1.87598688528969E-03
The	dd ngp S C I C CIZ	True	1.15265786539112E-03
The	dd ngp A_Res CIZ	True	1.1389118973499E-03
The	dd ngp S C I A CIZ	True	3.15130367976524E-04
The	dd ngp C C I C CIZ	True	2.4288166735345E-04
The	dd ngp cv pgp C CIZ	True	9.52268307187816E-05
The	dd ngp C C I A CIZ	True	6.64025219503215E-05
The	dd ngp cv pgp A CIZ	False	6.10465221425598E-05
The	dd ngp qd ngp S CIZ	False	5.57540618075914E-05
The	dd ngp S C I S CIZ	False	3.68240256459078E-05
The	dd ngp Excl CIZ	False	3.5068516165875E-05
The	dd ngp td ngp S CIZ	False	2.17658913637239E-05
The	dd ngp A C I C CIZ	False	1.72829267646785E-05
The	dd ngp qd ngp C CIZ	False	1.17481864308107E-05
The	dd ngp pd ngp S CIZ	False	1.10972694412562E-05
The	dd ngp A_Res C I C CIZ	False	1.04924672276053E-05
The	dd ngp C C I S CIZ	False	7.75935428550559E-06
The	dd ngp rd ngp S CIZ	False	5.67805861662363E-06

Functions:

- Stage zero: Start a new parse
- Stage one: Stage one grows a sentence initial item up to the sentence node
- Stage two: Stage two grows the next item in the sentence
- Stage three: Join a non-initial element to the built structures
- Stage four: Growing a non initial unit
- Stage five: Joining a non-initial unit to the built structures (coordination)
- Stage six: Backtrack!!
- Parse rest: Perform parse from current point to end of sentence in non-Step mode
- <-Go back: Go back one stage

What am i doing now?

Stage One is now complete. You may browse the Items table, the IVS results, the I2E results, see the built structures or press the Stage Two button

Save: Save the parse at the current state

Load: Reload a saved parse

Stage one is now complete. You may browse IVS and I2E data. You can move to S Trees:10

Figure 16.4: Stage 1 is complete and the DB_IVS table is displayed

Figure 16.5: The root likely built structure is shown

The Corpus Consulting Probabilistic Parser

Parsed: the

Input sentence:

the man saw me

BUILTSTRUCTURE(ID=1) {}

```

graph TD
    Z --- Cl
    Cl --- S
    S --- ngp
    ngp --- dd
    dd --- the
    
```

- h (78.8273615635179%)
- mo (5.86319218241042%)
- h_rcc (2.99674267100977%)
- rel_mo (2.89902280130293%)
- l (2.89902280130293%)
- th_mo (2.44299674267101%)
- h_n (1.40065146579805%)
- qd (0.749185667752443%)
- q_mo (0.684039087947893%)
- h_p (0.260586319218241%)
- pd (0.260586319218241%)
- sit_mo (0.195439739413681%)
- q (0.162866449511401%)
- od (0.130293159609121%)
- aff_mo (0.130293159609121%)
- dd (3.25732899022801E-02%)
- v (3.25732899022801E-02%)
- sd (3.25732899022801E-02%)
- M (33.6493808049536%)
- OM (24.2066563467492%)

TRE	COM	PROBABILITY	TREEL	ACT	XML
1	False	0.169436269648497	BUILTS	True	<TR I

NOI	PAF	TOKEN	PROBABILITY	LEVELPROB
1	2	the	0.971723518850987	0.971723518850987
2	3	dd	0.9005546453852	0.875090128931
3	4	ngp	0.401588382119356	0.3514260290861
4	5	S	0.581460245617943	0.2043402651889
5	6	Cl	0.829186893203884	0.1694362696484
6	-1	Z		

Tree 1 [GO] Load

Functions:

- Start a new parse
- Stage one grows a sentence initial item up to the sentence node
- Stage two grows the next item in the sentence
- Join a non-initial element to the built structures
- Growing a non initial unit
- Joining a non-initial unit to the built structures (coordination)
- Backtrack!!
- Perform parse from current point to end of sentence in non-Step mode
- Go back one stage

What am i doing now?

Stage One is now complete. You may browse the Items table, the IVS results, the I2E results, see the built structures or press the Stage Two button

Save the parse at the current state
 Reload a saved parse

Stage one is now complete. You may browse IVS and I2E data. You Trees:10 Inactive trees: 0 Position: 0 Parsed To parse:

Figure 16.5: The most likely built structure is shown

Figure 16.5 (on the previous page) shows the most likely built structure in the Parser WorkBench tree view. The forward predictions from the deictic determiner (dd) are shown in the left pane by right arrows with the next predicted element and its probability.⁴ The right-hand pane shows details of the tree and its nodes.

The parser is now at the end of Stage 1a and is waiting for the user to start Stage 2 by pressing the Stage Two button.

16.4.2.3 Stage 2

The user now presses the Stage Two button. In this stage the parser gets the next item and requests an I2E query for it and then builds candidate structures based on the results. The candidate structure may contain an item, an element and a unit or, if the item was an I2E2U2E item, then it will contain an extra element and unit (in both cases with the uppermost unit being the result of a U2E query). Figure 16.6 shows the Parser State Table (PST) after Stage 2 is complete.

The screenshot shows the 'The Corpus Consulting Probabilistic Parser' window. The 'Input sentence' is 'The man saw the'. The 'Parsed' text is 'The man'. The 'PST' table is visible, showing three entries for 'MOVEID'. The 'Functions' panel on the right includes buttons for 'Stage zero', 'Stage one', 'Stage two', 'Stage three', 'Stage four', 'Stage five', 'Stage six', 'Parse test', '<-Go back', 'Save', and 'Load'. A box labeled '1' points to the 'Show PST as Grid' radio button, and a box labeled '2' points to the 'Stage two' button.

MOVEID	BSTREEID	CS TREEID	BSNODEID	CSNODEID	PROBABILI	CYCLES	NEXTSTAT	STAGE4TR	FOLLOWED	REMARKS	LEN
2	-1	12	-1	-1	2204629476	0	3	False	False	CS: man h n	
2	-1	11	-1	-1	J1 62601626	0	3	False	False	CS: man h_r	
2	-1	13	-1	-1	897831E-04	0	3	False	False	CS: man h n	

Figure 16.6: The parser state table showing the entries for the move that involved Stage 2.

⁴ Details of the forward predictions from the Subject of the Clause can be seen if the user scrolls downwards.

The PST can be viewed either as a set of trees or as a grid, through the selection of the relevant button (Figure 16.6, Item 1). The left and right buttons at the bottom on the form (Figure 16.6, Item 2) allow the user to step through both the moves that the parser has made. Figure 16.7 shows the view of the PST in the form of a set of trees.⁵

In Figure 16.5, the built structure trees show the forward predictions that are being made for the unit(s) in the built structure, these being indicated by a right arrow and a percentage score. When a candidate structure is displayed (as in the PST view in Figure 16.7), a left pointing arrow and probability show the unit's backward predictions. As these were selected configurable parameters (see Figure 16.1), they will both be used to calculate a joining score in Stage 3, to which now turn.

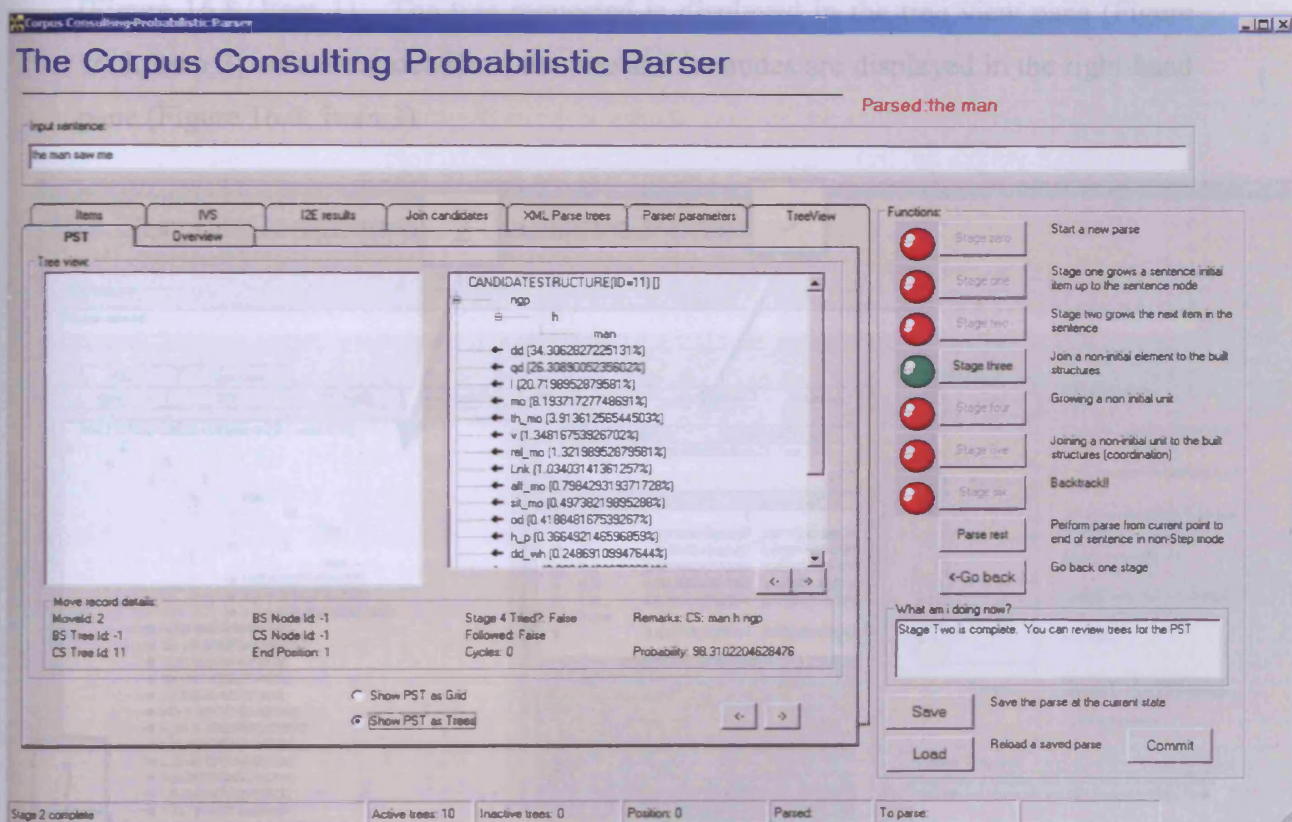


Figure 16.7: The parser state table displayed as a set of trees

16.4.2.4 Stage 3: Joining a candidate structure to a built structure

Next, the user presses the Stage Three button and the parser moves to Stage 3.⁶ After checking that the units above the elements to be joined match, it calculates the joining scores. It makes entries in the PST for all potential joins that have a score above (or

⁵ Note that the move shown only involved candidate structures and there are therefore no built structures in the left pane.

⁶ If the item was a linker, Stage Five would be an available button and not Stage Three.

equal to) the threshold value, and indicates that the tree-pair represented by the join will next move to Stage 3a. Any joins beneath the threshold value will be marked as needing to move to Stage 4, as will any tree-pairs that cannot be matched at all.

The parser now decides if it needs to move to Stage 3a or to Stage 4. If there are tree-pair records in the PST that have a joining score above (or equal) to the threshold value, it moves to Stage 3a and makes the joins for those tree-pairs. If there are no tree-pairs with a join score above the threshold, then the parser moves to Stage 4.

Figure 16.8 shows the most likely built structure following a successful join. This view allows the user to navigate through the built structures and candidate structures by using the forward and backward navigation buttons, or by going to a specific tree (Figure 16.8, Item 1). The tree requested is displayed in the tree view pane (Figure 16.8, Item 2), while the details of the tree and its nodes are displayed in the right-hand pane (Figure 16.8, Item 3).

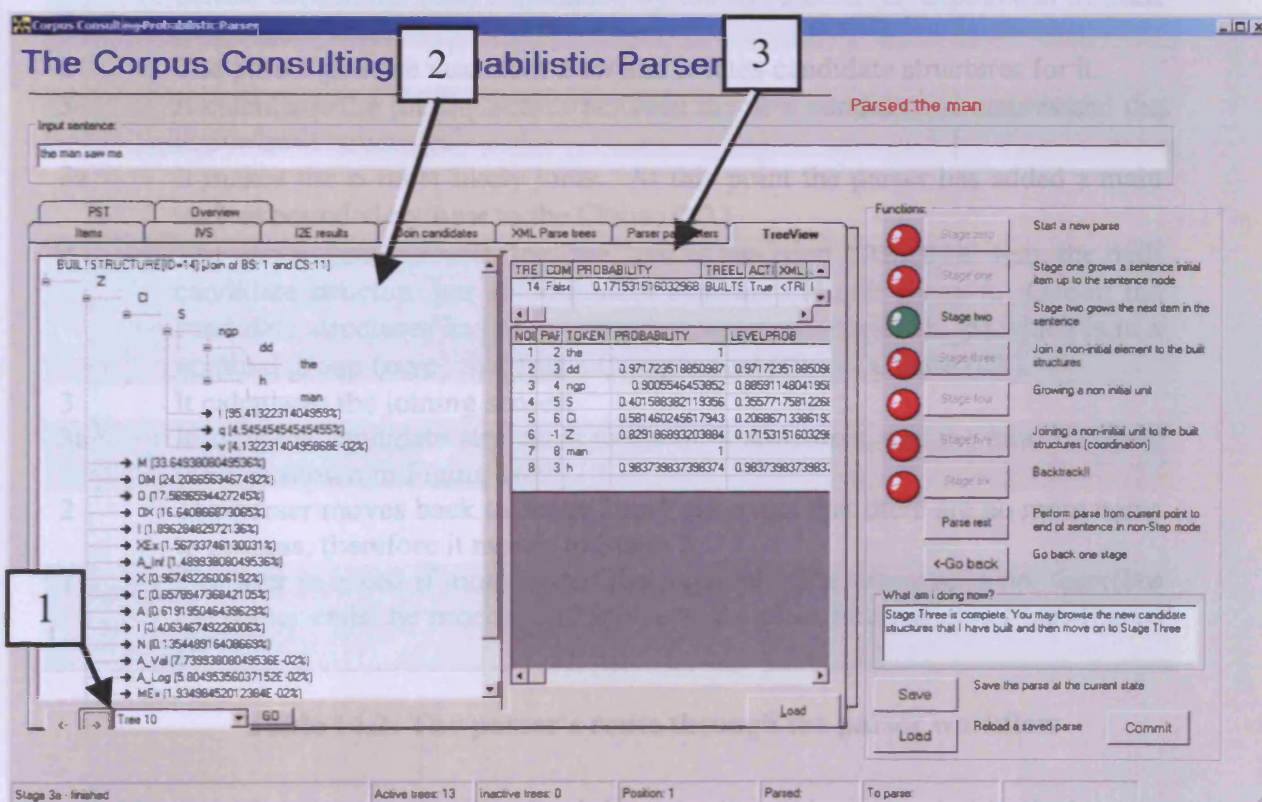


Figure 16.8: The most likely built structure in the tree view after a successful join

The tree being displayed in Figure 16.8 shows that the parser has identified, so far, a sentence (Z) that is filled by a Clause (C1), and that has a Subject (S) that is filled by a nominal group (ngp). This consists of a deictic determiner (dd) that is expounded by the item **the**, and a head (h) that is expounded by **man**.

Because there were successful joins, the parser returns to Stage 2 and the Stage Two button becomes available to the user. After this next Stage 2, the parser moves to Stage 3 again. The parser proceeds through the workflow and in arriving at its final analysis and it has followed the workflow path shown in Table 16.2.

Stage	Remarks
1	The parser gets the first item the , and creates a list of initial vertical strips in the DB-IVS table.
1a	It takes the n most likely vertical strips from the DB-IVS table and creates built structures for them.
2	It gets the next item man and creates candidate structures for it.
3	The parser calculates the joining scores between the active candidate structures and the active built structures.
3a	It makes the n most likely joins. At this point the parser has created the most likely built structure which has a Sentence (Z), filled by a Clause (C1) that contains a Subject (S) that is filled by a nominal group (ngp) that has a deictic determiner (dd) expounded by the and a head (h) expounded by man (see Figure 16.8).
2	The parser gets the next item saw and creates candidate structures for it.
3	It calculates the joining scores between the new candidate structures and the active built structures.
3a	It makes the n most likely joins. At this point the parser has added a main verb expounded by saw to the Clause (C1).
2	The parser gets the next item me , and as me is an I2E2U2E item, the built candidate structure has an additional element and unit above it. One of the candidate structures has me expounding a pronoun head (h_p) which is in a nominal group (ngp) that fills a Complement (C) in a Clause (C1).
3	It calculates the joining scores.
3a	It joins the candidate structures to the built structures, and the most likely of these is shown in Figure 16.9.
2	The parser moves back to Stage 2 and discovers that there are no more items to process, therefore it moves to Stage 7.
7	The user is asked if more parses are required. The response is no, therefore the parser ends the process and indicates the most likely analysis (see Figure 16.11).

Table 16.2: The parser's route through the parser workflow

Figure 16.9 shows the parser after it has completed the parse, and it displays the most likely analysis. The score for the tree is 20.8%. There are five complete trees at the end of the parse, and the others have low scores.

Corpus Consulting Probabilistic Parser

The Corpus Consulting Probabilistic Parser

Parsed: the man saw me

Input sentence:
the man saw me

PST Overview

Items IVS IZE results Join candidates XML Parse trees Parser parameters TreeView

BUILTSTRUCTURE(ID=42) [Join of BS: 25 and CS: 30]

TRE	COM	PROBABILITY	TREEL	ACTI	XML
42	False	0.208046373417035	BUILT	True	<TRI I

NOI	PAF	TOKEN	PROBABILITY	LEVELPROB
1	2	the		1
2	3	dd	0.971723518850987	0.971723518850987
3	4	ngp	0.9005546453852	0.88591148041958
4	5	S	0.401588382119356	0.35577175812268
5	6	CI	0.581460245617943	0.25090407858858
6	-1	Z	0.829186893203884	0.208046373417035
7	8	man		1
8	3	h	0.983739837398374	0.983739837398374
9	10	saw		1
10	5	M	0.947368421052632	0.947368421052632
11	12	me		1
12	13	h_p		1
13	14	ngp		1
14	5	C	0.431506849315069	0.431506849315069

Functions:

- Stage zero: Start a new parse
- Stage one: Stage one grows a sentence initial item up to the sentence node
- Stage two: Stage two grows the next item in the sentence
- Stage three: Join a non-initial element to the built structures
- Stage four: Growing a non initial unit
- Stage five: Joining a non-initial unit to the built structures (coordination)
- Stage six: Backtrack!
- Parse rest: Perform parse from current point to end of sentence in non-Step mode
- <-Go back: Go back one stage

What am i doing now?
Stage Two is complete. You can review trees for the PST

Save: Save the parse at the current state
Load: Reload a saved parse
Commit

Stage 2 complete Active trees: 17 Inactive trees: 29 Position: 3 Parsed To parse:

Figure 16.9: The parse is complete and the most likely built structure is displayed

There are two important facilities in the Parser WorkBench that we have not covered during this short walkthrough. These are the **XML Parse Tree View** (see Figure 16.10), and the **Overview** (see Figure 16.11). The XML Parse Tree View simply gives an alternative representation of the built and candidate structures; the XML is in the same form that is used in the corpus database.

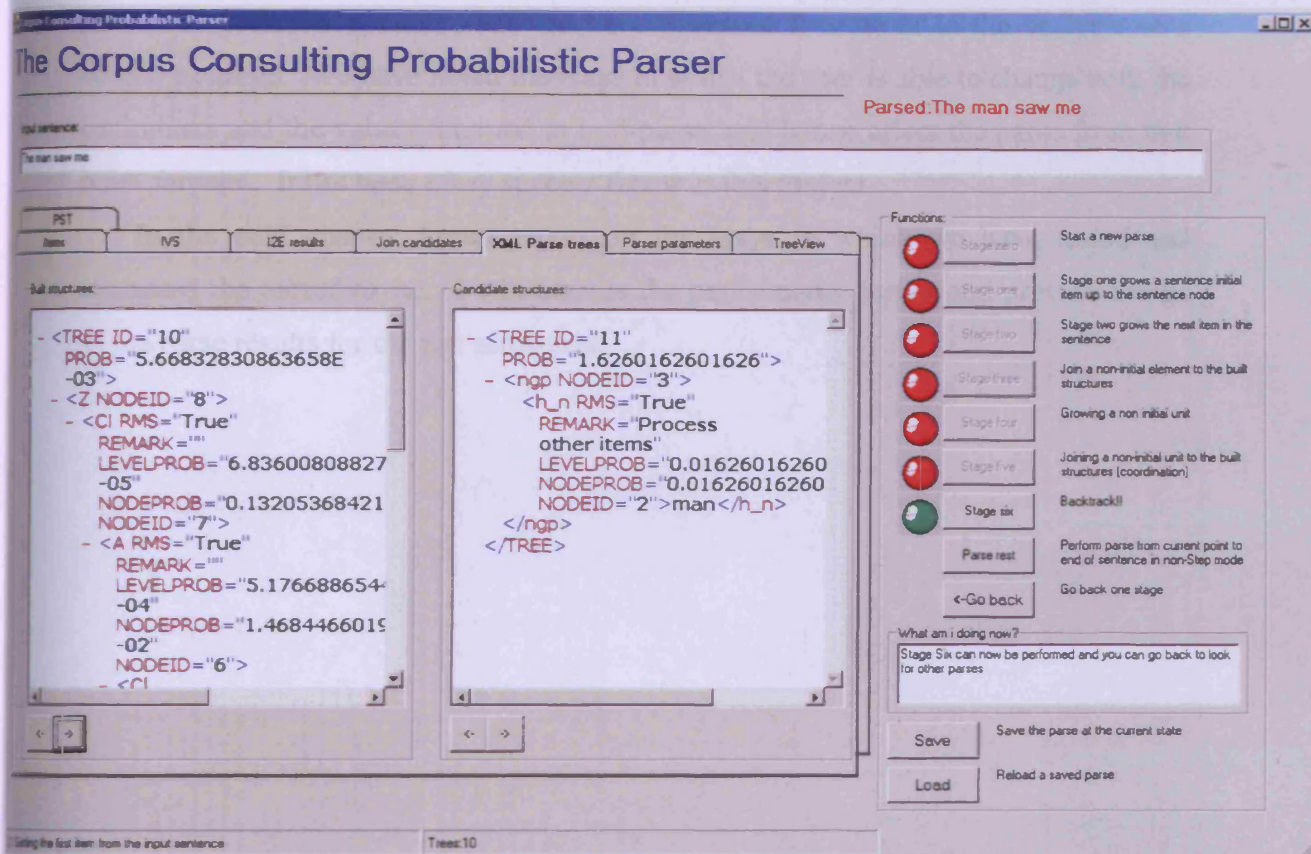


Figure 16.10: XML parse trees - built structures on the left and candidate structures on the right

The **Overview** (see Figure 16.11) provides a list of the active built structures at any point in the parse, and this allows the user to select one for display in an XML view. Figure 16.11 shows the Overview at the end of the parse, when the user has selected the most likely analysis for viewing in an XML format.

16.5 The commit button

When the parse is complete, the user can press the **commit** button to commit the built structure XML parse tree to the corpus database, so in turn updating the **corpus index tables** and the **probabilities tables**. The commit button is only available after a successful parse. At this phase of the project, only the user can decide whether or not

the parse will be evaluated as 'successful' and the user is able to modify the representation of the parse tree (in XML format) before it is committed to the database.

16.6 Summary

In this chapter, we have seen the user interface in which the Corpus Consulting Probabilistic Parser operates, and we have observed it at work in the analysis of a simple example. We have noted the ways in which the user is able to change both the parameters and the values returned in mid-parse, and hence affect the parse from that point forward. It has been a key success factor in this project.

In the next chapter I will report on the ways in which we have tested and evaluated the parser so far. I will discuss the performance issues and provide details of the parse results for the test sentences.

The Corpus Consulting Probabilistic Parser

Input sentence:
The man saw the

Items IVS I2E results Join candidates XML Parse trees Parameters TreeView

Parses at last stage:

MOVEID	TREEID	PROBABILITY	XML
10	46	0.208046373417035	<TREE ID='46' PROB='0.208046373417035'><Z><Cb><S><ngp>
10	46	0.108046373417035	<TREE ID='42' PROB='0.108046373417035'><Z><Cb><S><ngp>
10	43	0.139293193428697	<TREE ID='43' PROB='0.139293193428697'><Z><Cb><Voc><n>
10	44	4.72487849399941E-02	<TREE ID='44' PROB='4.72487849399941E-02'><Z><Cb><A><ngp>
10	45	1.78106307170634E-03	<TREE ID='45' PROB='1.78106307170634E-03'><Z><Cb><C><Cb>

Parsed: the man saw I

```

<TREE ID="42" PROB="20.8046373417035">
  <Z>
    <Cb>
      <S>
        <ngp>
          <dd>the</dd>
          <h>man</h>
        </ngp>
      </S>
    <M>saw</M>
  </Cb>
  <ngp>
    <h_p>me</h_p>
  </ngp>
</Cb>
</Z>
</TREE>
    
```

Functions:

- Stage zero
- Stage one
- Stage two
- Stage three
- Stage four
- Stage five
- Stage six
- Parse rest
- Go back

structures (coordination)
Backtrack!

Perform parse from current point to end of sentence in non-Step mode
Go back one stage

What am I doing now?
Stage Two is complete. You can review trees for the PST

Save Save the parse at the current state
Load Reload a saved parse Commit

Stage 2 complete Active trees: 17 Inactive trees: 29 Position: 3 Parsed: To parse:

Figure 16.11: The overview, which displays active built structures

Chapter Seventeen

Parser - testing and evaluation

This chapter describes how the parser was tested and evaluated against the aims and goals of this work.

Section 17.1 describes the methods that were used to test the parser, and reports the results. It starts with tests to determine the best values for the parser's configurable parameters, and concludes with an evaluation of the parser in terms of its **accuracy**, its **efficiency**, and its **speed**. It compares the results with those of the other parsers described in Chapters Eleven and Twelve.

Sections 17.2 and 17.3 provide an evaluation against the aims of this project that were specified in Chapter One.

17.1 Testing the parser

Testing was divided into three stages. The first stage is reported in Section 17.1.1, and involved determining the optimal values for the **configurable parameters** (see Section 16.4.1 of Chapter Sixteen). The second stage is reported in Section 17.1.2, and included testing the parser's backtracking mechanisms, and testing for sentences with attachment ambiguity. The third stage involved **extensive testing** of the parser with a large set of randomly selected, naturally occurring sentences. This latter set of tests used the optimal values found for the configurable parameters, and are described in Section 17.1.3.

17.1.1 Establishing the optimum configurable parameter settings

Before testing in the conventional sense could begin, it was necessary to determine what the optimum values for the **configurable parameters** should be, and this required a comprehensive set of tests in its own right. This section describes the tests that were designed to establish:

- the optimum values of **n** for **n-best** (see Section 15.1.6 of Chapter Fifteen and Section 17.1.1.1),
- the version of the **joining score formula** (see Section 15.2.4 of Chapter Fifteen and Section 17.1.1.2),
- the **join threshold value** (see Section 15.2.4 of Chapter Fifteen and Section 17.1.1.3).

- the **co-ordination joining score parameter values** (see Section 15.2.6.2 and Section 17.1.1.5)

During these preliminary tests, the sentences shown in Table 17.1 were used. They were chosen because they display a range of linguistic characteristics that present different challenges to the parser, and testing with these will indicate values for the configurable parameters that will suit a wide variety of situations.

These tests required the use of the **Parser Workbench** in the **step-by-step mode** (see Section 16.1 of Chapter Sixteen) which allowed us to follow the **built structures and candidate structures** around the **parser's workflow** (see Figure 15.5 of Chapter Fifteen). It was easy to see when it took the wrong path by examining the structures built, and the scores assigned to them. When the correct structures had not been considered in a particular parser move, adjustments were made to the relevant parameters.¹

Sentence No.	Sentence	Reason for selection
S1	The man saw me	Simple nominal group (ngp) as the Subject (S).
S2	The man saw me with the telescope	Prepositional group (pgp) as an Adjunct (A).
S3	I saw the man	A pronoun (h_p) as a Subject (S).
S4	You saw him	A pronoun (h_p) as a Complement (C).
S5	Robin saw him	A proper name (h_n) as a Subject (S).
S6	Who saw her?	Interrogative Subject (S).
S7	Yesterday I saw him	Thematised Adjunct (A).
S8	Where did you see him?	Interrogative Adjunct (A).
S9	When did you see him?	Interrogative Adjunct (A).
S10	We might not do any sums	Sentence in FPD Corpus
S11	When you saw him, was he happy?	Thematised Adjunct (A) with embedded clause (C1) + polarity seeker.
S12	And Robin	Ellipsis + binder (B).
S13	Then he went home	Thematised Adjunct (A) first.

Table 17.1: Some initial test sentences

The tests and the conclusions are described in the Sections 17.1.1.1, 17.1.1.2, and 17.1.1.3. After the best values and options were identified, the parser was allowed to run in the non-step-by-step mode while parsing the test sentences shown in Table 17.1, and parse details were recorded. These tests and the results are described in

¹ In this respect, the Parser WorkBench provided an excellent testing environment.

Section 17.1.1.4.

17.1.1.1 Establishing an optimum value of n in n -best

In the first version of the parser, the same value of n was applied at every point in the parser's workflow. These values specify for each parser move, the maximum number of trees (or tree-pairs) that can be created for these purposes:

- (a) to take forward from **Stage 1a**,
- (b) to allow to be joined in **Stage 3a**,
- (c) to allow to be grown in **Stage 4**,
- (d) to allow to be joined in a co-ordination relationship in **Stage 5a** (which will be described in Section 17.1.1.5).

Following the initial tests, it soon became obvious that it would be sensible to allow different values of n to be specified for each of the above parameters. In the original model, when n was set to a low value (e.g. 1), the joining worked well, provided that the correct built structure was represented by the most likely initial vertical strip in the **IVS-ITEM** and **IVS-ELEM** tables. As described in Chapter Fifteen, when it is performing Stage 1, the parser is at its most vulnerable, because it has no **parse history** to help it make its decisions.² If the correct built structure is not within the n -best vertical strips in the **IVS** tables for the given item, then later, the parser has to backtrack, and it is in this sense that it is 'vulnerable'.³ To reduce the need to backtrack to Stage 1a, therefore, the value of n needs to be set to a higher value than the value of n used to control the number of joins made in Stage 3a.⁴

A similar argument also applies to the number of candidate structures that the parser allows to be grown in Stage 4.⁵ When a candidate structure that has a Clause (**C1**) as its root element is grown, it will produce a new set of candidate structures - one for each element that a Clause can fill. Since a Clause can fill 25 different elements altogether, such a grow operation will create 25 new candidate structures.

² The parse history is defined as the structures that have been parsed so far.

³ Backtracking to Stage 1a proved to be computationally expensive, and a process that should be avoided if possible.

⁴ In Stage 3a, the join decisions are based on a parse history (and this increases in confidence with the number of items parsed). Stage 1a has no parse history, and therefore it makes sense to allow more structures to be built.

⁵ It should be made clear at this point that the value of n in this case is the number of candidate structures that are grown, and not the number of trees that are produced as the result of the operation.

If the value of n for the number of candidate structures to grow is too small, it will mean that the correct candidate structure may not exist until the parser backtracks. A candidate structure that has been grown may favour an element that is able to initiate a new unit over one that could be attached further up in an existing built structure. There is also less confidence in a candidate structure than there is in a built structure, since typically (i.e. from the third item onwards) it has a smaller parse history upon which to make its decisions. Therefore, it was also sensible to introduce a separate value of n to represent the number of candidate structures that can be grown.

The price to be paid in allowing n to be larger for grow operations is that the parser slows down when a grow operation is needed. There is an argument for the introduction of a more intelligent growing process, and we shall discuss this in Chapter Eighteen.

The parser's workflow path for the first of the sample sentences **the man saw me** can be seen in Table 17.2. This contains a mix of Stages 1, 1a, 2, 3, 3a, and 7. The first test can only therefore determine the optimum values for the number of trees to take forward from Stage 1a (Column A in Table 17.3) and the number of trees to take forward from Stage 3a (Column B in Table 17.3).

Pos	Item	Parser workflow (state paths)
1	the	1-1a
2	man	2-3-3a
3	saw	2-3-3a
4	me	2-3-3a-2-7

Table 17.2: The parser workflow path for the first sample sentence

A	B	Parse time (sec.)	No. trees	No complete trees	Score of correct tree	Rank of correct tree
5	5	25	46	5	20.8%	1
10	10	32	59	10	20.8%	1
20	20	38	63	14	20.8%	1
50	50	38	63	14	20.8%	1
5	1	18	34	1	20.8%	1

A = the number of trees to take forward from Stage 1a

B = the number of tree-pairs to take forward from Stage 3a

Table 17.3: The effect of changing the value of n in n -best

The second sentence **the man saw me with the telescope** requires the use of Stage 4 in order to grow the prepositional group (ppp) introduced by the

preposition **with** (p), to show it as filling an Adjunct (A). It also requires a Stage 4 to grow the nominal group **the telescope** into a complete (cv) to attach to the prepositional group (pgp). The parser's workflow path involved in this second sentence is given in Table 17.4, and the effect of changing the parameters is shown in Table 17.5. Here Column C represents the number of candidate structures to grow in Stage 4.

Pos	Item	Parser workflow (state paths)
1	the	1-1a
2	man	2-3-3a
3	saw	2-3-3a
4	me	2-3-3a-2-7
5	with	2-3-3a-4-3-3a
6	the	2-3-3a-4-3-3a
7	telescope	2-3-3a-2-7

Table 17.4: The parser workflow path for the first second sample sentence

A	B	C	Parse time (sec.)	No. trees	No complete trees	Score of correct tree	Back track?	Rank of correct tree
5	5	5	124	37	1	1.96%	No	1
10	10	10	180	350	4	1.96%	No	1
						(second 0.2%)		
20	20	20	220	407	9	1.96%	No	1
50	50	50	988	630	25	1.96%	No	1
5	1	1	140	242	2	1.96%	Yes	1
5	1	2	54	155	1	1.96%	No	1
5	1	8	120	448	1	1.96%	No	1

A = the number of trees to take forward from Stage 1a

B = the number of tree-pairs to take forward from Stage 3a

C = the number of candidate structures to grow in Stage 4

Table 17.5: The effect of changing the value of n in n-best

It can be seen that the parser is forced to backtrack when the number of candidate structures to grow or join is too low. This suggests that the value of **n** should be such that the correct join or grow is included within the number of trees that are selected for value that has been set for **n**. However, even though it is the case that if the value of **n** is greater, the parse is slower, the correct parse is nonetheless detected with a relatively low value of **n**.

The conclusions from these initial tests are that the following parameters are the optimum ones to use for the rest of the test sentences:

- (a) the number of trees to join in Stage 3a is 5,

(b) the number of trees to grow in Stage 4 is 5.⁶

The tests also show that, if we have a high value of n for the number of trees to take forward from Stage 1a, it improves the speed and accuracy of the parse. Indeed, all of the initial test sentences in Table 17.1 included the correct analysis within the three most likely initial vertical strips. It would be preferable, of course, if the first one was always the correct one. But the reasons for the selection of the 'wrong' ones are such that we are confident that this deficiency will be overcome when we introduce the revised and extended probabilities tables of Version Two (as described in Section 14.2 of Chapter Fourteen, Appendix H and Appendix I).

We turn next to consider the second of the parser's configurable parameters - the joining score formula's parameter configuration.

17.1.1.2 Establishing the best configuration of the joining score formula parameters

The Parser WorkBench allows the user to specify the parameters that will be used in the calculation of the joining score, and these are perhaps the most crucial of all the variable parameters (see Section 15.2.4 of Chapter Fifteen). They are implemented as check boxes in which the user can place a tick to indicate that the variable is wanted in the joining score formula (see Figure 16.1 of Chapter Sixteen). The purpose was to enable the user to perform experiments to determine the best algorithm to use, so that once the best combination has been found, then the other options could be removed from the Parser WorkBench. The options that can be combined are grouped into five pairs:

- (a) The forward prediction score (P_{Fwd}) and the backward prediction score (P_{Bwd}),
- (b) The probability that the lower units in the right-most strip of the built structure have been completed ($P_{LowerBSClosed}$) and the probability that the lower units in the candidate structure have no elements to the left of any element in the left-most strip ($P_{LowerCSOpen}$),
- (c) The built structure tree score (P_{BStree}) and the candidate structure tree score (P_{CStree}),

⁶ Despite finding that the optimum value of n is the same for the number of structures to grow and for the number of structures to join, it is still sensible to have two separate parameters as it is expected that more challenging sentences may need different values for each.

- (d) The built structure node probability (P_{BSNode}) and the candidate structure node probability (P_{CSNode}),
- (e) The built structure node's level probability ($P_{BSLevel}$) and the candidate structure node's level probability ($P_{CSLevel}$).⁷

A formula was developed which accepts all of the selected parameters and when it is applied it returns a joining score:

$$\begin{aligned}
 JS = & ((P_{Fwd} + P_{Bwd}) / nP1) * \\
 & ((P_{LowerBSClosed} + P_{LowerCSOpen}) / nP2) * \\
 & ((P_{BStree} + P_{CStree}) / nP3) * \\
 & ((P_{BSLevel} + P_{CSLevel}) / nP4) * \\
 & ((P_{BSNode} + P_{CSNode} / nP5))
 \end{aligned}$$

If none of the parameters in a given pair are required, both values of the pair are set to the value 1 (which means that they are both ignored). If only one of a set of two parameters in a pair is not required, the value of the other is set to 0 (which means that the parameter that is not required will be ignored). The variables $nP1$ through $nP5$ represent the number of parameters in the given pair that are required (i.e. 1 or 2). In this way, the values that are not selected will not be involved in the calculation, and this allowed us to test how far they affect the outcome.

As expected, initial testing revealed that the following joining score parameters are always required:

- (a) the forward and backward predictions (P_{Fwd} , and P_{Bwd}),
- (b) the probabilities that the lower units in the built structure have been completed, and the probabilities that the lower level elements in the candidate structures can start their units ($P_{LowerBSClosed}$ and $P_{LowerCSOpen}$).

The crucial finding of this batch of tests was that the **level probability** and **node probability** (see Section 15.1.7 of Chapter Fifteen) did not increase the ability of the joining score to reflect the judgement of a human analyst. Indeed they had the adverse effect of making the scores much smaller. It was therefore concluded that these two parameters should be excluded from the function. I next continued investigations with combinations of the other joining score parameters.

To see the effect of the discoveries of the best configuration of the joining score parameters, please refer to Figure 17.1. This shows an example of an incorrect analysis in the original POW Corpus, where the item **saw** is incorrectly analysed once

as an Operator conflated with a Main Verb (**OM**), when it should have simply been a Main Verb (**M**).

In the course of our work, we have normally corrected these errors as found them by using ICQF+'s **corpus editor** (see Chapter Eight). However, in this case I decided to proceed with testing before changing the sentence that contained the misanalysis. The reason for doing this was that I would then be able to ensure that, by selecting the correct joining score parameters, the most likely joins would be still preferred above the less likely incorrect ones. Further, there will inevitably be other such misanalyses in the corpus.

#	P_{Fwd}	P_{Bwd}	P_{Lower} BSClosed	P_{Lower} CSOpen	P_{BStree}	P_{CStree}	JS	Rank
A1	0.33	0.35	0.80	1.00	-	-	0.272	2
A2	0.24	0.71	0.80	1.00	-	-	0.380	1
B1	0.33	0.35	0.80	1.00	0.17	-	0.052	2
B2	0.24	0.71	0.80	1.00	0.17	-	0.072	1
C1	0.33	0.35	0.80	1.00	0.17	0.93	0.171	1
C2	0.24	0.71	0.80	1.00	0.17	0.25	0.079	2

Table 17.6: A comparison of different join score formula parameter configurations

In Table 17.6, the letters **A**, **B** and **C** represent different combinations of the joining score parameters, and the columns represent the values of the parameters. A dash in a column means that the parameter was not used in the test represented by the table row. **A1** represents the join of the built structure and the first candidate structure in Figure 17.1, and **A2** represents the join of the built structure and the second candidate structure using a particular set of parameters (i.e. P_{Fwd} , P_{Bwd} , $P_{LowerBSClosed}$ and $P_{LowerCSOpen}$). The pair **B1**, **B2**, and **C1**, **C2** have similar meanings for other sets of parameters.

The tests show that when the probability of the built structure and the candidate structure were not included in the formula, the incorrect join was favoured. This was because the choice was dominated by the backward prediction of an Operator conflated with a Main Verb (**OM**) being preceded by a Subject (**S**) (71%). However, by including the probability of the candidate structure (P_{CStree}) in the formula, the rarity of the word **saw** expounding an **OM** is taken into account by the formula, and the inclusion of both this, and the built structure score (P_{BStree}), ensures that the

⁷ Section 15.1.9 of Chapter Fifteen explains the terms **node probability** and **level probability**.

structures are correctly ranked. The final improvement was to add a weighting to represent the fact that we are more confident that the built structure is correct than we are in the correctness of the candidate structure (because it has a larger parse history). We have therefore doubled the score of the built structure tree in the join strength algorithm.⁸

The conclusion to be drawn from these tests is therefore that the following parameters are needed in the formula:

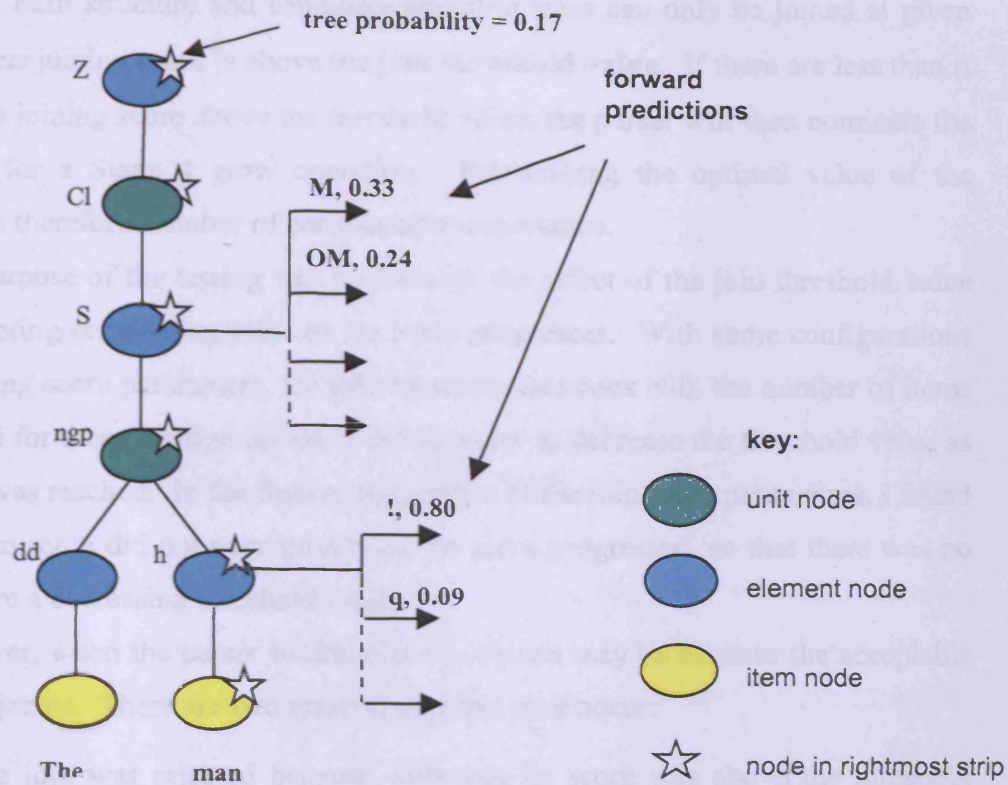
- (a) the forward prediction score (P_{Fwd}),
- (b) the backward prediction score (P_{Bwd}),
- (c) the probability that the lower units in right-most strip of the built structure have been completed ($P_{LowerBSClosed}$),
- (d) the probability that the lower units in the candidate structure have no elements to the left of any element in the left-most strip ($P_{LowerCSOpen}$),
- (e) the built structure tree score (P_{BStree}) multiplied by a weighting factor of two,
- (f) the candidate structure tree score (P_{CStree}).

The following are therefore not required:

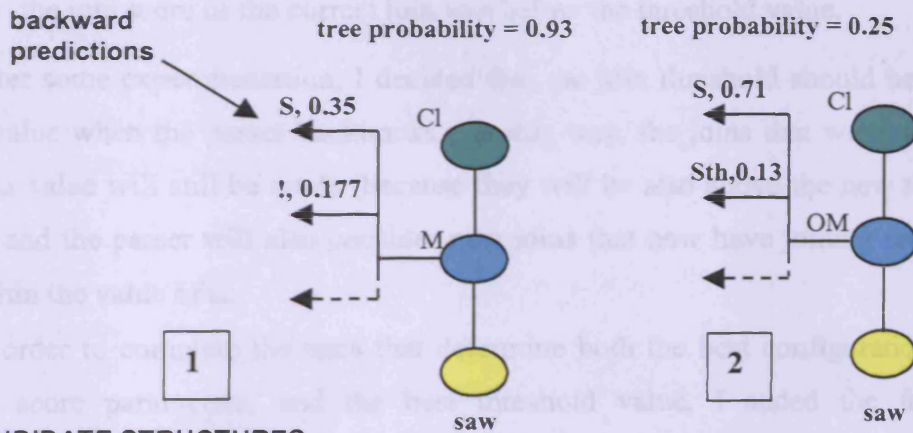
- (a) the built structure node probability (P_{BSNode}),
- (b) the candidate structure node probability (P_{CSNode}),
- (c) the built structure node's level probability ($P_{BSLevel}$),
- (d) the candidate structure node's level probability ($P_{CSLevel}$).

After determining the best parameters to use in calculating the joining score, the next task was to determine the best value for the **join threshold value**, and it is to this we turn now.

⁸ A further possible refinement would be to make the weighting value a variable configurable parameter, such that the weighting would increase with the number of words parsed. But the current weighting used in the tests conducted so far work well.



BUILT STRUCTURE



CANDIDATE STRUCTURES

Figure 17.1: A sample join to demonstrate the choice of joining score formula parameter configurations⁹

⁹ The occurrence of the element **Sth** may seem odd - and it is. It arises from the misanalysis of **saw** as **OM**. An examples of an **Sth** coming before an **OM** would be **there (Sth) are (OM) two red bricks in the box**.

17.1.1.3 Establishing the best join threshold value

The n -best built structure and candidate structure trees can only be joined at given nodes if their joining score is above the **join threshold value**. If there are less than n trees with a joining score above the threshold value, the parser will then nominate the remainder for a Stage 4 grow operation. Establishing the optimal value of the threshold is therefore a matter of considerable importance.

The purpose of the testing was to discover the effect of the join threshold value when accepting or rejecting joins as the parse progresses. With some configurations of the joining score parameters, the joining score decreases with the number of items parsed, and for these configurations, it is necessary to decrease the threshold value as each item was reached. In the final configuration of the join score parameters, I found that the join score did not vary greatly as the parse progressed, so that there was no need to have a decreasing threshold value.

However, when the parser backtracks, the reason may be because the acceptable join was rejected. There are two reasons why this may occur:

- (a) the join was rejected because, although its score was above the threshold value, there were more than n joins above the threshold, and the correct join was not in the best n ,
- (b) the join score of the correct join was below the threshold value.

After some experimentation, I decided that the join threshold should be given a lower value when the parser backtracks. In this way, the joins that were above the previous value will still be made (because they will be also above the new threshold value), and the parser will also consider new joins that now have joining scores that fall within the value of n .

In order to complete the tests that determine both the best configuration of the joining score parameters, and the best threshold value, I added the following additional parser configurable parameters:

- (a) the threshold **initial value**,
- (b) a threshold **decrement value**,
- (c) a threshold **division value**.

The **initial value** is used as the original value that will be applied for the first item. The **decrement value** is the value deducted from the initial value for each item

parsed, and this is set to 0 if no decrement is needed. The threshold **division value** is a percentage that is used to reduce the initial value after each backtracking event.¹⁰

At this stage, I had now determined:

- (a) that separate values are required for the value of **n** in the **n-best** algorithm at each different part of the parser's workflow,
- (b) the best values of **n** for each of these,
- (c) the best configuration of the join score parameters for the join score formula.

I was now in a position to determine the best starting value for the join threshold parameter. Initial tests using the sample sentences set this at a value of 0.006, and this allowed most reasonable joins to go through to the next stage of the parsing process.

The decisions about the values for the configurable parameters (described so far) are summarised in Table 17.7. The remaining parameters are (a) those that affect co-ordinated joins, and (b) the **number of cycles** of Stages 3, 4, 3 and 5 to attempt before backtracking. The first of these is discussed in Section 17.1.1.5. The value for the number of cycles parameter was set at 3 although the parser did not need more than a single cycle in any of the tests. Therefore, we can consider this value adequate for the initial test suite.

Number of trees to take forward from Stage 1a	10
Number of trees to join in Stage 3a	5
Number of trees to grow in Stage 4	5
Join score parameter configuration	$P_{Fwd}, P_{Bwd},$ $P_{LowerBSClosed},$ $P_{LowerCSOpen}, P_{Bstree} *$ $2, P_{Cstree}$
Initial join score threshold value	0.006
Join score decrement value	0
Join score division value¹¹	100
Number of cycles of Stages 3, 4, 3 and 5	3

Table 17.7: The final choice of parsing parameters for the initial test sentences

This now concludes the first stage of the testing with the goal of determining the best values for the configurable parameters. The tests to determine the configurable parameters for co-ordination joins are given in Section 7.1.1.5. We next turn to the results of the tests for the sentences shown in Table 17.1 when using these parameters.

¹⁰ A value of 1 can be used if no reduction is needed for each backtracking event.

17.1.1.4 Performing the basic tests

For all of the tests that follow in this and subsequent sections, we used the values of the parameters specified in Table 17.7 (unless otherwise stated) together with the Version One **probabilities tables**, and as reported in Section 14.2 of Chapter Fourteen, Appendix H and Appendix I, there were limitations in using these. The problems will be rectified when the Version Two tables are used (which are being created at the time of writing). One of the main problems with the Version One tables was the inadequate coverage of the items in the **I2E** and **I2E2U2E** tables. This meant that the test sentences had to be chosen carefully in order to ensure that the items used had occurred in the corpus, and that there were no instances of the item that represented a rare occurrence. These cases were typically detected during the testing, as they often caused the parser to follow an unexpected path, which was sometimes in preference to the correct one. It was the number of unexpected results and the lack of coverage of items that led to our development of the **Version Two probabilities tables**.¹²

The test sentences shown in Table 17.1 were created to test the parser using examples that contain different structures. For each sentence parsed, we recorded:

- (a) the parse times,
- (b) the final parse scores for the correct structures,
- (c) the total number of trees created (including those not in the final analyses),
- (d) the total number of complete trees (that are involved in the final analyses),
- (e) the route through the parser workflow,
- (f) if backtracking occurred.

The initial test results are shown in Table 17.8. The parser in all cases found the correct parse at the first attempt, and in no case did it need to backtrack. In all except one, the correct parse was reached first, and in that exceptional case, the reason can be attributed to the problems in the Version One tables. The parse times reported by the parser were encouraging. Parses that needed a Stage 4 grow operation, created a greater number of trees and this was due to the number of elements that a given class of unit can fill.

¹¹ See Section 17.1.1.3 for the reason for selection of this value.

¹² The Version One tables were also deficient in the number of items that were identified as requiring **I2E2U2E** processing.

Sentence	Back-tracking	Parse time (seconds)	Number trees	Number complete trees	Correct parse found?	Correct parse rank	Correct parse score	Workflow route
s1: the man saw me	N	25	46	5	Y	1	0.2085	1-1a-2-3-3a-2-3-3a-2-3-3a-2-7
s2: the man saw me with the telescope	N	100	270	3	Y	1	0.0196	1-1a-2-3-3a-2-3-3a-2-3-3a-2-3-4-3-3a-2-3-4-3-3a-2-3-3a-2-7
s3: I saw the man	N	60	193	2	Y	1	0.1595	1-1a-2-3-3a-2-3-4-3-3a-2-3-3a-2-7
s4: You saw him	N	12	32	5	Y	1	0.3643	1-1a-2-3-3a-2-3-3a-2-7
s5: Robin saw him	N	13	32	5	Y	1	0.3604	1-1a-2-3-3a-2-3-3a-2-7
s6: Who saw her?	N	12	34	4	Y	1	0.3502	1-1a-2-3-3a-2-3-3a-2-7
s7: Where did you see him?	N	25	60	5	Y	1	0.5738	1-1a-2-3-3a-2-3-3a-2-3-3a-2-3-3a-2-7
s8: When did you see him?	N	21	55	2	Y	2	0.5738	1-1a-2-3-3a-2-3-3a-2-3-3a-2-3-3a-2-7
s9: When you saw him, was he happy?	N	54	180	2	Y	1	0.6323	1-1a-2-3-3a-2-3-3a-2-3-3a-2-3-3a-2-3-3a-2-7
s10: We might not be doing any sums	N	48	219	2	Y	1	0.1607	1-1a-2-3-3a-2-3-3a-2-3-3a-2-3-3a-2-3-4-3-3a-2-3-3a-2-7
s11: And Robin	N	3	16	5	Y	1	0.0200	1-1a-2-3-3a-2-7
s12: Then he went home	N	24	110	5	Y	1/3	0.2181	1-1a-2-3-3a-2-3-3a-2-3-4-3-3a-2-7

Table 17.8: The parser results using the initial set of test sentences

17.1.1.5 Testing the parser's ability to handle co-ordination

The three simple sentences shown in Table 17.9 were used test the co-ordination stages of the parser. As with the earlier tests, I needed to establish (a) the optimum co-ordination joining score parameters and (b) the number of trees to take forward from Stage 5 to Stage 5a. The decisions made for these are detailed in Sections 17.1.1.5.1 and 17.1.1.5.2 respectively.

Sentence No.	Sentence	Reason for selection
S14	The man saw Cardiff and Treforest	Subject first, and co-ordinated nominal group in a Complement.
S15	Timothy and Robin built a farmhouse	Co-ordinated Subject first.
S16	Timothy, Robin and Christine built a car	Three co-ordinated nominal groups in a Subject first.

Table 17.9: Some sentences to test co-ordination

17.1.1.5.1 Calculating the co-ordination joining score

The parameters to the co-ordination joining score formula are based on a decision table as described in Section 15.2.6.2 of Chapter Fifteen, which are probability scores based on the values returned from three tests (see Table 17.10).

Test 1 (bLinker)	Test 2 (bUnitSame)	Test 3 (bPunct)	Score
true	true	true or false	0.910
true	false	false	0.001
true	false	true	0.001
false	true	true	0.600
false	true	false	0.001
false	false	true	0.001

Table 17.10: The decision table for assigning a co-ordination joining score

After these tests have been applied, a decision table determines the score for the join. The optimum values that were defined are shown in Table 17.10. These tests overwhelmingly favour a join that has a linker as the first item in the candidate structure, and have both units the same. The test for having punctuation after the last item in the built structure had to be increased from the original value assigned in Section 15.2.6.2 of Chapter Fifteen to allow co-ordinated groups such as **Timothy,**

Robin and Christine to have a score that meant that they were considered for a join without the need to backtrack.¹³

During testing, I found that the built structure trees needed to be ranked in order to determine the order that the joins are made in the **n-best** joins of Stage 5a. The final score was then determined by the following formula:

$$JS = P_{co-ord} * ((P_{Bstree} * 2) / (P_{Cstree}))$$

Here P_{co-ord} is the value from the decision table given in Table 17.10, and P_{Bstree} and P_{Cstree} are the scores for the built structure and candidate structure trees respectively.¹⁴

17.1.1.5.2 Determining the value of n in n-best co-ordinated joins to take forward

As in the case of the element-join, the optimum number of trees to take forward to make a unit-join had to be decided. I found that a value of between 3 and 5 captured all legal joins. The results of the co-ordination tests are described in the next section.

17.1.1.5.3 The results of the co-ordination tests

Table 17.11 shows the results of the co-ordination tests for the sample sentences given in Table 17.9.

For these relatively simple sentences, the parser worked very well and did not need to backtrack. In all cases the most likely analysis returned by the parser was the correct one.

¹³ This is an area of syntax in which it might be advantageous if the parser possessed some form of 'look-ahead' to detect the presence of a linker in a co-ordinated set of groups.

¹⁴ Note that, like in the element-join formula, the score for the built structure tree is weighted by a factor of two, because we have more confidence in the score for the built structure.

Sentence	Back-tracking	Parse time (sec)	Number trees	Number complete trees	Correct parse found?	Correct parse rank	Correct parse score	Workflow route
s14: the man saw Cardiff and Treforest	N	49	115	5	Y	1	0.0211	1-1a-2-3-3a-2-3-3a-2-3-4-3-3a-2-5-5a-2-3-2-7
s15: Timothy and Robin built a farmhouse	N	47	156	3	Y	1	0.1607	1-1a-2-5-5a-2-3-3a-2-3-3a-2-3-4-3-3a-2-3-3a-2-7
s16: Timothy, Robin and Christine built a car	N	60	241	2	Y	1	0.1595	1-1a-2-3-3a-2-3-4-3-3a-2-3-3a-2-7

Table 17.11: Results using the set of test sentences for co-ordination (see Section 17.1.1.5)

Sentence	Back-tracking	Parse time (sec)	Number trees	Number complete trees	Correct parse found?	Correct parse rank	Correct parse score	Workflow route
s17: I saw the man with the telescope	N	62	510	2	Y	1 and 2	0.0199 and 0,01967	1-1a-2-3-3a-2-3-4-3-3a-2-3-3a-2-3-4-3-3a-2-3-4-3-3a-2-3-3a-2-7

Table 17.12: Results using a test sentence that has attachment ambiguity (see Section 17.1.2.1)

Sentence	Back-tracking	Parse time (sec)	Number trees	Number complete trees	Correct parse found?	Correct parse rank	Correct parse score	Workflow route
s18: The players played last Saturday were dropped	Y	240	900	16	Y	8	0.0003	1-1a-2-3-3a-2-3-4-3-3a-2-3-3a-2-3-4-3-3a-2-3-4-3-3a-2-3-3a-2-7-6-3a-2-7-6.....etc

Table 17.13: Results using the set of test sentences for backtracking (see Section 17.1.2.2)

(b) the telescope is in the possession of the man.

These interpretations are shown in Figure 17.2:

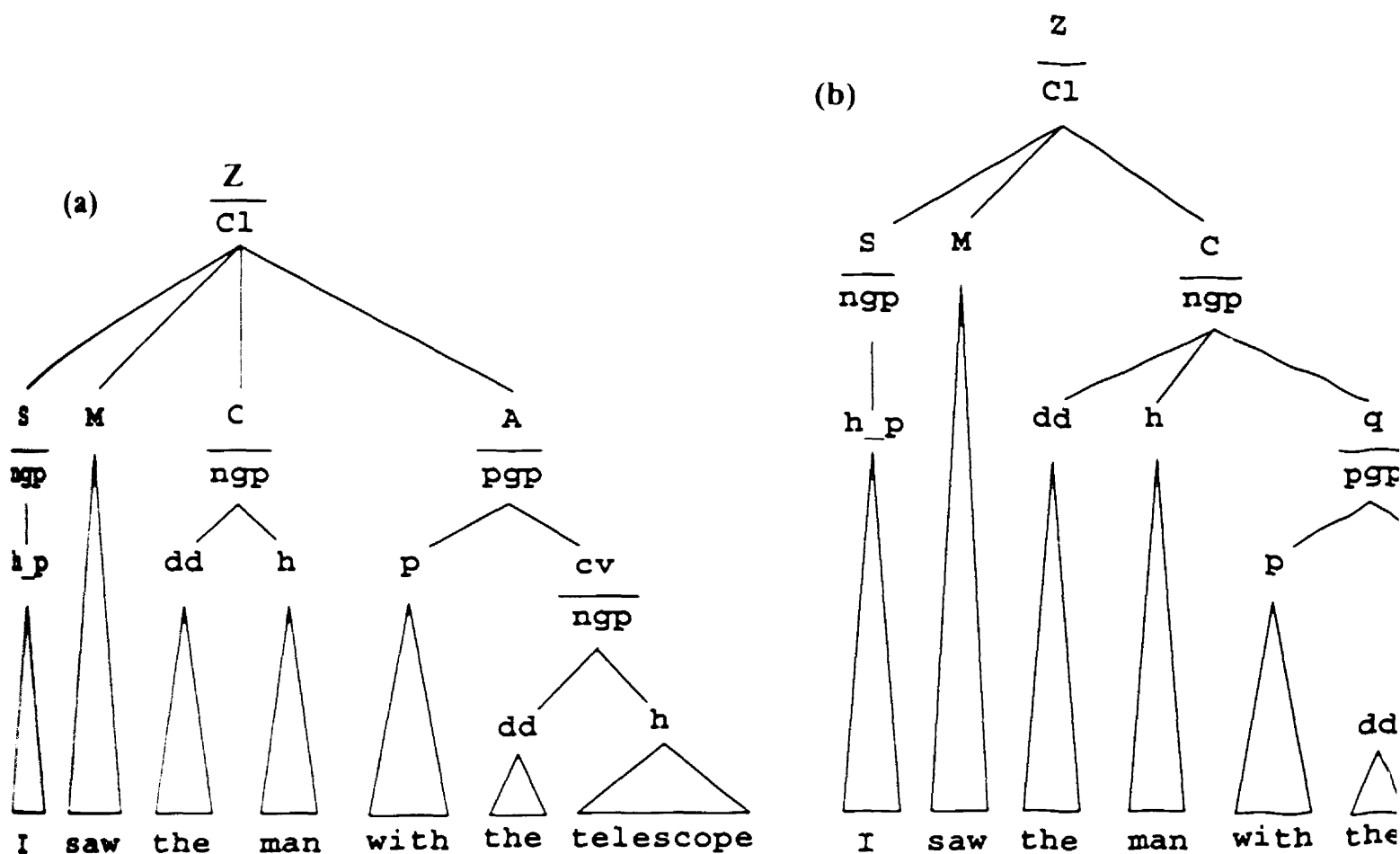


Figure 17.2: The two interpretations of test sentence S17

Table 17.12 shows that the parser found both analyses of the sentence shown in Figure 17.2. Figure 17.3 provides XML representations of the sentences that were provided by the Parser WorkBench.

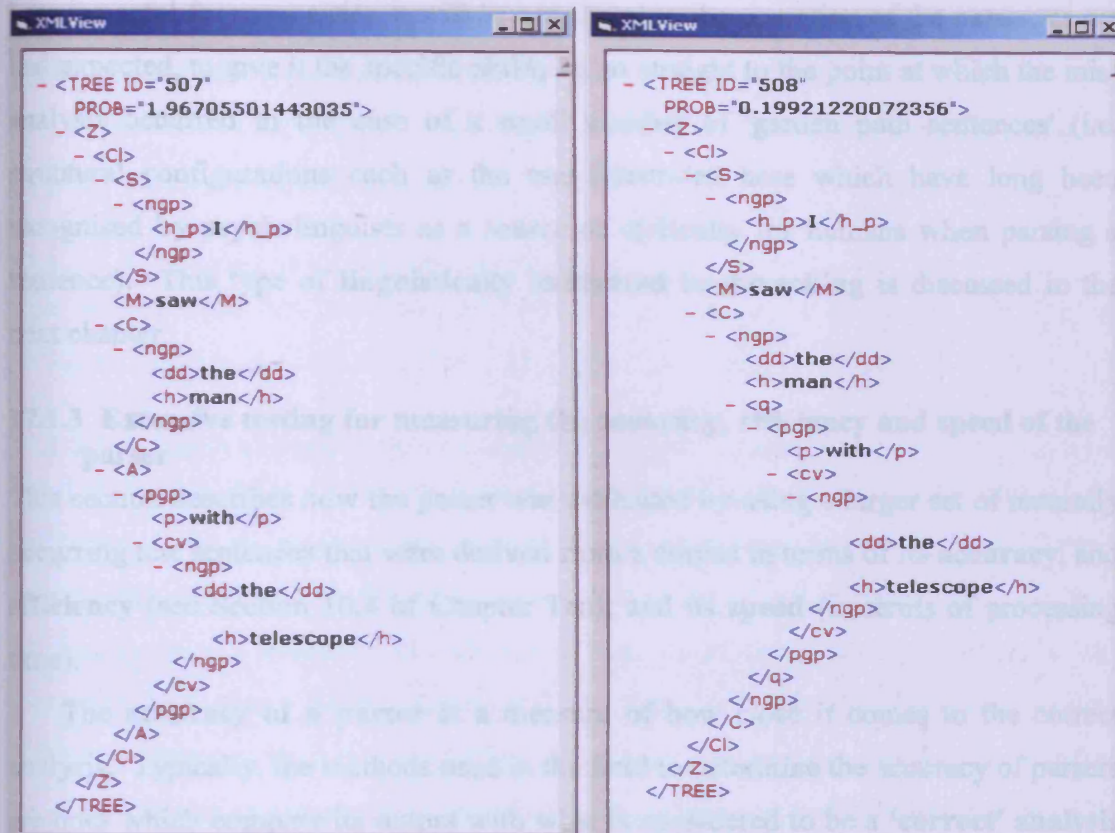


Figure 17.3: The two analyses of Sentence S17 in XML View

17.1.2.2 Tests using a sentence that required backtracking

To test backtracking, the sentence S18 was used.

S18: The players played Saturday were dropped.

This sentence caused the parser to backtrack because **played** is a Main Verb (M), and the candidate structure containing it can be joined to the primary Clause with a fairly high degree of confidence.¹⁵

The parser arrived at the correct interpretation after much backtracking. Each time backtracking occurred, the **join threshold** value was reduced by the **threshold division value** (see Section 17.1.1.3), and some unlikely joins were considered before the tree grow operation was performed to allow the Clause to fill a qualifier. Furthermore, as the qualifier was not the most likely element that a Clause can fill, other candidate structures were considered ahead of it.

¹⁵ This is in preference to the correct interpretation as a Main Verb in a Clause that fills the qualifier of the nominal group containing **the players**. This is a reasonable assumption. It is likely that the human parser acts in the same way, and it cannot tell that it should be attached as Clause that fills a qualifier until it reaches the item **were**.

The conclusions of this test were that, while the backtracking procedure described here is useful for most cases, it will be beneficial to the operation of the parser, as we had expected, to give it the specific ability to go straight to the point at which the mis-analysis occurred in the case of a small number of 'garden path sentences' (i.e. structural configurations such as the one illustrated here which have long been recognised by psycholinguists as a source of difficulty for humans when parsing a sentence). This type of **linguistically motivated backtracking** is discussed in the next chapter.

17.1.3 Extensive testing for measuring the accuracy, efficiency and speed of the parser

This section describes how the parser was evaluated by using a larger set of naturally occurring test sentences that were derived from a corpus in terms of its **accuracy**, and **efficiency** (see Section 10.4 of Chapter Ten), and its **speed** (in terms of processing time).

The **accuracy of a parser** is a measure of how close it comes to the correct analysis. Typically, the methods used in the field to determine the accuracy of parsers are ones which compare its output with what is considered to be a '**correct**' analysis for the same sentence. A score can be assigned which represents how close the two representations are, and when these tests are performed on a suitably sized set of sentences, an average score can be calculated that represents a more accurate indication of the overall accuracy of the parser. Details of the methods we used to determine the accuracy of the parser are given in Section 17.1.3.1 and the results are given in Section 17.1.3.2.1.

The **efficiency of a parser** can be defined as the amount of effort that the parser expends in reaching its conclusion. In terms of the parser described here, this can be measured with a count of the number of built structures and candidate structures it has created (including the ones which do not feature in the final analysis). A comparison of this count with similar measures for other parsers when parsing the same, or similar length sentences, can be used as a rough indication of the parser's efficiency.¹⁶ Details of how we tested the parser's efficiency, and the results of the tests are given in Section 17.1.3.2.2.

¹⁶ For example, a comparable measure could be the number of edges produced by a chart parser.

A general measure of the **speed of the parser** can be obtained by recording the length of time it takes to produce a complete analysis of each test sentence. Section 17.1.3.2.3 provides details of the results for the test sentences.

Opportunities for improving the algorithm that have been identified following the extensive testing are given in Section 17.1.3.2.4.

17.1.3.1 Measuring the accuracy of the parser

This section describes how the accuracy of a parser can be measured and it gives details of the results of the tests that were performed using the CCPP.

17.1.3.1.1 Current approaches to evaluating parser outputs

Since the mid-1990's there has been a growing interest in standardisation of the techniques used to evaluate the output of one parser against others, and Carroll, Briscoe and Sanfilippo (1999) give a comparative list of the methods that have been commonly used. Probably the most well known of these was developed by the Grammar Evaluation Interest Group (GEIG) (see Grishman, Macleod and Sterling 1992), and this was used in the PARSEVAL project. It provides a metric that is based on a comparison of the depth of two syntax diagrams – one of which is produced by the parser, and the other is what is considered to be the ‘**correct**’ analysis of a test sentence. A score is applied that indicates the number of matches found in both structures.¹⁷ A broad indication of the accuracy of the parser can be achieved by getting the average score for the parser when it is used on a suitably sized set of test sentences, and the scores for different parsers can be compared by having them all analyse the same test set. The PARSEVAL project has also shown that a comparison of the accuracy of parsers that use different linguistic theories can be achieved by ignoring the syntax labels and simply comparing the count the number of levels in each tree.¹⁸ The simulated annealing approach to parsing (see Section 11.2 of Chapter Eleven) is an example of a parser that requires an evaluation metric (Haigh et al 1988).

17.1.3.1.2 Selecting a set of test sentences

To evaluate the accuracy of our parser, we need a set of test sentences which will provide the 'correct analyses'. Because the FPD corpus contains analysed sentences

¹⁷ Typically, a bracketed notation has been used, and the measure is a count of matching brackets.

¹⁸ However, these methods have received criticism when they are used in this way due to the differences in the linguistic frameworks being used (see for example Sampson 2000).

that are annotated using the Cardiff Grammar, it gives an ideal source. We can assume that the parser will perform well with such a test sample (because it draws its probabilities from the same corpus). Therefore, we also plan to test the parser using other test sets that have been extracted from other corpora (such as the computer manuals corpus that is provided on the **AMALGAM** website (www.comp.leeds.ac.uk/amalgam/amalgam)). We cannot do this, however, until the coverage of the **I2E** and **I2E2U2E** probability tables has been increased in the Version Two tables during Phase Two of this project, see Section 14.2 of Chapter Fourteen, Appendix H and Appendix I).

To create the test set for Phase One, 100 sentences were selected at random from the FPD Corpus and modified using the following criteria:

- ellipsis markings were removed from the test set,
- sentences with the questionable analysis annotations were not included,
- annotations indicating that a unit is unfinished were removed,
- sentences of two words or less were not accepted.

The sentences were extracted from the corpus and saved in two formats:

- without annotations (these were used as an input to the parser),
- fully analysed version in XML format (which were used to provide the 'correct analyses' for comparison with the parser output).

Before testing commenced, some simple checks were performed on the test set to make sure that the set was representative of the corpus. Although we found some differences, we decided that the set was roughly representative because the differences were probably due to the fact that we did not include single word sentences or those that contained questionable analysis¹⁹

17.1.3.1.3 Defining the methods used for scoring parses

This section discusses the methods that can be used to score analyses by comparing the output of the parser with the 'correct' analysis in the test set.

As we have used the same XML annotations in the corpus and the output of the parser, I considered adapting the methods that are commonly used for detecting differences between two XML documents.

¹⁹ The test set was considered roughly representative although there were small differences. First, the average sentence was longer in the test set (7 compared to 11). Second, the longest sentence in the corpus contains 74 words, compared to 42 words in the test set. Finally, the spread of child ages in the test set was slightly different to that found in the corpus.

17.1.3.1.3.1 XML Differencing

Chawathe et al (1996) created an algorithm that is widely adopted by the creators of XML differencing tools.²⁰ They define the difference between two hierarchical structures as a **minimum set of edit operations** that can be applied to the first structure to transform it into the second. These operations, which are expressed in an **edit script**, are:

- **deletion** (of a sub-tree from the first structure, so it does not exist in the second)
- **insertion** (of a new sub-tree in the second structure that existed in the first)
- **update** (of node labels or values in the first structure, to give the same values in the second)
- **move** (of a sub-tree to a new location in the second structure, so it matches its location in the first)

When the two structures are identical, the edit script is empty. When they are different, the number of operations represented in the edit script can be used as a measure of their difference. The minimum set of operations is needed to transform the two analyses given in Figure 17.4 and Figure 17.5 are:

INSERT (P22, "q", P10, 3) = insert a new node with id **P22** with label **q** to parent **P10** at sibling position 3 (i.e. add a **q** to the **ngp** that contains **the fish** and place it after the **h**).

MOVE (C16, P22, 1) = move tree with parent **C16** so that its new parent is **P22** and place it in the first position (ie move the **pgp** so that it now fills the **q**).

DELETE (C15) = delete node **C15** (i.e. delete the old **C**).

The **move operation** is important when the technique is used for comparing structures that represent syntax diagrams. This is because it will be used in cases where a sub-tree has an incorrect attachment. Experiments with commercially, and freely available tools that I performed, revealed this operation was poorly supported, and many recognised a sub-tree was that was moved to a different location as a combination of an insertion and a deletion operation.²¹ The tools tested were:

- DeltaXML (www.deltaXML.com)
- XML Diff and Merge (by IBM) (<http://alphaworks.ibm.com/tech/xmltreediff>)
- ArborText Epic Editor (<http://www.ptc.com/products/arbortext>).

Because of the poor support for the move operation, the results of an XML difference returned by these tools, were not considered adequate for our purposes.

²⁰ An example of an XML difference tool based on Chawathe et al 1996 can be found in Mouat 2002. Such tools are in wide use in documentation systems where it is important to identify the changes between revisions of a document.

²¹ In fact, they recognised a moved structure as being a newly inserted structure.

Therefore I decided to implement my own scoring algorithm, and this is described next.

17.1.3.1.3.2 The XML vertical strip scoring algorithm

The task of detecting the difference between two hierarchical structures that represent different syntax analyses of the same sentence is more straightforward than comparing two general XML structures. This is because they have the following qualities:

- both have identical numbers of leaf nodes (and therefore identical numbers of vertical strips),
- leaf nodes are the only nodes that contain data.

A comparison of the pair of vertical strips that occur at the same **location** in both structures can be used to assign a score that represents the difference between these strips. The difference between the two structures can then be represented by considering the scores for all pairs of strips from both structures (for example, by taking the mean of the scores for each pair of strips).

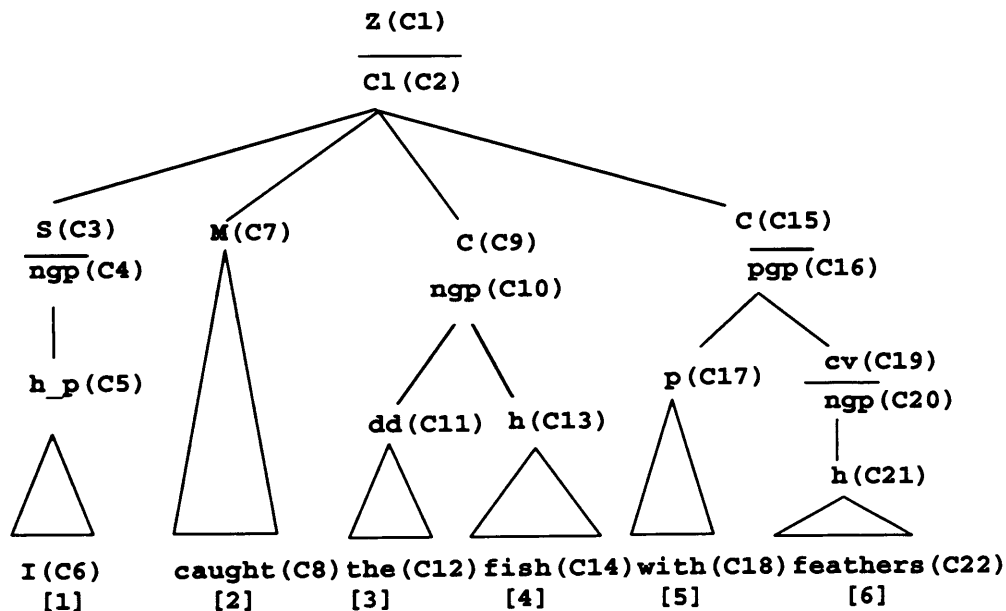


Figure 17.4: The 'correct' analysis of an example sentence

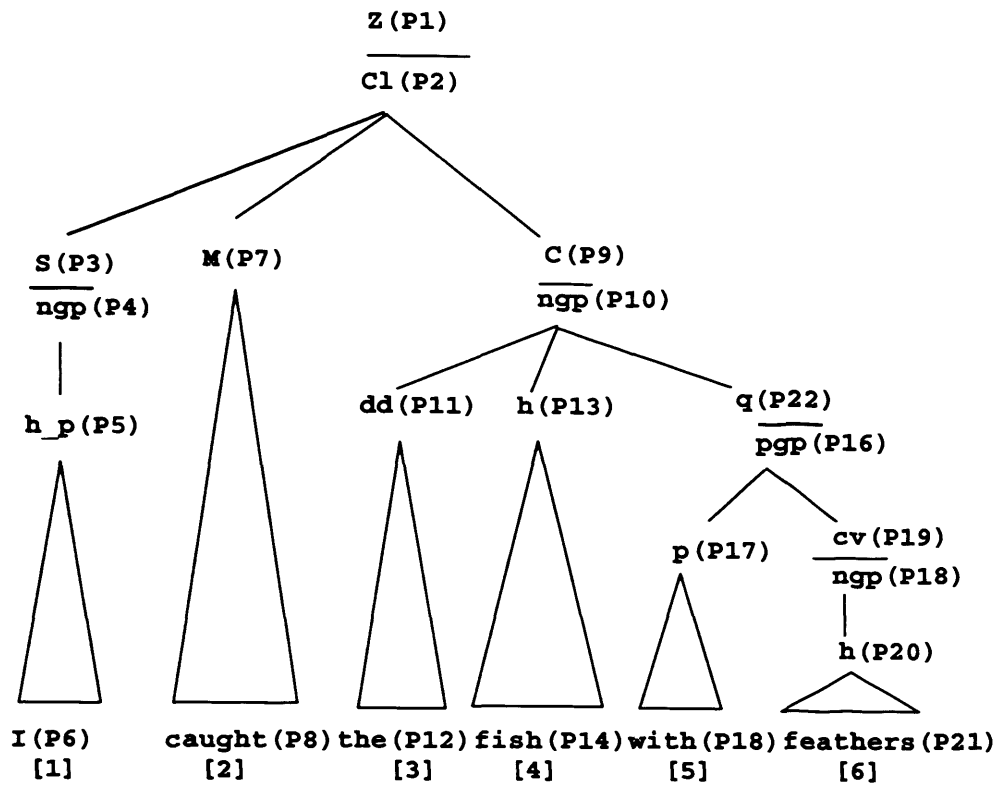


Figure 17.5: A possibly incorrect analysis from the parser²²

Strip number	Structure One The 'correct' vertical strip (CVS)	Structure Two The parser vertical strip (PVS)	Difference score
[1]	I h_p ngp S Cl Z	I h_p ngp S Cl Z	5/5 = 1
[2]	caught M Cl Z	caught M Cl Z	3/3 = 1
[3]	the dd ngp C Cl Z	the dd ngp C Cl Z	5/5 = 1
[4]	mackerel h ngp C Cl Z	mackerel h ngp C Cl Z	5/5 = 1
[5]	with p ppgp C Cl Z	with p ppgp q ngp C Cl Z	5/7 = 0.71
[6]	feathers h ngp cv ppgp C Cl Z	feathers h ngp cv ppgp q ngp C Cl Z	7/9 = 0.77
Total			5.48 / 6 = 0.91

Table 17.14: Calculating the score of an analysis

Figures 17.4 and 17.5 show two possible analyses of the same sentence. In these figures, the identifiers in round brackets represent the unique XML element identifiers, and the numbers in square brackets are the vertical strip numbers. Table 17.14 shows a simple technique for calculating a score that represents the difference

²² Note that the numbers in brackets represent the XML element Identifiers. I have added a prefix "C" in the 'correct analysis' and a prefix "P" in the parser analysis. This is to indicate in the examples that follow that the same identification number in both trees does not indicate that the element or unit is necessarily the same element or unit.

between the analyses in Figures 17.4 and 17.5. It can be seen that the same fault affects Strips 5 and 6 (i.e. the prepositional group (**ppp**) fills a qualifier (**q**) rather than a Complement (**C**)). Therefore, two modifications were made so that such differences affected the score once.

First, a score of 1 is given to each instance of an element or a unit in a vertical strip that has already been recognised as being incorrect in a previous vertical strip.

Second, because two strings that represent a vertical strip in both structures at the same position can be the same even though they contain different instances of particular elements and units. To apply a score correctly, we need to make sure that they represent the same element or unit. In Figure 17.4 and 17.5, for example, it can be seen that the two vertical strips that look upwards from the item **with** can be expressed as:

CVS: p[C17,5,5] ppp[C16,5,6] C[C15,5,6] C1[C2,1,6] Z[C1,1,6]

PVS: p[P17,5,5] ppp[P16,5,6] q[P22,5,6] npp[P10,3,6] C[P9,3,6]
C1[P2,1,6] Z[P1,1,6]

where **CVS** represents the 'correct vertical strip' (from Figure 17.4) and **PVS** represents the parser vertical strip (from Figure 17.5). The numbers in brackets that follow the syntax token represent the XML identifiers and the start position and the end position of the XML element.

There are two complements in the analysis in Figure 17.4; one (**the fish**) starts at Position [3] and ends at Position [4], and the other (**with feathers**) starts at Position [5] and ends at Position [6]. We can say that the **scope** of the XML elements that represent the first complement is (3, 4) and the second is (5, 6). The scope of an XML element in each vertical strip can be used to determine that it represents the same instance of the element or unit in both structures. Figure 17.6 is used to further explain the concept, it shows a parser's misanalysis of **the farm boy** as a pair of complements, and its correct analysis as a single complement.

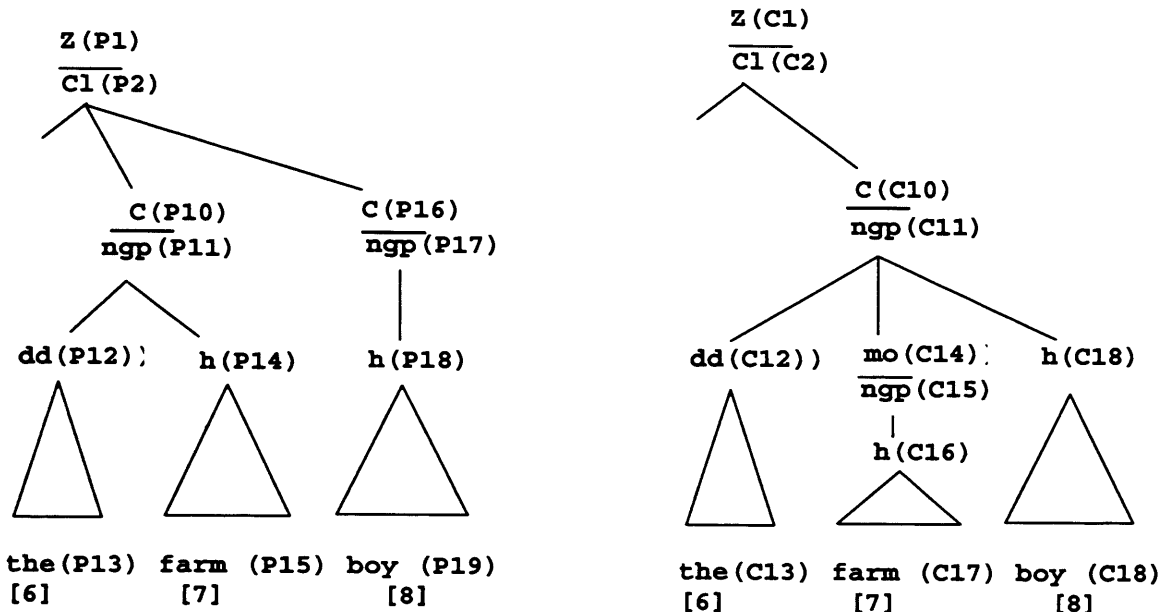


Figure 17.6: Two analyses of the farm boy

The parser's vertical strips (PVS) and the 'correct' vertical strips (CVS) in Figure 17.6 are given below:

PVS [6] = dd [P12, 6, 6] ngp [P11, 6, 7] C [P10, 6, 7] C1 [P2, 1, 8] Z [P1, 1, 8]
 CVS [6] = dd [C12, 6, 6] ngp [C11, 6, 8] C [C10, 6, 8] C1 [C2, 1, 8] Z [C1, 1, 8]

PVS [7] = h [P14, 7, 7] ngp [P11, 6, 7] C [P10, 6, 7] C1 [P2, 1, 8] Z [P1, 1, 8]
 CVS [7] = h [C16, 7, 7] ngp [C15, 7, 7] mo [C14, 7, 7] ngp [C11, 6, 8] C [C10, 6, 8]
 C1 [C2, 1, 8] Z [C1, 1, 8]

PVS [8] = h [P18, 8, 8] ngp [P17, 8, 8] C [P16, 8, 8] C1 [P2, 1, 8] Z [P1, 1, 8]
 CVS [8] = h [C18, 8, 8] ngp [C11, 6, 8] C [C10, 6, 8] C1 [C2, 1, 8] Z [C1, 1, 8]

It can be seen for example that although the syntax tokens for strip 8 (h ngp C C1 Z) are the same for the PVS and the CVS, the scope of the second complement in the PVS (with identifier P16) is (8, 8), and the scope of the single complement in the CVS (with identifier C10) is (6, 8). The difference in the scope can be used to indicate that the complements in both strips are different instances of complement.

The score ($S_{vs}[i]$) for each vertical strip at each position [i] is therefore defined as the number of correct XML elements (C) (after accounting for those that have been scored as being incorrect before, and the scope of the XML elements) divided by the maximum of the number of XML elements in the parser vertical strip and the number of XML elements in the correct vertical strip (N).

$$S_{vs}[i] = C / N$$

The score for the complete analysis (S_a) is the sum of the scores for each strip divided by the number of vertical strips in the analysis (N_{vs}):

$$S_a = \left(\sum_{i=1}^{i=N_{vs}} (S_{vs}[i]) \right) / N_{vs}$$

Using the two modifications to the simple scoring algorithm (which was shown by example in Table 17.14), the modified scores are shown in Table 17.15. In this example the score of 0.4 for Strip 5 was calculated as follows: 1 each for the preposition (p), the prepositional group (pgp), the Clause (Cl) and the Sentence (Z). All other elements and units score 0 - including the Complement (C) (because its start position in the PVS indicates that it is a different Complement than the one in Strip 4). Further, Strip 6 scores 1 because it was penalised for the attachment of the prepositional group (pgp) and the extra Complement (C) in Strip 5.

Strip number	Structure One The 'correct' vertical strip (CVS)	Structure Two The parser vertical strip (PVS)	The Score
[1]	I h_p ngp S Cl Z	I h_p ngp S Cl Z	5/5 = 1
[2]	caught M Cl Z	caught M Cl Z	3/3 = 1
[3]	the dd ngp C Cl Z	the dd ngp C Cl Z	5/5 = 1
[4]	mackerel h ngp C Cl Z	mackerel h ngp C Cl Z	5/5 = 1
[5]	with p pgp C Cl Z	with p pgp q ngp C Cl Z	5/7 = 0.71
[6]	feathers h ngp cv pgp C Cl Z	feathers h ngp cv pgp q ngp C Cl Z	9/9 = 1
Total			5.71/ 6 = 0.95

Table 17.15: Calculating the score of an analysis

17.1.3.2 Summary of the results of the tests using the FPD test set

This section provides a summary of the results of the extensive testing using 100 sentences in the test set that was extracted from the FPD corpus. Full details of the test set and the results of the tests can be found in Table M.1 of Appendix M.

Unless otherwise stated in the remarks column of Table M.1 of Appendix M, the tests were conducted using the parameters given in Table 17.7, and the parser was not allowed to backtrack. The results were noted in terms of the parse time, the number of structures created, and the score assigned by the XML vertical strip scoring algorithm. When more than one analysis was returned, only the highest XML vertical

strip score was recorded, and normally (unless otherwise stated in the remarks column of Table M.1) this was the most likely analysis returned by the parser.

The following sections provide a summary of these results in terms of the **accuracy**, the **efficiency**, and the **parsing speed**.

17.1.3.2.1 Tests for the accuracy of the output from the parser

The results from the tests were encouraging. The parser produced at least one analysis for all of the test sentences, and it can be seen in Table 17.16 that it arrived at the correct parse for approximately 50% of the test sentences, and a further 15% contained very minor errors. 78% of the sentences produced an analysis that scored above 90%. The remaining 22% of analyses contained either a greater number of minor errors, or the analysis was structurally incorrect.

Difference score range	Number sentences	Remark
1	49 (49%)	The parser analysis matched the corpus analysis
0.95 – 0.99	15 (15%)	Typically these analyses contained one or two incorrect element or attachments
0.90 – 0.94	14 (14%)	More than two problems occurred in this range
0.85 – 0.89	6 (6%)	
0.80 – 0.84	5 (5%)	
0.75 – 0.80	5 (5%)	
0.70 – 0.75	1 (1%)	
0.65 – 0.69	5 (5%)	Multiple types of problems of all sorts occurred in these sentences

Table 17.16: Summary of parser accuracy tests

17.1.3.2.2 Tests for the efficiency of the parser

The efficiency of the parser can be measured as the number of structures it creates in arriving at a solution - including those that do not form part of the final analysis. It can be seen in Table 17.17 that there is an increase in the number of structures created when longer sentences are parsed (although there is not a direct relationship between sentence length and the number of structures created). For example, a sentence of 17 words caused the parser to create more structures than sentences containing 18, 19, 22 or 23 words. This is due to the fact that different operations were involved in creating the analysis, and generally, a greater number of grow operations (see Section 15.2.5 of Chapter Fifteen) will provide a greater number of structures.

Although a **built structure** and a **candidate structure** are not equivalent to an edge in a chart parser, the number of structures produced by the CCPP and the number of edges produced by a chart parser can be used as a comparison of the efficiency of the two algorithms. The prototype chart parser that was created for this project (see Chapter Twelve) uses similar corpus queries to the CCPP. The chart parser uses the results of each query to produce a number of edges in the chart. The same query is used by the CCPP and this results in a tree operation which produces a new active tree (see Chapter Fifteen). If the CCPP had large enough values of n (for the n -best parameters), there could be the same number of trees produced as edges that would be in the chart. It is by using the n -best approach that the CCPP displays deterministic properties. Therefore, a comparison of the number of edges produced by the chart parser and the number of trees created by the CCPP for the same sentence, can be used to indicate the efficiency savings that can be attributed to the deterministic approach of the CCPP.

Souter (1996) lists the number of chart edges that his parser produced for a set of 23 sentences. Table 17.18 shows the average number of edges reported by Souter (1996:127), and the number of structures built by the CCPP for the sentences of the same length. There are approximately 98% fewer structures built by the CCPP, and we can therefore claim that the approach is more efficient than that of the standard chart parsing algorithm when it is used in the same linguistic framework.

Sentence length (words)	Number sentences	Average number of structures	Sentence length (words)	Number. sentences	Average number of structures
3	2	134.0	17	1	1072.0
4	11	121.4	18	3	886.3
5	6	174.1	19	1	569.0
6	11	229.3	21	2	1303.5
7	12	251.5	22	1	773.0
8	7	402.4	23	2	814.0
9	6	376.0	24	1	1259.0
10	1	265.0	25	2	1018.5
11	8	482.7	26	1	527.0
12	2	387.0	27	1	1264.0
13	2	598.0	28	2	1094.0
14	3	473.0	37	1	2201.0
15	2	914.0	42	1	2564.0
16	4	692.7	43	1	2193.0

Table 17.17: Measuring the efficiency of the parser

Sentence length	CCPP (No. structures)	Souter's chart parser (No. edges)	Difference
3	134.0	1696.8	92%
4	121.4	5851.0	97%
5	174.1	10548.0	98%
6	229.3	15073.0	98%
8	402.4	19518.0	98%

Table 17.18: Efficiency comparison - CCPP and chart parser

17.1.3.2.3 Tests for the speed of the parser

This section provides a summary of the results that show the speed of the parser in terms of the time it takes to produce an analysis for a sentence. The measure of the speed of the CCPP is defined as the time it takes from the end of the parser initialisation stage (Stage 0, which clears information from previous parses) to the time it reaches Stage 7, where it reports its success or failure.

Table 17.19 shows the average parse times (in seconds) for different sentence lengths, and has been provided for comparison between the performance of the CCPP and other parsers. Like the measure for efficiency, the time taken for a parse is determined by the number of structures produced rather than sentence length, and sentences that require a greater number of grow operations will take longer to process (this relationship can be seen in Table 17.20).

Sentence length (words)	Number sentences	Average parse time (seconds)	Sentence length (words)	Number sentences	Average parse time (seconds)
3	2	13.66	17	1	231.00
4	11	22.27	18	3	152.50
5	6	37.00	19	1	141.00
6	11	40.72	21	2	239.00
7	12	55.18	22	1	193.00
8	7	73.85	23	2	190.50
9	6	69.66	24	1	299.00
10	1	46.00	25	2	280.00
11	8	85.12	26	1	158.00
12	2	74.00	27	1	297.00
13	2	125.00	28	2	389.00
14	3	98.00	37	1	662.00
15	2	151.50	42	1	860.00
16	4	132.8	43	1	731.00

Table 17.19: Measuring the speed of the parser (sentence length / parse time)

Number trees	Number sentences	Average parse time (seconds)	Number trees	Number. sentences	Average parse time (seconds)
0 - 99	13	16.00	700-799	2	185.50
100-199	14	26.50	800-899	4	180.50
200-299	15	40.20	900-999	3	282.33
300-399	11	63.63	1000-1099	2	266.50
400-499	12	83.83	1100-1199	1	272.00
500-599	9	116.55	1200-1299	3	324.33
600-699	5	121.20	over 1300	6	536.80

Table 17.20: Measuring the speed of the parser (structures created / parse time)

The chart parser that was developed for this project (reported in Chapter Twelve) gave parse times for sentences of a similar length in tens of minutes, with Sentence S2 (see Table 17.8) requiring 1500 seconds (25 minutes). The time taken to parse this sentence in the CCPP was 100 seconds (1 minute 40 seconds). This represents an improvement of 1400 seconds (or 23 minutes 20 seconds); this was typical of the savings for all the test sentences.

The parse times were also a significant improvement over those reported by Souter (1996) for similar length sentences. Although the computer hardware technology has significantly improved since the times when these two parsers were developed, much of the improvement is due to the deterministic approach used by the CCPP.

While the parse times reported here are similar to those in Weerasinghe (1994), one has to take into consideration the very much larger number of syntactic relationships that are used by the CCPP. Weerasinghe's parser operated using comparatively small syntax lists and lexicons, while the present parser has been constructed from the start, in such a way as to enable it to operate on unrestricted natural texts.²³

Let us now consider the question 'what is a reasonable time for a parser to take to arrive at a correct analysis?' The answer depends, of course, on the length and complexity of the sentence being parsed. It also depends upon the use to which the parser is being put, and upon the richness of the analysis that is required from it. Parse times measured in tens of seconds are reasonable for, say, a batch processing algorithm which adds further sentences to a parsed corpus working on unanalysed files of natural data. It is also reasonably close to being acceptable in a system where a rapid response is needed. A slow parser, however, is very unsatisfactory in

²³ The Version Two probabilities tables contain over a million items (see Appendix I).

research. For example, when testing the chart parser reported in Chapter Twelve, it was annoying to have to wait for tens of minutes for the parser to finish only to find at the end that it has made a mistake. In contrast, it is acceptable to wait for tens of seconds for a result.

The point must be made, however, that the difference in parse times between Souter's (1996) parser and those of the chart parser described in Chapter Twelve, is likely to be due to the fact that the speed of computers have increased substantially in this period, and even without improvements to the parsing algorithm presented here, we can expect that parse times will improve even further in a relatively short space of time.

17.1.3.2.4 Opportunities for improvement

The most common type of problem experienced by the parser was the misanalysis of an element in the Clause and the most common was a particular unit filling a Complement rather than an Adjunct, and the correct selection of different types of adjunct, or replacements. The parser often did not recognise adequately that a unit is unfinished.

Further, the detection of embedded nominal groups as modifiers also caused a problem. Examples can be seen in the sentences below:

It's about the farm boy...(12dgism#15) (**farm** is analysed as being the head of the nominal group that contains the deictic determiner **the** instead of being in its own nominal group that fills a modifier, see Figure 17.6).

A lego boat sailing...(6abpscj#34) (**lego** is analysed in the same nominal group as the quantifying determiner **a**).

The nominal groups in the test set were of fairly simple construction, and improvements will be needed to handle more complex examples. The parser did, however, very reliably recognise handle quality and quantity groups as embedded modifiers and qualifiers.

The item **to** as the **infinitive element** in a few cases caused problems because the parser preferred its analysis as a preposition in a prepositional group.

17.2 An evaluation against the project's goals

As stated in Section 13.1 of Chapter Thirteen, the primary goal of this project is to build a parser that can take any string of items (words) that might occur in a natural text, and to turn this string into a **syntax tree diagram**, using the rich annotations of the Cardiff Grammar version of Systemic Functional Grammar (SFG).²⁴

In Chapters Eleven and Twelve, I discussed a number of other approaches to parsing that use SFG, including two early parsers that were built as part of this project. The conclusions of that work were that a more deterministic approach to parsing (i.e. one which simulates more closely that of a human when analysing a text sentence) gives the greatest promise of success, especially for parsing unrestricted natural texts. The second aim of the parser described here is therefore to 'get it right first time', whenever possible, and to use linguistic knowledge to help ensure that it backtracks only when it has to.

Our third aim was to demonstrate that a parser whose method of parsing is integrated with a **corpus database** gives an approach to parsing that provides both efficiency and a broad coverage of texts. Such a parser uses corpus-consulting, database-oriented methods to extract probabilities, and so determine the best paths to take.

The fourth aim was to prove that corpus-based parsing can benefit from an approach that closely follows the changing properties of real language by using a dynamic corpus. This will be discussed in Chapter Eighteen.

Did the parser meet the first three of these aims? The answers are discussed in Sections 17.2.1, 17.2.2, and 17.2.3 respectively.

17.2.1 How far is the output from a richly annotated syntax diagram

Figure 17.7 shows the most likely parse for the example sentence **S10** (see Table 17.1). It provides a horizontal representation of the tree diagram, in which the branches can easily be expanded and collapsed.²⁵ An XML representation of the sentence can be seen in Figure 17.8, and this is the same form in which it is loaded into the corpus database.

²⁴ However, we do not attempt the difficult task of parsing an analysis of the **Participant Roles** in Phase One; this important refinement will be added, with others, in Phase Two.

²⁵ Because the sentence is displayed in the Parser WorkBench, the forward predictions from the head (**h**) of the nominal group (**ngp**), and from the Complement (**C**) of the Clause (**C1**) can be clearly seen.

Figure 17.9 shows the structure of the sentence in a representation that conforms to the way in which linguists typically analyse sentences. This type of representation can be created by processing the XML representation (given in Figure 17.8) through an XML formatting program that uses an XSL-FO stylesheet.

Figure 17.10 shows the sentence as loaded in the corpus database and viewed in ICQF+'s Sentence Viewer. These five illustrations display the output formats available from the parser and hence achieve the goal of creating a richly annotated syntax diagram.

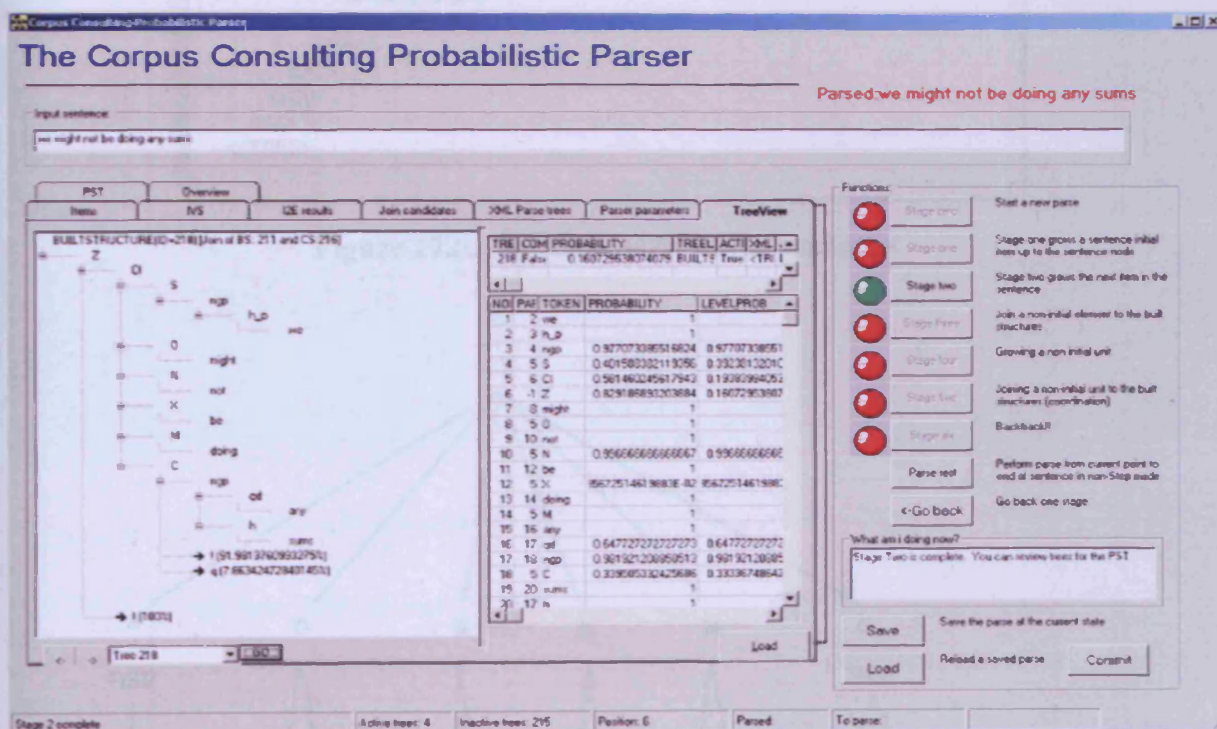


Figure 17.7: Formatted parse tree in the Parser WorkBench


```

XMLView
- <TREE ID="218" PROB="16.0729538074079">
- <Z>
- <Cl>
- <S>
- <ngp>
  <h_p>we</h_p>
  </ngp>
</S>
<O>might</O>
<N>not</N>
<X>be</X>
<M>doing</M>
- <C>
- <ngp>
  <qd>any</qd>
  <h>sums</h>
  </ngp>
</C>
</Cl>
</Z>
</TREE>

```

Figure 17.8: An XML view of the sentence

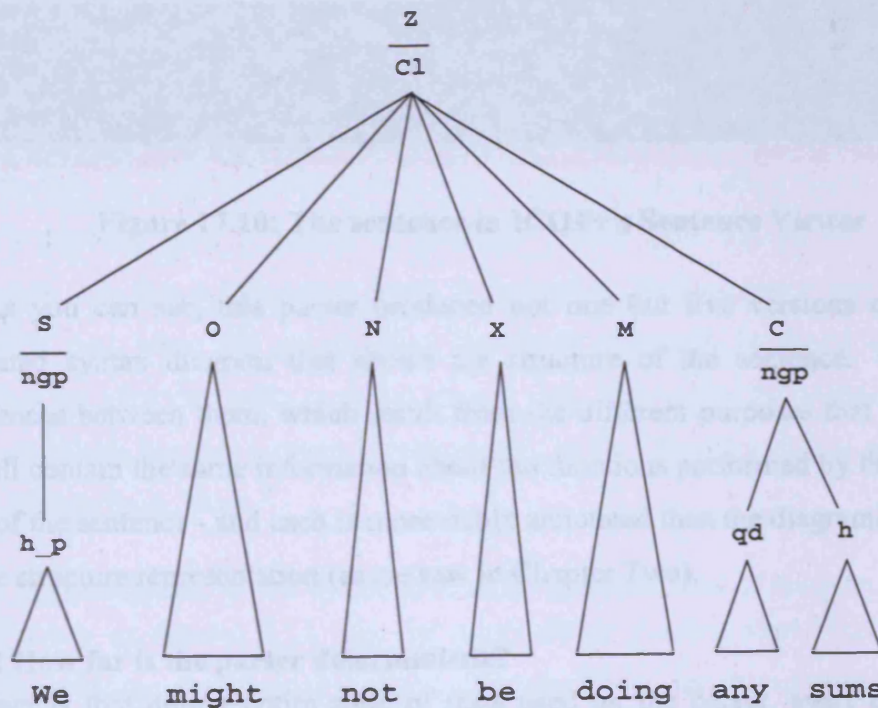


Figure 17.9: Formatted view of the parse tree

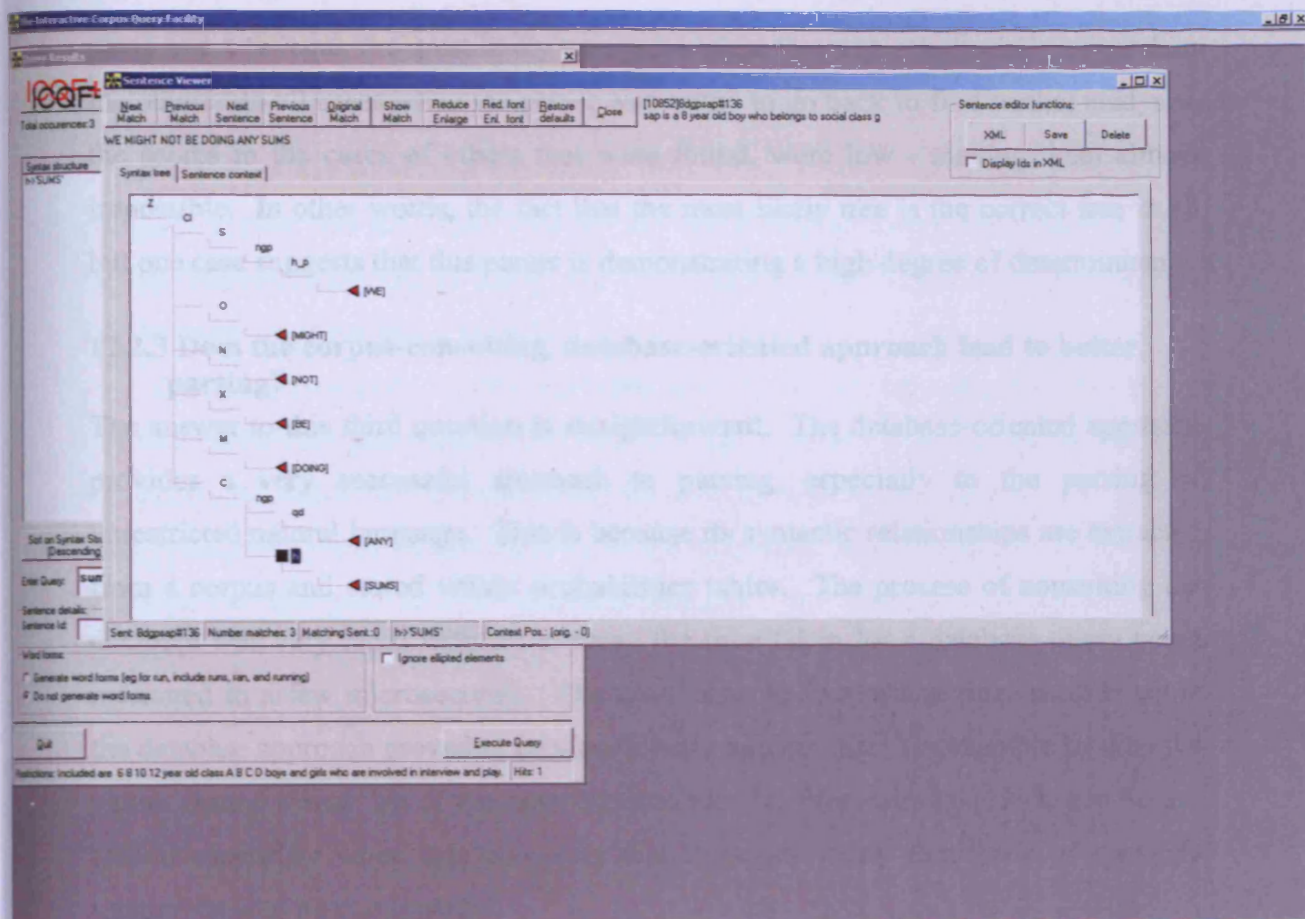


Figure 17.10: The sentence in ICQF+'s Sentence Viewer

As you can see, this parser produced not one but five versions of the richly annotated syntax diagram that shows the structure of the sentence. Despite the differences between them, which result from the different purposes that each serves, they all contain the same information about the functions performed by the words and units of the sentence - and each is more richly annotated than the diagrams of a typical phrase structure representation (as we saw in Chapter Two).

17.2.2 How far is the parser deterministic?

The fact is that during entire suite of tests used on the parser, apart from the test designed to involve backtracking, it only had to backtrack once. In most cases (ignoring the backtracking test), the most likely parse tree was ranked first, and in the case where it was not, the correct tree occurred in second position, close behind the incorrect tree. In the extensive tests, the analysis with the highest XML vertical strip score (i.e. the one that is 'most' correct) was very often ranked first. Moreover, in those cases where there were more than one parse tree produced and where only one

parse tree was legal, the most likely tree had a score that was substantially larger than the others. In all cases when the parser was asked to go back to find further analyses, the scores in the cases of others that were found, were low - making them almost impossible. In other words, the fact that the most likely tree is the correct tree in all but one case suggests that this parser is demonstrating a high degree of determinism.

17.2.3 Does the corpus-consulting, database-oriented approach lead to better parsing?

The answer to this third question is straightforward. The database-oriented approach provides a very successful approach to parsing, especially to the parsing of unrestricted natural language. This is because its syntactic relationships are extracted from a corpus and stored within probabilities tables. The process of consulting the database was very rapid with the average the time taken for a database query being measured in a few microseconds. The time taken to find and retrieve records using the database approach proved to be significantly quicker than, for example reading the values from a syntax list of the form implemented by Weerasinghe (1994) and Souter (1996) especially when one considers that there are many thousands of syntactic relationships to be manipulated.

We saw in Chapter Sixteen (and also in Section 17.1 above), the fact that the parser's working data is stored in database tables gave the parser a significant advantage. In the development stage, it helped in finding errors and bugs in the algorithm, and when the results were not those expected, it helped in answering questions on why the parser took the route it did. The ability to review the working data in this way resulted in many enhancements to the algorithm and in many changes to the probabilities tables.²⁶

17.2.4 Improvements in the speed, efficiency and accuracy of the parser

We saw in Section 17.1 that the approach to parsing used here provides an improvement over the prototype chart parser developed for this project, and Souter's and Weerasinghe's chart parsers that were reported in Chapter Twelve. The performance and accuracy will be further increased through the implementation of the Version Two probabilities tables and a number of enhancements to the algorithm (that

²⁶ Sometimes when problems were identified, they were attributed to errors in the FPD Corpus, and they were corrected in ICQF+.

will be identified in Section 18.2 of Chapter Eighteen) and include improvements for the types of problem identified in Section 17.1.3.2.4.

17.3 The results of the evaluation

While it is clear that the parser incorporates significant advances, the tests show that further improvements can be made, and I now turn to consider the ways in which we can enhance its speed and efficiency.

There are two parts of the algorithm that are computationally expensive that therefore affect the parse time. The first is Stage 6, which is backtracking, and it is clear that this procedure should clearly be avoided whenever possible. The way in which the parser seeks to achieve this, as we have seen, is to attempt to get the parse right on the first pass in a deterministic manner. But to do this, it crucial to the operation of the parser to get the optimum values and configurations of the parsing parameters (as reported in Section 17.1). For example, if the **join threshold** is too low, then the parser will let joins take place in preference to a preferable grow operation, and if the joining score formula's parameters are not correct, then the resulting parse trees will not have the correct ranking. However, as we saw in Section 13.5 of Chapter Thirteen, the method employed for backtracking in the Phase One model of the parser is one that uses **computational backtracking**. It is expected that Phase Two's **linguistically motivated backtracking** will significantly improve the efficiency of the parser in these situations when it has to backtrack.

The second part of the algorithm that is computationally expensive (though not as expensive as backtracking), is that of **growing** the candidate structures in Stage 4, after a join attempt fails in Stage 3. This is because of the number of elements that a class of unit can fill. For example, in the FPD Corpus upon which the initial unit-up-to-element tables are based:

- (a) the Clause (**C1**) can fill 24 elements,
- (b) the nominal group (**ngp**) can fill 35 elements,
- (c) the prepositional group (**pgp**) can fill 19 elements,
- (d) the quality group (**qlgp**) can fill 23 elements,
- (e) the quantity group (**qtgp**) can fill 16 elements.

The implications of this are that Stage 4 is the second most expensive part of the algorithm due solely to the number of candidate structures that are produced as a

result of the grow operation. As can be seen in Table 17.8, the sample parses that include one or more Stage 4 operations have significantly longer parse times, and result in a greater number of built and candidate structures in the parser's working tables. This is an area for improvement that will be discussed in Chapter Eighteen.

17.4 Summary

This chapter has described the ways in which we tested the parser. An important prerequisite of this work was the establishment of the parser's configurable parameters, these include for example, the value of n in the places of the algorithm where the n -best trees or vertical strips are taken forward, and the optimum configuration of the parser's joining score parameters. This work was presented in Section 17.1.

Also in Section 17.1 we looked at the results of the tests, and these were overall very encouraging. Using the simple set of sentences (in Table 17.1), the parser found the correct parse at the first attempt in all but one case. It only had to backtrack in one case, and this was avoided in a later run by changing the parser parameters. We identified the fact that the establishment of the optimum set of parameters is a central aspect in the operation of the parser, because this enables it to avoid it having to perform the most expensive part of the algorithm (i.e. backtracking in Stage 6). The success of our choice of these parameters was confirmed during the extensive testing using a set of sentences derived from the corpus.

In order to provide a metric that allowed us to determine the accuracy of the parser, a new XML vertical strip scoring algorithm was devised and used during the extensive testing. The results of the extensive test showed that the parser returned the correct analysis for approximately 50% of the test sentences and a further 15% contained just one error. Only a few analyses contained more serious errors, and improvements have been identified to circumvent these problems in Phase Two (see also Chapter Eighteen). Also in this chapter, the results were compared with the parsers reported in Chapter Twelve - the chart parsers of Weerasinghe (1994) and Souter (1996), and our own prototype chart parser, and it was concluded that this new approach offers improvements over these earlier attempts.

In this chapter, we discussed the suitability of the corpus-consulting database-oriented deterministic approach. The results were evaluated and the strengths and weaknesses of the algorithm were identified.

In Section 17.3 we introduced two areas of potential improvement - **linguistically motivated backtracking** and **intelligent tree growing**. These and other improvements will be described in the final chapter of this thesis - Chapter Eighteen, which also discusses potential improvements to the corpus database, and to ICQF+.

Chapter Eighteen

Further work and conclusions

This chapter describes (a) the improvements that the work in Phase One of the project suggests may be desirable (or at least worth considering), in either the **parser** or the **corpus database**, and (b) the major improvements that have been planned for the start of Phases Two and Three.

18.1 The Corpus Database and ICQF+: improvements and further work

18.1.1 The native XML tables

At the moment, the only constraint on whether the database allows sentences to be loaded or not is that the XML document that represents the sentence is valid.¹ This means that parse trees that contain errors may be loaded, and these errors could be stopped by checking that the XML complies with the rules of a DTD / Schema. For example, at present, it is possible to have:

- (a) a sentence with XML mark up elements with **generic identifiers** that do not match a syntax token in the Cardiff Grammar (some of these could be difficult to spot manually, e.g. **CL** instead of **C1** or **ADO** instead of **A_DO**)
- (b) incomplete mark up elements that should contain further 'child' mark up elements or data. For example, a pronoun head (**h_p**) that is supposed to be expounded by an item but is not (i.e. `<h_p></h_p>`), a Subject (**S**) that is supposed to be filled by a unit but is not: (i.e. `<S></S>`), and a Clause (**C1**) that is supposed to have component elements but does not (i.e. `<C1></C1>`).²
- (c) A mark up element that contains the mark up attribute **ellipted** and also other 'child' mark up elements. Mark up elements that represent ellipted Cardiff Grammar elements should contain no child mark up elements.³ Thus the following mark up example is not grammatically valid but is well-formed

¹ A valid XML document is one that is 'well-formed' as it has an end tag for every start tag, and it does not have to conform to any given DTD or Schema.

² These are real examples that were found in the FPD Corpus. They were either questionable items and units or ellipted elements that were missing the necessary annotations. In each case, they were corrected.

in XML terms:

```
<S ellipted="RapidSpeech"><ngp>..</ngp></S>
```

(d) an **ellipted** mark-up attribute that contains something other than the values **RapidSpeech** or **PreviousText**.

These problems can be easily overcome by implementing of an XML DTD or Schema. This would mean continuing the work reported in Chapter Six and Appendix C. In Phase 2, therefore, I will improve the **CreateDTD** program so that a complete analysis of the XML data in the corpus can be performed. This will mean that a usable XML Schema is produced. This will also be supported through the implementation of (a) a small number of **business rules**, and (b) a **business rules checking program** (see Section 6.1.4.3 of Chapter Six).⁴ Such programs are proving extremely valuable in industry (see Day (2006), and Day and Ichizli-Bartels (2007)).

18.1.2 ICQF+

This section describes some of the modifications that could be made to improve ICQF so that can be used more widely. ICQF+ was, as we saw in Chapter Eight, a complete redevelopment of the original version (presented in Day (1993a)), and it incorporated many substantial improvements. However, most of the changes made were designed to meet the specific needs of this project, as we needed a research tool to answer the questions we needed to ask in order to develop transformation scripts to create the FPD Corpus, or to edit sentences (which we did using ICQF+'s sentence editor).

Many of the changes made were made with the needs of other users in mind, and in some cases, we drew on the experiences of the various users of the original system and / or consulting linguists on their requirements. However, there is still scope for further improvements that could significantly enhance ICQF, and these are listed here.

18.1.2.1 A graphical query builder for ICQF+

Students working in the study of language who use ICQF may not be familiar with the use of computers and writing queries in the Corpus Query Language (CQL) (see Chapter Eight) may be a challenge. For these users, a graphical query builder will allow them to express queries as diagrams and not by words (i.e. as trees). The trees can then be translated into CQL automatically when the query is executed.

³ Recall from Chapter Six that the term **element** has a meaning in mark up languages, and another meaning in Systemic Functional Grammar. I will use the term **mark up element** to disambiguate the terms, as before.

This facility could display partial parse trees and allow parts of other parse trees to be 'dragged and dropped' or 'cut and pasted' to form a new query. It would also be useful if queries could be saved and restored.⁵

18.1.2.2 XML tree grapher

Although ICQF+ provides a fully functional sentence viewer, the syntax diagrams produced are not of the format used in the field of linguistics (e.g. see Figure 2.5 of Chapter Two, or the many other examples throughout this thesis). This format could be achieved through the use of a **tree grapher**. The ability to use these diagrams in a word processor would be a great advantage when writing papers or books. It is possible to achieve this using our XML by using an XSL-T or XSL-FO stylesheet. This is not a trivial task and work is underway to achieve it.⁶ This tree grapher could also be used by the Corpus-Consulting Probabilistic Parser to display the final complete parse trees.

18.1.2.3 Controlling access to the corpus editor

The ability to modify the corpus should be restricted to certain users and should only be done in a controlled manner by individuals who realise the implications of the change. ICQF+ should therefore include a user access system.⁷

18.1.2.4 Multi-user and web-enabled support

ICQF+ is a single-user system and further work would be necessary to provide database record locking and feedback that is necessary to allow it to be used by multiple users at the same time. The work to achieve this would be fairly trivial.

A more significant improvement would be to allow ICQF+ to run on the internet, allowing users request queries and see the results online. ICQF+ was not developed in a web environment, and the work to achieve this would not be as trivial but it would make it significantly more attractive to other researchers.

18.1.3 Using the Text Encoding Initiative (TEI) standard

One of the major advantages of using XML in this project, is that it, like its

⁴ Business rules checkers are able to check, for example, that a pronoun head (**h_p**) does not occur in any unit other than a nominal group (**ngp**), and that the root element of the parse tree is a sentence (**Z**)

⁵ It would be important to retain the CQL interface for more experienced users as it remains the quickest way of retrieving information.

⁶ XSL-FO is able to create files in Scalable Vector Graphics (SVG) or Portable Document Format (PDF) among other formats.

⁷ Perhaps the easiest method of implementing this is to provide a password that must be entered correctly before such changes can be made. Alternatively, we could release a version of ICQF without its embedded corpus editor.

predecessor SGML, is a so-called 'neutral standard'. This means that it is possible to process the data in different systems. To do this with a corpus requires a DTD or Schema and a set of **business rules**. We saw in Section 6.2.1 of Chapter Six that the Text Encoding Initiative has made considerable progress in this direction.

There are significant benefits to be gained from being able to provide material from the corpus database that conforms to the TEI standard. This would mean that the FPD Corpus could, in theory, be used in other TEI compliant systems. This includes the corpus query tools developed by others, so raising interesting possibilities for collaboration with other research projects.

Equally, the import of TEI-compliant corpora into our corpus database would mean that corpora developed by others could be used, after the necessary minimum modifications within ICQF+. It could be used to prepare alternative probabilistic models of syntax (after making the appropriate adjustments), and this could provide alternative models for the parser to use.

18.1.3.1 Export in today's TEI format

For export to today's TEI format, support for the TEI headers is needed. While these can be provided in the native XML tables some effort would be required to populate these values in the format that is required for the TEI. Much of this work could be done automatically, through the intelligence provided in the corpus cell identifiers (i.e. age, sex, class, initials, situation and sentence number).

At the sentence level, export of the data could be in XML, so that it conforms to the abstract mark up model demanded by the TEI DTD (see Figure 6.7 and Section 6.2.1 of Chapter Six). In doing this, however, one would have to have knowledge of the Cardiff Grammar's relationships and what constitutes Cardiff elements, units and items, for the reasons explained in Section 6.2.1 of Chapter Six.

18.1.3.2 Import from today's TEI format

It is accepted that some mapping work may be necessary in order to be able to use the TEI data generated by others within our corpus database, but this work is made considerably easier because the data is in the TEI standard. It would be a valuable exercise to have another parsed corpus and work with it within ICQF+.

18.1.3.3 Extending the TEI model to provide a more usable format

It would certainly be worth exploring the concept of extensibility in XML, such that the mark up method below the sentence level could be the same descriptive scheme

used in this project (see Section 6.2.1 of Chapter Six). The benefits of doing this would be that the Cardiff Grammar could be properly handled in the TEI model (provided that the knowledge of how to process the information is also exchanged).⁸

18.1.4 Summary

I have identified here a surprisingly large number of potential improvements - some with an interesting major potential. Some are quite minor but nevertheless desirable. I turn next to the improvements that can be made to the parser.

18.2 The parser: improvements and further work

We are developing the parser in three distinct phases (see Section 13.4 of Chapter Thirteen), and we are currently at the end of Phase One. This phase includes building the fully functional parser and its complementary database etc, as described here. Further improvements are discussed in this section.

18.2.1 Improvements for Phase Two and Phase Three

Phase Two will include a number of additions to the model, each of which is confidently expected to constitute a significant advance on the current version. The following are discussed in this section:

- (a) testing and refining of the Version Two of the probabilities tables,
- (b) the detection of multi-word items,
- (c) the treatment of punctuation,
- (d) intelligent tree growing,
- (e) linguistically motivated backtracking,
- (f) making use of morphological information,
- (g) the introduction of units to handle names of people, dates and times etc.

The additional problems caused by discontinuous units (see Section 4.3 of Chapter Four) will be handled in Phase Three, as also will the identification of further functionally different types of Adjunct and modifier.

18.2.2 Testing and refining the Version Two probabilities tables

Work on the Version Two probabilities tables is, at the time of writing, in the final stages of completion (see Section 14.2 of Chapter Fourteen). These will be tested and developed further in Phase Two, and we anticipate that these new tables will significantly increase (a) the coverage of the parser, (b) its usability to fields other than

⁸ This extensibility would benefit other projects who wish to use the TEI format and keep the model of

children's speech, and (c) its speed and accuracy. See Appendix I for full details of the work involved in creating these new tables, and the advantages of establishing them.

18.2.3 Improved item recognition: detecting multi-word items etc.

Phase One was restricted to recognising single word items. Although these are by far the most common type, it is sometimes the case that an item can also contain more than one word (e.g. **in spite of**, **according to**, **out of**) in fact items such as these have an internal structure, and will be provided for in Phase Two. Further, a word can constitute more than one item, as in **John's book** where **John** and the **'s** are treated as two elements in a genitive cluster; other examples are **cannot**, **isn't** etc. One relatively simple extension to the parser in Phase Two will be to employ a more intelligent item recognition algorithm. This would rely on a knowledge of problem words and patterns that typically form multi-item words, and it will be permitted a small amount of 'look-ahead' in order to ascertain whether an item such as **in** is complete or is part of **in spite of** etc.

A modified algorithm will be introduced, with steps such as the following:

- (a) Check to see if the word ends in 'apostrophe-s' (**'s**) or simply 'apostrophe' (**'**). If it does, then identify both items.
- (b) If the word is a multi-item word (e.g. **cannot** or **another**), create two items (**can** and **not**, and **an** and **other**).
- (c) Many one-word items (e.g. **out**) are also potentially words in a multi-word item. When one of these words is encountered, the parser reads the next three words from the input string and consults the item-up-to-element table to see if two or more of these words form a multi-word item (e.g. **out** followed by **of**).

It will also be necessary to provide for such items within the new backtracking model (see Section 18.2.6). This is because when a parse fails, or when alternative analyses are required, the parser needs the ability to consider multi-word items as well as individual items.

18.2.4 Improved punctuation treatment

The version of the parser implemented here has support for only one type of punctuation. This occurs in Stage 5, when the comma is used to detect co-ordination

syntax that they use in a descriptive mark up annotation method.

(as in **Robin, Mike and Andrew**). Phase Two will implement procedures for the handling of the full range of punctuation as defined in Fawcett (2000a).

Punctuation marks can be handled in the parser by treating them as items, so assigning them a position, as in the following example:

0 Robin 1 , 2 Timothy 3 and 4 Christine 5 built 6 a 7 farmhouse 8 . 9

The punctuation characters can then be identified by the parser and shown to expound the starter (**st**) and ender (**e**) elements within the units. This information will also be used in the joining score formula, since an ender will always end the unit and the starter will always start its unit. The treatment of punctuation will include special functions to detect cases where the punctuation is within an item: for example, a decimal point in a number can be represented by **25.4** or **25,4** etc.

18.2.5 Intelligent tree growing

We saw in Section 17.1 of Chapter Seventeen that the existing Stage 4 algorithm, which is responsible for the growing of candidate structures after a failed join attempt, could benefit from a more 'intelligent' approach.⁹ This part of the algorithm must be improved, and there are a number of ways in which this can be done.

First, a limit could be set on the number of trees that are the result of a grow operation. Currently, the parser grows the **n** most likely candidate structures. The problem is that, if the value of **n** is set to 5, and the number of elements that each class of unit for each of those best five candidate structures can fill is (say) 25, the number of resulting new candidate structures will be 125. An arbitrary limit of say, 20 new candidate structures could therefore be set, and any others could be captured if the parser backtracks.

The first method is not sufficient, however because new candidate structures that are less likely may be built before ones that are more likely. This drawback could be overcome by using the following method instead: allow the parser to generate more than **n** trees, but have it stop when it has generated **n** trees that have a score of over a threshold value of **x%**. Here **n** and **x%** will be, of course, new parser parameters.

A third method would be to favour the growth of a candidate structure that we know can be joined to existing built structures. This would be **intelligent tree growing**, and it can be achieved by a query that determines whether the elements in

the **forward predictions** from the active built structures can be filled by the units that are at the root node of the candidate structure.

A further enhancement to this improved algorithm would be for the parser only to allow candidate structures to be grown which cannot be joined to an existing built structure only if it is likely that the element (i.e. the element that the unit of the candidate structure will fill) is able to begin a new unit. If it can, further combinations of the results of element-up-to-unit (**E2U**) and unit-up-to-element (**U2E**) queries could be used to determine the likelihood of the new candidate structure before growing it.

These modifications, I believe, would substantially improve the speed and efficiency of the parser.

I next turn to improvements that can be made to the second computationally expensive part of the algorithm - Stage 6 (backtracking)

18.2.6 Linguistically motivated backtracking

The current model of backtracking (as we saw in Section 13.5 of Chapter Thirteen and Section 15.2.7 of Chapter Fifteen) is based on the Hamilton Path Problem (Ore 1960), and is thus **computationally motivated**. As we saw in Chapter Seventeen, this means that it blindly follows the next most likely paths that have not been followed when it goes back, and it does so in the order that they occur.

In Phase Two we intend to implement a backtracking algorithm that is **linguistically motivated** and initial ideas are presented in Appendix L.

In an example such as **the people moved from their villages were returned by the NATO forces**, the discovery of **were**, which is an Operator conflated with an Auxiliary (**OX**) or an Operator conflated with a Main Verb (**OM**) that expects a preceding Subject (**S**) and fails to find one. This alerts the parser to check whether the words **moved from their villages** can be interpreted as a Clause (**C1**) that fills the qualifier of the nominal group whose head is **people**. With **linguistically motivated backtracking**, the parser would backtrack to the join between **people** and **moved** and attempt to follow the unlikely but correct path in which **moved from their villages** is a qualifier to **people**.

⁹ Testing revealed that, because some classes of unit can fill up to twenty or thirty different elements, the process of growing trees is computationally expensive and has the effect of slowing the parser down, both when it grows the trees and later when it uses the trees in subsequent join attempts.

18.2.7 Participant roles

The concept of **Participant Roles** (PRs) was introduced in Section 4.2.1.3 of Chapter Four. PRs are functions that are inherently associated with the process in a Clause (C1) - and typically with the Main Verb (M), and their value in parsing is that they can be used to predict the number of Complements (C). For example, the Main Verb **eat** has two Participant Roles: an Agent (Ag) which represents the person, animal or thing doing the eating, and the Affected (Af) which represents the person, animal or thing being eaten. The Agent (Ag) is typically, but not inevitably, conflated with the Subject (S) and the Affected (Af) is typically, but not inevitably, conflated with the Complement (C). So, when the parser encounters the Main Verb **eat**, it knows that it should expect a single Complement (C), and the parser can use this information to influence its decisions. It can, for example, favour a Complement (C) attachment over an Adjunct (A) attachment for **in the house** in **John is in the house**. Identifying PRs correctly provides certain problems, but we will tackle this major challenge in Phase Two. We will be greatly helped in this by being able to incorporate information from Amy Neale's Process Type Database (PTDB) (2002a, 2002b) into the **probabilities tables**. This crucial table contains a list of over 5,000 verbs together with their expected participant roles (see Table 18.1 which shows an excerpt).

Main Verb Form	Meaning	Cardiff Grammar Feature	Participant role configuration
earn	of wages (I'm not earning any money)	possessive, agent carrier	Ag-Ca + Pos
earn	merit (he has earned his place in history)	possessive, agent carrier	Ag-Ca + Pos
ease	alleviate (community groups were making efforts to ease tension)	two role, plus affected	Ag + Af
ease	loosen (ease the door open)	attributive, plus 3 p Ag	Ag + Af-Ca + At
ease	decrease (the snow had eased)	one role, affected only	Af
eat	food (she had never eaten Chinese food before)(was eaten by a lion)	two role, plus affected	Ag + Af
eat away	idea of corrosion (acid eats away metal)	two role, plus affected	Ag + Af

Table 18.1: An excerpt from Neale's Process Task Database

18.2.8 Making use of morphological information

The current parser makes less use of morphological information than many other current parsers, relying instead on the use of its knowledge of co-occurrences of elements. Some use of morphological information, however, would increase the

parser's efficiency. For example, it would be useful for the parser to know that a verb form ending in **-s** makes the **backward prediction** that it will not be preceded by an Operator (**O**) or an Auxiliary (**X**).

18.2.9 Units that handle names, dates, times etc.

Fawcett (2000a:213) describes the need for several new classes of unit that have not been included in the FPD Corpus (and hence the probabilities tables). The most important of these is the **human proper name cluster (hpnc1r)**. Currently, the version of the Cardiff Grammar used in this project treats proper names as multi-word items and expound a proper name head (**h_n**). Within the **hpnc1r** unit, the main elements are **title**, **forenames**, and **family name**. Further new classes of unit are available that can handle the **address**, the **date** and the **clock time clusters**. During Phase Two of the project, ICQF+ will be used (as described in Chapter Nine) to find all proper name heads (**h_n**) and modify them to include the structure of the new units, and similar operations will be performed for the other new units.

18.2.10 Other improvements

Further improvements that could be implemented after Phase Two (i.e. in Phase Three) are discussed in this section.

18.2.10.1 Web-enabled parsing

A trend in the design of computer systems that has increased with the popularity of the internet has been web-enabled technologies. Here, the application is hosted within a web browser (such as Internet Explorer) and accessed via an intranet or the internet itself.

As we have seen, the design of the parser is such that it strongly relies on the eXtensible Markup Language (XML). This is a major step towards making the parser web enabled because XML is 'the language of the internet'. However, the core applications are **executables** and **dynamic linked libraries** and these have to be run on a local computer. The next natural step would be to change the programs so that they can run within a web browser.

The advantages of doing this would be that users would be able to run the parser through the internet, and allow them to submit texts for parsing by visiting a web site.

18.2.10.2 The recognition of unknown items

Although the probability that the parser will encounter items that it does not recognise is significantly reduced through the implementation of vast Version Two probabilities

tables (See Section 14.2 of Chapter Fourteen and Appendix I), there will still be cases when items are found that are not included in the item tables. This could be because the item does not occur in the BNC (or the other sources), or because it has never been used before, and in either case it would be missing from the Version Two tables.¹⁰

However, there are likely to be items such as names, technical terms and manufacturing processes that are not included in the tables, because the texts of the source corpora did not incorporate texts of the type that would include all of them. Furthermore, since language is dynamic (see Section 1.1 of Chapter One), new words and uses are being created all the time, e.g. through the process of 'verbification' that is so widespread in the USA and in computer science (e.g. the use of **web-enable**) as a verb, and these will not be included in the item tables.

It is sensible, therefore, to create routines that will be able to handle these unknown items. This can be achieved by creating candidate structure trees that are marked as unknown and have the elements and probabilities that are defined in the **forward predictions** from the active open units in the built structures. These can be joined in the normal manner to the built structures and when the parser moves to the item after the unknown item, the **backward predictions** can be used to detect the most probable element that the unknown item expounds.

This method, however, will not cater for unknown items that open a new unit (i.e. one that needs to be grown because there is no incomplete unit that could include the item within it), this problem requires its own solution. The most obvious strategy is to implement a limited look-ahead facility at such points.

18.2.10.3 The use of semantic features

Weerasinghe (1994:57) followed the example of many of his predecessors in supplementing the syntactic representation in his parser with the use of semantic features. But this can never be the complete answer, and in this project we are following the principle of doing as much as possible at the level of language (syntax) before passing the outputs to the semantic interpreter.

However, as Weerasinghe and others have demonstrated some success with this approach, we will, possibly in Phase Three, investigate their use in this project. To do this would require a trivial modification to the item tables. The semantic features could be derived from the GENESYS system networks relatively easily, but to use this

¹⁰ There are nearly a million items in the Version Two items tables, and the coverage should therefore include most items that occur in natural text.

technique widely would breach the principle that I have just identified. It would be possible, however, to apply these labels selectively at known points where syntactic ambiguity can be reduced by introducing semantic information associated with specific items.

18.2.10.4 Modelling the dynamic properties of language

One of the original aims of this project was allow the parser to learn about syntax and the probabilities of given syntactic relationships as it parses new sentences. This is achieved by adding the newly parsed sentences to the corpus database, and automatically updating the corpus index and the probabilities tables automatically.

The Commit button on the Parser Workbench allows the user to save selected parse trees in the **corpus tables** (see Section 16.5 of Chapter Sixteen). When this is done, the **corpus index tables** and the **probabilities tables** are also automatically updated.¹¹ It will, of course, take a great number of new sentences to make a significant impact on the probabilities tables and thus affect the parser. The results of using this feature will therefore not be observable - and so testable - for some time.

This concludes the discussion of the improvements that will be made to the parser in Phase Two and Three. As you can see, this is a considerable amount of work to be done, mostly in refining the linguistic analysis even further - so making the syntactic representation even better equipped to do its work.

18.3 Conclusions

The Corpus-Consulting Probabilistic Parser, together with its associated corpus database, has proved to be a very practical parser for parsing unrestricted texts using Systemic Functional Grammar. It has provided encouraging results, and it has demonstrated that this new type of parsing algorithm (with its deterministic properties, its knowledge of probabilities and its strong corpus base) is well on the way to becoming a parser that fits the requirements of parsing in the twenty-first century. The aims of the work, as set out in Section 1.1 of Chapter One, have been met.

The parser incorporates a number of new concepts, each of which has performed well and has proved its value in the parsing process. The parsing algorithm itself introduces a combination of new concepts and approaches that make it unlike any other parser built so far:

¹¹ Ideally, the parser ideally would be able to decide which analyses are added to the corpus without the need for human intervention.

- (a) The techniques of (i) building an **initial strip** that becomes the **built structure**, and then (ii) creating **candidate structures** to be joined to it, coupled with (iii) a new approach to handling co-ordination proved very practicable, and it is interesting that these are derived directly from the nature of language itself, as represented in the Cardiff Grammar version of SFG and in the corpora used to create the probabilities tables.¹²
- (b) The overall algorithm was well served by modelling it as a **workflow**, and this proved to be an apt method to mould the way the parser operates as it draws upon its **working data**.
- (c) The **database-oriented** approach was ideal for storing the data needed for the **probabilities tables**. Although the amount of data was enormous, the parser's performance was excellent, and fully acceptable for use in a research environment.
- (d) The inclusion of **linguistic knowledge** at various points was vital to the efficient operation of the parser. Examples are (i) its use in the item-up-to-element-up-to-unit-up-to-element tables (**I2E2U2E**) and (ii) the forward and backward predictions in the joining procedures.
- (e) The storage of the parser's **working data** in the tables of the database was an important feature, and it proved to be an excellent research tool in the development of the parser.
- (f) The ability of the **Parser WorkBench** to operate in a step-by-step mode when determining the optimum values and combinations of the parser's configurable parameters was a key element in the success of this project, because it was this that allowed us to stop the parser and see why it has followed certain paths.
- (g) Finally, the value of the use of mark up languages for annotating corpus data was demonstrated at many points. The storage of marked up corpus data in native XML tables in a relational corpus database is particularly useful when coupled with **corpus index tables** both in the parsing process and in the **corpus query tool**.
- (h) Outside of the parser itself, I should mention the value to the project of the corpus query tool, ICQF+. It was an important aid for: (i) the creation of the

¹² Co-ordinated units, for example, could not be parsed in the way used here in terms of a typical Phrase Structure Grammar, since they require the concept that a unit **fills** an element of a higher unit.

new version of the corpus (the FPD Corpus), and (ii) the development of the Version Two probabilities tables that will be used by the parser in Phase Two.

18.4 The final word

The work of Weerasinghe (1994) and of Souter (1996) demonstrated that a chart parsing approach is a possible solution to the problem of parsing natural language in terms of Systemic Functional Grammar, but each ran into major problems (as described in Chapter Eleven). The author's own attempt at using chart parsing techniques had similar problems, as reported in Chapter Twelve. The conclusions of Chapter Twelve were that chart parsers are inherently unsuited to parsing unrestricted texts in terms of the rich functional syntax of a Systemic Functional Grammar, and it was this that led to the adoption of the new set of ideas that have been explored here. The Corpus-Consulting Probabilistic Parser therefore offers an alternative approach which addresses these problems and solves them in a manner that, at the end of Phase One of the project, can be described as successful. We are confident, moreover, that there will be further significant improvements in Phases Two and Three.

We have met the primary aim set in Chapter One, which was to demonstrate that it is possible to parse texts successfully in terms of systemic functional grammar, by using: (a) probabilistic data that are automatically drawn from a dynamic corpus database in which the probabilities may be modified as new sentences are satisfactorily parsed, and (b) introducing knowledge of functional syntax into the parsing algorithm. I have shown that a parser can be built that is database-oriented in that it (a) retrieves its knowledge of syntactic relationships to be used in a parse, and (b) stores its working data in the database. As we have seen, this provides an ideal environment for developing a parser.

Bibliography

- Abney (1991):** Abney, S. *Parsing by Chunks*. In Berwick et al, 1991.
- Allen (1987):** Allen, J. *Natural Language Understanding*. Benjamin Cummings.
- ASD (1985):** ASD (formerly, AECMA) Specification *ASD-STE100 Simplified Technical English*. Aerospace and Defence Industries of Europe (ASD).
- Atwell (1988):** Atwell, E.S. *Transforming a Parsed Corpus into a Corpus Parser*. In Kyto et. al, 1988.
- Atwell et al (1988a):** Atwell, E.S., Souter, C. *Experiments with a Very Large Corpus-based Grammar*. In Proceedings of the 15th ALLC Conference, June 5-13 1988, Jerusalem.
- Atwell et al (1988b):** Atwell, E.S., Souter, C., O'Donoghue, T. *Prototype Parser 1 - COMMUNAL Research Report No. 17*. University of Wales College of Cardiff.
- Benson and Greaves (1985):** Benson, J. and Greaves, W. (eds) *Systemic Functional Approaches to Discourse: Selected Papers from the 12th International Systemic Workshop*. Norwood, N.J.: Ablex.
- Berwick et al (1991):** Bewick, R., Abney, S., Tenny, C. (eds) *Principle-based Parsing* Kluwer Academic Publishers.
- Biber et al (1999):** Biber, D., Johansson, S., Leech, G., Conrad, S., and Finegan, E. *Longman Grammar of Spoken and Written English*. Harlow: Pearson
- Black et al (1993):** Black, E., Jelinek F, Lafferty, J.D., Magerman, D.M, Mercer, R.L., Roukos, S. *Towards History-Based Grammars: Using Richer Models for Probabilistic Parsing*. In Meeting of the Association for Computational Linguistics 1993.
- Bobrow (1978):** Bobrow, R.J. *The RUS System*. Quarterly Technical Report, Bolt, Baranek and Newman, Cambridge, MA.
- Brady and Berwick (1983):** Brady, M., and Berwick, R.C. (eds) *Computational Models of Discourse*. Cambridge, Mass: MIT Press.
- Brill (1992):** Brill, E. *A Simple Rule Based Part-of-speech Tagger*. In Proceedings of the (ANLP)-92, 3rd Conference on Applied Natural Language Processing.
- Brill (1995):** Brill, E. *Transformation-Based Error-Driven Learning and Natural Language Processing: A Case Study in Part of Speech Tagging*. In Computational Linguistics, December, 1995 - John Hopkins University.
- Bryan (1988):** Bryan, M. *SGML - an Author's Guide*. Addison Wesley Longman.
- Butler (2003a):** Butler, C.S. *Structure and Function: An introduction to three major structural-functional theories. Part 1: Approaches to the simplex clause*. Amsterdam: John Benjamins.
- Butler (2003b):** Butler, C.S. *Structure and Function: An introduction to three major structural-functional theories. Part 2: From clause to discourse and beyond*. Amsterdam: John Benjamins.
- Butler et al (2007):** Butler, C.S., Hidalgo Downing, R., and Lavid, J. *Functional Perspectives on Grammar and Discourse: Papers In Honour of Angela Downing*. Amsterdam: John Benjamins, pp. 165-204.
- Calhoun et al (2005):** Calhoun, S., Nissim, M., Steedman, M., Brenier, J. *A Framework for Annotating Information Structure in Discourse*. In Proceedings of the Workshop on Frontiers in Corpus Annotation: Pie in the Sky pp 45-52.
- Carletta, McKelvie and Isard (2002):** Carletta, J., McKelvie, D., Isard, *A Supporting Linguistic Annotation using XML and Stylesheets*. In *Corpus Linguistics: readings in a widening discipline*, eds G.Sampson and D.McCarthy (London and New York: Continuum Interpretations).

- Carroll, Briscoe and Sanfilippo (1999):** Carroll, J., Briscoe, T., Sanfilippo, A., *Parser Evaluation: Current Practice In Evaluation of Natural Language Processing Systems: Final Report*, EC DG-XIII LRE EAGLES Document EAG-II-EWG-PR.1. 140-150.
- Chawathe et al (1996):** Chawathe, S., Rajaraman, A., Garcia-Molina, H., Widom, J. *Change Detection in Hierarchically Structured Information*. Stanford University, California.
- Charniak (1983):** Charniak, E. *A Parser with Something for Everyone*. In King (1983).
- Charniak and Johnson (2005):** Charniak, E. and Johnson, M. *Coarse-to-find n-best Parsing and MaxEnt Discriminative Ranking*. In 43rd Annual Meeting of the ACL, pp 170-180, Brown University.
- Chomsky (1957):** Chomsky, N. *Syntactic Structures*. The Hague, Mouton.
- Chomsky (1965):** Chomsky, N. *Aspects of the Theory of Syntax*. Cambridge, Mass: MIT Press.
- Church (1988):** Church, K. *A Stochastic Parts Program and Noun Phrase Parser for Unrestricted Text*. In Proceedings of the Second Conference on Applied Natural Language Processing, 26th Annual Meeting of the Association for Computational Linguistics pp 136-143).
- Clocksin and Mellish (1994):** Clocksin, W., Mellish, C. *Programming in Prolog (4th Edition)* Springer-Verlag.
- Collins (1996):** Collins, M. J. *A New Statistical Parser Based on Bi-gram Lexical Dependancies*. In Proceedings of the Thirty-Fourth Annual Meeting of the Association for Computational Linguistics, Morgan Kaufmann Publishers - University of Pennsylvania.
- Collins (1999):** Collins, M. *Head Driven Statistical Models for Natural Language Parsing* Ph.D. Thesis, University of Pennsylvania.
- Colmerauer (1978):** Colmerauer, A. *Metamorphosis grammars*. In Bloc, L. (Ed.) *Natural Language Communication with Computers*, Springer-Verlag.
- Costa et al (2003):** Costa, F., Frasoni, P., Lombardo, V., Soda, G. *Towards Incremental Parsing of Natural Language using Recursive Neural Networks*. Applied Intelligence (available at <http://cogprints.org/2089>).
- Dale et al (1992):** Dale, R., Hovy, E.H., Roesner, D., and Stock, O., (eds.) *Aspects of Automated Natural Language Generation*. Berlin: Springer.
- Davey (1978):** Davey, A. *Discourse Production: a Computer Model of some Aspects of a Speaker* Edinburgh University Press
- Day (1993a):** Day, M.D. *The Interactive Corpus Query Facility and Other Tools for Exploiting Parsed Natural Language Corpora*. MSc. Thesis, Cardiff University
- Day (1993b):** Day, M.D. *Content Specific Mark-up*. Rolls-Royce internal report (available from author).
- Day (1993c):** Day, M.D. *Customer Logistic Support Prototype Integrated LSA/CSDB/ATA 100 Publishing System*. Rolls-Royce internal report (available from author).
- Day (1995):** Day, M.D. *SGML Parsing is not Enough!* Rolls-Royce internal report (available from author).
- Day (2006):** Day, M.D. *Business Rules - a Tutorial*. A presentation given given at the ASD/AIA/ATA/ADL S1000D Users' Forum, Clearwater, Florida, May 2006 and available from www.s1000d.org.
- Day and Ichizli-Bartels (2007):** Day, M.D., Ichizli-Bartels, V. *Business Rules, Business Rules Builders, and Business Rules Checkers*. White paper S1000D Technical Publications Specification Maintenance Group.
- Déjean (2000):** Déjean, H. *Learning Syntactic Structures with XML*. Presented at the Seminar fur Sprachwissenschaft, University of Tübingen.
- Dowty et al (1985):** Dowty, D.R, Karttunen, L. and Zwicky, A. (eds). *Natural language parsing*. New York, Cambridge U. Press

- Earley (1970):** Earley, J. *An Efficient Context Free Parsing Algorithm*. Communication of the ACM Vol 13).
- Elhadad (1993):** Elhadad, M. *Using Argumentation to Control Lexical Choice: A Functional Unification Implementation*. Ph.D thesis, Columbia University.
- Fawcett (1980):** Fawcett, R. P. *Cognitive Linguistics and Social Interaction: Towards an Integrated Model of a Systemic Functional Grammar and the Other Components of an Interacting Mind*. Heidelberg: Julius Groos and Exeter University.
- Fawcett (1988):** Fawcett, R.P. *Language generation as choice in social interaction*. In Zock and Sabah (1988).
- Fawcett (1990):** *The computer generation of speech with semantically and discoursally motivated intonation*. In *Proceedings of the 5th International Workshop on Natural Language Generation*, Pittsburgh: University of Pittsburgh. 164-73.
- Fawcett (1994):** Fawcett, R.P. *A generationist approach to grammar reversibility in natural language processing*. In Strzalkowski (ed.) 1994, 365-413.
- Fawcett (2000a):** Fawcett, R.P. *A Theory of Syntax for Systemic Functional Linguistics*. Current Issues in Linguistic Theory 206. Amsterdam: John Benjamins
- Fawcett(2000b):** Fawcett, R.P. *In place of Halliday's "verbal group", Part 1: Evidence from the problems of Halliday's representations and the relative simplicity of the proposed alternative*. Word 51.2. 157-203
- Fawcett (2000c):** Fawcett, R.P. *In place of Halliday's "verbal group", Part 2: Evidence from generation, semantics and interruptability*. Word 51.3. 327-75
- Fawcett (2007a):** Fawcett, R.P. *Auxiliary Extensions: six new elements for describing English*. In Hasan et al (2007).
- Fawcett (2007b):** Fawcett, R.P. *Modelling "selection" between referents in the English nominal group: an essay in scientific inquiry in linguistics*. In Butler et al (2007).
- Fawcett (forthcoming, 2007):** Fawcett, R.P. *Functional Syntax Handbook: Analyzing English at the Level of Form*. London: Equinox
- Fawcett and Davies (1992):** Fawcett,R.P., Davies,B.D. *Monologue as a turn in dialogue: towards an integration of exchange structure and rhetorical structure theory*. In Dale et al. (1992), 151-66.
- Fawcett and Perkins (1980):** Fawcett, R.P., and Perkins, M.R *Child Language Transcripts 6-12*, Pontypridd, Wales: Polytechnic of Wales (now University of Glamorgan).
- Fawcett et al (1984):** Fawcett, R.P., Halliday, M.A.K., Lamb, S.M., and Makkai, A., (eds.) *The Semiotics of Culture and Language, Vol 1 Language as Social Semiotic* London: Pinter
- Fawcett, Tucker and Lin (1993):** Fawcett, R.P., Tucker, G.H., and Lin, Y.Q., *How a systemic functional grammar works: the role of realization in realization*. In Horacek and Zock (1993), 114-86.
- Fawcett, Tucker and Young (1988):** Fawcett, R.P., Tucker,G.H., Young, D.J. *Issues Concerning Levels and Channels in a Generator with both Graphological and Phonological Outputs* Report 8 of COMMUNAL, University of Wales College of Cardiff.
- Firth (1957):** Firth, J.R. *Papers in linguistics 1934-1951*. Oxford University Press.
- Garside et al (1987):** Garside, R., Leech, G. *The Computational Analysis of English - A Corpus Based Approach* Longman.
- Gazdar and Mellish(1989):** Gazdar, G., Mellish, C. *Natural Language Parsing in Prolog*. Wokingham, England: Addison-Wesley Addison-Wesley
- Gazdar et al (1985):** Gazdar, G., Klein, E., Pullum, G., and Sag, I. *Generalized Phrase Structure Grammar*. Oxford: Blackwell. Oxford. Blackwell.
- Godfrey et al (1992):** Godfrey, J., Holliman, E., McDaniel, J. *SWITCHBOARD: The Telephone Speech Corpus for Research and Development*. In Proceedings ICASSP-92 pp 517-520).

- Goldfarb (1991):** Goldfarb, C. *The SGML Handbook*. Oxford University Press.
- Grishman, Macleod and Sterling (1992):** Grishman, R., Macleod, C., Sterling, J. *Evaluating parsing strategies using standardized parse files*. In *Proceedings of the 3rd ACL Conference on Applied Natural Language Processing*, 156-161. Trento, Italy.
- Groz et al (1970):** Groz, B.J., Sparck Jones, K., Webber, B.L. (Eds). *Readings in Natural Language Processing*, Morgan Kaufmann.
- Grune and Jacobs (1990):** Grune, D., Jacobs, C.J.H. *Parsing Techniques a Practical Guide*. Ellis Horwood, Chichester, England.
- Guyon and Pereira (1995):** Guyon, I., Pereira, F. *Design of a Linguistic Postprocessor Using Variable Memory Length Markov Models*. In the proceedings of the International Conference on Document Analysis and Recognition.
- Haigh et al (1988):** Haigh, R. Sampson, G., Atwell, E. S.: *Project APRIL - a progress report* in *Proceedings of ACL, the 26th Conference of the Association for Computational Linguistics* pp104-112, New Jersey, ACL 1988.
- Hall (2005):** Hall, K.B. *Best-first Word Lattice Parsing - Techniques for Integrated Syntactic Modelling*. PhD. Thesis, Brown University.
- Halliday (1956/76):** Halliday M.A.K. *Grammatical categories in Modern Chinese*. In *Transactions of the Philological Society* 1956, 177-224. Reprinted in part in Halliday, M.A.K., 1976, *System and Function in Language: Selected Papers by M.A.K. Halliday* (ed. G.R. Kress), London: Oxford University Press, 36-51.
- Halliday (1961):** Halliday, M.A.K. *Categories of the theory of Grammar*. *Word* (17) pp 241-292.
- Halliday (1975):** Halliday, M.A.K. *Learning How to Mean*. London: Arnold
- Halliday (1976):** Halliday, M.A.K. *System and Function in Language: Selected Papers by M.A.K. Halliday* (ed. G.R. Kress). London: Oxford University Press
- Halliday (1984):** Halliday, M.A.K. *Language as code and language as behaviour: a systemic-functional interpretation of the nature and ontogenesis of dialogue*. In Fawcett et al. 1984. 3-35.
- Halliday (1981):** Halliday, M.A.K., 1981. 'Introduction'. In Halliday, M.A.K., and Martin, J.R., 1981 (eds.), *Readings in Systemic Linguistics*, London: Batsford, 13-16.
- Halliday (1985):** Halliday, M.A.K. *An Introduction to Functional Grammar*. London: Arnold.
- Halliday (1994):** Halliday, M.A.K. *An Introduction to Functional Grammar (Second Edition)*. London: Arnold.
- Harold (1999):** Harold, R. E. *XML Bible*. IDG Books World-wide.
- Hasan et al (2005):** Hasan, R., Matthiessen, C. & Webster, J. *Continuing Discourse on Language: a Functional Perspective: Vol 1*. London: Equinox.
- Hasan et al (2006):** Hasan, R., Matthiessen, C. & Webster, J. *Continuing Discourse on Language: a Functional Perspective: Vol 2*. London: Equinox.
- Henderson (2000):** Henderson, J. *Estimating a probabilistic grammar using a Neural Network*. In *Workshop ROMAD 2000 Lausanne*, Exeter University.
- Henderson (2003a):** Henderson, J. *Generative Versus Discriminative Models for Statistical Left-Corner Parsing*. In *IWPT2003 - 8th International Workshop of Parsing Technologies*, 23-25 April 2003 - Nancy, France.
- Henderson (2003b):** Henderson, J. *Inducing history representations for broad coverage statistical parsing*, In *Proceedings of HLT-NAACL 2003*.
- Henrici (1965):** Henrici, A. *Some notes on the systemic generation of a paradigm of the English Clause*. Working paper for the OSTI programme in the linguistic properties of scientific English. Reprinted in Halliday (1981).
- Horacek and Zock (1993):** Horacek, H., and Zock, M., (eds.). *New Concepts in Natural Language Generation*. London: Pinter.

- Hudson (1976):** Hudson, R.A. *Arguments for a Non-transformational Grammar*. Chicago: Chicago University Press.
- Isard et al (2003):** Isard, A., McKelvie, D., Mengel, A, Moller, M. *The MATE Workbench - A tool for annotating XML corpora*. In *Speech Communication* Vol. 33 pp 97-112.
- ISO (1986):** The International Standards Organization. *ISO 8879: Information processing - Text and office systems - Standard Generalized Markup Language (SGML)*
- Jagadish et al (2002):** Jagadish, H.V., Al-Khalifa, S., Chapman, A., Lakshmanan, Laks V.S., Nierman, A., Paparizos, S, Patel, J.M., Srivastava, D., Wiwatwattana, N. Wu, Y., Yu,C. *Timber: A native XML database*. Technical report, University of Michigan, April 2002.
- Jarvinen (1994):** Jarvinen, T. *Annotating 200 Million Words: the Bank of English Project*. In *Proceedings of COLING-94*.
- Johnson (1983):** Johnson, R. *Parsing with Transition Networks*. In King (1983).
- Joshi (1985):** Joshi, A.K. *Tree adjoining grammars: How much context sensitivity is required to provide structural descriptions?* (In Dowty et al 1985).
- Joshi and Schabes (1997):** Joshi, A., Schabes, Y. *Tree-Adjoining Grammars*. In Rozenberg et al (1997).
- Joshi et al (1975):** Joshi, A.K., Levy, L. and Takahashi, M. *Tree Adjunct Grammars*. In *Journal of Computer and System Sciences*.
- Kasper (1988):** Kasper, R. *An experimental parser for systemic grammars*. In *Proceedings of the 12th International Conference on Computational Linguistics 309-12*. Association for Computational Linguistics: Budapest.
- Kay (1979):** Kay, M. *Functional Grammar*. In *proceedings of the 5th meeting of the Berkeley Linguistics Society*.
- Kay (1989):** Kay, M. *Head Driven Parsing*. In the *proceedings of the first international workshop on parsing technologies, Pittsburgh, USA*. Carnegie-Mellon University.
- King (1983):** King, M ed. *Parsing Natural Language*. Academic Press.
- Klein and Manning (2003):** Klein, D., Manning, C. *Accurate Unlexicalised Parsing*. Stanford University, Stanford CA.
- König, Lezius and Voormann (2003):** König, E., Lezius, W., Voormann, H. *TIGERSearch 2.1 User's Manual*. IMS University of Stuttgart.
- Kuhn (1962/70):** Kuhn, T. *The Structure of Scientific Revolutions*. Chicago: Chicago University Press.
- Kyto et al (1988):** Kyto, M., Ihalainen, O., Rissanen, M. (Eds). *Corpus Linguistics Hard and Soft*. Amsterdam, Rodopi.
- Langkilde and Knight (1998):** Langkilde, I. Knight, K. *Generation that Exploits Corpus-Based Statistical Knowledge*. In *COLING, 1998*.
- Leech et al (1994):** Leech, G., Garside, R., Bryant, M. *CLAWS4: The Tagging of the British National Corpus*. In the *Proceedings of the 15th International Conference on Computational Linguistics (COLING '94) Kyoto, Japan*.
- Lezius and König (2000):** Lezius, W., König, E. *Towards a search engine for syntactically annotated corpora* in: Ernst G. Schukat-Talamazzini Werner Zühlke (editor): *KONVENS-2000 Sprachkommunikation*, pp. 113-116, VDE-Verlag, Ilmenau, Germany
- Magerman and Marcus (1991a):** Magerman, D.M., Marcus, M.P. *Parsing a natural language using mutual information statistics*. Tech Rep, CIS Dept. University of Pennsylvania.
- Magerman and Marcus (1991b):** Magerman, D.M., Marcus, M.P. *Pearl: A probabilistic chart parser*. In *Proceedings of the 2nd International Workshop on Parsing Technologies, Cancun, Mexico*.
- Mann and Matthiessen (1985):** Mann, W.C. and Matthiessen, C.M.I.M. *Demonstration of the Nigel text generation computer program*. In *Benson and Greaves (1985)*.
- Mann and Thompson (1988):** Mann, W.C., Thompson, S.A. *Rhetorical Structure Theory: toward a structural theory of text organisation*. (in *Text 8(3)*) quoted by O'Donnell and Bateman (2005).

- Marcus (1980):** Marcus, M.P. *A Theory of Syntactic Recognition for Natural Language*. MIT Press.
- Marcus et al(1993):** Marcus, M.P., Santorini, B., Marcinkiewicz, M.A. *Building a Large Annotated Corpus of English: the Penn Treebank*. In *Computational Linguistics*, 19.
- Marslen-Wilson (1973):** Marslen-Wilson, W. *Linguistic structure and speech shadowing at very short latencies* (Nature Vol 244).
- Matthiessen and Bateman (1991):** Matthiessen, C., Bateman, J. *Text Generation and Systemic Functional Linguistics: experiences from English and Japanese*. London and New York: Frances Pinter and St. Martin's Press.
- McDonald (1983):** McDonald, D. *Natural language generation as a computational problem*. In Brady and Berwick 1983, 209-65.
- McKelvie and Mikheev (1997):** McKelvie, D., Mikheev, A. *Indexing SGML files using LT NSL*. Technical Report, Language Technology Group, University of Edinburgh.
- Mel'cuk (1998):** Mel'cuk, I. *The Meaning-Text Approach to the Study of Natural Language and Linguistic Functional Models*. In Embleton, S. (ed.), LACUS Forum 24, Chapel Hill: LACUS, 3-20.
- Mengel and Lezius (2000):** Megel, A., Lezius, W. *An XML-based encoding format for syntactically annotated corpora*. In: Proceedings of the Second International Conference on Language Resources and Engineering (LREC), volume 1 pp 121-126 Athens, Greece.
- Meteer (1989):** Meteer, M. *The Spokesman Natural Language Generation System*. Report 7090, BBN Systems and Technologies, Cambridge, Mass, 1989.
- Mouat (2002):** Mouat, A. *XML Diff and Patch Utilities*. Master's thesis, Heriot Watt University, School of Mathematical and Computer Sciences
- Neale (2002a):** Neale, A. *More Delicate TRANSITIVITY: Extending the PROCESS TYPE system networks for English to include full semantic classifications*. PhD Thesis. Cardiff: School of English, Communication and Philosophy, Cardiff University. Available on request via neale@qmul.ac.uk.
- Neale (2002b):** Neale, A. *The Process Type Data Base*. Available on request via neale@qmul.ac.uk.
- Neale (2006):** Neale, A. *Matching corpus data and system networks*. In Thompson and Hunston (2006).
- O'Donnell (1993):** O'Donnell, M. *Reducing complexity in a systemic parser*. In Proceedings of the Third International Workshop on Parsing Technologies 203-17. Tilburg, the Netherlands.
- O'Donnell (1994):** O'Donnell, M. *Sentence Analysis and Generation: a systemic perspective*. PhD thesis. Department of Linguistics, University of Sydney, Australia.
- O'Donnell (1996):** O'Donnell, M. *Input Specification to the WAG Sentence Generation System*. In Proceedings of the 8th International Workshop on Natural Language Generation.
- O'Donnell (2005):** O'Donnell, M. *The UAM Systemic Parser*. In Proceedings of the 1st Computational Systemic Functional Grammar Conference, University of Sydney, Sydney, Australia. 16 July 2005.
- O'Donnell and Bateman (2005):** O'Donnell, M., Bateman, J. *SFL in computational contexts: a contemporary history*. In Hasan, Matthiessen and Webster (2005).
- O'Donoghue (1991a):** O'Donoghue, T.F.. 'The vertical strip parser: a lazy approach to parsing'. Report 91.15. Leeds: School of Computer Studies, University of Leeds.
- O'Donoghue (1991b):** O'Donoghue, T.F. *A Semantic Interpreter for Systemic Grammars*. COMMUNAL Report AL Report 91:15
- O'Donoghue (1991c):** O'Donoghue, T.F. *A semantic interpreter for systemic grammars*. In *Proceedings of the ACL Workshop on Reversible Grammars*. Berkely, California. 1991. 129-38.
- Ore (1960):** Ore, O. *A Note on Hamilton Circuits* (in American Math Monthly 67).

- Pereira and Shieber (1987):** Pereira, F.C.N., Sheiber, S. *Prolog and Natural Language Analysis* (CSLI Lecture Notes - quoted in Weerasinghe, (1994)).
- Pocock and Atwell (1993):** Pocock, R., Atwell, E. *Probabilistic Grammatical Models for Treebank-Trained Lattice Disambiguation*. Research Report 93.30, School of Computer Studies, Leeds University.
- Priestley (2001):** Priestley, M. *Specializing Topic Types in DITA* www-128.ibm.com/developerworks/xml/library/x-dita2.
- Quirk et al (1985):** Quirk, R., Greenbaum S., Leech G., and Svartvik J. *A Comprehensive Grammar of the English Language*. London, Longman.
- Rambow and Korelsky (1992):** Rambow, O., Korelsky, T. *Applied Text Generation*. In proceedings of the third conference on applied natural language generation.
- Randall (2000):** Randall, B. *CorpusSearch User's Manual*. Technical Report, University of Pennsylvania (www.ling.upenn.edu/mideng/ppcme2dir)
- Ratnaparkhi (1999):** Ratnaparkhi, A. *Learning to Parse Natural Language using Maximum Entropy Models*. University of Philadelphia.
- Reiter (1994):** Reiter, E. *Has a Consensus NL Generation Architecture Appeared, and is it Psycholinguistically Plausible?* In Proceedings of the 7th. International Workshop on Natural Language generation. CoGenTex, Inc.
- Reiter, Mellish and Levine (1995):** Reiter, E., Mellish, C., Levine, J. *Automatic Generation of Technical Documentation*. Applied Artificial Intelligence (Vol. 9) University of Edinburgh.
- Rhode (2005):** Rhode, D.L.T. *TGrep2 User Manual*. Available from www.tedlab.mit.edu.
- Ritchie (1983):** Ritchie, G. *Semantics in Parsing*. In King (1983).
- Rozenberg et al (1997):** Rozenberg, G. Salomaa A. (eds.), *Handbook of Formal Languages Vol. 3*. Springer, Berlin, New York, 1997, pp 69 - 124.
- Rozenkrantz and Lewis (1970):** Rozenkrantz, D.J., Lewis, P.M. *Deterministic left corner parsing*. In Proceedings of the 11th Symposium on Switching and Automata Theory.
- Rustin (1973):** Rustin, R. (Ed.). *Natural Language Processing*. New York, Algorithmics Press.
- S1000D (2006):** Aerospace Industries Association (AIA), Aerospace and Defence Industries of Europe (ASD) *Specification for Technical Publications Utilizing a Common Source Database (Issue 2.3)* (available from www.s1000d.org).
- Sampson (1983):** Sampson, G.R. *Deterministic Parsing*. In King (1983).
- Sampson (1995):** Sampson, G.R. *English for the Computer* Oxford University Press.
- Sampson (2000):** Sampson, G.R. *A proposal for improving the measurement of parse accuracy* in International Journal of Corpus Linguistics vol. 5, pp. 53-68, 2000
- Shieber (1984):** Sheiber, S.M. *The design of a computer language for linguistic information (PATR)*. In Proceedings of the 10th International Conference on Computational Linguistics (COLING 1984), California, USA.
- Sinclair (1990):** Sinclair, J. (editor-in-chief) *Collins COBUILD English Grammar*. London: HarperCollins.
- Sinclair (1991):** Sinclair, J. *Corpus, concordance, collocation*. Oxford: Oxford University Press.
- Sommerville (1992):** Sommerville, I. *Software Engineering Fourth Edition*. Addison-Wesley.
- Souter (1996):** Souter, D.C. *A Corpus Trained Parser for Systemic Functional Syntax*. PhD Thesis, Leeds University.
- Sprecht et al (1995):** Sprecht, G., Freitag, B. AMOS: *A Natural Language Parser Implemented as a Deductive Database in LOLA*. In Workshop on Programming with Logic Databases.
- Stegmann et al (2000):** Stegmann, R., Telljohann, H., Hinrichs, E.W. *Stylebook for the German Treebank in VERBMOBIL*. Technical Report 239, Verbmobil.

- Steinar and Kallmeyer (2002):** Steinar, I., Kallmeyer, L. *VIQTORIA - A Visual Query Tool for Syntactically Annotated Corpora*. In Proceedings of the Third International Conference on Language Resources and Evaluation (LREC 2002) pp 1704-1711.
- Stolcke (1993):** Stolcke, A. *An efficient probabilistic context-free parsing algorithm that computes prefix probabilities*. Technical Report TR-93-065. International Computer Science Institute, University of California at Berkeley, CA.
- Stolcke and Omohundro (1994):** Stolcke, A., Omohundro, S.M. *Best-First Model Merging for Hidden Markov Model Induction*. TR-94-003 University of California at Berkeley, CA International Computer Science Institute.
- Strzalkowski (1994):** Strzalkowski, T. (ed.). *Reversible Grammar in Natural Language Generation*. Dordrecht: Kluwer.
- Swartout (1978):** Swartout, W.R. *A Comparison of PARSIFAL with ATNs* (quoted by Sampson, 1983).
- Taylor (2003):** Taylor, A. *CorpusSearch Reference Manual* (www-users.york.ac.uk).
- Teich (1999):** Teich, E. *Systemic Function Grammar in Natural Language Generation: Linguistic Description and Computational Representation* Cassell, London.
- Thompson and Hunston (2006):** Thompson, G., Hunston, S. (eds) *System and Corpus: Exploring Connections* London: Equinox
- Tucker (1988):** Tucker, G. *The Lexicogrammar of Adjectives: a Systemic Functional Approach to Lexis*. London: Cassell Academic.
- Tucker (1989):** Tucker, G. *Natural language generation with a systemic functional grammar*. In Laboratorio degli studi linguistici 1989/1. Camerino, Italy: Università degli Studi di Camerino pp.7-27.
- Tucker (2006a):** Tucker, G. *Systemic incorporation: on the relationship between corpus and Systemic Functional Grammar*. In Thompson and Hunston (2006).
- Tucker (2006b):** Tucker, G. *Between lexis and grammar: towards a systemic functional approach to phraseology*. In Hasan, Matthiessen, and Webster (2006).
- van Herwijnen (1994):** van Herwijnen, E. *Practical SGML*. Kluwer Academic Publishers.
- W3C (1998):** W3C RFC-2413-September 1998 - *IETF recommendation: Dublin Core - Metadata for Resource Discovery*.
- W3C (2004a):** W3C REC-xml-20040204 - *Extensible Markup Language (XML) 1.0 (Third Edition)*.
- W3C (2004b):** W3C XML Schema Part 0: *Primer Second Edition*.
- W3C (2006):** W3C REC-xsl11-20061205 - *W3C Recommendation Extensible Stylesheet Language (XSL) Version 1.1*.
- W3C (2007):** W3C REC-xquery-20070123 - *W3C Recommendation XQuery 1.0: An XML Query Language*.
- Warren and Pereira (1982):** Warren, D. H. D, Pereira, F.C.N. *An Efficient and Easily Adaptable System for Interpreting Natural Language*. American Journal of Computational Linguistics 8.
- Webster (2005):** Webster, J. *M.A.K Halliday: the early years, 1925 - 1970* (in Hasan, Matthiessen and Webster (2005)).
- Weerasinghe (1994):** Weerasinghe, A. R. *Probabilistic Parsing in Systemic Functional Grammar*. PhD Thesis, Cardiff University.
- West (1965):** West, M. *A general service list of English words*. Longmans.
- Winograd (1972):** Winograd, T. *Understanding Natural Language*. Edinburgh University Press.
- Winograd (1983):** Winograd, T. *Language as a Cognitive Process*. Reading Mass Addison Wesley.
- Wirén (1989):** Wirén, M. *Interactive Incremental Chart Parsing*. In Proceedings of the Fourth Conference of the European Chapter of the Association for Computational Linguistics, pp 241-248, Manchester, England, 1989.

- Wirth (1976):** Wirth, N. *Algorithms + Data Structures = Programs*. Prentice-Hall.
- Woods (1970):** Woods, W.A. *Transition network grammars for natural language analysis*. In Groz et al (1970) pp 71-88.
- Woods (1973):** Woods, W.A. *An experimental parsing system for transition network grammars*. In Rustin (1973).
- Woods (1980):** Woods, W.A. *Cascaded ATN grammars*. *AJCL* 6, 1, 1-12.
- Xtag (1995):** XTag Research Group *A lexicalized Tree Adjoining Grammar for English*. Technical Report IRCS Report 95-03, The Institute for Research in Cognitive Science, Univ. of Pennsylvania.
- Yang (1994):** Yang, W. *Incremental LR Parsing*. In 1994 International Computer Symposium Conference Proceedings vol. 1. National Chiao Tung University, Hsinchu, Taiwan, 577-583.
- Zipf (1935):** Zipf, G.K. *The Psychobiology of Language*. Cambridge, Mass.
- Zock and Sabah (1988):** Zock, M., and Sabah, G., (eds) *Advances in Natural Language Generation Vol 1 and 2*. London: Pinter.

Appendix A

The Cardiff Grammar

This appendix lists the units and elements that form the Cardiff Grammar as defined by Fawcett (2000a). For a definition of terms, and a description of the Cardiff Grammar please see Chapter Four. A complete alphabetical list of elements and units that occur in the Fawcett-Perkins-Day Corpus (FPD) is provided in Appendix B¹.

A.1 The Cardiff Grammar Units

A syntax unit contains elements of structure. The units in the Cardiff Grammar are shown below. A list of elements that occur in any unit is given in Section A.8.

- (a) Clause (**C1**) (see Section A.2),
- (b) nominal group (**ngp**) (see Section A.3),
- (c) prepositional group (**pgp**) (see Section A.4),
- (d) quality group (**qlgp**) (see Section A.4),
- (e) quantity group (**qtgp**) (see Section A.5),
- (f) genitive cluster (**genc1r**) (see Section A.6), and
- (g) text (**TEXT**) (see Section A.7).

A.2 The Clause (C1)

The elements of the Clause are given in Table A.1.

Element	Meaning
A*	Adjunct (many types)
B	Binder (e.g. that)
C	Complement
F	Formula
I	Infinitive element (i.e. to)
L	Let element (i.e. Let 's)
M	Main Verb
Mex	Main Verb Extension (e.g. off in switch off the light)
N	Negator (e.g. not)
O	Operator (O, O/X or O/M where / is conflation)
S	Subject
V	Vocative
X	Auxiliary Verb
Xex	Auxiliary Verb Extension
Fr	Frame
Lnk	Linker (& in Fawcett 2000a)

Table A.1: Elements of the Clause

¹ Appendix B contains a complete list including the different types of Adjunct, modifier and head that are not given here.

A.3 The nominal group (ngp)

The elements of the nominal group are shown in Table A.2.

Element	Meaning
rd	representational determiner
v	selector
pd	partitive determiner
fd	fractionative determiner
qd	quantifying determiner
sd	superlative determiner
od	ordinative determiner
qid	qualifier introducing determiner
td	typic determiner
dd	deictic determiner
m	modifier (different types eg rel_mo (relative) aff_mo (affective))
h	head - h_p (pronoun head), h_n (proper name head) or h (ordinary head h)
q	qualifier

Table A.2: Elements of the nominal group

A.4 The prepositional group (pgp)

The elements of the prepositional group are shown in Table A.3

Element	Meaning
p	preposition
pt	prepositional temperer
cv	completive

Table A.3: Elements of the prepositional group

A.5 The quality group (qlgp)

The elements of the quality group are given in Table A.4.

Element	Meaning
qld	quality group deictic determiner
qlq	quality group quantifier
et	emphasising temperer
dt	degree temperer
at	adjectival temperer
ax	apex
sc	scope
fi	finisher

Table A.4: Elements of the quality group

A.6 The quantity group (qtgp)

The elements of the quantity group are given in Table A.5.

Element	Meaning
qtd	quantity group determiner
ad	adjustor
am	amount
qtf	quantity group finisher

Table A.5: Elements of the Quantity Group

A.7 The genitive cluster (genclr)

The elements of the quantity group are given in Table A.6.

Element	Meaning
po	possessor
g	genitive element
own	owner element

Table A.6: Elements of the genitive cluster

A.8 Elements that can occur in any class of unit

The elements of that can occur in any class of unit are given in Table A.7.

Element	Meaning
Lnk	Linker ("&" in Fawcett 2000a)
Inf	Inferer

Table A.7: Elements that can occur in any Group

Appendix B

An alphabetical list of the Cardiff Grammar units and elements

This appendix provides a reference list of the units and elements that are found in the Fawcett-Perkins-Day (FPD) Corpus.

In the table that follows, the **FREQ** column represents the number of occurrences of the syntax token in the FPD Corpus. The **TYPE** column states if the syntax token represents an **element (ELEM)** or a **unit (UNIT)**.¹

TOKEN	MEANING	FREQ	TYPE
A	Adjunct expressing Experiential meaning	2487	ELEM
A_Aff	Adjunct expressing only Affective meaning (e.g. "luckily")	2	ELEM
A_Anc	Ancillary Grammar Adjunct (e.g. "flipping")	2	ELEM
A_AS	Acknowledgement-Seeking Adjunct	269	ELEM
A_CSTag	Confirmation-Seeking Tag Adjunct	401	ELEM
A_DO	Adjunct expressing Discourse Organisational meaning	11	ELEM
A_Inf	Adjunct expressing Inferential meaning	318	ELEM
A_Log	Adjunct expressing Logical relationship meaning	821	ELEM
A_Log_Repl	Replacement for an Adjunct expressing Logical relationship	9	ELEM
A_Log_Wh	Wh-Adjunct expressing Logical relationship meaning	1	ELEM
A_ML	Metalingual Adjunct		ELEM
A_Pol	Politeness Adjunct	13	ELEM
A_Repl	Replacement for Adjunct expressing Experiential meaning	11	ELEM
A_Res	Respect Adjunct ('with respect to X')	62	ELEM
A_Val	Adjunct expressing Validity meaning	60	ELEM
A_Wh	Wh-Adjunct expressing Experiential meaning sought	164	ELEM
ad	adjustor	167	ELEM
aff_mo	modifier expressing affective meaning	71	ELEM
AL_Repl	Replacement for Adjunct expressing Logical relationship		ELEM
am	amount	279	ELEM
ax	apex	2386	ELEM
ax_wh	Wh-apex	4	ELEM
B	Binder, i.e. a 'subordinating conjunction'	809	ELEM
C	Complement	11891	ELEM
C_Repl	Replacement for a Complement	121	ELEM
C_Wh	Wh-Complement	392	ELEM
Cl	Clause	16603	UNIT
Cl_Un	Unfinished Clause	558	UNIT
cv	completive	2902	ELEM
cv_repl	replacement for a completive	12	ELEM
cv_wh	wh-completive	4	ELEM

¹ This table is one of the corpus index tables (see Chapter Seven) and its main use is to provide the user of ICQF+ with help on the meaning of the syntax tokens.

TOKEN	MEANING	FREQ	TYPE
DATA	DATA - i.e. lexical items	61377	ITEM
dd	deictic determiner	3436	ELEM
dd_wh	wh-deictic determiner	32	ELEM
do	Ordinal determiner	0	ELEM
dt	degree temperer	274	ELEM
dt_wh	wh degree temperer	13	ELEM
ELEMQUERY	Element that was questionable in the analysis	170	ELEM
et	emphasizing temperer	10	ELEM
Excl	Exclamation Formula	160	ELEM
F	Formula	2367	ELEM
fi	finisher	52	ELEM
Fr	Frame	99	ELEM
g	genitive element	68	ELEM
genclr	genitive cluster	74	UNIT
h	head expounded by a 'common noun'	7789	ELEM
h_n	head expounded by a name (ie a 'proper noun')	1418	ELEM
h_p	head expounded by a 'pronoun'	11372	ELEM
h_rcc	Head, recoverable from cultural classification	520	ELEM
h_sit	situation head (i.e. a head filled by a clause)	19	ELEM
h_wh	wh-head	623	ELEM
I	Infinitive Element	1248	ELEM
inf	inferer	27	ELEM
ITEMQUERY	The analyst could not identify the item (word)	4	ITEM
L	Let element	22	ELEM
Lnk	Linker (i.e. 'co-ordinating conjunction; ampersand in GENESYS)	2366	ELEM
M	Main Verb	10873	ELEM
MEx	Main Verb Extension	1357	ELEM
mo	modifier expressing experiential meaning	1090	ELEM
N	Negator	324	ELEM
ngp	nominal group	22248	UNIT
ngp_un	Uninished nominal group	131	UNIT
O	Operator (form of "do" or a 'modal verb')	2288	ELEM
od	ordinative determiner	47	ELEM
OM	Operator conflated with Main Verb	3104	ELEM
own	owner	7	ELEM
OX	Operator conflated with Auxiliary	1904	ELEM
p	preposition	2986	ELEM
pd	partative determiner	179	ELEM
pex	preposition extension (e.g. 'with his hat on')	2	ELEM
pgp	prepositional group	2938	UNIT
pgp_un	Unfinished prepositional group	52	UNIT
po	possessor	73	ELEM
pt	prepositional temperer	63	ELEM
q	qualifier	659	ELEM
q_mo	quantifying modifier	42	ELEM
q_repl	Replacement qualifier	2	ELEM
qd	quantifying determiner	3753	ELEM
qd_wh	wh-quantifying determiner	6	ELEM
qld	quality group deictic	60	ELEM
qlgp	quality group	2392	UNIT

TOKEN	MEANING	FREQ	TYPE
qlgp_un	unfinished quality group	9	UNIT
qtd	quantity group deictic	13	ELEM
qtf	quantity group finisher	8	ELEM
qtgp	quantity group	290	UNIT
qtgp_un	Unfinished quantity group	1	UNIT
rd	representative determiner	5	ELEM
rel_mo	relational modifier	161	ELEM
s	Subject	10832	ELEM
s_it	Experientially empty 'it' Subject	85	ELEM
s_Repl	Replacement Subject	28	ELEM
s_th	Experientially empty 'there' Subject	420	ELEM
s_Wh	Wh-Subject	113	ELEM
sc	scope	47	ELEM
sd	superlative determiner	25	ELEM
sit_mo	'situation' modifier	43	ELEM
t_Wh	wh-temperer		ELEM
td	typic determiner	24	ELEM
text	text	133	ELEM
th_mo	'thing' modifier	342	ELEM
v	selector (always "of", hence 'v')	421	ELEM
Voc	Vocative	375	ELEM
x	Auxiliary Verb	463	ELEM
XEx	Auxiliary Extension	470	ELEM
z	Sentence (Z representing Greek 'S', i.e. sigma)	10791	ELEM

Table B.1: Alphabetical list of syntax tokens in the FPD Corpus

Appendix C

Marking up language texts

This appendix serves two purposes:

- (a) it gives the necessary background to mark up languages for the readers that need it in preparation for Chapter Six (see Section C.1),
- (b) it provides details of the artefacts from the experiments with mark up languages in this project (see Section C.2).

C.1 Defining Mark up

In this section, we provide the reader with the necessary information about mark up languages in order to present the models we use in this project. For more information about SGML, refer to, for example ISO specification (ISO 1986), Bryan (1988), Goldfarb (1991) and van Herwijnen (1994). For a reliable reference to XML, see Harold (1999) and the W3C specification (W3C 2004a).

C.1.1 Definitions

A marked-up document or file (called an **instance**) is a hierarchical collection of mark up tags and data. A mark up element, which can contain data or other mark up elements, is delimited by a **start tag** and an **end tag**. A start tag starts with a **mark up declaration open** (MDO) character `<`,¹ followed by a **generic identifier** (the tag name), a set of **attributes** (if any) and a **mark up declaration close** (MDC) character `>`. An end tag starts with an MDO and is followed by a slash `/`, the generic identifier and then an MDC. For example, the start tag for a paragraph may be `<para>` and the end tag will be `</para>`; the start tag for a nominal group could be `<ngp>` and its end tag `</ngp>`.

There are rules that govern the creation of generic identifiers that have had an impact on our work here. In both SGML and XML, it is mandatory that generic identifiers start with a letter; in SGML generic identifiers are not case sensitive and in XML they are. There are also rules about which characters can be used in a generic identifier.²

¹ In XML, `<` is the only character that can be used for MDO; in SGML, any character can be defined in the **SGML declaration** (see ISO 1986), however, I am yet to find an application that uses anything other than `<`. The same note applies to MDC.

² This is the reason why certain elements in Fawcett (2000a) have been renamed (such as `&` being `Lnk`).

Start tags can contain attributes which further qualifies the element. For example, in a document, it may state that a table's border colour is red or, for a Subject in a Clause, that it is ellipted. The format of an attribute is attribute name, followed by the equals sign and a quote delimited attribute value. Examples of mark up elements with attributes follow.

```
<table bordercolour="red" width="100%">
<para securityclass="restricted">
<S ellipted="RapidSpeech">
```

A mark up element may be defined as **empty**; this means that it is not allowed to contain any data or other mark up elements. Empty elements are signified by only a start tag in SGML. XML has alternative representations for empty tags. They can be shown with no content or with a / before the MDC. The following shows some example XML elements that are equivalent. Hence:

```
<qa type="first_verified"/>
<qa type="first_verified"></qa>
```

and

```
<S ellipted="Rapid Speech"/>
<S ellipted="Rapid Speech"></S>
```

are both allowed in XML to represent the same thing.

A **Document Type Definition (DTD)** or **XML Schema** (see Section C.1.2) supplies the rules which govern the mark up elements and their attributes that may occur in a given document type. They state the order in which mark up elements may or must occur, the attributes (and types of attributes) they may or must have and the content that they may or must have. An SGML or XML **parser** is a computer program that checks that an SGML or XML instance conforms to the DTD or Schema.

C.1.2 Document Type Definitions and Schemas

There are rules that govern which mark up elements may occur in an instance and what they are to contain in terms of other mark up elements, attributes and data. Further rules are applied to the types of values that a mark up attribute can take. In SGML, it is the Document Type Definition (DTD) that does this; XML adopted SGML's DTD (with some changes) and the World Wide Web Consortium (W3C) decided that it needed strengthening (in terms of data typing, for example) and introduced a specification for an XML Schema (W3C 2004b). In this project, because we started with SGML, we have been using DTDs rather than Schemas. SGML

demands that an instance conforms to a DTD; XML, when there is no reference to a DTD or Schema, only demands that the instance is **canonical** (or **well-formed**, having a end tag for every start tag and it does not break the basic rules of XML). An explanation of a DTD and its structures is given by example in the next section.

C.1.3 Document Type Definitions (DTDs)

An example DTD for an abstract mark up scheme (see Section 6.1.4.2 of Chapter Six) is shown in Figure C.1. Each SGML or XML document represents an individual sentence in the corpus. The **!DOCTYPE** statement identifies the DTD with a **public identifier** (see ISO 1986) and this must be specified in every sentence instance that conforms to the DTD. The **!ELEMENT** statements define a mark up element: the generic identifier appears first (e.g. **sentence**), this is followed by two dashes which indicate that both the mark up element's start tag and its end tag must be present.³ Next is the element's **content model** which contains a list of the elements that may occur within this element's content. The symbol **+** means '1 or more' (the element sentence comprises one or more units), **?** means '0 or none', and **|** means 'or' (hence, **elementS** can contain one or more co-ordinated units **or** an item, **or** it can contain nothing when it is ellipted)⁴.

!ATTLIST defines the mark up attributes that can be applied to the mark up element's start tag. Here the attribute name comes first and is followed by the type of data that the attribute value can be. **CDATA** means that the attribute can contain any characters, **ID** means that it is to contain a **unique identifier**, and values in brackets (e.g. (**RapidSpeech|Recoverable**)) provide a list of choices. If there is a default value, it is indicated in brackets after the list.⁵ Non-list attributes are followed by an optionality indicator **#REQUIRED** means that the attribute must be present and **#IMPLIED** means it is optional. Figure C.2 shows a sample sentence marked up according to the rules of this DTD:

```
<DOCTYPE sentence PUBLIC "-//DTD/SENTENCE COMMUNAL VERSION 1/EN" [  
  
<!ELEMENT sentence - - (unit+) >  
<!ATTLIST sentence      sentid CDATA #IMPLIED>  
<!ELEMENT unit      - - (elementS+) >
```

³ This feature does not exist in XML because it does not allow optional end tags.

⁴ The symbol “,” which is not in this DTD means sequence and “*”, also not in the DTD, means “0 or more”.

⁵ See the SGML specification (ISO 1986) for complete details of SGML data types.

```

<!ATTLIST unit
            unitid ID #IMPLIED
            syntaxtoken CDATA #REQUIRED>
<!ELEMENT elements - - (unit+ | item)?>
<!ATTLIST elements
            elemid CDATA #IMPLIED
            syntaxtoken #REQUIRED
            ellipted(RapidSpeech|Recoverable)
            "RapidSpeech">
<!ELEMENT item - - (#PCDATA)>
<!ATTLIST item
            itemid ID #IMPLIED
            intonationstart CDATA #IMPLIED
            intonationend CDATA #IMPLIED>
] >

```

Figure C.1: a simplified DTD for the Cardiff Grammar (version 1)

```

<!DOCTYPE sentence PUBLIC "-//DTD/SENTENCE COMMUNAL VERSION 1/EN" []>
<sentence sentid="10A...">
  <unit syntaxtoken="C1">
    <elementS syntaxtoken="S">
      <unit syntaxtoken="ngp">
        <elementS syntaxtoken="dd">
          <item>The</item>
        </elementS>
        <elementS syntaxtoken="h">
          <item>seagulls</item>
        </elementS>
      </unit>
    </elementS>
    <elementS syntaxtoken="M">
      <item>ate</item>
    </elementS>
    <elementS syntaxtoken="C">
      <unit syntaxtoken="ngp">
        <elementS syntaxtoken="dd">
          <item>The</item>
        </elementS>
        <elementS syntaxtoken="h">
          <item>mackerel</item>
        </elementS>
      </unit>
    </elementS>
  </unit>
</sentence>

```

Figure C.2: A sample sentence marked up according to the DTD in Figure C.1

C.1.4 Using mark up languages for non-hierarchical structures

In order for mark up languages to be useful for annotating parse structures, we had to prove that they could be used for non-hierarchical and therefore discontinuous structures. SGML and XML provide a mechanism called **ID/IDREF** that was designed for document cross-referencing (see `itemid` attribute of the `item` element in the DTD of Figure C.1).

The following example demonstrates the use of ID/IDREF within documentation; Section 6.1.6.1 of Chapter Six discusses how we used these constructs in discontinuous parse trees.

```
<figure id="F0921">
<title>A view of Chesil Beach from the Bridging Camp</title>
<graphic boardno="ICN-E2A000000RK0378A011"/>
</figure>
```

The author of the document is then able to cross-refer to the figure by specifying the ID in a cross-reference element as below:

```
<para>A view of the beach from the Bridging Camp is shown in Figure
<xref xrefid="F0921"/>.</para>
```

TigerXML relies heavily on XML's ID/IDREF mechanism as can be seen in Figure 6.8 of Chapter Six.

C.2 Artefacts from the experiments with mark up languages

Chapter Six described the experiments with mark up languages in this project. Section 6.1.5.1 of Chapter Six described my efforts in creating an SGML DTD both manually and automatically from the POW Corpus (using the **createDTD** program). This section gives the DTD that was created manually for the Cardiff Grammar.

It contains an SGML content model for each class of unit presented in Appendix A of Fawcett (2000a)..

C.2.1 The nominal group

The nominal group was modelled using the SGML DTD content model diagram in Figure C.3. This states that it can optionally start with a linker, followed by various types of determiner with optional selectors, zero or more modifiers and then a mandatory head, proper name head or pronoun head, followed by zero or more qualifiers.⁶ In this DTD note that a modifier can be filled by a **ngp**, or a **Cl**, or a **qlgp** or a **genc1r** or be expounded by an **item**.⁷

⁶ A head is of a certain type and could have been (but wasn't) modelled as attributes as in **<h type="pronoun">**.

⁷ At the time I created the DTD, I didn't realise that elements of structure may be filled by coordinated units of different types (Fawcett, personal communication, 2005), however, this is extremely rare.


```

<!ELEMENT ngp - - (Lnk?, (rd,v?)?, (pd,v?)?, sd|od), v?)?,
(td,v?)?, dd?, mo*, (h|h_n|h_p), q*)>
<!ELEMENT Lnk - - (item)>
<!ELEMENT rd - - (ngp+ | item)>
<!ELEMENT pd - - (ngp+ | item)>
<!ELEMENT sd - - (ngp+ | qlgp+ | item)>
<!ELEMENT od - - (ngp+ | item)>
<!ELEMENT v - - (item)>
<!ELEMENT sd - - (qtgp+ | item)>
<!ELEMENT od - - (qlgp+ | item)>
<!ELEMENT td - - (ngp+ | item)>
<!ELEMENT dd - - (genclr | item)>
<!ELEMENT mo - - (ngp+ | Cl+ | qlgp+ | genclr | item)>
<!ELEMENT h|h_p|h_n - - (genclr | item)>
<!ELEMENT q - - (ngp+|Cl+|qlgp+|pgp+)

```

Figure C.3: SGML DTD content model for the nominal group

C.2.2 The quality group

The structure of the quality group is represented by the SGML DTD content model shown in Figure C.4. Note that only one type of temperer can occur before an apex and a degree temperer can occur after it; the content model forbids any temperers before the apex, if a degree temperer occurs after the apex. I have assumed that two scopes can occur before a finisher and one afterwards, and my model assumes that they are not mutually exclusive.

```

<!ELEMENT qlgp - - (Lnk?, qld?, qlq?, ((et|dt|at)?, ax) |,
(ax, dt?)), sc?, sc?, fi?, sc?>
<!ELEMENT qld - - (genclr | item)>
<!ELEMENT qlq - - (item)>
<!ELEMENT et - - (item)>
<!ELEMENT at - - (qlgp+|item)>
<!ELEMENT dt - - (ngp+|item) >
<!ELEMENT ax - - (item) >
<!ELEMENT sc - - (pgp+ | item)>
<!ELEMENT fi - - (Cl+|pgp+|item)>

```

Figure C.4: SGML DTD content model for the quality group

C.2.3 The quantity group

The SGML DTD content model for quantity group is shown in Figure C.5.

```

<!ELEMENT qtgp - - (ad?, am, qtf)>
<!ELEMENT ad - - (qtgp+|item)>
<!ELEMENT am - - (item)>
<!ELEMENT qtf - - (Cl+|pgp+|item)>

```

Figure C.5: SGML DTD Content model for quantity group

C.2.4 The prepositional group

The SGML DTD content model for quantity group is shown in Figure C.6. Note that there are two alternate models, the first and most common is with the preposition first, which is represented by the content model `(pt?,p, cv)`, as in:

up on the mountain

The second is when the preposition follows the completive `(pt?,cv,p)`, as in:

Up to two weeks ago

These structures are mutually exclusive and modelled in the SGML content model as alternative options. Note that preposition and completive are both mandatory in these models.

```
<!ELEMENT pgp - - ((pt?,p, cv) | (pt?,cv,p))>
<!ELEMENT pt - - (item)>
<!ELEMENT p - - (qlgp+|qtgp+|item)>
<!ELEMENT cv - - (ngp+|pgp+)>
```

Figure C.6: SGML DTD content model for the prepositional group

C.2.5 The genitive cluster

The SGML DTD content model given in Figure C.7 represents the genitive cluster.

```
<!ELEMENT genclr - - (Lnk?,po,g?,own?)>
<!ELEMENT po - - (ngp+)>
<!ELEMENT g - - (item)>
<!ELEMENT own - - (qlgp+|item)>
```

Figure C.7: SGML DTD content model for the genitive cluster

C.2.6 The text unit

Text caused me a problem in creating the DTD. This was because text contains one or more sentences and sentence is the root node of the parse tree; in SGML and XML, you can only have one element at the root and it cannot appear as a child of any other elements. Therefore, I created the element `<Ztext>` to represent a sentence that is within the TEXT unit. The content model for TEXT then became very simple and is shown in Figure C.8.

```
<!ELEMENT TEXT - - (Ztext+)>
<!ELEMENT Ztext - - (C1+)>
```

Figure C.8: SGML DTD content model for the TEXT unit

C.2.7 Sentence

The sentence node is the root node in the DTD and has the content model shown in Figure C.9; note that certain attributes were added to identify the sentence in a corpus database.

```
<!ELEMENT Z - - (Cl+)>
<!ATTLIST Z - - ID ID #REQUIRED
          POWCELL CDATA #REQUIRED>
```

Figure C.9: SGML DTD content model for Sentence

C.2.8 The Clause

I introduce the Clause, which was modelled using the SGML content model shown in Figure C.10 last, as it is by far the most complex structure to model in an SGML DTD because of optionality and mutual exclusivity. The alternate structures shown in the content model are for operator before subject and complement before main verb, operator after subject and complement before main verb, operator before subject and complement after main verb, and operator after subject and complement after main verb.

```
<!ELEMENT Cl - - (Lnk?,B?,A*,
  (C*,O?,S?,N?,A*,I?,(X*|Xex*),M,Mex?) |
  (C*,S?,O?,N?,A*,I?,(X*|Xex*),M,Mex?) |
  (S?,O?,N?,A*,I?,(X*|Xex*),M,C*,Mex?) |
  (S?,O?,N?,A*,I?,(X*|Xex*),M,C*,Mex?) |
  (O?,S?,N?,A*,I?,(X*|Xex*),M,Mex?,C*) |
  (C?,S?,O?,N?,A*,I?,(X*|Xex*),M,Mex?,C*) |
  (S?,O?,N?,A*,I?,(X*|Xex*),M,Mex?,C*) |
  (S?,O?,N?,A*,I?,(X*|Xex*),M,Mex?,C*))
,A*,Voc) >

<!ELEMENT Lnk - - (item)>
<!ELEMENT B - - (item)>
<!ELEMENT A - - (Cl+|ngp+|pgp+|qlgp+|qtgp+|item)>
<!ELEMENT C - - (TEXT|Cl+|ngp+|pgp+|qlgp+|item)>
<!ATTLIST C - - PR (Agent|Affected|Attribute)
          #IMPLIED>
<!ELEMENT S - - (TEXT|Cl+|ngp+)>
<!ATTLIST S - - PR (Agent|Affected|Attribute)
          #IMPLIED>
<!ELEMENT O - - (item)>
<!ELEMENT N - - (item)>
<!ELEMENT I - - (item)>
<!ELEMENT X - - (qlgp+|item)>
<!ELEMENT Xex - - (qlgp+|item)>
<!ELEMENT M - - (item)>
<!ELEMENT Mex - - (ngp+|qlgp+|pgp+|Cl+|item)>
<!ELEMENT Voc - - (ngp+|item)>
<!ELEMENT item - - (#PCDATA)>
```

Figure C.10: SGML DTD content model for the clause

C.3 Summary

The use of a DTD or Schema in conjunction with a business rules checker is seen as a valuable component of this project. Its use would allow error checking to be performed on the sentences in the corpus database and to check sentences before they are added to the corpus.

As reported in Chapter Six, the DTDs created as part of this project were not used in the final system due to:

- (a) the manually created DTD (reported here) was not suitable for naturally occurring texts because many of the syntactic representations were not covered in the DTD structure.
- (b) the work on the automatically created DTD was more promising but the **createDTD** program would need to be substantially updated in order to simplify the SGML content models it produces.

The **createDTD** program will be modified in or after Phase Two of this project and an XML Schema will be produced for the Cardiff Grammar.⁸

⁸ Although **createDTD** may have to be renamed **createSchema**.

Appendix D

The database schema

This appendix provides details of the complete database schema. It includes:

- (a) The **corpus tables** (see Section D.1),
- (b) The **corpus index tables** (see Section D.2),
- (c) The **probabilities tables** (see Section D.3), and
- (d) The **parser working tables** (see Section D.4).

D.1 The corpus tables

The corpus tables are used to store the native XML data that forms the corpus.¹ They comprise of the following tables:

- (a) the **DOCUMENT** table, which stores high level metadata about the XML documents in the database,
- (b) The **ELEMENT** table, which stores XML mark up element details,
- (c) The **ATTRIBUTE** table, which stores XML attribute information,
- (d) The **PCDATA** table, which stores parsed character data.

The structure of the corpus tables is described in Table D.1.

D.2 The corpus index tables

The corpus index tables provide a quick indexed reference to the data held within the corpus tables. They are used with ICQF+ and its data transformation programs to create the FPD Corpus (see Chapters Eight and Nine). There is one main corpus index table and other less significant ones (see Section 7.2.2 of Chapter Seven).

When new sentences are added to the corpus, or existing ones are modified, the corpus index tables are automatically updated.

The main corpus index table is called **MARKUPINDEX** and provides information about mark up elements, their parents, ancestors, children and siblings. This index allows ICQF+ to rapidly locate sentences in the corpus that match given search criteria.

The other index tables include:

¹ The corpus tables are essentially native XML tables that are able to store any XML (or SGML) document, but are used only to store corpus information in this project.

- (a) Indexes of units, elements, and items that are used for ICQF+'s reporting functions
- (b) An index of syntactic tokens and their associated meanings. This is used for ICQF+'s user help and can be seen in Appendix B.

Only the **MARKUPINDEX** table is described here in Table D.2.

D.3 The probabilities tables

These tables store the data that is used for the queries needed by the parser. They are automatically updated as new sentences are added to the corpus provided that this feature is enabled. The tables are:

- (a) the **IVS_ITEM** table - used for **initial vertical strip item** queries,
- (b) the **IVS_ELEM** table - used for **initial vertical strip element** queries,
- (c) the **I2E** table - used for **item-up-to element** queries,
- (d) the **I2E2U2E** table - used for **item-up-to-element-up-to-unit-up-to-element** queries,
- (e) the **E2U** table - used for **element-up-to-unit** queries,
- (f) the **U2E** table - used for **unit-up-to-element** queries,
- (g) the **FUS** table - used for **forward unit structure** queries,
- (h) the **BUS** table - used for **backward unit structure** queries.

These tables are described in Table D.3. See also Chapter Fourteen and Appendix H. See Appendix I for details of the improved Version Two probabilities tables.

D.4 The parser working tables

These tables store the parser's working data:

- (a) the **DB_PARSE_ITEM** table - stores the items in the sentence being parsed,
- (b) the **DB_PARSE_TREE** table - stores details about the built and candidate structures,
- (c) the **DB_PARSE_NODE** table - stores details about the nodes in the built and candidate structures,
- (d) the **DB_IVS** table - stores the initial vertical strips created in Stage 1,
- (e) the **DB_PARSE_NEXT_NODE** table - stores details of the syntax token that can follow a given node in a forward prediction,
- (f) the **DB_PARSE_PREV_NODE** table - stores details of the syntax token that can precede a given node in a backward prediction,

(g) the **DB_PST** table - the parser state table.

These tables are given in Table D.4 and have a set of shadow tables (eg **DB_PARSE_NODE_SAVE** which are used in the Parser WorkBench to save and restore partial parses.

Note that there are also other parser working tables that are not covered here. These include tables that are used to save and restore the parser's configurable parameters, and tables that are used to store the contents of a parse when the parse is saved (see Chapter Sixteen for the reasons why these tables are needed).

Table D.1: The structure of the native XML corpus tables

Table name	Field	Description
the mark up document table (DOCUMENT)		This table stores the details about the mark up document
	DOCID	A unique identifier given to the document (sentence)
	DOCNAME	The name of the document (e.g. POW Cell Id)
	REMARKS	A field for general remarks
	TAG1...TAG10	General fields for storing metadata (used for POW Cell fields)
	DATETIME REMARKS	The date and time that the document was added or modified Any remarks that may be applied (for example by the batch modification programs)
the mark up element table (ELEMENT)		This table stores mark up elements
	DOCID	Foreign key to the DOCUMENT table
	ELEMID	A unique identifier within the document that identifies the element
	GI	The generic identifier of the mark up element
	PARENTID	The unique identifier of the element's parent
	DATETIME REMARKS	The date and time that the element was added or modified Any remarks that may be applied (for example by the batch modification programs)
the mark up attribute table (ATTRIBUTE)		This table stores the details of the mark up attribute
	DOCID	Foreign key to the DOCUMENT table (composite key with ELEMID)
	ELEMID	Foreign key to the ELEMENT table (composite key with DOCID)
	ATTNAME	Name of the attribute
	ATTVAL ATTID	The value of the attribute A unique identifier for the attribute used to maintain sequence of attributes in the element (although SGML and XML do not state that attributes are to be in any order)

Table D.2: The MARKUPINDEX table

Table name	Field	Description
The mark up index table (MARKUPINDEX)		This table is the main index to the corpus tables
<i>mark up field</i>	DOCID	A unique identifier given to the document (sentence) (composite key with ELEMID to the ELEMENT table)
<i>mark up field</i>	ELEMID	The unique identifier of the element in the document (composite key with DOCID to the ELEMENT table)
<i>mark up field</i>	GI	The generic identifier of the element in ELEMID
<i>mark up field</i>	CHILDIDS	A string of the unique identifiers of the element's children
<i>mark up field</i>	CHILDGIS	A string containing a list of the child generic identifiers separated by spaces
<i>mark up field</i>	LEFTSIBIDS	A string of IDs of the elements to the left of this element within its parent
<i>mark up field</i>	LEFTSIBGIS	A string of generic identifiers of the elements to the left of this element within its parent
<i>mark up field</i>	RIGHTSIBIDS	A string of unique identifiers of the elements to the right of this elements within its parent
<i>mark up field</i>	RIGHTSIBGIS	A string of generic identifiers of the elements to the right of this element within its parent
<i>mark up field</i>	ABOVEIDS	A unique identifier within the document that identifies the element
<i>mark up field</i>	ABOVEGIS	A string of generic identifiers of the elements that occur above this element up to the root element
<i>mark up field</i>	ABOVEIDS	A string of unique identifiers of the elements that occur above this element up to the root element
<i>mark up field</i>	PARID	The unique identifier of the element's parent (see Note 1)
<i>mark up field</i>	PARPARID	The unique identifier of the element's grandparent (see Note 1)
<i>mark up field</i>	PARPARPARID	The unique identifier of the element's great grandparent (see Note 1)
<i>mark up field</i>	PARGI	The generic identifier of the element's parent (see Note 1)
<i>mark up field</i>	PARPARGI	The generic identifier of the element's grandparent (see Note 1)
<i>mark up field</i>	PARPARPARGI	The generic identifier of the element's great grandparent (see Note 1)
<i>mark up field</i>	TAG1...TAG10	Ten fields that can store any metadata (as in DOCUMENT table) - store the POW Cell Identifiers (see Note 4)
<i>linguistic field</i>	TYPE	One of ELEMent , UNIT , or ITEM
<i>linguistic field</i>	INITIAL_IN_SENTENCE	True if this mark up element is initial in the sentence (i.e. on the left most strip of the parse tree) (see Note 2)
<i>linguistic field</i>	INITIAL_IN_PARENT	True if the mark up element is initial within its parent (see Note 2)
<i>linguistic field</i>	INITIAL_IN_SENTENCE_NO_ELLIPSIS	True if this mark up element is initial in the sentence after any ellipted elements have been ignored (see Note 2)
<i>linguistic field</i>	INITIAL_IN_PARENT_NO_ELLIPSIS	True if this mark up element is initial in its parent after any ellipted elements are ignored (see Note 2)

Table name	Field	Description
<i>linguistic field</i>	ELLIPTED	True if this mark up element has an ELLIPTED attribute (see Note 3)
<i>linguistic field</i>	ELLIPTED_RS	True if this element has an ELLIPTED attribute set to RapidSpeech (see Note 3)
<i>linguistic field</i>	ELLIPTED_PT	True if this element has an ELLIPTED attribute set to PreviousText (see Note 3)

Note 1: These fields are replicated fields as the information is also in ABOVEIDS and ABOVEGIS. They are replicated to speed up queries.

Note 2: These fields are used for ICQF+ queries for initial in sentence and initial in parent.

Note 3: These fields are used in ICQF+ for ignoring ellipsis.

Note 4: These fields are used in ICQF+ for query restrictions.

Table D.3: The probabilities tables

Table name	Field	Description
initial vertical strip item (IVS-ITEM)	ITEM	This table gives a set of vertical strips that can occur above a given item
	VERTSTRIP	the item
	PROBABILITY	a string representing the elements and units that occur above the item the probability of the given vertical strip among all possible strips
initial vertical strip elem (IVS-ELEM)	ELEM	This table gives a set of vertical strips that occur above a given element
	VERTSTRIP	the element
	PROBABILITY	a string representing the units and elements that can occur above the element the probability of the given vertical strip among all possible strips
item-up-to-element (I2E)		the I2E probabilistic table gives a list of elements that a given item can expound together with their associated probabilities.
	ITEM	the item itself
	ELEM1	the element that the item expounds
	PROBI2E	the probability that the item expounds the given element
	I2E2U2NEEDED	a Boolean field that is set to true if an I2E2U2E is needed for this item-element pair
item-up-to-element-up-to-unit-up-to-element (I2E2U2E)	IVSITEM	a Boolean field that is set to true if this item is an IVS-ITEM
		Certain items behave quite differently from others when we take into account the element of structure that occurs above the unit above the expounded element. These are stored in the I2E2U2E table.
	ITEM	the item itself
	ELEM1	the element that the item expounds
	UNIT	the unit that ELEM1 is a component of

Table name	Field	Description
	ELEM2	the element that is filled by UNIT
	PROB1E2U2E	the probability of the item-element-unit-element quadruple
element-up-to-unit (E2U)		the E2U table stores a list of units that a given element can be a component of (usually only one)
	ELEM	the element
	UNIT	the unit that ELEM is a component of
	UNITPROB	the probability that the ELEM is a component of the UNIT (normally 100%)
unit-up-to-element (U2E)		the U2E table gives a list of elements that a given unit can fill
	UNIT	the unit
	ELEM	the element that UNIT can fill
	ELEMPROB	the probability that the given UNIT fills the given ELEM
forward structure query (FUS)		The forward unit structure table gives a list of elements that can follow a given set of elements in a unit
	UNIT	the unit
	ELEMENTS	the set of elements that have already been found in the unit
	NEXTELEM	the element that can follow the set of elements in ELEMENTS
	PROBABILITY	the probability that the given NEXTELEM can follow ELEMENTS
backward structure query (BUS)		This table stores the element that can precede a given element in a given unit
	UNIT	the unit
	ELEMENT	the element that is the subject of the query
	PREVELEM	the element that can precede ELEMENT
	PROBABILITY	the probability that PREVELEM precedes ELEMENT

Table D.4: The parser working tables

Table name	Field	Description
the parse items (DB_PARSE_ITEM)	ITEM	This table stores the items in the sentence being parsed the item
	STARTPOS	the position that the item starts
	ENDPOS	the position that the item ends (see Note 5)
the parse trees (DB_PARSE_TREE)	TREEID	This table stores the built and candidate structure trees a unique identifier given to the tree
	TREELABEL	a label that says if this is a candidate structure or a built structure
	PROBABILITY	the tree's score
	COMPLETE	a Boolean that is true if the tree is one that represents the complete sentence
	ACTIVE	a Boolean that is true if the tree is active
	STARTPOS	an integer that represents the start position of the tree
	ENDPOS	an integer that represents the end position of the tree
the parse tree nodes (DB_PARSE_NODE)		this table stores the nodes that belong to the parse trees
	TREEID	the identifier of the tree
	NODEID	the identifier of the node in the tree
	PARENTID	the identifier of the parent node of this node
	LEFTSIBID	the identifier of the immediate left sibling of the node
	RIGHTSIBID	the identifier of the immediate right sibling of the node
	TOKEN	the syntax token stored in the node (e.g. S, C, Cl, ngp, Z,...)
	TYPE	a label that states if this is an ITEM node, an ELEMENT node or a UNIT node
	LEVEL_PROB	the level probability of the node
	NODE_PROB	the node probability
	RMS	a Boolean set to true if the node is in the right-most strip
	LMS	a Boolean set to true if the node is in the left-most strip
	STRUCTURE	a string containing a list of the node's child elements (redundant but incorporated for efficiency of the parser)
the initial vertical strip table (DB_IVS)		the initial vertical strip table created in Stage 1
	ITEM	the initial item of the parse
	VERTSTRIP	the vertical strip from the element that the item expounds up to a sentence node

Table name	Field	Description
	FOLLOWED	true if this strip has been taken forward
	PROBABILITY	the score of this strip among the other strips
the next node table (DB_PARSE_NEXT_NODE)		this table holds the forward predictions from a DB_PARSE_NODE that is a built structure are the result of a FUS query
	TREEID	the unique identifier of the tree to which this prediction belongs
	NODEID	the unique identifier of the node
	NEXTTOKEN	the syntax token of the next element
the previous node table (DB_PARSE_PREV_NODE)		this table holds the backward predictions from the a DB_PARSE_NODE that is in a candidate structure and is the result of a BUS query
	TREEID	the unique identifier of the tree to which this prediction belongs
	NODEID	the unique identifier of the node to which this prediction belongs
	PREVTOKEN	the predicted previous syntax token
	PROBABILITY	the score associated with the prediction
the parser state table (DB_PST)		the parser state table
	BSTREEID	the identifier of the BS tree involved in the parser move
	CSTREEID	the identifier of the CS tree involved in the parser move
	BSNODEID	the identifier of the BS node that is involved in for e.g. a join
	CSNODEID	the identifier of the CS node that is involved in for e.g. a join
	NEXTSTATE	the next state in which the tree or tree-pair that is represented by this PST record will move to if selected
	SCORE	the score assigned to the join, grow etc that is represented by the record
	CYCLES	the number of stage 3,4,3,5 cycles undertaken by the tree-pair
	STAGE4TRIED	true if a Stage 4 has been tried on this candidate structure
	FOLLOWED	true if this PST record has been followed
	BACKTRACK_POINT	true if this represents a linguistically motivated backtrack point (see Note 6)
	ENDPOS	the end position of the BSTREE or the CSTREE

Note 5: An item may cover more than one position (e.g. *in spite of* - covers three).

Note 6: This will be used in Phase 2 for linguistically motivated backtracking along with another table **DB_PARSE_BACKTRACK_STACK**.

Appendix E

Example ICQF+ reports

This appendix provides examples of the reports extracted from the FPD Corpus. The reports are exported from ICQF+'s Report functions into XML and formatted using XSL-FO (see Section 8.4.3 of Chapter Eight).¹ The following reports are provided:

- (a) the item-up-to-element report (see Figure E.1),
- (b) the unit-up-to-element report (see Figure E.2),
- (c) the element-up-to-unit report (see Figure E.3),
- (d) the complete item report (see Figure E4).

Item	Total Freq.	Element	Freq./Elem.	Probability
the	2228	dd (deictic determiner)	2165	97.17%
		qld (quality group determiner)	50	2.24%
		qtd (quantity group determiner)	13	0.59%
...				
them	331	dd (deictic determiner)	35	10.57%
		h_p (pronoun head)	296	89.42%

Figure E.1: Excerpt from the item up to element report (from the FPD Corpus)

Unit: ngp - nominal group			
Occurs: 22248			
Fills		Frequency	Probability
S	Subject	8882	40.64%
C	Complement	7012	32.08%
cv	completive	2737	12.52%
A	Adjunct	681	3.11%
Voc	Vocative	375	1.66%
...

Figure E.2: Excerpt from the unit-up-to-element report for the nominal group (from the FPD Corpus)

¹ The first three figures show the data replicated in the word processor and the fourth is an image of a page of the large item report which exists as a PDF file.

Element: Lnk - Linker**Occurs: 2351**

In unit		Frequency	Probability
Cl	Clause	1829	84.48%
ngp	nominal group	301	13.74%
qlgp	quality group	33	1.40%
pgp	prepositional group	9	0.38%

Figure E.3: Excerpt from the element-up-to-unit table for linker (from the FPD Corpus)

ICQF+ Item report

Pos.	Item	Element	Total this element	Total item	Probability
0	I	h_p	2645	2645	100%
1	THE	dd	2228	2165	97.1724%
2	A	qd	1882	1881	99.9469%
3	AND	Lrk	1658	1658	100%
4	IT	h_p	1582	1509	95.3856%
5	YOU	h_p	1291	1291	100%
6	S	OM	1407	1153	81.9474%
7	WE	h_p	1131	1131	100%
8	TO	I	1289	1098	85.1823%
9	YEAH	F	987	987	100%
10	THAT	h_p	842	830	98.5748%
11	GOT	M	902	694	76.9401%
12	THEY	h_p	609	609	100%
13	NO	F	690	578	83.7681%
14	IN	p	559	559	100%
15	PUT	M	472	470	99.5763%
16	DONT	O	419	414	98.8067%
17	HE	h_p	412	412	100%
18	OF	v	398	398	100%
19	ON	p	393	391	99.4911%
20	THERE	h_p	786	387	50.5222%
21	ONE	h_roc	561	386	68.8057%
22	THERE	S_th	379	379	100%
23	CAN	O	367	367	100%
24	YES	F	350	350	100%
25	MY	dd	352	346	98.2955%
26	LOOK	M	345	345	100%
27	HAVE	M	620	339	54.6774%
28	KNOW	M	316	314	99.3671%

Item report

PAGE 2

Figure E.4: The item report (sorted by item frequency)

Appendix F

Modifying the corpus: details of the changes made in Stage One

In Chapter Nine, I described how we updated the Polytechnic of Wales (POW) Corpus to create the Fawcett-Perkins-Day (FPD) Corpus, which is analysed according to the latest version of the Cardiff Grammar. I also explained how many of the changes were performed automatically in **Stage One** of the conversion. This Appendix is a record of those changes.

During **Stage One**, we kept the uppercase syntax tokens used in the POW Corpus, and when we inserted a new syntax token that was not in the earlier version of the Cardiff Grammar, we inserted it in uppercase. **Stage Two** of the conversion converted all the syntax tokens into their modern mixed case versions and details of these tokens are given in Appendix G.

The automatic changes are detailed in the table that follows. It is important to note that the changes were executed in the order implied by the identifier. This was because some earlier changes affect the requirements for those that were applied later.

In addition to the changes that are described here, there were other non-automated changes, which were performed using ICQF+'s **corpus editor**, and these are not described here.

Id	Description of change	Remarks
0	Find all instances of the items this, that, these and those which expound a deictic determiner (DD) in a nominal group (NGP) or an unfinished nominal group (NGPUN) where the element is the final element in its unit, and change the deictic determiner (DD) into a pronoun head (HP).	
1	Find all instances of the item which , that expounds a wh-deictic determiner (DDWH) in a nominal group (NGP) or an unfinished nominal group (NGPUN) where the element is the final element in its unit, and change the wh-deictic determiner (DDWH) into a wh-head (HWH).	
2	Find all instances of the items which and what where it expounds a wh-deictic determiner (DDWH) in a nominal group (NGP) or an unfinished nominal group (NGPUN), and it is the final element in its group. Change the wh-deictic determiner (DDWH) into a wh-head (HWH).	
3	Find all instances of the items here, now, then and when where it expounds an apex (AX) in a quantity-quality group (QQGP) or an unfinished quantity-quality group (QQGPUN), and it is the only element in its group. Change the apex (AX) into a pronoun head (HP) and the quantity-quality group (QQGP) into a nominal group (NGP), or the unfinished quantity-quality group (QQGPUN) into an unfinished nominal group (NGPUN).	
4	Find all instances of the items where , and when where it expounds a wh-apex (AXWH) in a quantity-quality group (QQGP) or an unfinished quantity-quality group (QQGPUN), and it is the only element in its group. Change the wh-apex (AXWH) into a wh-head (HWH) and the quantity-quality group (QQGP) into a nominal group (NGP), or the unfinished quantity-quality group (QQGPUN) into an unfinished nominal group (NGPUN).	
5	Find all instances of the items can, could, will, would, shall, should, might, must, and ought where it expounds a modal operator (OM) in a Clause (CL) or an unfinished Clause (CLUN) and change the modal operator (OM) into an ordinary operator (O).	Note that an error was discovered in the POW Corpus where the syntax token (OM) was being used to represent both (a) a modal operator, and (b) an operator conflated with a main verb. This error was corrected by this, and subsequent changes.

Id	Description of change	Remarks
6	Find all instances of the items cant, couldnt, wont, wouldnt, shouldnt, shant, maynt, mightnt, oughtnt, can't, couldn't, won't, wouldn't, shouldn't, shan't, mayn't, mightn't, and oughtn't where it expounds a modal operator (OM) in a Clause (CL) or an unfinished Clause (CLUN) and change the modal operator (OM) into an ordinary operator (O).	Similar to change 5, but with the 'nt forms of the words.
7	Find all instances of the items am, is, was, are, 's, and were , and were where it expounds the modal operator (OMO, OM) in a Clause (CL) or an unfinished Clause (CLUN) and the modal operator (OMO) is followed by the infinitive element (I). Change the modal operator (OMO, OM) into an operator (O).	Note that modal operators where labelled OM and OMO in the POW corpus. See also Change 5 and Change 8.
8	Find all instances of the items have, had, has, 'ave, and 'ad , and were where it expounds the modal operator (OMO, OM) in a Clause (CL) or an unfinished Clause (CLUN) and the modal operator (OMO) is followed by the infinitive element (I). Change the modal operator (OMO, OM) into an operator (O).	
9	Find all instances of the items a-few, any, a-bit, bit, a-little, a-lot, about, another, as, enough, hardly, how, nearly, plenty, quite, so, something-like, and that , where it expounds a temporer (T) in a quantity-quality group (QQGP), or an unfinished quantity-quality group (QQGPUN) which fill with a quantifying determiner (DQ) or a negative quantifying determiner (DQN). Change the temporer (T) into an adjuster (AD) and the quantity-quality group into a quantity group (QTGP), or the unfinished quantity-quality group (QQGPUN) into an unfinished quantity group (QTGPUN). If there is a apex (AX) as a sibling, change it to an amount (AM). If there is a finisher (FI) present as a sibling, change it to a quantity group finisher (QTF).	Although the quantity group (QTGP) is not a syntax token in the earlier version of the Cardiff Grammar used in the POW Corpus, I decided to keep the syntax tokens in uppercase letters until the global change into mixed case tokens was made later.

Id	Description of change	Remarks
10	Find all instances of the item just , where it expounds a temporer (T) in a quantity-quality group (QQGP), or an unfinished quantity-quality group (QQGPUN) which fill with a quantifying determiner (DQ) or a negative quantifying determiner (DQN). Change the temporer (T) into an inferer (INF) and the quantity-quality group into a quantity group (QTGP), or the unfinished quantity-quality group (QQGPUN) into an unfinished quantity group (QTGPUN). If there is a apex (AX) as a sibling, change it to an amount (AM). If there is a finisher (FI) present as a sibling, change it to a quantity group finisher (QTF).	Just was forgotten in Change 9.
11	Find all instances of the item more , where it expounds a scope (SC) in a quantity-quality group (QQGP), or an unfinished quantity-quality group (QQGPUN) which fill with a quantifying determiner (DQ) or a negative quantifying determiner (DQN). Change the scope (SC) into an amount (AM) and the quantity-quality group into a quantity group (QTGP), or the unfinished quantity-quality group (QQGPUN) into an unfinished quantity group (QTGPUN). If there is an apex (AX) present as a sibling to the left or right, change it to an adjuster (AD). If there is a temporer (T) present change it also to an adjuster (AD). If there is a finisher present (FI), change it to a quantity group finisher (QTF). Warn the user if more than one adjuster in the resulting quantity group.	
12	Find all instances of the item outside , where it expounds an apex (AX) in a quantity-quality group (QQGP), or an unfinished quantity-quality group (QQGPUN) which fill with a preposition (P) in a prepositional group (PGP), or an unfinished prepositional group (PGPUN). Change the quantity-quality group (QQGP) into a quantity group (QTGP), or the unfinished quantity-quality group (QQGPUN) into an unfinished quantity group (QTGPUN). If there is a temporer (T) present as a sibling, change it also to an adjuster (AD). If there is a finisher (FI) present as a sibling, change it to a quantity group finisher (QTF).	
13	Find all instances of the items a-bit , almost , more , too , that , down , further , as and about where it expounds a temporer (T) in a quantity-quality group (QQGP), or an unfinished quantity-quality group (QQGPUN) which fill with a temporer (T) in a higher quantity-quality group (QQGP), or an unfinished quantity-quality group (QQGPUN). Change the temporer (T) into an adjuster (AD). Change the quantity-quality group (QQGP) into a quantity group (QTGP), or the unfinished quantity-quality group (QQGPUN) into an unfinished quantity group (QTGPUN). If there is a sibling apex (AX), change it to an amount (AM). If there is a sibling finisher (FI), change it to a quantity group finisher (QTF). Do not change the higher quantity-quality group (QQGP) as this will be the subject of a later change.	Change 13 was implemented as two sub-changes for groups of the words given.

Id	Description of change	Remarks
14	Find all instances of the items much, more, a-lot, a-little, most, a-bit, and how-much, where it expounds an apex (AX) or a temporing apex (AXT) in a quantity-quality group (QQGP), or an unfinished quantity-quality group (QQGPUN) and where the unit fills either of these elements in the Clause: Adjunct (A), Replacement Adjunct (AREPL), Metalingual Adjunct (AM), and wh-Adjunct (AWH). Change the apex (AX) into an amount (AM). Change the quantity-quality group (QQGP) into a quantity group (QTGP), or the unfinished quantity-quality group (QQGPUN) into an unfinished quantity group (QTGPUN). If there is a sibling temporer (T), change it to an adjuster (AD). If there is a sibling scope (SC) element, change it to a quantity group finisher (QTF).	Change 14 was implemented as two changes: one for the AX and the other for the AXT .
15	Find all instances of the elements modifier (MO), affective modifier (MOA), relative modifier (MOC), thing modifier (MOTH), ordinative determiner (DO), or superlative determiner (DS) that are filled by one or more quantity-quality groups (QQGP) or an unfinished quantity-quality group (QQGPUN). Change any child deictic determiner (DD) into a quality group determiner (QLD). Change the quantity-quality group to a quality group (QLD).	
16	Find all instances of the elements Complement (C), anticipated Complement (CANTIC), wh-Complement (CWH), or replacement Complement (CREPL) that are filled by one or more quantity-quality groups (QQGP) or an unfinished quantity-quality group (QQGPUN). Change the quantity-quality group into a quality group (QLGP), or the unfinished quantity-quality group into an unfinished quality group (QLGPUN). Change any child deictic determiner (DD) into a quality group determiner (QLD). Change the quantity-quality group to a quality group (QLD).	
17	Find all instances of the items again, never, normally, anyway, better, different, easy, fast, first, second, hard, later, neat, preferably, right, slowly, soon, together, wrong, yellow, full, any, and good, which expound an apex in a quantity-quality group (QQGP), or an unfinished quantity-quality group (QQGPUN). Change the quantity-quality group into a quality group (QLGP), or the unfinished quantity-quality group into an unfinished quality group (QLGPUN).	The list of words was compiled after querying in ICQF+. The list shown here was extended to cover more words, and the change was run several times.

Id	Description of change	Remarks
18	Find all quantity-quality groups (QQGP), or unfinished quantity-quality groups (QQGPUN) that have a single child element and fill a Main Verb Extension (CM). Make the item that expounds the single child element of the unit directly expound the Main Verb Extension (CM) and discard the quantity-quality group (QQGP), or unfinished quantity-quality group (QQGPUN).	
19	Find all quantity-quality groups (QQGP), or unfinished quantity-quality groups (QQGPUN) that have more than one child element and fill a Main Verb Extension (CM) and the quantity-quality group does not contain a scope (SC) element. Change the quantity-quality group (QQGP) into a quantity group (QTGP), or the unfinished quantity-quality group (QQGPUN) into an unfinished quantity group (QTGPUN). Change any apex (AX) in the group into an amount (AM). Change any temporer (T), into an adjuster (AD).	
20	Find all temporerers (T) that are in prepositional groups (PGP), or unfinished prepositional groups (PGPUN) and change them to prepositional temporerers (PT).	
21	Find all instances of the items by-here , by-ere , b-there , by-there , somewhere , everywhere , nowhere , there , and where , where it expounds an apex (AX). Change the apex (AX) into a pronoun head (HP). Change the parent unit (QQGP or QQGPUN) into a nominal group (NGP or NGPUN). Change any sibling temporerers (T) into modifiers (MO). Change any sibling scope elements (SC) into qualifiers (Q).	Note that by-here , and by-there are common in the Welsh dialect.
22	Find all instances of temporer (T) in a nominal group (NGP) and change them to modifiers (MO).	At this point, some QQGP that had been changed to NGP were discovered to contain temporerers which hadn't been changed.
23	Find all instances of the items more and most where it expounds a tempering apex (AXT) in a quantity-quality group (QQGP), or an unfinished quantity-quality group (QQGPUN). Change the quantity-quality group (QQGP) into a quantity-group (QTGP), or the unfinished quantity-quality group (QQGPUN) into an unfinished quantity group (QTGPUN). Change the tempering apex (AXT) into an amount (AM). Change any sibling apex (AX) into an adjuster (AD) and change any sibling scope (SC) into a quantity group finisher (QTF).	

Id	Description of change	Remarks
24	Find all instances of the items still , yet and anymore where it expounds an apex (AX) in a quantity-quality group (QQGP), or an unfinished quantity-quality group (QQGPUN). Change the quantity-quality group (QQGP) into a quantity-group (QTGP), or the unfinished quantity-quality group (QQGPUN) into an unfinished quantity group (QTGPUN). Change the (AX) into an amount (AM). Change any sibling temperer (T) into an adjuster (AD) and change any sibling scope (SC) into a quantity group finisher (QTF).	
25	Find all instances of the item gonna where it expounds an auxiliary (X). Change the item gonna into going and the auxiliary (X) into and auxiliary extension (XEX). Add a new item to as the immediate right sibling of the target item, which expounds an infinitive element (I).	
26	Find all instances of the item gotta where it expounds an auxiliary (X). Change the item gotta into got and the auxiliary (X) into and auxiliary extension (XEX). Add a new item to as the immediate right sibling of the target item, which expounds an infinitive element (I).	
27	Find all instances of the item going-to and goin-to where it expounds an auxiliary (X). Change the item going-to or goin-to into got and the auxiliary (X) into and auxiliary extension (XEX). Add a new item to as the immediate right sibling of the target item, which expounds an infinitive element (I).	
29	Find all instances of a Subject (S) followed by an auxiliary extension (XEX) where there is not also an operator conflated with an auxiliary (OX) in the same clause. Find also examples where the subject and or the auxiliary extension are ellipted. Add an ellipted operator conflated with an auxiliary (OX) between the Subject (S) and the auxiliary extension (XEX).	Change 28 was abandoned.
30	Find all instances of the item had that is followed by the item better . Change the item had so that it expounds an operator conflated with and auxiliary (OX) and the item better so that it expounds an auxiliary extension (XEX).	
31	Find all instances of the item let 's that expound a Subject (S). Add a new element to the left of the Subject (S) for the let element (L), directly expound it by the item let . Change the Subject (S) so that it fills a nominal group (NGP) which has a pronoun head expounded by 's .	The element L is not discussed in Chapter Four.

Id	Description of change	Remarks
32	Find all occurrences of the items a - few, all, any, enough, more, most, much, less, one, two, three, four, five, six, seven, eight, nine, ten, and eleven which expound an apex (AX) as the only element in a quantity-quality group (QQGP), or an unfinished quantity-quality group (QQGPUN) which fills a quantifying determiner (DQ) or a negative quantifying determiner (DQN). Remove the quantity-quality group (QQGP), or the unfinished quantity-quality group (QQGPUN) and let the item directly expound the quantifying determiner (DQ), or the negative quantifying determiner (DQN).	
33	Find all occurrences of the items a - few, all, any, enough, more, most, much, less, one, two, three, four, five, six, seven, eight, nine, ten, and eleven which expound an apex (AX) which is not the only element in a quantity-quality group (QQGP), or an unfinished quantity-quality group (QQGPUN) which fills a quantifying determiner (DQ) or a negative quantifying determiner (DQN). Change the apex (AX) to an amount (AM). Change any sibling temperer (T) to an adjuster (AD). Change any sibling scope (SC) to a quantity group finisher (QTF). Change the quantity-quality group (QQGP) to a quantity group (QTGP), or the unfinished quantity-quality group (QQGPUN) to an unfinished quantity group (QTGPUN).	
34	Find all instances of the items 'er, 'ere, here, there, now, then, how, why, anyhow, somehow, anywhere, somewhere, everywhere, and nowhere where it expounds an apex (AX) in a quantity-quality group (QQGP), or an unfinished quantity-quality group (QQGPUN). Change the quantity-quality group (QQGP) to a nominal group (NGP), or the unfinished quantity-quality group (QQGPUN) to an unfinished nominal group (NGPUN). Change the apex (AX) to a pronoun head (HP). Change any sibling temperer (T) to a modifier (MO), and change any sibling scope (SC) to a qualifier (Q).	
35	Find all instances of the items if, once, twice, afterwards, after, beforehand, before, forwards, backwards, upwards, downwards, indoors, outdoors, inside, and outside where it expounds an apex (AX) in a quantity-quality group (QQGP), or an unfinished quantity-quality group (QQGPUN). Change the quantity-quality group (QQGP) to a nominal group (NGP), or the unfinished quantity-quality group (QQGPUN) to an unfinished nominal group (NGPUN). Change the apex (AX) to a pronoun head (HP). Change any sibling temperer (T) to a modifier (MO), and change any sibling scope (SC) to a qualifier (Q).	

Id	Description of change	Remarks
36	Find all instances of the item gorra where it expounds an auxiliary (X). Change the item gorra into got and the auxiliary (X) into an auxiliary extension (XEX). Add a new item to as the immediate right sibling of the target item, which expounds an infinitive element (I).	
37	Find all instances of the item some where it expounds an apex (AX) in a quantity-quality group (QQGP), or an unfinished quantity-quality group (QQGPUN) that fills a quantifying determiner (DQ), or a negative quantifying determiner (DQN). Change the quantity-quality group (QQGP) to a nominal group (NGP), or the unfinished quantity-quality group (QQGPUN) to an unfinished nominal group (NGPUN). Change the apex (AX) to a pronoun head (HP). Change any sibling temperer (T) to a modifier (MO), and change any sibling scope (SC) to a qualifier (Q).	
38	Find all instances of the items after , afterwards , inside , indoors , in , out , down , before , on-top , in-front , beneath , below , outside , outdoors , underneath , upwards , downwards , up , and upwards , where it expounds a head (H) in a nominal group (NGP), or an unfinished nominal group (NGPUN). Change the head (H) into an amount (AM). Change the nominal group (NGP) to a quantity group (QTGP), or the unfinished nominal group (NGPUN) to an unfinished quantity group (QTGPUN). Change any sibling modifiers (MO*) into an adjuster (AD). Change any qualifiers (Q) to a quantity group finisher (QTF).	
39	Find all instances of the items how and why where they expound a wh-apex (AXWH) in a quantity-quality group (QQGP), or an unfinished quantity-quality group (QQGPUN), or a quantity group (QTGP), or an unfinished quantity group (QTGPUN). Change the unit (QQGP , QQGPUN , QTGP , QTGPUN) to a nominal group (NGP), or an unfinished nominal group (NGPUN). Change the wh-apex (AXWH) to a wh-head (HWH). Change any sibling temperer (T) to a modifier (MO). Change any scope (SC) to a qualifier (Q).	
44	Find any item an ' and a linker (Lnk) and change to and	Following a query in ICQF to confirm they were all and . Changes 40 thru 43 abandoned.

Table F.1: Changes made during Stage One of the corpus modification

Appendix G

Modifying the corpus - a table of syntax token mappings used in Stage Two

The process that was used to create the Fawcett-Perkins-Day (FPD) Corpus from the Polytechnic of Wales (POW) Corpus is given in Chapter Nine. While reading this appendix, it is important to realise that the modification was not simply a matter of changing syntax tokens and it represented many man-months of work. It involved two stages, and it is the data for the latter stage that is given here; details of the changes made for Stage One are given in Appendix F.

OLDLABEL	NEWLABEL	CONTEXT
&N	Lnk	
AA	A_Aff	
AD	A_DO	
AF	A_AS	
AI	A_Inf	
AL	A_Log	
ALREPL	A_Log_Repl	
ALWH	A_L_Wh	
AM	A_Val	CL
AM	A_Val	CLUN
AN	A	
AP	A_Pol	
AREPL	A_Repl	
ATG	A_CSTag	
AWH	A_Wh	
AX	ax	
AXT	ax	
AXWH	ax_wh	
BM	B	
BN	B	
CANTIC	A_Res	
CL	Cl	
CM	MEx	
CP	Xex	
CREPL	C_Repl	
CV	cv	
CVREPL	cv_repl	
CVWH	cv_wh	
CWH	C_Wh	
DD	dd	
DDWH	dd_wh	

OLDLABEL	NEWLABEL	CONTEXT
DO	od	
DP	pd	
DQ	qd	
DQN	qd	
DQWH	qd_wh	
DR	rd	
DS	sd	
DT	td	
ET	et	
EX	Excl	
FI	fi	
G	g	
GC	genclr	
H	h	
HN	h_n	
HP	h_p	
HPN	h_p	
HSIT	h_sit	
HWH	h_wh	
INF	inf	
LNK	Lnk	
MO	mo	
MOA	aff_mo	
MOC	rel_mo	
MOQ	q_mo	
MOSIT	sit_mo	
MOTH	th_mo	
NGP	ngp	
OMN	O	
ON	O	
OWN	own	
P	p	
PGP	pgp	
PM	p	
PS	po	
PT	pt	
Q	q	
QLGP	qlgp	
QREPL	q_repl	
QTF	qtf	
QTGP	qtgp	
SANTIC	A_Res	
SC	sc	
SIT	S_it	
SREPL	S_Repl	
STH	S_th	
SWH	S_Wh	
T	dt	QLGP
TD	td	
TWH	dt_wh	
TWH	dt_wh	

OLDLABEL	NEWLABEL	CONTEXT
V	Voc	
VO	v	NGP
XEX	Xex	
XM	X	
XMN	X	
XMO	X	
XMON	X	

Table G.1 - The data used in Stage Two for creating the FPD Corpus

Appendix H

The probabilities tables, parser working tables and data structures

This appendix describes:

- (a) the **queries** and the **probabilities tables** (see Section H.1),
- (b) the **data structures** used by the parser that are stored in the **parser working tables** (see Section H.2), and
- (c) the **functions** that operate on those data structures (see Section H.3).

H.1 The probabilities tables and their queries

This section describes **Version One** the probabilities tables. It provides details of the different types of table and the queries that are performed upon them.

H.1.1 Types of table and their queries

As we saw in Chapter Fourteen that there are two groups of probability queries: **vertical queries** and **horizontal queries**. **Vertical queries** have this name because they ask about the relationship upward between an item, element or unit and the syntactic category (or categories) above it in the tree. This relationship may reach up to the 'root' of the tree (i.e. the Sentence node(**Z**)). The vertical queries return **vertical strips**, or parts of vertical strips, as we saw in Section 5.2.4 of Chapter Five and Section 11.5 of Chapter Eleven, this concept first used by O'Donoghue (1991b)¹. The vertical strips used by the parser can be sub-classified as:

- (a) **complete-with-item**,
- (b) **complete-without-item**,
- (c) **partial-with-item**,
- (d) **partial-without-item**.

Figure H.1 shows a diagram of the syntactic analysis of a sentence that illustrates the **complete-with-item** type of vertical strip. Such vertical strips include the item, the elements and units in the order in which they occur, starting from the item and going up to the sentence node. Thus, the first vertical strip in Figure H.1, consists of **"The" dd ngp S C1 Z**. **Complete-without-item** vertical strips are the same but exclude the item (eg **dd ngp S C1 Z**). Partial vertical strips are simply parts of a

¹ O'Donoghue's vertical strips are of the type **complete-without-item**.

vertical strip that occur anywhere in a complete-with-item vertical strip, e.g. "The" dd ngp, "The" dd, ngp S, S Cl etc.

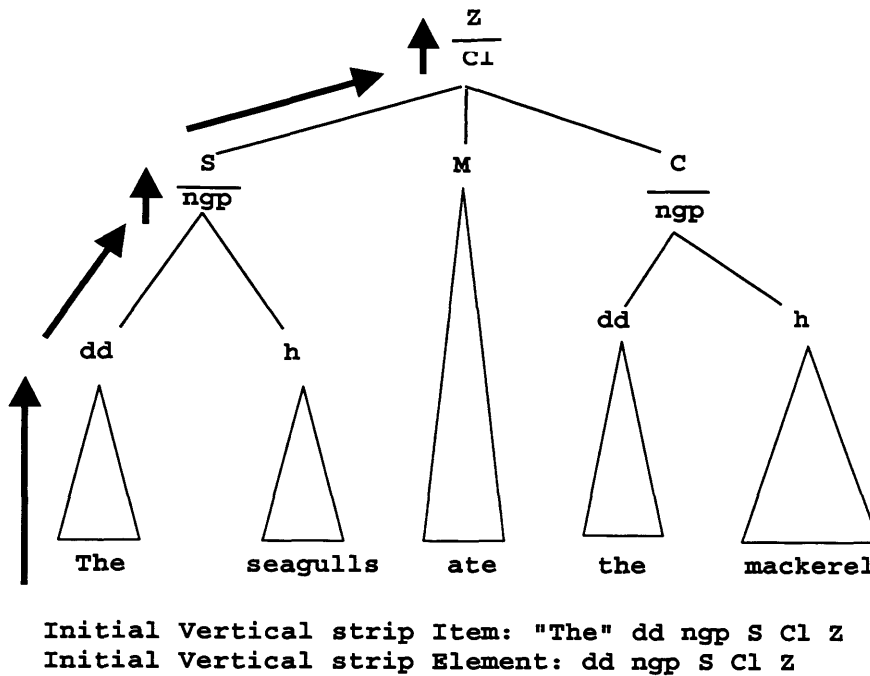


Figure H.1 - A sample sentence and a set of initial vertical strips extracted from it

The types of vertical queries used by the parser are:

- (a) Initial vertical strip based on Item (**IVS - ITEM**), see Section H.1.2,
- (b) Initial vertical strip based on element of structure (**IVS - ELEM**), see Section H.1.2,
- (c) Item-up-to-Element (**I2E**), see Section H.1.3,
- (d) Item-up-to-Element-up-to-Unit (**I2E2U2E**), see Section H.1.3,
- (e) Unit-up-to-Element (**E2U**), see Section H.1.4,
- (f) Element-up-to-Unit (**U2E**), see Section H.1.5.

Horizontal queries return **elements**, or groups of elements together with as associated probability. They operate using **horizontal slices**, or partial **horizontal slices** at any level of the tree which represents a **unit**. There are two horizontal queries called **Unit Structure queries**:

- (a) Forward Unit Structure (**FUS**), see Section H.1.6.1,
- (b) Backward Unit Structure (**BUS**), see Section H.1.6.2.

H.1.2 Queries to the initial vertical strip tables

H.1.2.1 Rationale in creating the tables

The first item in the input string requires a different sort of query from all the other items, because there is no preceding analyses that make forward predictions that could assist the parser in its decisions. In the early implementations of the parser, the algorithm analysed the first item in the same way as all the others, i.e. by performing an **I2E** query followed by a sequence of **E2U**, and **U2E** queries until it reached the sentence node. We discovered, however that this method led to the creation of too many vertical strips, many of which could not occur in this position in the sentence.

I therefore developed new **Query Restriction** features in ICQF+ (see Section 8.3.4 of Chapter Eight), which enabled information to be extracted about items, elements and units when they occur in the sentence initial position. These queries confirmed that the items, elements and units in the sentence initial vertical strip have significantly different statistical properties from the others.

In fact, I automatically created two sets of initial vertical strips: one that was **complete-with-item**, and another that was **complete-without-item** for use in the Initial Vertical Strip (**IVS**) probabilities tables. The problem that led to this was that the FPD Corpus contains approximately 11,000 sentences, and so yielding 11,000 initial vertical strips, and this is too small a database from which to create probabilities tables that would be representative of, and so applicable to, unrestricted language. Substantial time and effort was expended by both researchers in an attempt to ensure that the **IVS** probabilities tables have a far wider coverage than that which could be derived from the FPD corpus. And the main means by which this was accomplished was to amend the **IVS** tables based on the data extracted from the vast tables of lexical probabilities that were derived from the British National Corpus (BNC).

Two types of **IVS** probabilities tables are used. These are the **initial vertical strip element (IVS-ELEM)** and the **initial vertical strip item (IVS-ITEM)** tables, and it is to the distinction between these that we next turn.

We have therefore also introduced the **initial vertical strip item (IVS-ITEM)** probabilities table, which contains a set of **complete-with-item** vertical strips that occur in the sentence initial position.

The **initial vertical strip element (IVS-ELEM)** probabilities table contains a set of **complete-without-item** vertical strips that occur in the sentence initial position, and this is the table that is used for most items.

However, there is a small set of about twenty items, each of which can expound two or more elements, and each of which has significantly different probabilities of expounding one or other of those elements when the item occurs initially. The item **that**, for example, is far more likely to be a **pronoun head (h_p)** or a **deictic determiner (dd)** in a nominal group (**ngp**) than a **Binder (B)** in a Clause (**C1**) when it is sentence initial. The probabilities for these items (which include many of the personal pronouns and the more frequent co-ordinating conjunctions), are contained in the **IVS-ITEM** table.

H.1.2.2 The structure of the tables

The structure of the **IVS-ITEM** table is given in Appendix D.

Table H.1 shows an excerpt from the results of an **IVS-ITEM** query for the item **that**. It shows a 32.28% chance of being a pronoun head (**h**) in a nominal group (**ngp**) that fills the Subject (**S**) in the primary clause (**C1**).

ITEM	VERTSTRIP	PROBABILITY
that	h_p ngp S C1 Z	0.3228
that	dd ngp S C1 Z	0.3199
that	h_p ngp C C1 Z	0.2796
that	dd ngp C C1 Z	0.2771
that	h_p ngp cv ppp C C1 Z	0.0547

Table H.1 - An excerpt from the IVS-ITEM query for the item "that"

Table H.2 shows an **IVS-ELEM** query for a head (**h**). It indicates that there is a 32.28% chance of the pronoun head being in a nominal group (**ngp**) that fills the Subject (**S**) in the primary clause (**C1**) and 27.96% chance of being in a nominal group that fills a Complement (**C**) in the primary clause.

ELEMENT	VERTSTRIP	PROBABILITY
h	h ngp S C1 Z	0.1400
h	h ngp C C1 Z	0.0300
h	h ngp A C1 Z	0.0247
h	h ngp S C1 C C1 Z	0.0059
h	h ngp C C1 C C1 Z	0.0040

Table H.2 - An excerpt from the IVS-ELEM query for the element-of-structure "h"²

H.1.2.3 Criteria for including items in the Initial Vertical Strip Item table

Many items in English may expound two or more different elements of structure. Many of these are already given special treatment by the parser, by including them in the Item-up-to-element-up-to-unit (**I2E2U2E**) table (see Section H.1.3). However, some of these items can be seen to be operating differently when they occur in the sentence initial position, in that their potential structures and their associated probabilities are different.

While most personal pronouns (such as **I** and **me**) are included in the **I2E2U2E** table, there are other frequent items whose probabilities are significantly different when they are sentence initial, namely: **you, mine, yours, his, hers, ours, theirs, that, these, those, it, there, then, and, or, nor, neither** and **so**.

The reason why the Linkers (**Lnk**), **or, nor**, and, **neither** are included is that, while they can occur in any class of unit when not sentence-initial, they are extremely likely to be a linker in a Clause when they are sentence-initial.

That concludes the description of the tables used for the initial item in a sentence. The rest of the tables to be described here are the tables that are used when the parser is working out the structure above for all the subsequent items in a sentence.

H.1.3 Item-up-to-element (**I2E**) and item-up-to-element-up-to-Unit-up-to-Element (**I2E2U2E**) tables

An **I2E** query operates on the **I2E** probabilities table, and it obtains a list of elements that a given item can expound together with their associated probabilities. However, certain items behave quite differently from others when we take into account the element of structure that occurs above the unit above the expounded element. For these items, there is a Boolean value to indicate that the parser must ask for a further item-up-to-element-up-to-unit-up-to-element query (**I2E2U2E**).

² The query returns 147 rows.

Let us look first at the **I2E** probabilities table, since it is the simpler of the two. The structure of the table is given in Appendix D. There are two important fields in this table that need to be discussed in more detail here. These are the fields named **I2E2U2ENEEEDED** and **IVSITEMNEEEDED** which are Boolean fields that are used by the parser. The first indicates that the parser should ask for a query to the **I2E2U2E** table for this item when it expounds the given element. The second indicates that the parser should ask for an **IVS_ITEM** query for the given item when it is parsing the first item of a sentence.

Table H.3 shows an **I2E** query for the item **the**. Note that, when **the** expounds a quality group determiner (**qld**), a quantity group determiner (**qtd**) or a linker (**Lnk**), they are worthy of further analysis which will be provided by an **I2E2U2E** query. There is a 97.14% chance that **the** expounds a deictic determiner (**dd**), a 2.24% chance that it is a quality group determiner (**qld**), a 0.59% chance that it is a quantity group determiner (**qtd**), and a 0.01% chance that it is a linker (**Lnk**)³.

ITEM	ELEM1	PROBI2E1	I2E2U2ENEEEDED
the	dd	0.9714	False
the	qld	0.0224	True
the	qtd	0.0059	True
the	Lnk	0.0001	True

Table H.3 - Entries in the I2E table for the item the

There are certain items which predict not only the element and unit above them, but also the element above that. For example, an item such as **happy** that expounds the apex (**ax**) of a quality group (**qlgp**), strongly predicts the range of elements that the quality group may be fulfilling (e.g. a modifier (**mo**) or a Complement (**C**), but not an Adjunct (**A**)). This knowledge is provided by the **I2E2U2E** table. This is one of the particular areas in which the CCPP has additional linguistic knowledge, so giving it an advantage over other approaches.

The fields of the **I2E2U2E** table are: **ITEM**, and **ELEM1** (which are the same fields found in the **I2E** table and are used as a matching set of fields in the **I2E2U2E** table), and then **UNIT** (the unit that **ELEM1** is a component of), **ELEM2** (which is

³ Some readers may be surprised to find that, in this grammar, **the** in **the biggest** is not a simple deictic determiner (**dd**), for reasons explained in Fawcett (2007a). The use of **the** as a linker that occurs in sentences such as **the faster we work, the sooner we'll finish**.

the element that the UNIT fills), and the probability of this I2E2U2E record when compared against other matching I2E2U2E records is given in **PROBI2E2U2E**.

Table H.4 shows the **I2E2U2E** values for **the** as a quality group determiner (**qld**); it shows that it can act as a superlative determiner (**sd**) (85% chance), a Complement (**C**) (14.5% chance), and an ordinative determiner (**od**) (0.5% chance).

ITEM	ELEM1	UNIT	ELEM2	PROB I2E2U2E
the	qld	qlgp	sd	0.8500
the	qld	qlgp	C	0.1450
the	qld	qlgp	od	0.0050

Table H.4 - Entries in the I2E2U2E table for the item the as a qld

We come now to the last two tables, each of which is small in comparison, with most of those considered so far, since they do not involve items.

H.1.4 Queries to the element-up-to-unit (E2U) table

This table provides the probability that an element of structure is a component of a given unit. Apart from the two elements of the Linker (**Lnk**) and the Inferer (**inf**), no elements can only belong to more than one unit⁴. The fields of the **E2U** table are given in Appendix D.

In most cases the element can only belong to one class of unit and its **PROBABILITY** score will be 1 (100%). However, the element linker (**Lnk**), which may belong to a number of classes of unit, and is therefore more interesting, is shown in Table H.5. Here we can see that a linker is most likely to occur in a Clause (**C1**) (80.00%) and that the second most likely unit that it occurs in is a nominal group (**ngp**) (18.00%).

ELEM	UNIT	UNITPROB
Lnk	C1	0.8000
Lnk	ngp	0.1800
Lnk	qlgp	0.0019
Lnk	qtgp	0.0001

Table H.5 - Entries in the E2U table for the element Lnk

Finally, we turn to the rather more complex Unit-up-to-Element (**U2E**) table.

⁴ In the FPD Corpus, however, we distinguish unfinished units from units that have been completed by using the token for the unit followed by an underscore and a "un" (eg **C1_un**).

H.1.5 Queries to the unit-up-to-element (U2E) table

This table gives the probability that a given unit fills one of one or more elements of structure. So, like the **I2E** table, this table extends the vertical strip upwards by one layer of structure represented by an element.

The structure of the **U2E** table is given in Appendix D. Table H.6 shows gives a portion of the results of a **U2E** query for a nominal group (**ngp**). It shows that the elements that a nominal group is most likely to fill a Subject (**S**) (40%) and a Complement (**C**) (33%).

UNIT	ELEM	ELEMPROB
ngp	S	0.40188
ngp	C	0.33950
ngp	cv	0.12716
ngp	A	0.03067
ngp	Voc	0.01697
ngp	C_Wh	0.01633
ngp	th_mo	0.01488

Table H.6 - A portion of the results of a Unit-up-to-Element query for a nominal group

H.1.6 An overview of the unit structure (US) tables and their queries

They are of two types: **forward** and **backward**.

In contrast with the tables and their queries that have been described so far, unit structure queries ask about the horizontal relationships in the parse tree. Forward Unit Structure (**FUS**) queries elicit the probability of the next element in a unit that has a given string of elements before it. Backward Unit Structure (**BUS**) queries elicit the probability of the previous element in a unit for any given element.

Note that, in unit structure queries, the start and the end of the unit are treated, for computational purposes, as if they were elements in the unit, each being represented by an exclamation mark (!).

H.1.6.1 Queries to the forward unit structure tables (FUS)

The structure of the **FUS** table is given in Appendix D.

Table H.7 shows a portion of the **FUS** table for a Clause (**C1**) for which the parser has already found a Subject (**S**) and an Operator (**O**). The table shows that there is a 76% probability that the next element will be a Main verb (**M**) (as in **I can't see...**), a 5% probability that it will be an auxiliary (**X**) (as in **He may have...**), and

several other lower probability elements. There is also an 11% probability that the Operator (O) will be the last element in the unit⁵.

UNIT	ELEMENTS	NEXTELEM	PROBABILITY
Cl	! S O	M	0.7643
Cl	! S O	!	0.1134
Cl	! S O	X	0.0539
Cl	! S O	A_Log	0.0121
Cl	! S O	A_Inf	0.0087
Cl	! S O	A	0.0066
Cl	! S O	I	0.0055

Table H.7 - Forward unit structure query for the next element in a Clause (Cl) that has found a Subject (S) and an Operator (O)

H.1.6.2 Queries to the backward unit structure table (BUS)

The structure of the BUS table is given in Appendix D.

Table H.8 shows a Backward Unit Structure Query for a Clause (Cl). It tells us which elements of structure may appear before a Main Verb (M), and their probabilities. It shows that there is a 35% chance of the preceding element being a Subject (S) (as in I ate it), a 13% chance of it being an Operator (O) (as in I might eat), an 11% chance of it being an Infinitive element (I) (as in I am going to eat sausages)⁶. There is also a 17% chance of the Main Verb being the first element in the clause (as in eat it).

UNIT	ELEM	PREVELEM	PROBABILITY
Cl	M	S	0.3597
Cl	M	!	0.1774
Cl	M	O	0.1373
Cl	M	I	0.1122
Cl	M	OX	0.0861
Cl	M	OM	0.0336
Cl	M	Lnk	0.0251
Cl	M	A_Inf	0.0204
Cl	M	X	0.0146
Cl	M	A	0.0100
Cl	M	N	0.0067
Cl	M	S_wh	0.0055

Table H.8 - Entries in the backward unit structure table for elements that can precede a Main Verb (M) in a Clause (Cl)

⁵ This figure is untypically high because the FPD Corpus, which was used as a source of the probabilities table, contains spoken texts which has unfinished units and ellipted elements. There are many unfinished Clauses, and many Clauses with ellipted elements.

⁶ Here, the item going is an auxiliary extension (XEx).

This now concludes the description of the probabilities tables and the types of query that are used to extract data from them. Next, we turn to a very different set of tables, and algorithms that are used to populate them. These are the tables that store the parser's working data including the data structures.

H.2 The parser data structures

The way in which the parser is **database-oriented** is that its working data are stored in database tables. This section describes (a) those tables, and (b) the functions that operate on them.

H.2.1 Trees and Nodes

We saw in Chapter Fifteen that the basic data structure in the parser is a **database tree** and that there are two types of tree:

- (a) ones that represent what the parser has built so far (called **built-structures**),
- (b) ones that represent what the parser is attempting to attach to the built structures (called the **candidate structures**)⁷.

Both types of tree are stored in the **DB_PARSE_TREE** table which has a structure as defined in Appendix D.

The field named **TREEID** contains a unique identifier of the tree and is its primary key. The field named **TREELABEL** contains either the value **BUILTSTRUCTURE** or the value **CANDIDATESTRUCTURE** and indicates the type of the tree. The field named **PROBABILITY** contains the associated score for the tree and is used to rank it. Chapter Fifteen describes how this value is calculated. The field named **COMPLETE** contains a Boolean value, and if set to **true**, this indicates that this tree represents the parse of the complete sentence. The field named **ACTIVE** contains a Boolean value, and if it is set to **true**, it means that this tree will be involved in the next 'move' that the parser makes. The **STARTPOS** and **ENDPOS** fields indicate the 'starting position' and 'end position', and so the span of the tree.

A tree may have a collection of **nodes**. The nodes of a tree are stored in the **DB_PARSE_NODE** table, which has the structure defined in Appendix D.

The field named **TREEID** identifies (hence 'ID') the tree of which the node is a part. The field named **NODEID** is a unique identifier of the **node** within the tree. The

⁷ Technically, the candidate structure is a vertical strip as it does not contain any branching. It is convenient to store both forms in the same data structure.

parent of the node is indicated in the field named **PARENTID**, where a value of -1 indicates that the node is the root node. The node to the left of the given node is indicated in the field named **LEFTSIBID**, where a value of -1 indicates that the node is the first node within its parent. The node to the right of the node is indicated in a field named **RIGHTSIBID**, where a value of -1 means that the node is (currently) the right-most node within its parent. The field named **TOKEN** stores the syntax token, which can be a item, an element or a unit, where the field named **TYPE** indicates which. The field named **LEVEL_PROBABILITY** gives the probability of this node, and this is based on its parse history (see Section 15.1.7 of Chapter Fifteen). The field named **NODE_PROBABILITY** stores the probability of this node, and this is the result of a calculation that has involved an **I2E**, **I2E2U2E**, **E2U**, **U2E** or **unit structure** query. The Boolean field **RMS** (right-most strip), if true, indicates that this node is in the tree's right-most strip. Similarly, the Boolean field **LMS** (left-most strip), if true indicates that this node is in the tree's left-most strip.

H.2.2 Forward and backward predictions

Each unit node in the built structure tree has predictions about the element of structure that may follow the last element in the unit. This information, which will be used in an attempt to join a candidate structure to the built structure, is stored in the **FORWARDPREDICTIONS** table. The structure of this table is given in Appendix D.

The fields named **BSTREEID** (identifying the built structure tree) and **BSNODEID** (identifying a node in the built structure tree) together identify the tree and the node for which the prediction applies. This node always represents a unit. The field named **ELEM** contains the syntax token of the next element predicted (or ! for the end of the unit). The field named **PROBABILITY** contains the probability that the element follows the structure of the unit for which the node.

A similar table exists for the backward predictions from each unit node in the candidate structures. The **BACKWARDPREDICTIONS** table contains the following fields as defined in Appendix D.

The fields named **CSTREEID** (identifying a candidate structure tree) and **CSNODEID** (identifying a node in the candidate structure tree), together represent the unit node for which the **backward prediction** applies. The field called **ELEM** gives the element that can occur before the first element in the given unit. The field called

PROBABILITY contains the probability that the given element comes before the first element in the given unit.

H.2.3 The initial vertical strip table

The Initial Vertical Strip (**IVS**) table is used to store initial vertical strips which may be used in the creation of built structures that represent the first item in the sentence. It acts in the same way as the Parser State Table (**PST**), which will be described in the next section. It contains the fields defined in Appendix D.

The field named **ITEM** contains the first item found in the sentence, and the field named **VERTSTRIP** contains a string of **element+unit** pairs up to the sentence element, and together form a legal vertical strip that can occur above the item. The field named **PROBABILITY** contains the probabilities score for this strip when it occurs in the sentence initial position. The field named **FOLLOWED** contains a Boolean value that is set to true when the parser has used this vertical strip. The table is used in Stage One of the parsing algorithm, and the strips are taken in order of probability in groups of **n**, where **n** is the value in **n-best** (see Chapter Fifteen). Stage One is a backtrackable state, and when the parser backtracks, the next **n** vertical strips are converted into built structures, and will then have their **FOLLOWED** fields set to the value True.

H.2.4 The parser state table

The parser state table is used to record the moves that the parser has made and the trees that were involved in the move. This table is closely associated with the parsing algorithm and was therefore described in Section 15.1.5 of Chapter Fifteen.

H.3 Tree functions

The following operations are available to the parser to manipulate tree data structures.

Operation	Arguments	Value returned	Description
newTree	sTreeType	Id of the new tree	Creates a new active database tree where tree type is "built structure" or "candidate structure".
copyTree	iTreeId	Id of the new tree	Creates a new tree containing copies of the nodes in the source tree; makes the new tree active.
createTreeFromXML	sXML	Id of the new tree	Creates a tree from the XML in the input string. If XML is invalid, returns -1.
createTreeFromStrip	sStrip		Creates a new tree from a string containing syntax tokens (probabilities set to 1).
getTreeXML	iTreeId, iNodeId	sXML	Creates an XML string containing the structure below and including the nodeId.
getLeftToks	iTreeId, iNodeId	string containing a list of syntax tokens	Gets a list of all tokens in the nodes to the left of the given node (if any).
getRightToks	iTreeId, iNodeId	string containing a list of syntax tokens	Gets a list of all tokens in the nodes to the right of the given node (if any).
getNodeTok	iTreeId, iNodeId	string	Gets the token in the given node.
getLvlProb, getNodeProb addNode	iTreeId, iNodeId iTreeId, iParentId, sToken, sType, rNodeProb, rLvlProb bRMS	rProb True or false	Gets the value of the level or node probability in the given node. Creates a new tree node and adds it as a child of the node indicated in iParentId.
deactivateTree treeExists	iTreeId iTreeId	True or False	Makes the tree indicated by iTreeId inactive. Returns true if a tree exists with the given Id.
emptyParseTree	iTreeId	True or False	Returns true if iTreeId contains no nodes or false if it contains nodes.
deleteBranch	iTreeId, iBranchParentId	True or false	Deletes the node indicated by iBranchParentId and any children and descendants.
joinTree	iTree1Id, iTree2Id, iTree1ParentId bDestroy	True or false	Joins tree 2 to tree 1 at node tree1 parentId. If bDestroy is true, tree1 is set to inactive.
getTreeRoot	iTreeId	Id of the node that is the root node of the tree	Returns the Id of the root node.
activateTree, deactivateTree	iTreeId	True or false	Makes given tree active or inactive.

Operation	Arguments	Value returned	Description
setRMS	iTreeId	True or false	Adjusts the RMS of the given tree after a join operation.
editNode	sToken, sType, rNodeProb rLvlProb	True or false	Changes the values held in the tree node to the new values supplied.

Figure H.9: Table of the tree functions

H.4 Summary

This appendix has provided in Section H.1 full details of the **probabilities tables**. Therefore, we have seen the types of query that is used by the parser, and the types of data that is returned. This then, forms necessary extra detail for Chapter Fourteen and for Chapter Fifteen, which fully describes the parsing algorithm.

Two versions of the probabilities tables were created. The first, which was in the main automatically generated from the FPD Corpus, was adequate for the development and testing of the parser, but was found to be lacking for parsing unrestricted English.

Therefore, a **Version Two** of the probabilities tables was created, which drew its information from a number of sources, but mainly the British National Corpus. Appendix I describes these new tables and the procedures used to create them.

This appendix also covered the parser working tables, its data structures, and the operations used for manipulating them.

Appendix I

The Version Two probabilities tables

This appendix describes how the Version Two **probabilities tables** were constructed (see Chapter Fourteen and Appendix H). These tables are due to be implemented at the start of Phase Two of this project and, at the time of writing, their construction is nearing completion.

I.1 The development of the Version Two tables

In constructing the original probabilities tables introduced in Chapter Fourteen and Appendix H, we have noted several places in which the probabilities were skewed by the fact that the corpus was one that contained children's spoken texts. This led us to develop a second version of the tables so that there are now two versions of the probabilities tables, called **Version One** and **Version Two**. The two versions are stored in database tables that have the same structures, the difference being the different data in each version. The version used by the parser will be available for selection from the **Parser WorkBench** (see Chapter Sixteen).

I.1.1 Weaknesses in the Version One tables

The Version One tables were created automatically from the **corpus tables** (and the **corpus index tables**) that store the FPD Corpus (as described in Chapter Seven). After their creation they were modified in relatively small ways to create a set of tables that would be considered adequate for the development and testing of the parser. However, as the work on developing the parser got underway, we realised both (a) that we needed to develop a set of tables that would be capable of handling all types of text, and (b) that we might be able to devise ways of incorporating data from large tagged corpora into what came to be known as the Version Two of the probabilities tables.

However, some major changes were made to Version One after their automatic creation, ie (a) the identification of those items which needed **item-up-to-element-up-to-unit-up-to-element (I2E2U2E)** treatment, and (b) the identification of the items for the **initial vertical strip item table (IVS - ITEM)**.

The Version One tables ignored any of the following:

- (a) items that were marked in the FPD Corpus as questionable analyses (by containing a question mark),
- (b) items where the analyst was not sure which word was uttered (ie where two items were separated by a slash eg **SAY/SAW?**),
- (c) items that were missing, and so were identified by the mark up element **<ITEMQUERY>**,
- (d) elements that were missing, and so were identified by the mark up element **<ELEMQUERY>**,
- (e) ellipted elements that were identified by the presence of the **ellipted** mark up attribute.

The **initial vertical strip item (IVS-ITEM)** and **initial vertical strip element (IVS-ELEM)** tables were derived from the left-most strip of the parse trees. Even for Version One, however, these had to be substantially extended to increase the coverage so that the parser would not be limited to the texts in the FPD Corpus.

Despite their shortcomings, this initial set of probabilities tables enabled us to complete the development of the Phase One version of the parsing algorithm and to demonstrate that it is indeed a suitable method of parsing. However, one of our primary goals is to develop a parser that is capable of parsing general English that is not restricted in any way to the text type of the source corpus that was used to generate the probabilities tables. The Version One tables had three main issues, which we needed to address in the improved Version Two tables.

The first and perhaps, the most obvious one was connected to the coverage of items in the item tables (**I2E** and **I2E2U2E**). Since item tables in the Version One set were extracted directly from the corpus, and since relatively few items were added during our initial modifications to it, we were only able to parse sentences which contained items that occur in the FPD Corpus. In order to have a parser that is able to cover texts of any type, the item tables would need to contain vastly greater numbers of items.

The second drawback was the problem of misanalysis found in the corpus. The most common problem was that an item was assigned to the wrong element. These analysis errors, which had somehow eluded the rigorous checking procedures used in the original analysis, caused an unexpected number of problems during the testing of

the parser. A considerable amount of research time was devoted to identifying and then correcting them using ICQF+ and its **sentence editor** (see Chapter Eight).

The third drawback was undue weight being given to possible analyses that were relatively rare occurrences within the corpus. In these cases, the syntactic structures were perfectly legal according to the rules of the grammar, but because other structures would occur more frequently in adult, written texts had not occurred. Therefore, the probabilities in the Version One tables gave certain structures higher than expected weighting.

A small corpus of the size of the FPD Corpus cannot be expected to include all of the items nor even all of the syntactic structures that one would expect to occur in a large corpus with general coverage of English. Most of the core structures of English do actually occur in the corpus, but the type of text skews their probabilities. Ideally, when parsing, we would like to be able to draw on probabilities that were extracted from a parsed corpus that contains texts of the type that we are attempting to parse. In the absence of such corpora, the next best type of corpus would be one that is subdivided into representative range of types of texts. Unfortunately no such corpus exists, and in order to build the Version Two tables, we draw mainly on a corpus that is large and general, but undivided - this was the British National Corpus (BNC).

The full range of sources on which we drew, were the following:

- (a) a word list with parts-of-speech and frequencies extracted from the BNC (see Section 3.2.2 of Chapter Three),
- (b) the Moby Project's Part-of-Speech word list (www.dcs.shef.ac.uk/research/ilash/Moby),
- (c) data from Biber et al (1999),
- (d) the system networks of COMMUNAL's GENESYS (see Section 2.5.1 of Chapter Two),
- (e) syntax data in Fawcett (2000a). See Chapter Four and Appendix A and B.
- (f) the Collins COBUILD word usage dictionaries (see Section 3.2.1 of Chapter Three),
- (g) ICQF+, to extract examples from the FPD Corpus (see Chapter Eight), and finally, but most importantly,
- (h) the wide knowledge of an experienced linguist.

As the Version One tables were directly extracted from the **corpus tables**, it is

clear that the parser is '**corpus-consulting**'. With the use of the Version Two tables, it can be argued that the parser is corpus-consulting in two ways. First the main source of the additional data comes predominantly from another corpus - the larger British National Corpus. Second, the Version Two tables can be considered to be part of a 'bootstrap' set of tables, in that their role is simply to give the parser a good start. Then, as new sentences are successfully parsed, they will be added to the corpus tables, and the probabilities tables will then be updated with the information from the new parses. Although it will take a long time for the additional data to have significant effect, the new data will gradually modify the original probability data, as the corpus grows.

I next turn to the methods that were used to create Version Two of the probabilities tables.

I.1.2 How the Version Two tables were developed

It was the **I2E** and **I2E2U2E** tables that were affected by the introduction of the data from the BNC. This is because the BNC is a tagged corpus and therefore does not contain full parse tree analyses. Consideration was given to drawing on another parsed corpus (such as the Penn Treebank), so that the syntax-oriented tables (**E2U**, **U2E**, **FUS** and **BUS**) could also be updated. This was considered to be too complex a task to attempt, as the only parsed corpora available contain analyses in terms of a Chomskyan Phrase Structure Grammar approach. The task might be impossible since so many assumptions about the nature of syntax are different. Specifically, other parsed corpora are not labeled in a sufficiently rich manner to allow such a conversion to be performed. Therefore, the main source for updating these tables was the FPD Corpus (the revised version of the original POW Corpus), and Fawcett (2000a) was used to cover any structures or partial vertical strips that did not appear in the FPD, and for adjusting the probabilities of those that did.

Before we could start, we obtained a list of items from the BNC corpus¹. I loaded the list into a corpus database table in order to allow the extraction of records based on SQL queries. The temporary table contained the following information:

- (a) **ITEM** (word),
- (b) **TOKEN** (BNC category),

¹ I am extremely grateful for the efforts of Adam Kilgarrieff at Brighton University for creating and supplying the BNC frequency lists that we used in this project.

(c) **FREQUENCY.**

The field named **ITEM** contains the word, as extracted from the BNC word list. The field **TOKEN** is the BNC syntax token (see Appendix J for a list of these). The field named **FREQUENCY** is the count of the item when it expounds the given syntax token in the BNC.

I also loaded the Moby Part-of-Speech Word List into a database table with a similar structure.

Once we had these resources in the database, we could begin the task of creating the Version Two tables. This took a considerable amount of time and effort, and the work had to be performed with great care to avoid making mistakes and corrupting the tables.

Our procedure normally required us to obtain from the BNC a list of words that were identified as corresponding to each major word class, i.e. Main Verbs (**M**), heads (**h**) and apexes (**ax**), in Cardiff Grammar terms.² We then copied them in the **I2E** sub-table for the element or group of elements being worked on.

The method of extraction varied considerably. In its simplest form, it involved collecting items from the BNC which had a syntax token which could be mapped directly to an equivalent Cardiff Grammar element. In a more complex form, it involved, for example, taking all items that ended in certain groups of letters such as **-er** and **-est**, and that has a syntax token that could be matched. However, nearly all the lists had to be checked by hand, and some were complemented or replaced by data from other sources listed above. We found occasional problems with the word tagging in the BNC word list (e.g. it identified as adjectives a number of words ending in **-er**, and **-est** that were not; in this case, we used the Moby Word List as a source).

For certain elements this process of extraction was either supplemented or replaced by using data from the FPD Corpus (accessed via ICQF+), Biber et al (1999), the COMMUNAL's system networks, or by using the Collins-COBUILD reference books.

The **I2E** tables, at this point, did not contain probabilities that indicate the elements of structure that the item expounds. This information was created later in the process. Once we had enlarged the **I2E** table (and had checked it by hand), the next

² Although for some elements, we found the Moby Word List a better source than the BNC word lists.

task was to create the equivalent **I2E2U2E** table, if it was needed. This involved identifying item-element pairs that needed an **I2E2U2E** treatment.

After having gathered a complete set of **I2E** and **I2E2U2E** tables, we began the task of merging the sub-tables. To do this, we used a program to query the BNC table and extract the frequencies with which an item expounds each particular BNC category. These were then mapped to the corresponding Cardiff Grammar elements (which were often not in a one to one relationship to the BNC categories, so requiring other conversion logic), and so populating the Version Two **I2E** table with the BNC items. At the time of writing, we are currently performing and checking this merge stage.

I.2 Summary

The Version Two probabilities tables are almost ready to use in the parser. The result of this work will be that the parser now will have access to a database of approximately a 16,000 Main Verb forms, 200,000 nouns, 16,000 adjective forms, and 13,000 manner adverb forms, and a multitude of other elements such as prepositions, Main Verb extensions and so on.

Appendix J

The British National Corpus tag set and its mapping to Cardiff Grammar elements

The British National Corpus (BNC) uses a set of 'tags' called the C5 tagset, which represents the parts of speech that the BNC items expound. This appendix gives the BNC tag set and shows the mappings to the equivalent Cardiff Grammar elements that were used in the creation of the **Version Two probabilities tables** as reported in Chapter Fourteen and Appendix I. The source of this information, which includes the examples, was the BNC website at www.natcorp.ox.ac.uk/corpus.

Note that where the CLAWS automatic tagger returned a probability for the most likely part-of-speech, and the second most likely that was considered too low to disambiguate, the BNC uses 'ambiguity tags' to indicate the two likely parts-of-speech. The ambiguity tags were not used in the creation of the probabilities tables, and hence are not shown here.

BNC tag	Description	Equivalent Cardiff Grammar element
AJO	Adjective (general or positive) (eg good, old, beautiful)	ax
AJC	Comparative adjective (eg better, older)	ax
AJS	Superlative adjective (eg best, oldest)	ax
ATO	Article (eg a, an)	dd or dq
AVO	General adverb: an adverb not sub-classified as AVP or AVQ (eg often, well, longer (adv.), furthest)	ax
AVP	Adverb particle (eg up, off, put)	MEx
AVQ	Wh-adverb (eg when, where, how, why, wherever)	h_wh
CJC	Co-ordinating conjunction (eg and, or, but)	Lnk
CJS	Subordinating conjunction (eg although, when)	B
CJT	The subordinating conjunction that	B
CRD	Cardinal number (eg one, 3, fifty-five, 3609)	qld, am or h (hundred = h)
DPS	Possessive determiner-pronoun (eg your, their, his)	dd
DTO	General determiner-pronoun: ie a determiner-pronoun which is not a DTQ or an ATO.	dd
DTQ	Wh-determiner-pronoun (eg which, what, whose, whichever)	dd_wh
EXO	Existential there: ie there occurring in the there is... or there are... construction.	S_it
ITJ	Interjection or other isolate (eg oh, yes, hmm, wow).	F

BNC tag	Description	Equivalent Cardiff Grammar element
NN0	Common noun, neutral for number (eg aircraft, data, committee)	h
NN1	Singular noun (eg pencil, goose, time, revelation)	h
NN2	Plural common noun (eg pencils, geese, times, revelations)	h
NP0	Proper noun (eg London, Michael, Mars, IBM)	h_n
ORD	Ordinal number (eg first, sixth, 77th, last)	ax
PNI	Infinite pronoun (eg none, everything, one (as pronoun), nobody)	h_p
PNP	Personal pronoun (eg I, you, them, ours)	h_p
PNQ	Wh-pronoun (eg who, whoever, whom)	h_wh
PNX	Reflexive pronoun (eg myself, yourself, itself, ourselves)	h_p
POS	The possessive, or genitive marker 's or '	g
PRF	The preposition of	p
PUL	Punctuation: left-bracket ie (or [-
PUN	Punctuation: general separating mark: ie . , ! : ; or ?	ender 'e'
PUQ	Punctuation: quotation mark: ie ' or "	-
PUR	Punctuation: right bracket: ie) or]	-
TOQ	Infinitive marker to	I
UNC	Unclassified items which are not appropriately considered as items of the English lexicon.	ITEMQUERY
VBB	The present tense forms of the verb BE , except for is, 's , ie am, are, 'm, 're and be (subjective or imperative)	OM or OX
VBD	The past tense forms of the verb BE : was and were .	OM or OX
VBG	The -ing form of the verb BE : being .	M or X
VBI	The infinitive form of the verb BE : be .	M or X
VBN	The past participle form of the verb BE : been .	M or X
VBZ	The -s form of the verb BE : is, 's .	OM or OX
VDB	The finite base form of the verb DO : do .	OM or OX
VDD	The past tense form of the verb DO : did .	M or O
VDG	The -ing form of the verb DO : doing .	M or O
VDI	The infinite form of the verb DO : do .	M
VDN	The past participle of the verb DO : done .	M
VDZ	The -s form of the verb DO : does, 's .	M or O
VHB	The finite base form of the verb HAVE : have, 've .	OX or ON
VHD	The past tense form of the verb HAVE : had, 'd .	OX or OM
VHG	The -ing form of the verb HAVE : having .	X or M
VHI	The infinitive form of the verb HAVE : have .	X or M
VHN	The past participle form of the verb HAVE : had .	X or M
VHZ	The -s form of the verb HAVE : has, 's .	OX or OM
VMO	Modal auxiliary verb (eg will, would, can, could, 'll, 'd)	O
VVB	The finite base form of lexical verbs (eg forgot, send, live, return (including the imperative and present subjunctive))	M
VVD	The past tense form of lexical verbs (eg forgot, sent, lived, returned)	M
VVG	The -ing form of lexical verbs (eg forgetting, sending, living, returning)	M
VVI	The infinitive form of lexical verbs (eg forget, send, live, return)	M
VVN	The past participle form of lexical verbs (eg forgotten, sent, lived, returned)	M

BNC tag	Description	Equivalent Cardiff Grammar element
VVZ	The -s form of lexical verbs (eg forgets, sends, lives, returns)	M
XX0	The negative particle not or n't .	N, O, OX or OM
ZZ0	Alphabetical symbols eg A, a, B, b, C, c, d .	various

Table J.1 The BNC C5 tag set and its mapping to Cardiff Grammar elements

Appendix K

Parser walkthrough

The algorithm that we use in our parser is very different to the traditional algorithms used in other approaches. In order to assist the reader in understanding our processes, I provide a walkthrough using two of the sentences that were used to test the parser in Chapter Seventeen.

Please see Figure 15.5 of Chapter Fifteen (which shows the parser's workflow diagram). Figure 15.8 of Chapter Fifteen shows how the node and level probabilities are calculated.

K.1 The test sentences

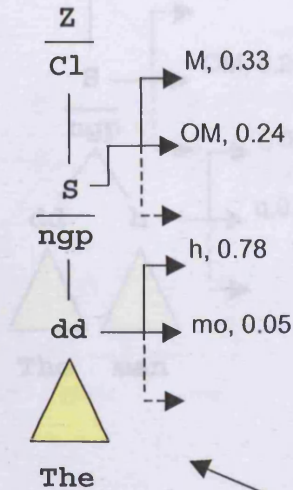
To demonstrate the parser, I have chosen the following two sentences:

- (a) **the man saw me, and**
- (b) **the man saw Cardiff and Treforest**

These sentences together demonstrate each of the stages of the parsing algorithm (apart from Stage 6 - backtracking, which is discussed in Appendix L). The first sentence includes the item **me** (which is an **I2E2U2E** item), and the second includes a co-ordinated unit.

A. Stage 1 and Stage 1a:

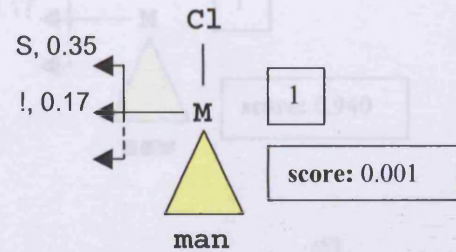
- Stage 1:** the parser requests an **I2E** query for **the** and as **the** is not an **IVS-ITEM** item, it requests an **IVS-ELEM** query. It adds all strips returned to the **DB_IVS** table.
- Stage 1a:** The **FOLLOWED** field for the top **n** of these is set to **true** and built structures are created for them. The most likely of which is shown below. At this stage, the **node** and **level probabilities** are calculated by using **I2E**, **E2U** and **U2E** queries (see Figure 15.8 of Chapter Fifteen).



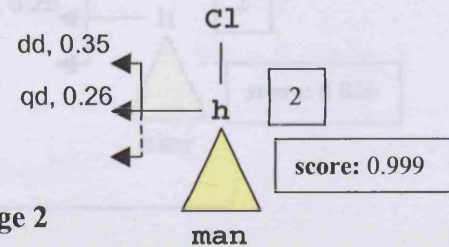
Move 1 and 2 - Stage 1 and 1a

B. Stage 2 :

- Stage 2:** The parser moves to the next item and requests an **I2E** query for it. The item **man** can expound a head (**h**) and a Main Verb (**M**), therefore two candidate structures are created. The node and level probabilities are calculated using the **I2E**, and **E2U** queries (see Figure 15.8 of Chapter Fifteen). As **man** is more likely to expound a head (**h**), this is the more likely candidate structure.



Move 3 - Stage 2



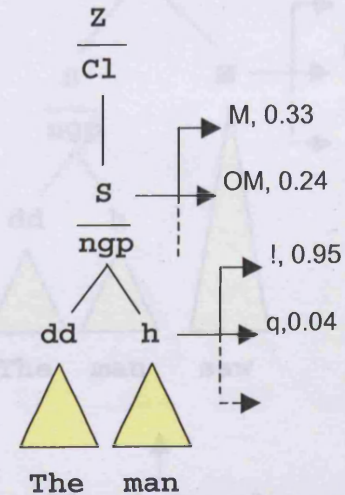
Move 4 - Stage 3

C. Stage 3 and Stage 3a

- Stage 3:** The parser moves to **Stage 3** and determines any possible joins assigning a joining score by using the joining score formula (see Section 15.2.4 of Chapter Fifteen).

It orders any possible joins by their scores (in the PST) and determines that candidate structure 1 cannot be joined to the most likely built structure because the lower unit (**ngp**) cannot be closed. The most likely join is between candidate structure 2 and the most likely built structure. This is because the **forward** and **backward predictions** match.

- Stage 3a:** The parser makes the **n** most likely joins and creates new built structures and calculates the new **node** and **level probabilities** (see Figure 15.8 of Chapter Fifteen). The level probability for the joined node is calculated as the product of the level probabilities of the built and candidate structure nodes involved in the join.

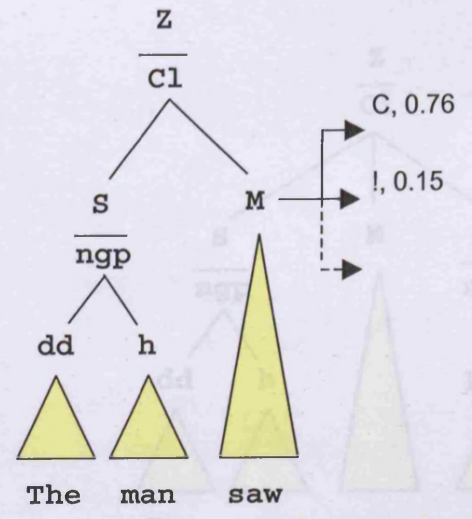
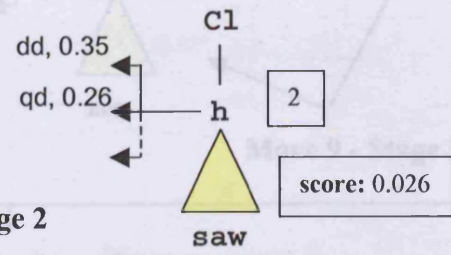
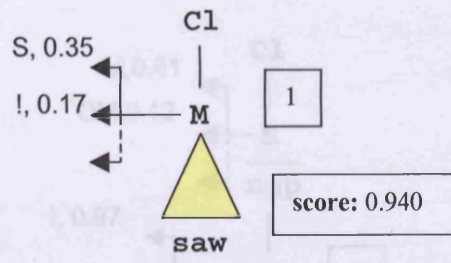
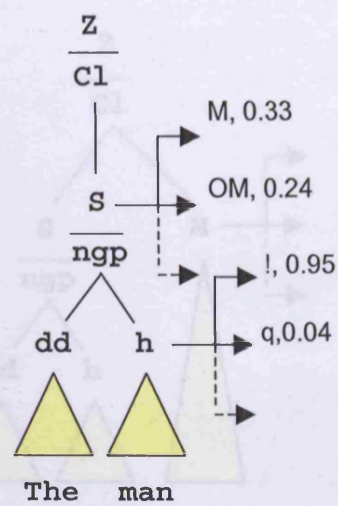


Move 5 - Stage 3a

E. Stage 2, Stage 3 and Stage 3a

6. **Stage 2:** the parser requests an **I2E** query for **saw**. It finds that it can expound a head (**h**) or a Main Verb (**M**) and it therefore creates two **candidate structures**. It uses the **I2E** and **E2U** to create the level and node probabilities and the level probability of the unit in this case is the score for the candidate structure. Because saw is more likely to be a Main Verb, the candidate structure that represents this is the most likely.
7. **Stage 3:** determines the positions in the built structures where the candidate structures can attach. Candidate structure 1 (**saw** as an **M**) can attach to the Clause (**C1**).
8. **Stage 3a:** the parser makes the most likely of these, and sets their **FOLLOWED** field to true in the PST. As before, it calculates the new level probability of the joined node by taking the product of the built and candidate structures involved in the join.

410



Move 6 - Stage 2

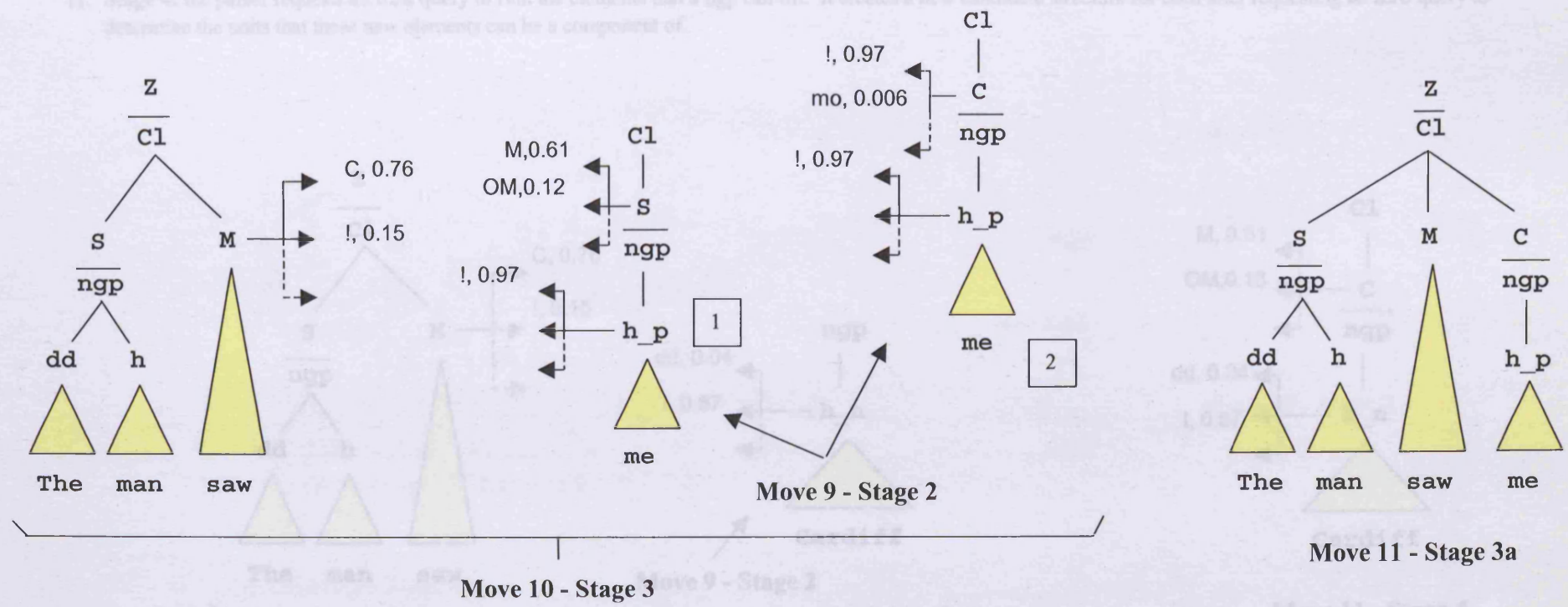
Move 8 - Stage 3a

Move 7 - Stage 3

F. Stage 2, Stage 3 and Stage 3a

9. **Stage 2:** the parser requests an **I2E** query for **me**. **me** is an **I2E2U2E** item, therefore the parser requests one and creates a number of candidate structures, the most likely are shown.
10. **Stage 3:** determines the positions in the built structures where the candidate structures can attach. Candidate structure 2 (the Complement (**c**)) can attach to the Clause as shown.
11. **Stage 3a:** the parser makes the most likely of these, and sets their **FOLLOWED** field to true in the PST. As before, the level and node probabilities are recalculated.
12. **Stage 2, Stage 7:** The parser moves to Stage 2 and finds that there are no more items to process. Therefore, it moves to Stage 7 and as the user does not want any more analyses, this completes the parse for the first sentence. The parse is only successful if all elements in the right-most strip of the final parse predict that they can finish their units. The parse score is the recalculated level probability of the parse tree. When there is more than one analysis, the most likely is the one with the highest score.

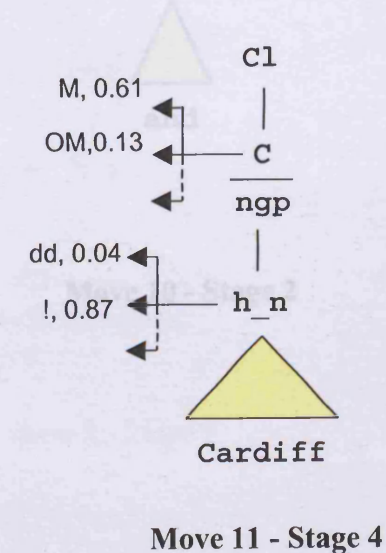
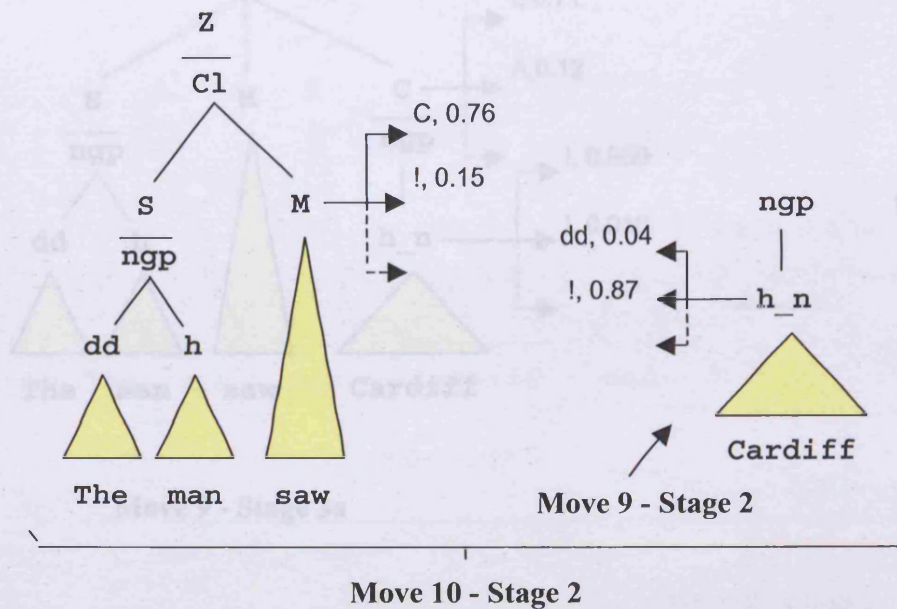
411



E. Stage 2, Stage 3, Stage 4

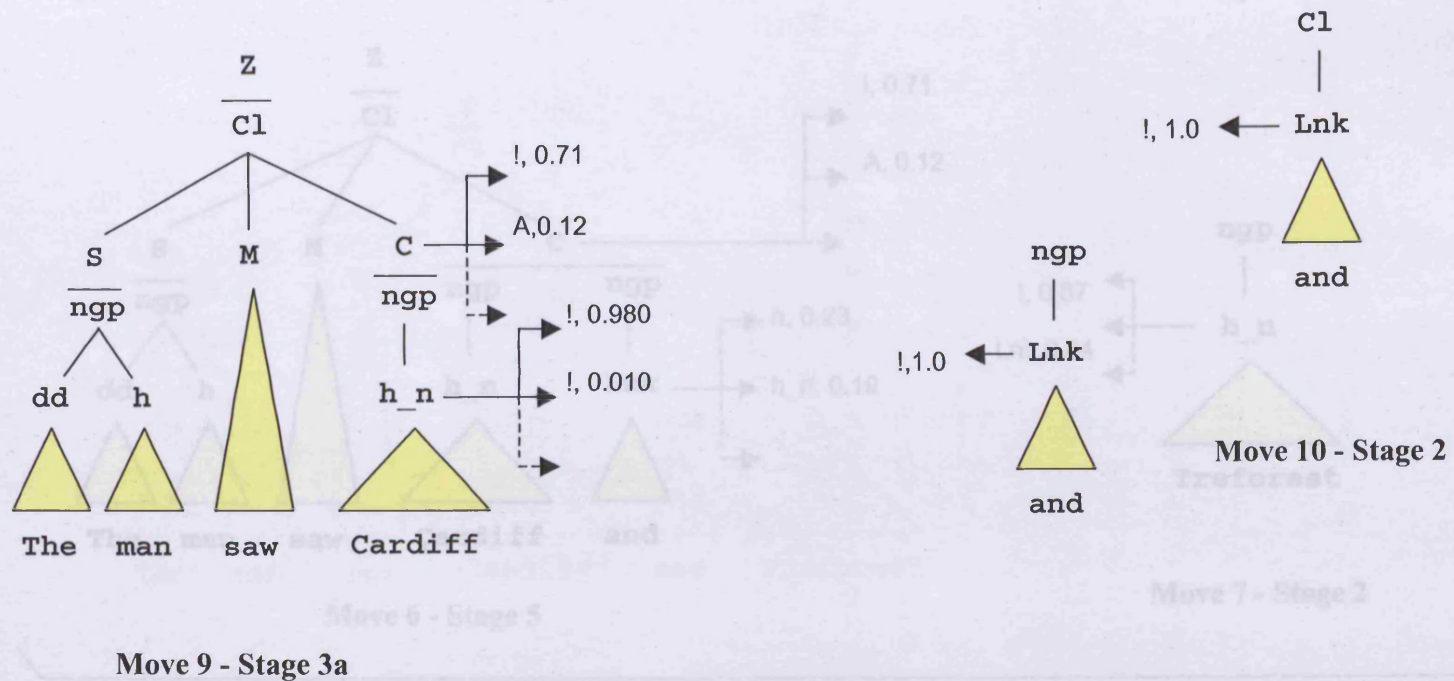
The start of the parse for the second sentence is the same as the one we have already seen. Therefore, we start with the parse after it has processed the item **saw** and created the built structure shown below.

9. **Stage 2:** the parser requests an **I2E** query for **Cardiff** and this states that it can only be a proper name head (**h_n**). It therefore builds a candidate structure for it after performing a **E2U** query that indicates the **h_n** is a component of a **ngp**. It compiles the backward predictions after requesting a **BUS** query. The score of the candidate structure is the product of the **I2E** and the **E2U** that created it (these are the **node probabilities** of the **candidate structure** nodes).
10. **Stage 3:** the parser moves to Stage 3 and finds that the candidate structure cannot join at any position in the most likely built structure and therefore moves to Stage 4.
11. **Stage 4:** the parser requests an **U2E** query to find the elements that a **ngp** can fill. It creates a new candidate structure for each after requesting an **E2U** query to determine the units that these new elements can be a component of.



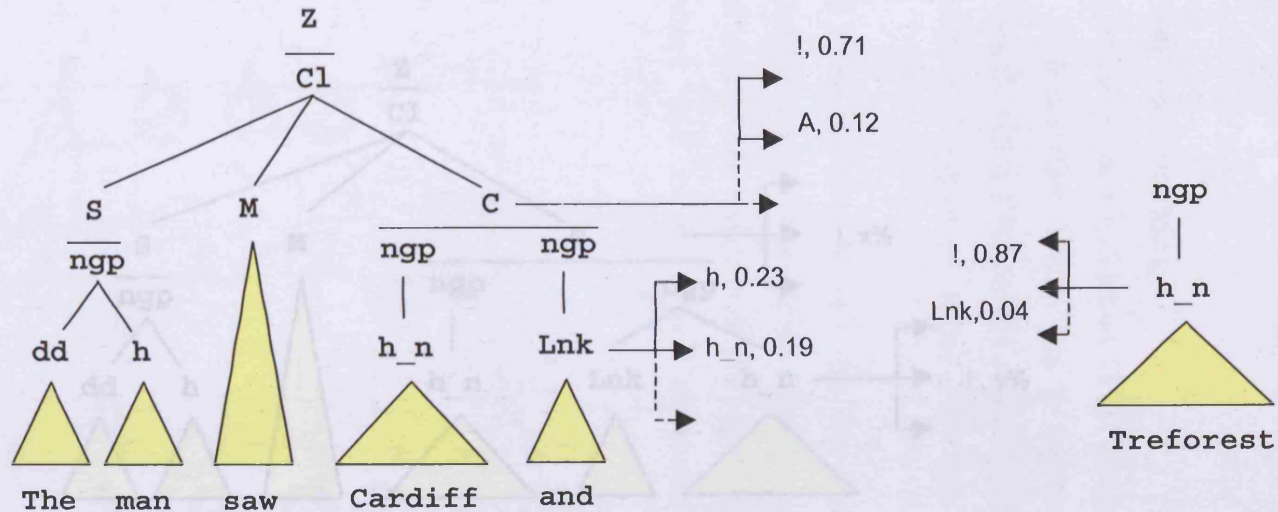
F. Stage 3, Stage 3a, Stage 2, Stage 5

9. **Stage 3:** after growing the candidate structures in Stage 4, the parser performs a Stage 3 and finds that the most likely candidate structure can now join to the built structure and this is done in **Stage 3a**. The forward predictions are created for the new built structure.
10. **Stage 2:** the parser moves to **Stage 2** and requests an **I2E** for the item **and** and finds that it is a linker (**Lnk**). It therefore builds a candidate structure that represents each unit that a linker can occur in.
11. **Stage 5:** the parser now moves directly to **Stage 5** (because the item is a linker)....



G. Stage 5, Stage 5a, Stage 2

12. **Stage 5:** as **and** is a linker, the parser moves to **Stage 5** and determines if a co-ordinated join is possible. It finds that a join to the nominal group (**ngp**) is possible and, in **Stage 5a**, makes the join. After the join, it creates the forward predictions from the right most elements in the new built structure.
13. **Stage 2:** the parser moves to **Stage 2** and requests an **I2E** for the item **Treforest** and finds that it is a proper name head (**h_n**) and therefore it creates a candidate structure after performing a **E2U**. It creates the backward predictions for the candidate structure.
14. **Stage 3:** the parser moves to Stage 3 and determines that the most likely candidate structure is able to join to the most likely built structure....



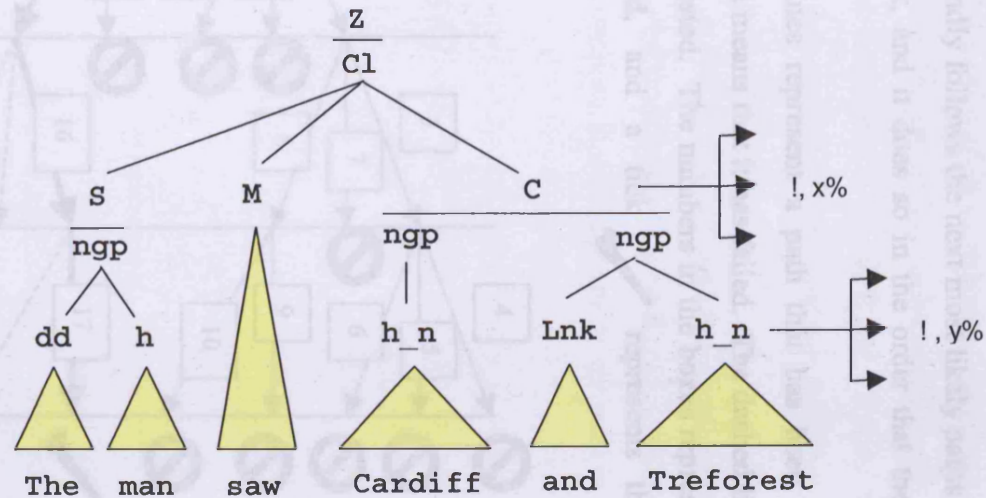
Move 6 - Stage 5

Move 7 - Stage 2

Move 8 - Stage 3

H. Stage 3a, Stage 2, Stage 7

15. **Stage 3a:** makes the join between the most likely built and candidate structures and creates the new forward predictions from the right most elements in the new built structure.
16. **Stage 2:** the parser moves to Stage 2 and finds that there are no more items in the input string, therefore it moves to Stage 7.
17. **Stage 7:** if it is operating in the Parser WorkBench in step-by-step mode, Stage 7 causes a question to be asked to the user if more analyses are required. If they are, or if the parser is not operating in step-by-step mode and the required number of analyses has not been reached, then the parser backtracks (via Stage 6). The correct parse has been found, and the user does not want more, therefore the parser stops.



Move 9 - Stage 3a



Appendix L

Ideas for linguistically motivated backtracking

This appendix details some initial ideas for the implementation of linguistically motivated backtracking.

L.1 Linguistically motivated backtracking

The current model of backtracking (as we saw in Section 15.2.7 of Chapter Fifteen) is based on the Hamilton Path Problem (Ore 1960), and is thus **computationally motivated**. This means that it blindly follows the next most likely paths that have not been followed when it goes back, and it does so in the order that they occur (see Figure L.1).

In Figure L.1, the solid lines represent a path that has been tested. The symbol  at the end of the path means that it has failed. The dashed lines represent paths that the parser has not yet tested. The numbers in the boxes represent the order in which the paths are tested, and a tick  represents the end of a successful path.

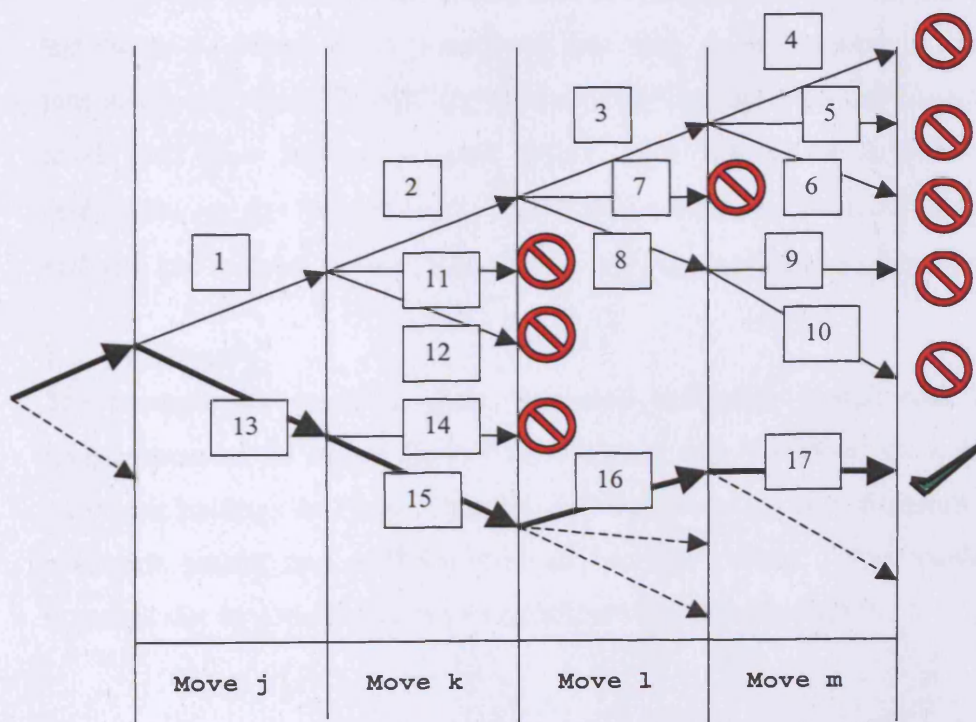



Figure L.1: How the computational backtracking works

Compare this with Figure L.2. This uses the same notations as Figure L.1 but has a circle  to represent a point at which the parser has identified as a backtrack point.

The decision that the parser had to make at this point was one in which the parser has identified as being syntactically significant, and could, for example, represent an ambiguous attachment of a prepositional group, as say, a qualifier to a nominal group or an Adjunct to a Clause. To make sure that the most recent point is considered first, these linguistically motivated backtrack points can be stored in a last-in-first-out stack.¹ When the parser reaches a 'fail', it pops the top point from the stack and moves back to it. Thus it ignores any 'not followed' paths between the point of failure and the popped backtrack point.

This approach can work in tandem with computational backtracking, as shown in Figure L.2, path number 6. Here the parser has backtracked to the linguistically motivated backtrack point and then tried path 6, which failed. It then finds there are no other backtrack points and therefore moves to the next not followed path and tries that.

If you compare Figures L.1 and L.2, you will see that linguistically motivated backtracking controls the order in which paths are followed, and so reduces the time it takes to find the correct path. Furthermore, it will produce fewer trees in its analysis.

These likely backtrack points can be identified in syntactic terms, so allowing the parser to return to constructions and also cause further a backtrack when processing sentences. These happen in so-called 'garden path' sentences, such as in the much used The horse raced past the barn fell. It is only when the reader sees (or the listener hears) the Main Verb fell, that the need to backtrack is realised, and raced past the barn is recognised as a truncated relative clause.

L.2 Summary

The method for marking these required backtrack points has in fact, been implemented in the Parser State Table together with the 'stack' to store the preferred backtrack points. In Phase Two we will complete the specifications for the set of backtrack points and will be able to test the theory. The implementation of linguistically motivated backtracks should provide a better parser.

¹ Another method of handling the order of the backtrack points that will be tested, is to consider them in order of their probabilistic score.

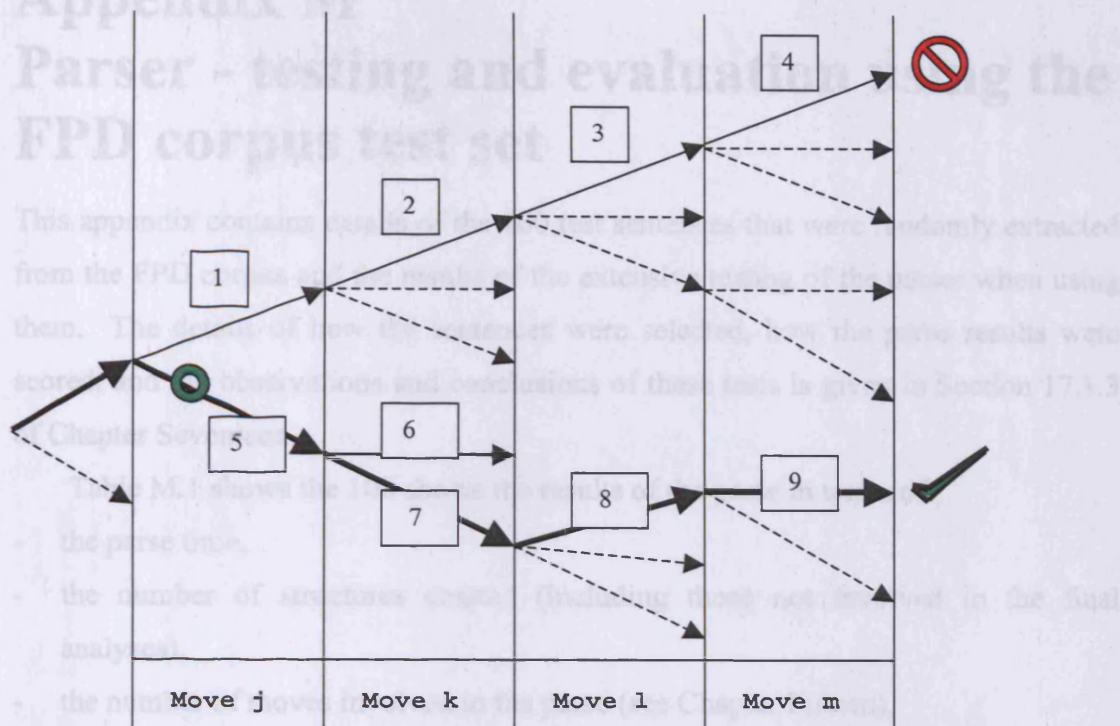


Figure L.2: How the linguistically motivated backtracking works

These likely backtrack points can be identified in syntactic terms, so allowing the parser to return to constructions that also cause humans to backtrack when processing sentences. These happen in so-called 'garden path' sentences, such as in the much used **The horse raced past the barn fell**. It is only when the reader sees (or the listener hears) the Main Verb **fell**, that the need to backtrack is realised, and **raced past the barn** is recognised as a truncated relative clause.

L.2 Summary

The method for marking these favoured backtrack points has, in fact, been implemented in the Parser State Table together with the 'stack' to store the preferred backtrack points. In Phase Two we will complete the specifications for the set of backtrack points and will be able to test the theory. The implementation of linguistically motivated backtracking will provide a better parser.

Appendix M

Parser - testing and evaluation using the FPD corpus test set

This appendix contains details of the 100 test sentences that were randomly extracted from the FPD corpus and the results of the extensive testing of the parser when using them. The details of how the sentences were selected, how the parse results were scored, and the observations and conclusions of these tests is given in Section 17.1.3 of Chapter Seventeen.

Table M.1 shows the 100 shows the results of the parse in terms of:

- the parse time,
- the number of structures created (including those not involved in the final analyses),
- the number of moves involved in the parse (see Chapter Fifteen),
- the score assigned to the parser's analysis (when it is compared to the 'correct' analysis in the corpus) by the XML vertical strip scoring algorithm (see Section 17.1.3.1.2 of Chapter Seventeen).

Note that a score of 1 indicates an exact match between the 'correct' analysis from the corpus, and an analysis returned by the parser for the same sentence.

No.	FPD Cell	Sentence	No. words	Parse time (sec)	No. trees	No. moves	Score against correct analysis	Remarks
1	10abigt#36	I know how to do things.	6	43	203	13	1	
2	10abihs#1	We were talking about what we were going to build and-then we all decided on building a farmhouse.	18	124	488	52	0.68	A complement and an adjunct analysed instead of one complement in the first primary clause ("about" in a complement of its own and "what we were going to build" as an adjunct. "on" (in "decided on") analysed as an MEX and therefore "building a farmhouse" was not within a prepositional group.
3	10abihs#19	You buy lots of property and houses and-then when someone comes along they land on you.	16	80	336	46	0.65	
4	10abihs#2	And-then Richard started building the white and Timothy did the little cars and I did the field and I was building a lorry but when we came in we took the bottom off and used it as the roof of a house instead.	43	731	2193	127	0.93	did analysed as Operator (corpus as Main Verb). "was" analysed as OM (corpus as OX). Partitive determiners not analysed. Linker in "but when..." analysed in a Complement (corpus direct in Clause). "when we cam in" analysed in Complement (Adjunct in corpus). "in" in "came in" analysed as a preposition. "a house" analysed as a cv replacement.
5	10abihs#21	We always play football.	4	26	135	10	0.85	always analysed as a modifier to previous ngp. Investigation showed that the corpus has examples of mo following h_p in ngp (as in "these little"). Although low probability, these made the top n joins.
6	10abirl#42	It was quite recently I saw the-zulu again.	8	39	192	22	0.98	It analysed as Subject instead of S_it (not penalised). "I saw zulu again" analysed as an Adjunct instead of a Complement (in corpus analysis).

Table M.1: The test set of 100 sentences from the FPD Corpus, results of the test using the CCPP, and observations

No.	FPD Cell	Sentence	No. words	Parse time (sec)	No. trees	No. moves	Score against correct analysis	Remarks
7	10abirl#49	So the Zulus got angry and started attacking them again and the commander was shouting south wall fifty men or something like that.	23	203	858	67	0.74	Text ("fifty men south wall") not supported in Phase One. C_Repl ("Or something like that") analysed as a co-ordinated clause. "Was" analysed as O/M by the parser (corpus O/X). The clause starting with "attacking" was not embedded correctly.
8	10abirl#55	And the hospital was where the zulus were invading.	9	72	402	25	0.65	Embedded clause not analysed correctly
9	10abitg#37	have to know people and what they do.	8	42	219	22	0.79	and what they do analysed as a co-ordinated clause instead of a clause in a qualifier.
10	10abitg#63	So the board goes down on the stone and he lights the fires underneath and the tosses the shark at them.	21	272	1121	61	1	
11	10abitg#64	And he puts all the potatoes on there and the fries the chips.	13	128	653	37	0.95	on analysed as an MEx.
12	10abprsl#220	I 'll have to add a-few more windows.	9	25	148	22	0.9	qtgp for "a few more" analysed as directly filling a C, CI had two C
13	10abprsl#52	Then I 'll have the windows after.	7	38	255	16	1	
14	10abprsl#61	Now I 'll have the windows.	6	36	258	17	1	
15	10abpshs#15	Look at this door.	4	20	98	10	1	
16	10abpshs#24	We can make a little.	5	67			1	
17	10abpshs#24	We can make the telephone-box like this.	8	67	403	19	1	Preferred analysis had 'like this' as a complement (C)
18	10abpshs#27	That isn't a very good car is it.	8	103	528	22	1	A_Log analysed correctly
19	10abpsrl#105	Tim here 's a good door.	6	81	456	16	0.94	Tim analysed as a Complement instead of a Vocative.

Table M.1: The test set of 100 sentences from the FPD Corpus, results of the test using the CCPP, and observations (cont'd)

No.	FPD Cell	Sentence	No. words	Parse time (sec)	No. trees	No. moves	Score against correct analysis	Remarks
20	10abpstg#19	I 'll just do the roof.	6	44	239	16	1	
21	10abpstg#22	Shall we put the trees in there look.	8	86	492	29	1	
22	10abpstg#24	You do the cars and I 'll do the bus-stop now.	11	104	569	31	1	
23	10abpstg#30	We have to take one of these.	7	45	258	25	1	
24	10agikp#48	I 'd like to run a stable.	7	60	344	19	1	
25	10agilb#1	We decided on a bungalow because we thought it would save time because if you are doing a house you wouldn't do enough of it.	25	302	1062	73	0.93	"on" analysed as MEx instead of a preposition. "it would save..." analysed as a Complement instead of an A_Log and "if you are doing" was also a Complement instead of A_Log. "You wouldn't do enough of it" analysed as a qualifier instead of an A_Log
26	10agpskp#11	We forgot to put the windows in.	7	62	331	20	0.92	C analysed as an "A"
27	10agpskp#17	Put some more windows in.	5	29	153	13	0.64	qtgp "some more" analysed directly as C, "in" analysed as a preposition and not an MEx
28	10agpskp#66	We should have a back door really.	7	48	281	19	0.97	a back and "door" analysed as two nominal groups and not a modifier.
29	10agpslb#16	Look what are we going to do with that little gap.	11	69	296	28	0.95	look was analysed as a main verb but an A_AS in the corpus and hence "we going..." was analysed as a clause in a Complement of the main clause and not elements of the main clause.
30	10agpslb#33	Look we 'll leave a space for the door and-then we 'll put it on after.	16	139	829	46	0.91	Look analysed as a Complement of the primary clause (in own clause in Corpus). "for the door" analysed as a qualifier after "space" and not an Adjunct.

Table M.1: The test set of 100 sentences from the FPD Corpus, results of the test using the CCPP, and observations (cont'd)

No.	FPD Cell	Sentence	No. words	Parse time (sec)	No. trees	No. moves	Score against correct analysis	Remarks
31	10bbigi#16	When all the monies have run out you just add your money up and see who 's got the most and they 've won.	23	178	770	70	0.95	When all the monies have run out not analysed as an Adjunct but in the main Clause. "You just add.." and "who's got the most.." not embedded correctly. All clauses analysed correctly but not embedded correctly.
32	10bbihw#43	A man got sent to prison.	6	19	126	16	0.91	In some of the 'successful parses', "got" analysed as a main verb in a clause with two main verbs. "to prison" analysed as Adjunct instead of Complement. Note that this was allowed because of an error in the corpus where one clause was analysed as S M M A (two main verbs followed by Adjunct) Score of parse was low. In other parses, "got" was an XEx (X in corpus analysis).
33	10bbihw#57	Because you can get into football-matches free.	7	30	157	19	0.97	Whole sentence analysed as an A_Log in the corpus. Parser analysed the clause that is supposed to fill the A_Log as the primary clause.
34	10bbpsgi#16	Look at this ladder.	4	19	98	10	1	
35	10bbpshw#1	We don't really need windows.	5	16	82	13	1	
36	10bbpshw#1	I wonder if there 's a dog in there.	9	110	624	25	0.92	The pgp "in there" analysed as a qualifier of the nominal group "a dog".
37	10bbpshw#2	We won't be able to do another layer though will we.	11	35	155	31	1	
38	10bbpshw#3	We can make a cat for them.	7	91	497	19	1	

Table M.1: The test set of 100 sentences from the FPD Corpus, results of the test using the CCPP, and observations (cont'd)

No.	FPD Cell	Sentence	No. words	Parse time (sec)	No. trees	No. moves	Score against correct analysis	Remarks
39	10bgicl#38	I usually have a fruit when I go home and-then I have my tea and I have my supper a-few hours after and-then I go to bed then.	28	401	917	82	0.96	Usually analysed as directly filling an Adjunct whereas it is an apex in the corpus. "home" is analysed as a nominal group filling an MEx. "a-few hours" analysed as a Complement instead of an Adjunct. "after" analysed as a separate Adjunct.
40	10bgiee#31	I mean a man that changed into a dog.	9	101	513	25	1	S_Wh analysed as S (not penalised)
41	10bgire#49	No but my auntie 's Spanish so she teaches me mostly Spanish.	12	71	327	34	1	
42	10bgpscl#21 2	I 'll have to take the roof off to stick him in.	11	62	337	34	1	
43	12dgmism#17	His father was a doctor and his father got the police and the men were arrested and they found they were after money trying to rob something	27	297	1264	79	0.94	Final clause ("trying to rob something") analysed as a qualifier and not an Adjunct.
44	10cbpsat#12 3	Cor wish they didn't have this carpet.	7	31	166	19	0.95	Cor analysed as a Formula instead of an Exclamation.
45	10cbpslj#178	I 'm having this ladder.	5	19	88	13	1	
46	10cbpsmh#8 5	Do you like your teacher.	5	21	106	13	1	
47	10cgied#13	Sometimes we run about.	4	14	72	10	1	
48	10cgisp#42	She sells houses to people and gets all the deeds done.	11	89	470	31	0.78	1. Error in corpus analysis ("all the deeds done" analysed as "S")
49	10cgisp#52	She would get the people who are most noisy out the front and-then tell them to shut up.	18	223	2119	52	0.86	Qualifier ("who are most noisy") analysed as Complement. Tempering preposition "the front" not recognised.

Table M.1: The test set of 100 sentences from the FPD Corpus, results of the test using the CCPP, and observations (cont'd)

No.	FPD Cell	Sentence	No. words	Parse time (sec)	No. trees	No. moves	Score against correct analysis	Remarks
50	10dbipl#43	At the end he climbed up a building and he had a woman in his hand.	16	208	1992	46	0.86	The Adjunct ("in the end") was analysed as a Complement and "up the building" was analysed as an Adjunct instead of a Complement.
51	10dgidh#25	Sometimes he do pull them down by the hair and give them a smack or just send them to mr-Rhys where they have a cane.	25	258	975	73	0.87	Sometimes analysed directly as filling Adjunct (Corpus had apex of qlgp filling an Adjunct). "smack" analysed as a main verb instead of a head and "them" and "a" as h qd in the same nominal group. "to" analysed as an I in a clause (instead of a preposition and hence a clause instead of a prepositional group. "where they have a cane" analysed as a qualifier of "Mr Rhys".
52	10dgism#13	I ride my bike.	4	17	96	10	1	Correct parse ranked in second position.
53	10dgism#52	But as-soon-as I come off I sat down and I had something to eat and-then I was alright again then.	21	206	805	58	0.93	Sat down analysed as an Adjunct rather than directly in the clause. "to eat" analysed as a clause filling an Adjunct rather than a qualifier of the nominal group. The first clause analysed as filling a Sentence which is TEXT in a Complement (structure inside was correct).
54	10dgiss#11	And we put an aerial on the roof and the chimney.	11	128	696	31	1	
55	10dgpsdh#18	Talks like a queen.	4	54	277	10	1	
56	10gbsre#156	You should have seen this film it was revolting it was.	11	116	614	34	0.9	It was revolting... was analysed as A_CSTag (corpus "it was" was A_CSTag). "Oh God it" was analysed as a separate Complement and not a co-ordinated clause. "it" was analysed in the same clause as "Oh God". "It was" was analysed as a qualifier not A_CSTag.

Table M.1: The test set of 100 sentences from the FPD Corpus, results of the test using the CCPP, and observations (cont'd)

No.	FPD Cell	Sentence	No. words	Parse time (sec)	No. trees	No. moves	Score against correct analysis	Remarks
57	12abiaw#14	When some people run away and they look for you and you got to get to the stone first.	19	141	569	55	0.87	When analysed as a Binder an not an A_Wh. "to" analysed as I in an embedded cause instead of a preposition.
58	12abpsaw#2 26	She 's alright for a free lesson but i 'm not sure she 's particularly good.	15	188	914	46	0.82	In most likely "for a free lesson" analysed as an Adjunct instead of a Complement (complement version lower in rank). "not" analysed as part of an embedded clause instead of in the main clause. "'s particularly good" analysed as a clause filling a quallfier instead of a scope. Note that modifiers analysed correctly..
59	12abpspg#8a	What 's the point.	4	35	247	10	1	
60	12abpssm#7 6	There 's ten.	3	23	166	7	1	
61	12agibr#37	No we 'ad scampi and chips.	6	31	122	16	1	Note that most likely analysis had co-ordinated clauses instead of co-ordinated nominal groups
62	12agind#50	I 'd like to go over and see Disneyland and see my auntie over there.	15	115	534	43	0.65	Two embedded complements analysed ("to go" and "over ") instead of one. Embedded clause not analysed at right level.
63	12bbimb#29	I 've seen star-wars.	4	14	71	10	1	
64	12bbimn#34	And she went back to the pet shop and she said my hamster died.	13	122	543	40	0.81	Text not supported in Phase One. "to" analysed as an I in the Clause instead of a preposition. "pet" analysed as the head of the nominal group and not a modifier. "shop" was therefore analysed as a separate Complement in the clause.
65	12bgiah#35	He was on the back of a cart.	8	88	525	22	0.87	Partitive determiner ("the back of") not analysed - was separate nominal group in a clause with two complements instead of one.

Table M.1: The test set of 100 sentences from the FPD Corpus, results of the test using the CCPP, and observations (cont'd)

No.	FPD Cell	Sentence	No. words	Parse time (sec)	No. trees	No. moves	Score against correct analysis	Remarks
66	12bgihl#16	And play ball.	3	13	102	7	1	
67	12dbiaf#21	Their legs go forward and they kick the ball.	9	48	253	25	1	
68	12dbipl#52	Put him in a barrier or a pole.	8	92	448	22	0.79	The linker "or" co-ordinates the clauses instead of the nominal groups. The nominal group "a barrier" is not analysed as a completive of the prepositional group. Analysis showed that correct joins were not in top n in latter moves. Corpus analysis questioned because "in" may be an MEx - "put in".
69	6abicij#30	I Saw the TV yesterday.	5	46	262	13	1	
70	6abpscj#34	A lego boat sailing and people was in it.	9	62	316	25	0.92	Lego analysed as filling the head of the nominal group and not a modifier, "boat" was in a different .nominal group. The clause "people was in it" filled a qualifier instead of a Complement in the clause.
71	6agika#10	They play star-wars sometimes.	4	12	67	10	1	
72	6agika#13	I like princess-leela the best in star-wars.	7	86	438	19	0.7	the best in Star-Wars analysed as dd mo h_n with "in" starting a finisher. Corpus analysis is qld ax ("the best") with "in Star-Wars" being an Adjunct.
73	6agika#20	Because they had guns and they shoted and they had star-wars swords as well.	14	53	214	37	1	

Table M.1: The test set of 100 sentences from the FPD Corpus, results of the test using the CCPP, and observations (cont'd)

No.	FPD Cell	Sentence	No. words	Parse time (sec)	No. trees	No. moves	Score against correct analysis	Remarks
74	6agika#6	You all make a ring and somebody 's the farmer and-then you all say the-farmer-wants-the-wife and-then you make somebody as the wife and-then the nurse then the child then the dog and then the bone and we all pat the bone.	42	860	2564	121	0.81	TEXT element "the farmer.." was not analysed because TEXT is not supported in Phase 1. Ellipted prepositions (e.g. "as the wife") not recognised (applies to 5 primary clauses). 4 of 9 clauses correctly analysed.
75	6agpspn#261	I can make a big huge aeroplane.	5	61	354	19	1	
76	6bgicl#28	But I didn't want to go.	6	18	91	16	1	
77	6bgicl#31	They were trying to bash and trying to play a game and-then they came to a water and-then they saw some meat.	22	193	773	64	0.98	a game was analysed as a separate Complement in the higher clause rather than the one containing "to play..".
78	6bgihj#42	The tin man wanted a brain.	6	58	343	16	0.94	tin not analysed as a modifier.
79	6bgpscl#195	Stop screaming or laughing tell me tidy.	7	19	97	19	0.68	Stop analysed as a head instead of a main verb. "Tidy" analysed as a Complement and not an Adjunct. "Or laughing" analysed as a co-ordinated primary clause.
80	6cgirs#24	Sabrina was up in an aeroplane.	6	36	239	16	1	
81	6dbimj#7	Got my Star-Wars gun.	4	26	134	10	1	Preferred analysis had two complements ("Star Wars" and "Gun").
82	6dgikj#15	A little mouse fell in the water.	7	66	323	19	1	
83	6dgikj#17	Eat him all up.	4	8	40	10	1	
84	12dgism#15	Its about a farm boy and he lived on a farm and he had a horse black beauty	18	745	52	10	0.98	"boy" analysed as a qualifier of "farm" instead of a modifier. "Black Beauty" analysed as a Complement and not a Replacement Complement (not penalised)
85	6dgisd#19	There 's a boat in the water and they sink and there 's cars.	14	124	630	140	0.98	in the water was analysed as an Adjunct whereas it was a Complement in the corpus.

Table M.1: The test set of 100 sentences from the FPD Corpus, results of the test using the CCPP, and observations (cont'd)

No.	FPD Cell	Sentence	No. words	Parse time (sec)	No. trees	No. moves	Score against correct analysis	Remarks
86	6dgisd#25	A wolf from the forest and a girl was there and the farmer came right and he found them and he shot the wolf.	24	299	1259	70	0.98	The qualifier ("from the forest") was analysed as a Complement.
87	6dpsmj#197	Is that going to be a boat 'cause boats don't have wheels.	12	77	447	34	1	
88	8abiji#16	And it cost me about ten pound to buy it.	10	46	265	81	1	
89	8abiji#35	Sometimes I play on my bike.	6	26	131	16	0.98	On my bike analysed as a Complement by the parser and an Adjunct in the corpus.
90	8abijs#21	We play football in the street and	7	62	368	19	1	
91	8abijs#34	And-then scooby was acting and-then they just take him and he kepted on switching it until they all came around and all clothes fell off him.	26	158	527	76	0.98	Subject instead of Complement in ("switching it"). Last clause ("and all clothes fell off him") was analysed as a primary clause instead of an embedded one.
92	8abijs#35	And-then he started running into the bush and the gun fell in a bush and the bush started leaping and the witch-doctor started running away from them then.	28	377	1271	82	0.9	The final "then" was analysed as a linker and not an Adjunct
93	8agijo#4	So we made a little street.	6	56	305	16	1	
94	8agijr#48	And-then after they 've learnt that i' ll let them try and swim on their own.	16	63	414	46	0.84	"after" analysed as a preposition rather than a Binder. The elements of the clause "they 've learnt that" analysed in the main clause whereas the corpus analysis has the clause filling an Adjunct. "Try and swim on their own" analysed as a clause filling a qualifier and not a complement. "Their" not analysed in the genitive cluster, but "own" was.
95	8agish#28	I don't make friends in my street because it 's on the main road.	14	117	575	40	1	

Table M.1: The test set of 100 sentences from the FPD Corpus, results of the test using the CCP, and observations (cont'd)

No.	FPD Cell	Sentence	No. words	Parse time (sec)	No. trees	No. moves	Score against correct analysis	Remarks
96	8bbipj#3	And they put an aerial on top to pretend there was a tv set in it.	16	174	861	46	1	
97	8bgjbc#28	I can't swim.	3	5			1	
98	8dbimh#71	And the cat picked it up and sucked it in the	11	78	452	31	0.95	The linker "and" was analysed in the nominal group "the cat". "Sucked it in the.." was analysed as an Adjunct rather than a Complement.
99	8dgpssr#148	I think i 'll put a man standing up there and-then a girl on the blue one.	17	231	1072	49	0.82	First clause incorrectly embedded within a complement. "Standing up there" analysed as an Adjunct and not a qualifier (as it is in the corpus). "on" analysed as an MEx and not a preposition. "the blue one" analysed as a complement and not an adjunct.
100		You put straws into a glass tube with holes in and-then you put the straws in the holes and-then you put marbles down and-then pull a straw out to see if a marble goes into a point.	37	662	2201	112	0.94	Embedded modifier ("glass") (two ngp). "In the holes" analysed as a Complement instead of an Adjunct. "To see if a ..." analysed as an Adjunct instead of a Complement.

Table M.1: The test set of 100 sentences from the FPD Corpus, results of the test using the CCPP, and observations

