

Disabled Person's Control, Communication and Entertainment Aid

T.24
2

**An Investigation of the Feasibility of Using Speech
Control and Natural Language Understanding to
Control a Manipulator and a Software Application and
Development Environment**

**A thesis submitted in fulfilment of the requirements for the
award of the degree of**

Master of Engineering

from

Victoria University of Technology

by

Malcolm James Dow

**Department of Electrical and Electronic Engineering
March, 1994**



FTS THESIS
617.030285 DOW
30001002327619
Dow, Malcolm James
Disabled person's control,
communication and
entertainment aid : an

DECLARATION

I, Malcolm James Dow, hereby declare that this submission is my own work and that, to the best of my knowledge, it contains no material previously published or written by another person nor material which to a substantial extent has been accepted for the award of any other degree or diploma of a university or other institute of higher learning, except where due acknowledgment is made in the text.



Malcolm James Dow

ACKNOWLEDGMENTS

The author of this thesis acknowledges the following people and organisations for directly or indirectly providing assistance and guidance while carrying out this research and writing this thesis.

Mrs. Elizabeth Haywood, Mrs. Ann Pleasants and Mr. Alec Simcock, the academic supervisors, for their patience, encouragement and constructive criticism during the research and preparation of the thesis.

Dr Chula na Ranong, for encouraging me to begin investigating the possibilities of speech control systems.

Mr. Ted Walker and Mr. Wally Evans, previous and present Heads of Department, for allowing time from my teaching duties, and facilities to complete this research and thesis.

Victoria University of Technology, for the award of a Commonwealth Staff Development Fund scholarship, which gave me a semester free of teaching to concentrate on preparation of this thesis.

To my wife, Diana, who endured this work and the loss of my undivided attention for many years, I express my love and gratitude.

Finally, I dedicate this work to the Lord Jesus Christ, who is my sole motivation, and who gave me a mind with which to create for Him and to communicate His glory to others.

PUBLICATIONS ARISING FROM THIS THESIS

- [1] Dow, Malcolm J. and Stevens, Mark, "Adaptive Audio Noise Cancelling Unit", Footscray Institute of Technology, Department of Electrical and Electronic Engineering, unpublished project report, Footscray 1986.
- [2] Dow, Malcolm J. and na Ranong, Chula, "Natural Language Interface for a Disabled Person's Aid" in *Proceedings of the Australian Colleges of Advanced Education 18th Annual Computer Conference*, South Australian Institute of Technology, Adelaide 1987.
- [3] Dow, Malcolm J. and na Ranong, Chula, "An Interrupt Driven Memory Resident Control Program for a General Purpose Speech Recognition System", *Conference on Computing Systems and Information Technology*, The Institution of Engineers, Australia, Brisbane 1987.
- [4] Dow, Malcolm J., "SARLIB: A Library of SAR-10 Speech Recognition and Audio Response Interface Routines", Technical Report 1, Computer Application Software and Hardware Group, Department of Electrical and Electronic Engineering, Victoria University of Technology, Footscray, Victoria 1994.
- [5] Dow, Malcolm J., "A Machine Readable Dictionary for Natural Language Understanding Systems", Technical Report 2, Computer Application Software and

Hardware Group, Department of Electrical and Electronic Engineering, Victoria University of Technology, Footscray, Victoria 1994.

[6] Dow, Malcolm J., "Affix Transforms for a Machine Readable Dictionary for Natural Language Understanding Systems", Technical Report 3, Computer Application Software and Hardware Group, Department of Electrical and Electronic Engineering, Victoria University of Technology, Footscray, Victoria 1994.

[7] na Ranong, C. and Dow, M.J., "A Voice Controlled Robot", *Robots in Australia's Future Conference*, Perth, Western Australia 1986.

ABSTRACT

The work reported in this thesis is a feasibility study of the possibilities and practical problems of applying speech control and natural language understanding techniques to the use of a computer by a physically disabled person. Solutions are proposed for the overcoming of some of the difficulties and limitations of the available equipment, and guidance given for the application of such systems to real tasks.

The use of voice control with a low cost industrial robot is described. The limitations introduced by the speech control hardware, such as restricted vocabulary size and artificial manner of speaking are partially overcome by software extensions to the operating system and the application of natural language understanding techniques.

The application of voice control and audio response to common application packages and a programming environment are explored.

Tools are developed to aid the construction of natural language understanding systems. These include an extension to the use of an existing context-free parser generator to enable it to handle context-sensitive grammars, and an efficient parallel parser which is able to find all possible parses of a sentence simultaneously.

Machine readable dictionary construction is investigated, incorporating the analysis of complex words in terms of their root forms using affix transformations, and the

incorporation of semantic information using a variety of techniques, such as semantic fields, the previously mentioned affix transforms, and object-oriented semantic trees.

The software developed for the system is written in *Borland Pascal* on an *IBM* compatible PC, and is produced in the form of library modules and a toolkit to facilitate its application to any desired task.

TABLE OF CONTENTS

	<u>Page</u>
ACKNOWLEDGMENTS	i
PUBLICATIONS ARISING FROM THESIS	iii
ABSTRACT	v
TABLE OF CONTENTS	vii
LIST OF FIGURES	xv
LIST OF TABLES	xviii
CHAPTER 1 INTRODUCTION	1-1
<u>Part I Voice Control</u>	
CHAPTER 2. VOICE CONTROL SYSTEMS	2-1
2.1 Two Approaches to Speech Recognition	2-1
2.1.1 Continuous Speech Recognition	2-1
2.1.2 Isolated Word and Phrase Recognition	2-2
2.2 Acoustic Analysis for Isolated Word Recognition	2-3
2.3 Dynamic Time Warping	2-7
2.4 Speech output	2-9

2.5. The Speech Control and Audio Response System	2-11
2.5.1 The <i>SAR-10</i> System	2-11
2.5.2 <i>SAR-10</i> Software	2-12
2.5.3 <i>SAR-10</i> Command Structure	2-13
2.5.4 Communication Between the <i>SAR-10</i> and the PC	2-15
2.5.5 The <i>SAR-10</i> Speech Recognition Function	2-17
2.5.6 Speech Recognition Vocabulary Table	2-18
2.5.7 Recognition Clusters	2-19
2.5.8 Speech Recognition Parameters and Flags	2-20
2.5.9 The <i>SAR-10</i> Audio Response Function	2-21
2.5.10 Audio Response Parameters	2-22
CHAPTER 3 SIMPLE APPLICATIONS OF THE <i>SAR-10</i> SYSTEM	3-1
3.1 Application Program Interfacing	3-1
3.2 Word Processing	3-2
3.2.1 Entry of Text	3-3
3.2.2 Navigation Within a File and Command Sequences	3-5
3.2.3 Execution of Recognised Commands	3-8
3.2.4 Controlling Command Context	3-10
3.2.5 Results	3-11
3.2.6 Conclusions	3-13

3.3 Spread Sheet	3-13
3.3.1 Data Entry	3-14
3.3.2 Commands	3-14
3.3.3 Results	3-15
3.4 DOS Shell	3-15
3.5 Keyboard Macro Processor	3-16
3.6 Programming Environment	3-17
3.6.1 <i>Turbo Pascal</i>	3-17
3.6.2 Reusable Code	3-19
3.6.3 Vocabulary Selection and Programming Technique	3-20
3.6.4 Programming Aids	3-23
3.7 <i>MicroExpert</i> Expert System Shell	3-25
3.8 Use with Mouse Based Applications	3-26
3.9 Conclusion	3-27
CHAPTER 4. APPLICATION: A VOICE CONTROLLED ROBOT	4-1
4.1 The Robot and its Controller	4-1
4.2 The Robot Control Program	4-2
4.3 The Robot Control Language	4-3
4.3.1 Vocabulary Orthogonality	4-4
4.3.2 Speaker Independence	4-7

4.4 Performance of the Robot Control System	4-8
4.4.1 Operation in a Noisy Environment	4-8
4.4.2 Operation Under Stress	4-9
4.4.3 Command Verification	4-11
CHAPTER 5. OPTIMISING THE <i>SAR-10</i> 's PERFORMANCE	5-1
5.1 Audio Pre-processing	5-1
5.1.1 Microphone Choice	5-1
5.1.2 Electronic Speech Processing	5-2
5.1.3 Program Control of Microphone On/Off	5-4
5.2 Vocabulary and Reference Pattern Control	5-5
5.2.1 Cluster Control	5-5
5.2.2 Retraining and Updating Reference Patterns	5-6
5.3 Parameter Control	5-7
5.4 User Interface Optimisation	5-8
CHAPTER 6. EXPANDING THE VOCABULARY	6-1
6.1 Vocabulary Context Swapping from Disk	6-1
6.2 Determining and Controlling the Context	6-2
6.2.1 Context Determination	6-2
6.2.2 Manual Context Control	6-3
6.2.3 Automatic Context Control	6-4

6.2.4	Programmed Context Control	6-5
6.2.5	Artificial Intelligence and Context Control	6-6
6.3	Storage and Transfer Considerations	6-7

CHAPTER 7. *VOICEDOS*: A VOICE CONTROLLED OPERATING

	SYSTEM EXTENSION	7-1
7.1	Methods of Extending an Operating System	7-1
7.2	The <i>VOICEDOS</i> Control Program	7-2
7.2.1	The <i>VOICEDOS</i> Command Interpreter	7-3
7.2.2	The <i>SARLIB</i> Interface Routines	7-4
7.2.3	Speech Control Utilities	7-5
7.3	Implementing Memory Resident Utilities	7-6
7.3.1	Requirements of Memory Resident Programs	7-6
7.3.2	Construction of Memory Resident Programs	7-7
7.4	Making <i>VOICEDOS</i> Utilities Memory Resident	7-7
7.4.1	The Installation Program	7-7
7.4.2	The Interrupt Service Routine	7-9
7.5	<i>VOICEDOS</i> Utilities	7-10
7.5.1	Setting the <i>SAR-10</i> Recognition Parameters	7-11
7.5.2	Training the <i>SAR-10</i> for Speech recognition	7-12
7.5.3	Testing the <i>SAR-10's</i> Recognition	7-14
7.5.4	Training the <i>SAR-10</i> for Audio Output	7-14

CHAPTER 8. INTEGRATING VOICE CONTROL INTO A NEW

APPLICATION PROGRAM	8-1
8.1 The <i>SARLIB</i> Interface Routines	8-1
8.1.1 An Example <i>SARLIB</i> Routine	8-2
8.1.2 <i>SARLIB</i> Utility Routines	8-2
8.1.3 <i>SARLIB</i> Error Handling	8-3
8.1.4 <i>SARLIB</i> Data Structures	8-3
8.2 Using <i>SARLIB</i> in an Application Program	8-6

Part II Natural Language Understanding

CHAPTER 9. NATURAL LANGUAGE UNDERSTANDING SYSTEMS	9-1
9.1 Natural Language Interfaces	9-1
9.1.1 Approaches to Language Understanding Systems	9-2
9.1.2 Processes of Language Analysis	9-2
9.2 Syntax Analysis	9-4
9.3 Semantic Analysis	9-5
9.4 Discourse Analysis and Knowledge Representation	9-6

CHAPTER 10. SYNTAX ANALYSIS AND GRAMMARS FOR

NATURAL LANGUAGE SUBSETS	10-1
10.1 Words and Word Categories	10-1

10.2 Context-free Grammars and Parsing	10-2
10.2.1 Context-free Grammars	10-2
10.2.2 Parsing	10-6
10.2.3 Recursive Descent parsing	10-8
10.2.4 LALR(1) Parsing	10-10
10.3 Augmented Transition Networks	10-14
CHAPTER 11. A NATURAL LANGUAGE PARSING SYSTEM	11-1
11.1 A Recursive Descent Parser	11-1
11.2 The <i>LALR</i> Compiler Generator	11-3
11.2.1 An <i>LALR</i> Skeleton Using <i>Turbo Pascal</i>	11-4
11.3 A State Machine Parser Using <i>LALR</i>	11-5
11.4 A Parallel Parser	11-7
11.5 Adapting <i>LALR</i> for Use with Augmented Grammars	11-14
11.5.1 The Grammar for <i>LALR</i>	11-15
11.5.2 Grammar Augmentation Using Descriptors	11-16
11.5.3 Building an Augmented Parser	11-20
CHAPTER 12. THE DICTIONARY	12-1
12.1 The Choice of Words for the Dictionary	12-1
12.2 A Small Dictionary	12-1
12.2.1 Prefix Processing	12-3
12.2.2 Suffix Processing	12-4

12.3 A Larger Dictionary	12-8
12.3.1 Word Features	12-9
12.3.2 Suffixes and Intermediate Forms	12-17
12.4 Problems with this Dictionary Design	12-25
12.5 Dictionary Enhancements	12-28
12.6 Semantic Information	12-29
CHAPTER 13. DICTIONARY IMPLEMENTATION	13-1
13.1 The Structure of the Dictionary	13-1
13.1.1 String Storage	13-1
13.1.2 Dictionary Entry Storage	13-2
13.1.3 Static Two Character Indexing	13-5
13.1.4 Dynamic Three Character Indexing	13-11
13.2 Affix Processing	13-14
13.2.1 Affix Transformations	13-14
13.2.2 Affix Transformation Storage	13-19
13.2.3 Affix Removal	13-20
13.3 Semantic Information Storage	13-24
13.3.1 Auxiliary Verbs	13-25
13.3.2 Affix Transforms	13-25
13.3.3 Meaning Field	13-27
13.3.4 Inter-word Links	13-27
13.3.5 Semantic Reference Field	13-28

13.4 Dictionary Files	13-28
13.4.1 Dictionary File Format	13-28
13.4.2 Compiling the Dictionary	13-30

CHAPTER 14. SEMANTIC ANALYSIS AND KNOWLEDGE

REPRESENTATION	14-1
14.1 Canonical Primitives	14-1
14.1.1 Verbs	14-1
14.1.2 Nouns and Adjectives	14-2
14.2 Object Oriented Semantic Actions and Descriptors	14-4

Part III Conclusion

CHAPTER 15. FINDINGS AND FUTURE DIRECTIONS	15-1
15.1 Summary of the Final Results	15-1
15.2 Future Enhancements	15-3
15.3 Construction of Natural Language Programming Tools	15-6
15.4 Conclusions	15-7

BIBLIOGRAPHY

APPENDIX A: Parts of Speech for English

APPENDIX B: *LALR* Skeleton for a Parallel Parser

APPENDIX C: *LALR* Generated Parallel Parser for an English language Subset

APPENDIX D: An ATN Grammar

APPENDIX E: Augmented Grammar for an English Language Subset

APPENDIX F: Grammar for the Augmentation language

APPENDIX G: English Prefixes

APPENDIX H: English Suffixes

APPENDIX I: Suffix Transforms and their Effects

APPENDIX J: Suffix Rules and their Transforms

APPENDIX K: SARLIB Library Routines and Error Codes

LIST OF FIGURES

Figure 1.1	Speech Controlled System	1-6
Figure 2.1	Speech Recognition with Feedback Between the Analysis Stages	2-2
Figure 2.2	Fourier Analysis Using Bandpass Filters and Energy Detectors	2-5
Figure 2.3	Matrix of Speech Sound Similarity Between Reference and Unknown	2-8
Figure 2.4	Total Accumulated Distance Matrix.	2-9
Figure 2.5	A Simple Speech Synthesiser	2-10
Figure 2.6	<i>SAR-10</i> Data and Status Ports	2-16
Figure 2.7	Writing data to the <i>SAR-10</i>	2-17
Figure 3.1	<i>Turbo Pascal</i> Block Skeleton	3-21
Figure 3.2	<i>Turbo Pascal</i> Repeat Loop Skeleton	3-22
Figure 3.3	<i>Turbo Pascal</i> Unit Skeleton	3-22
Figure 4.1	Robot Controller Command Language	4-2
Figure 4.2	Voice Controlled Robot Commands	4-3
Figure 4.3	Difference and Accumulated Distance Matrices	4-5

Figure 5.1 Adaptive Audio Filter and Noise Canceller	5-3
Figure 7.1 Use of the <i>VOICEDOS</i> Command Interpreter	7-3
Figure 7.2 <i>VoiceDosInstall</i> Memory Resident Routine Design	7-8
Figure 7.3 <i>VoiceDosIsr</i> Interrupt Service Routine Design	7-10
Figure 8.1 Recognise First Speech Reference Pattern Candidate Algorithm	8-2
Figure 10.1 Subtree for Parsing the Production $A \rightarrow X Y Z$	10-5
Figure 10.2 Context-free Grammar for a Subset of English	10-5
Figure 10.3 Parse Tree for the Sentence: <i>The small tabby cat scratched the baby with a claw</i>	10-6
Figure 10.4 Block Diagram of a Parser	10-7
Figure 10.5 A Recursive Descent Parser	10-9
Figure 10.6 Building a State Machine Parser	10-12
Figure 10.7 Operation of an LR parser	10-12
Figure 10.8 Equivalent Context Free and RTN Grammars	10-15
Figure 10.9 An ATN for a Sentence	10-16
Figure 10.10 An ATN for a Noun Phrase	10-18
Figure 10.11 An ATN for a Preposition Phrase	10-19

Figure 11.1 Recursive Descent Parser State Skeleton	11-2
Figure 11.2 Error Correcting Parser	11-6
Figure 11.3 Error Correcting Parser Algorithm	11-7
Figure 11.4 Switching Shift-Reduce Parsing Algorithm	11-9
Figure 11.5 Context Record for a Parallel Parser	11-10
Figure 11.6 Context Switching Parallel Parser Algorithm	11-12
Figure 11.7 A Parallel Parser	11-13
Figure 11.8 Augmentation List Notation	11-18
Figure 11.9 Segmenting the Sentence ATN	11-19
Figure 11.10 Some Augmented Grammar Rules	11-20
Figure 12.1 Word Feature Values	12-10
Figure 13.1 Dynamic Data Storage Structure for a Dictionary Entry	13-4
Figure 13.2 Linked List Dictionary Structure	13-5
Figure 13.3 Static Single Character Indexing	13-6
Figure 13.4 Static Two Character Indexing	13-7
Figure 13.5 Number of Entries Indexed by Each Array Element	13-9
Figure 13.6 Linked List Length Distribution	13-10
Figure 13.7 Dynamic Three Character Indexing	13-12
Figure 13.8 Data Structure to Hold Altered Words Information	13-21
Figure 13.9 Dictionary Entries	13-29
Figure 13.10 Dictionary Entries in Compact Format	13-29

Figure 14.1	Semantic Descriptor Tree for Nouns and Adjectives	14-3
Figure 14.2	Alternative Treatment for Concrete Objects	14-4
Figure 14.3	Semantic Object Library Implementation	14-6
Figure 14.4	Register Structure for a Sample Sentence	14-7
Figure 14.5	Register Structure Augmented by Semantic Objects	14-8

LIST OF TABLES

Table 2.1 <i>SAR-10</i> speech recognition and audio response specifications	2-12
Table 2.2 <i>SAR-10</i> speech recognition commands	2-14
Table 2.3 <i>SAR-10</i> audio response commands	2-15
Table 2.4 <i>SAR-10</i> control and testing commands	2-15
Table 12.1 Prefixes Used to Modify Root Words in the Small Dictionary	12-4
Table 12.2 Suffixes Used to Modify Root Words in the Small Dictionary	12-7
Table 12.3 Word Derivation Table - Part 1	12-18
Table 12.4 Word Derivation Table - Part 2	12-18
Table 12.5 Word Derivation Table - Part 3	12-19
Table 12.6 Word Derivation Table - Part 4	12-20
Table 12.7 Complex Versus Simple Transformation Rules	12-21
Table 12.8 Singular Transformation Rules	12-24
Table 13.1 Memory Occupied by Word Feature Sets	13-3
Table 13.2 List Length Data for Static 2-Character Indexing	13-13
Table 13.3 Effect of Maximum List Length on Actual List Length Distribution	13-14
Table 13.4 The Suffix <i>+ion</i> and Some of its Variants	13-16
Table 13.5 The Suffix <i>+ar</i> and Some of its Variants	13-17

Table 13.6 The Suffix <i>+ary</i> and Some of its Variants	13-18
Table 13.7 The Suffix <i>+meter</i> and Some of its Variants	13-19
Table 13.8 Examples of the <i>+ary</i> Transformation	13-26
Table 13.9 Example of the <i>+smanship</i> Transformation	13-26
Table 13.10 Examples of the <i>+meter</i> Transformation	13-27

1. INTRODUCTION

The benefits of being able to interact with a computer via spoken language are obvious and extremely attractive, speech being the most natural means of communication for human beings. Initial language skills are acquired painlessly by most people, and communication is carried on with little apparent effort, in contrast to the difficulty experienced in learning other languages later in life. The acquisition of non-verbal language skills is also difficult, as evidenced by the labour involved in learning to type or use Morse code, or the development of proficiency in computer languages. Thus, it is unfortunate that the primary means of communication with computers is limited to the less natural skills of reading and typing rather than listening and speaking.

There seem, then, to be considerable advantages in producing machines which can respond reliably to the means of communication most comfortable for humans, rather than humans being required to accommodate themselves to the artificial requirements of machines. However, the optimism of early workers in the area of natural language communication with computers was soon damped when the true magnitude of the task was realised. Initial aims of producing fluent understanding by machine of human language, either written or spoken, gave way to the more modest goal of achieving halting communication using a small subset of a natural language [MARK93].

Linguistic theory, advanced in the description of human communication [ROBI79] [ROBI80], had not until then, come up against the requirements of precision, completeness and freedom from ambiguity needed for implementation of workable automatic systems. The sheer computing power needed to carry out tasks that the human brain seems to take in its stride was severely underestimated. In particular, the limited nature of computing equipment - finite storage capacity, low speed of processing and retrieval from storage - proved to be major handicaps when attempting to cope with

language structures that are very large - possibly infinite - in the number of possible constructions and at the same time frequently imprecise and ambiguous in usage.

Recent natural language research has concentrated on achieving far more modest goals than the early dreams of automatic translation of speech from one language to another in real time. Even the more limited aim of translation of written language proved to be relatively impractical, without having to cope with the vagaries of different speakers' voice characteristics, speaking rates, pronunciation and dialect.

Progress has been made in speech recognition hardware that copes with a limited vocabulary and with isolated words and phrases rather than continuous speech. Such equipment, initially built using discrete electronic devices, is now produced economically using advanced digital signal processing microprocessors. The reverse process, that of producing speech from textual information, while not as difficult as the recognition of speech, can also be carried out using equipment of similar complexity. A complete voice control system ideally provides facilities for speech output as well as speech input.

A SAR-10 speech recognition and audio response system was acquired by Footscray Institute of Technology (now Victoria University of Technology) Department of Electrical and Electronic Engineering in 1986. It was decided to explore its potential by applying it to a task where its limited capabilities could be of real benefit - the use of voice control as an aid to physically disabled people, using only low-cost, easily obtainable equipment. Even with a limited vocabulary and a slow response time, the system could assist someone otherwise forced to use a far less convenient means to control and communicate with their environment.

A simple speech controlled computer and robot system was constructed, and its limitations soon became apparent. The project then became one of seeking means of overcoming these limitations, and of improving its performance and general applicability.

The work branched down several paths; real-time control of manipulators, natural language understanding, automatic parser generation, control of environmental context, operating systems and user interfaces for effective voice control, and development of specialised software tools to assist in producing code for natural language understanding systems. Chapters 5 to 14 describe this work.

The aim was to eliminate use of a keyboard and screen entirely, except for maintenance and modification of the software, unless even these functions could be carried out by voice control. The traditional arrangement of the personal computer could be replaced by a physical device far more amenable to the needs and environment of the users it serves. The logical extension of this is to compile the entire system onto silicon and use it as an embedded controller in the devices presently controlled by separate computer equipment.

The task was approached in two phases. First, an investigation was made into obtaining the maximum possible performance out of the limited facilities provided by the speech interface. The second phase involved extending the system by incorporating a natural language understanding capability.

The overall goal of the research was to determine the feasibility of using a voice controlled system to aid a physically disabled person. The simplest way was to use voice to control existing computer applications, such as word processing, spread sheet operation, operating system tasks, programming, and the use of an expert system. This work shows that these applications are amenable to speech control, although proving rather restrictive to an able bodied person who has the option of typing or using a mouse. This work is described in Chapter 3.

For speech control to be of maximum benefit to a disabled person in the ordinary tasks of living, it must carry out some of the physical tasks otherwise denied to the user, such as turning the pages of a book or handling a cup of tea. To this end, the speech recognition

hardware and software were used in 1986 to control a small industrial robot arm, a Rhino educational robot, as described in Chapter 4, and experiments carried out to determine the possibilities and problems inherent in such an arrangement. It was soon discovered that given the state of the art then, this was a risky process for the disabled user. However, the identification of these dangers, and of possible approaches to their solution, are considered to be a useful outcome of this work.

The use of a robot in such a way proved not only dangerous but also tedious. Consequently, a further goal was defined: to expand the facilities provided and improve the performance of the system using suitable software.

Various methods of improving the performance of the speech system were tried, such as pre-processing of the speech signal, clustering of small vocabularies into distinct groups and enabling only those groups relevant to a particular context, and manipulation of the speech control parameters under program control. The control of vocabulary clusters proved to be particularly successful, greatly enhancing the reliability of command recognition, as shown in Chapter 5.

Enhancement of the speech control system was carried out by the development of operating system extensions in the form of the *VOICEDOS* memory resident utilities to control the system, and construction of the *SARLIB* library of speech control and audio response routines, as described in Chapters 6 and 7. The memory resident utilities enable the speech system to control existing application programs for which the source code is not available, since no modifications to the application program are needed. On the other hand, the library routines can be incorporated into new applications, and allow the full range of speech control and audio response capabilities of the speech system to be accessed.

The second phase of the research involved the development of a natural language understanding facility. Such a capability removes some of the artificiality of adapting one's speech to a limited vocabulary isolated word and phrase system. To this end, a suitable grammar, a parallel parser, and a dictionary system were produced. With these, a flexible natural language understanding system was constructed, as described in Chapters 9 to 14. It can be used with the speech control system, or as a text based system using keyboard control. The full potential of this part of the work will be especially realised when a better speech recognition system becomes available.

Apart from investigating the feasibility of speech control for disabled users, and the production of the practical systems for speech control and natural language understanding, an important spinoff from this research has been the construction of powerful tools to support the development of such systems. These include the memory resident operating system extensions, and the speech control and audio response library routines. In addition, methods and tools for rapid and automatic production of natural language parsers have been produced. These greatly facilitate the investigation of such techniques, and will form the core of the author's future research in this area.

Figure 1.1 summarises the system developed in this project.

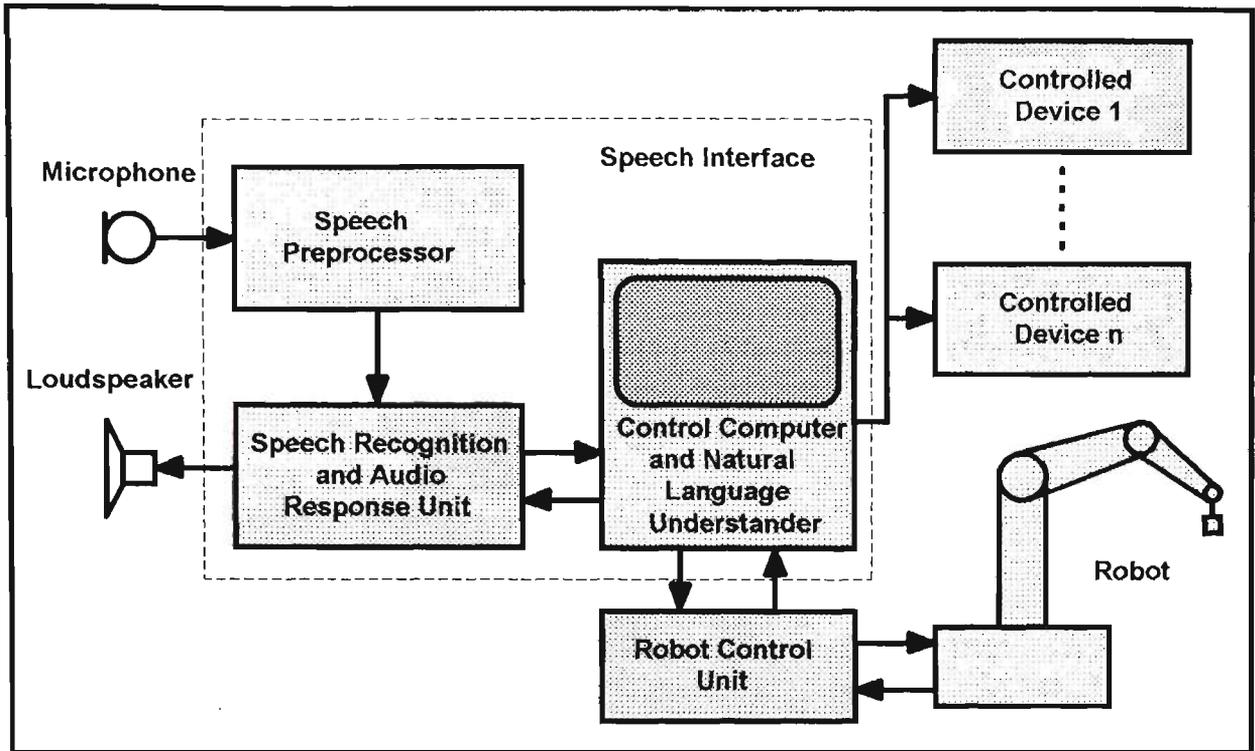


Figure 1.1 Speech Controlled System

Part I

Voice Control

2. THE VOICE CONTROL SYSTEM

This chapter describes two methods of speech recognition, and an implementation of one of these in a particular speech control system.

2.1 Two Approaches to Speech Recognition

The two approaches to achieving a useful speech recognition system are the recognition of continuous speech, or, as used in this project, isolated word or phrase recognition.

2.1.1 Continuous Speech Recognition

Continuous speech recognition is achievable with today's very fast processing systems, but is still a formidable and expensive task. In normal speech a person's pronunciation is often careless, speaker differences are significant, speaking rates vary greatly, and co-articulation effects (the way the meaning of a sound depends upon the sounds preceding and following it) exist within and between words. In addition, the importance of a word within a sentence affects its intonation and stress, and has an affect on how it is articulated. When listening to a speaker, it is extremely difficult to detect the word boundaries. Looking for periods of silence is not a reliable guide; sophisticated, and even intuitive, knowledge is required. This knowledge is difficult to codify into an automatic recognition algorithm. Any form of reliable continuous recognition requires the simultaneous application of syntactic and semantic analysis. If the component words of an utterance were already known, the syntactic and semantic information they contain could be used to aid the further recognition of words in the speech stream. But as this word extraction is what is being attempted, a circular process is involved. It becomes clear that a system of analysis based on only partially determined knowledge, and

complex feedback paths, is involved [AINS76]. The recognition becomes a recursive process, as illustrated in Figure 2.1.

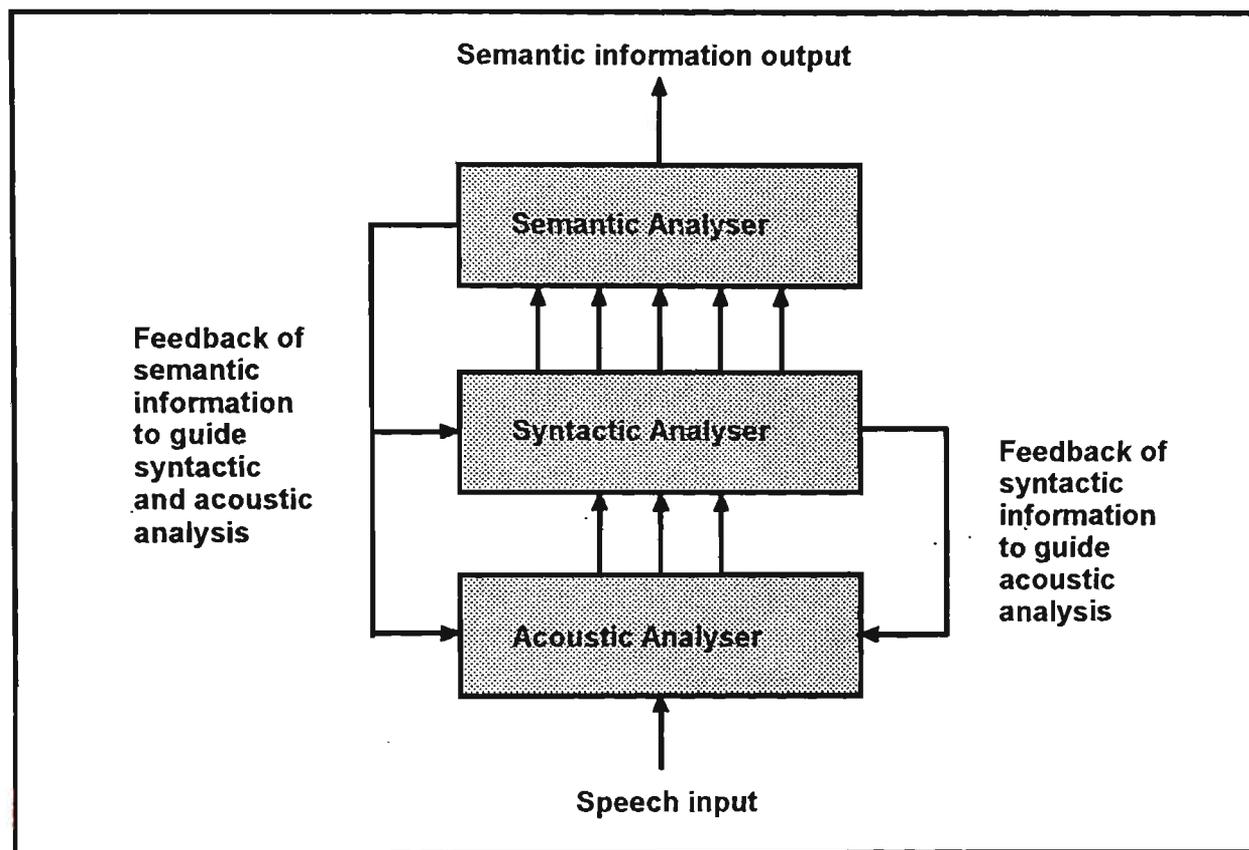


Figure 2.1 Speech recognition with feedback between the analysis stages.

2.1.2 Isolated Word and Phrase Recognition

One simplification that makes a recognition system achievable using relatively simple, inexpensive equipment is to limit the recognition task to a fixed vocabulary of isolated words or phrases. The term *isolated* here means that each word or phrase is separated by a distinct period of silence long enough to be detected and to allow the recognition process to be completed before the next utterance arrives.

Most isolated word recognition systems are based on an acoustic analyser followed by some form of pattern classifier. The acoustic analyser attempts to extract from the input speech certain features which are used to form unique identifiers capable of distinguishing that utterance from all other utterances received. With the vagaries of human spoken languages, and the variations and inconsistencies between different peoples' voices and pronunciation, this is, at present, impossible unless severe restrictions are placed on the total system. These restrictions may take the form of a reduced vocabulary, a limitation on the number of users, or even a restriction on use of the system to one particular person for whom it has been trained. Limitations on the way the user may talk are also likely.

Since a major goal of the use of speech recognition systems is to make communication between humans and machines more natural and efficient, it is clear that any such restrictions are highly undesirable. Unfortunately, given the present state of the art of automatic speech recognition, they are unavoidable, especially in low cost systems. The aim of the present project is, working within the limitations of the available equipment, to achieve a user interface to a computer which has maximum usability while hiding the restrictions as much as possible.

2.2 Acoustic Analysis for Isolated Word Recognition

Fourier analysis of human speech [AINS76] reveals that voiced sounds, such as vowels and nasals, are produced by a train of pulses at a fundamental frequency corresponding to the vibration of the larynx. The spectrum of this oscillation is shaped by two major formants (resonances in the frequency response), the frequencies of which are largely controlled by the position of the tongue and the shape of the mouth. Other formants and anti-formants (absence of resonance) can also be significant for the production of certain

sounds. Non-voiced sounds, such as fricatives, are generated by the turbulent flow of air over a constricted larynx, rather than a more periodic vibration of the larynx.

Thus, important information for the identification of the component sounds of a speech pattern is found in the fundamental frequency of the larynx vibration, the frequencies of the major formant peaks and nulls, and the relative sound energy contained in the various frequency bands. This information can be used to identify the component sounds, or phonemes, in an utterance.

Phonemes are radically affected by the sounds preceding and following them, complicating the recognition process. However, even with this simplified approach it is still possible to achieve low cost automatic recognition. Much of the complexity is then moved into the processing phase following the extraction of sound features.

The usual method of quantifying the sound features mentioned above is to slice the incoming speech signal into short time periods and to perform a Fourier analysis on each sample. This gives a summary of the principal features as they vary with time. This process is carried out relatively simply, efficiently and inexpensively using Digital Signal Processing (DSP) microprocessors synthesising a bank of bandpass filters followed by peak detectors to determine the energy content in each frequency band, as shown in Figure 2.2.

A more direct method divides the signal into two or more frequency bands using low-pass, band-pass and high-pass filters, then counts the zero crossings to determine the frequencies of the fundamental and the formants, and uses detectors to measure the energy content in each band. A workable model of just such a simple system was developed by Davis, Biddulph and Balashek in 1952, reported by Ainsworth [AINS76].

In practice, feature extraction needs to be considerably more complex than these simple representations indicate. Preprocessing, such as filtering, noise cancelling and gating, thresholding and logarithmic compression, to enhance the signal to noise ratio of the speech, is needed to increase the dynamic range handling ability of the system, and reduce the effect of interfering signals or redundant components of the wanted signal. In addition, some method of extraction of features related to unvoiced components of the speech might be included.

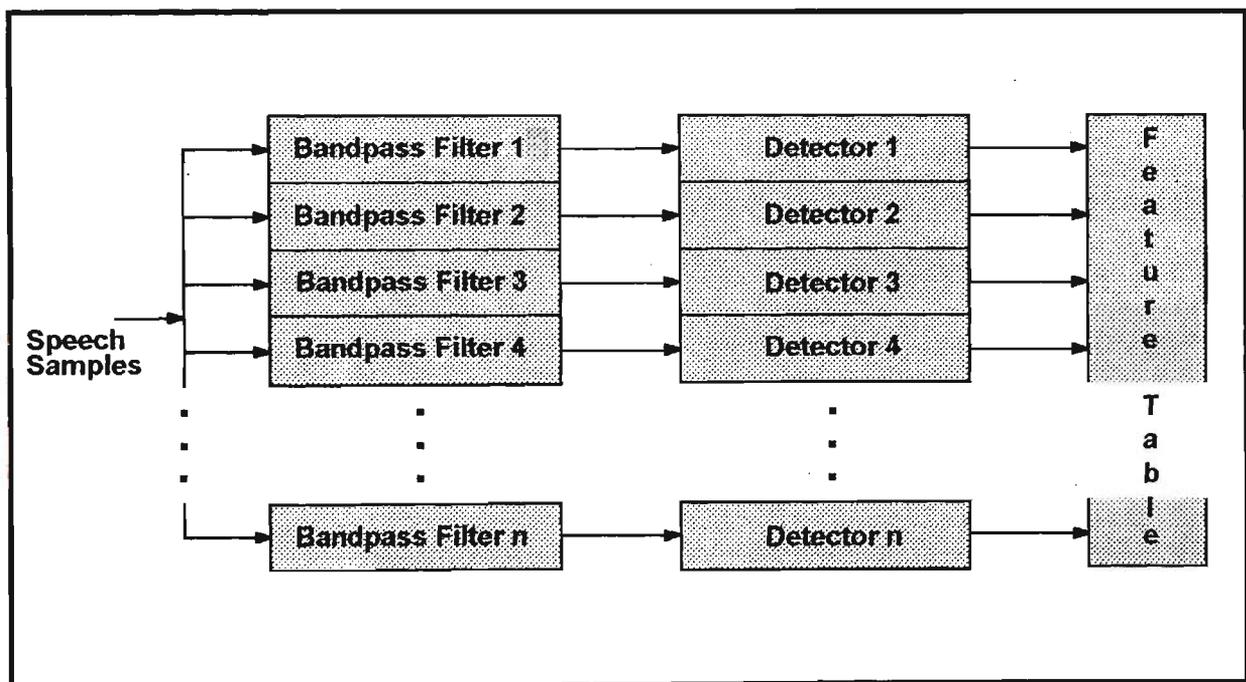


Figure 2.2 Fourier analysis using bandpass filters and energy detectors

Acoustic analysis results in a table of speech features against time. Once the system is trained by speaking into it all of the utterances to be recognised, these features, or a suitable statistical summary of them, are used as templates against which future utterances are compared. The difference between the stored features and input speech features is quantified to give a measure of how well the input matches any of the trained

words. If a close enough match is achieved then the input utterance is said to be recognised, and it is assumed to be the same as the trained word against which it was measured. What constitutes a *close enough match* is discussed later.

The key to successful isolated word recognition is finding a set of words or phrases (a *vocabulary*) in which each entry is as different as possible. If every entry matches a different template, then the vocabulary possesses what I have chosen to call *orthogonality* (analogous to the orthogonality of vector space basis vectors in mathematics). Orthogonality will be given a more complete definition in Chapter 4, when the robot control language is described.

The method of measuring the difference between input utterances and trained utterances is the source of a number of difficulties, which will be discussed when practical application of the SAR-10 system is considered in Chapters 3 and 4.

Analysis of the features of incoming speech during the training phase provides information useful in adapting the preprocessing parameters so that the signal to noise ratio of the input is improved. Also, once the system has been trained and is in use, an adaptive process based on the features extracted from the new speech patterns enables the system to adapt to a different speaker from the one who initially trained the system. Additionally, updating of the trained templates and processing parameters is feasible using the new information. These techniques also assist in coping with a changing external environment and variations in a speaker's voice due to fatigue and other factors. This adaption has been tested under operator control, and improved the performance of recognition to some extent. Automatic adaption will be explored in a future version.

2.3 Dynamic Time Warping

To be useful a speech recognition system needs to be tolerant of reasonable variation in a user's speaking rate. One problem encountered with template matching schemes is that of normalising each utterance to the same length in the time domain.

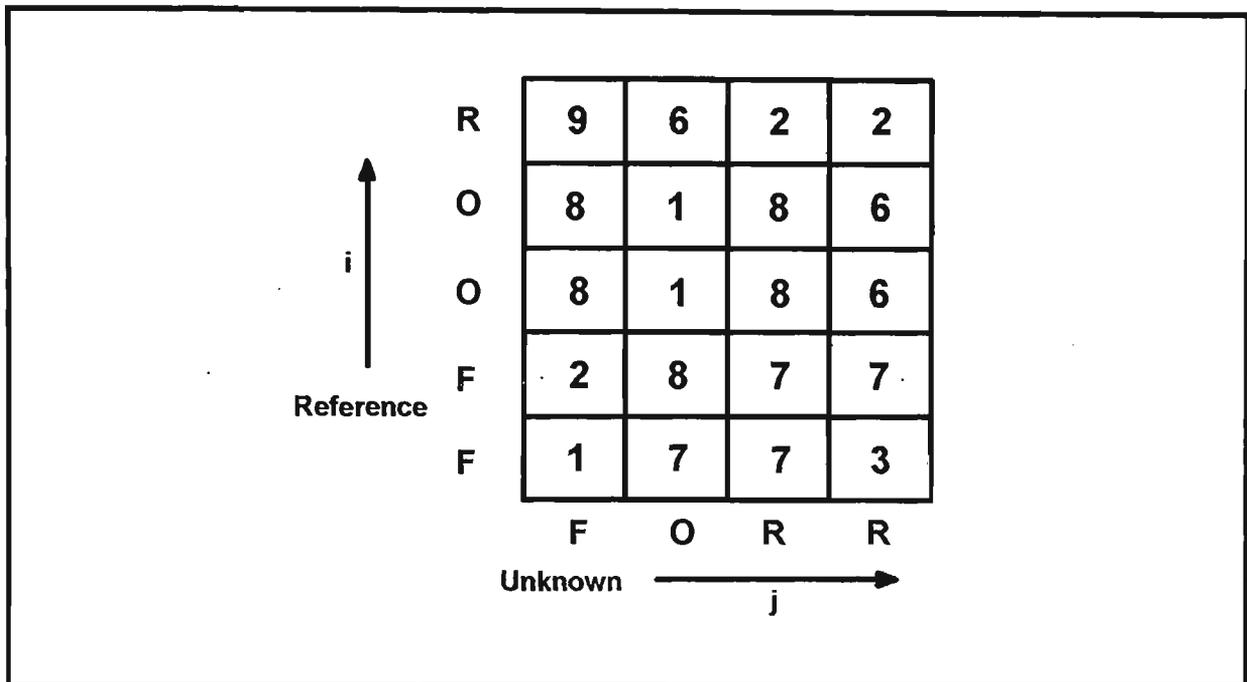
One approach used is a dynamic programming algorithm called *dynamic time warping*, described by Smith and Sambur [SMIT80]. This is used in the SAR-10 Speech Recognition and Audio Response system. A simplified explanation of how it functions, adapted from Smith and Sambur, will help later when the problems produced by the fact that dynamic time warping can result in very different utterances producing similar or identical templates are considered.

Utterances are divided into fixed length time frames and analysed to extract measurable features on which to base comparisons. A matrix method allows comparison between reference templates with different numbers of time frames. As an example, suppose that the input utterance consisting of the word "four" has been analysed to have four time frames, representing (in a purely hypothetical case) the phonemes **FORR**. Further, suppose that the reference pattern is stored as a five frame sequence representing the phonemes **FFOOR**. To produce a measure of the *distance* between these two patterns, a matrix is set up whose entries represent $\text{DIST}(i,j)$, the difference between the reference at time i and the utterance at time j , as determined by some suitable measure from the extracted acoustic features of each pattern. An example of such a matrix can be seen in Figure 2.3.

The shortest path from bottom-left to top-right of this matrix is a measure of the correlation between the reference utterance and the unknown utterance. The optimal path

to a point (i, j) in this matrix must pass through one of the points (i-1, j), (i-1, j-1), or (i, j-1). Therefore the minimum accumulated distance to point (i, j) is given by:

$$D(i,j) = \text{DIST}(i,j) + \text{Min}\{ D(i-1,j), D(i-1,j-1), D(i, j-1) \}$$



**Figure 2.3 Matrix of speech sound similarity
between reference and unknown**

These values are shown in the matrix in Figure 2.4. The dynamic programming algorithm recursively determines the minimum path length for the unknown against each reference in the vocabulary. The best acceptable match is chosen.

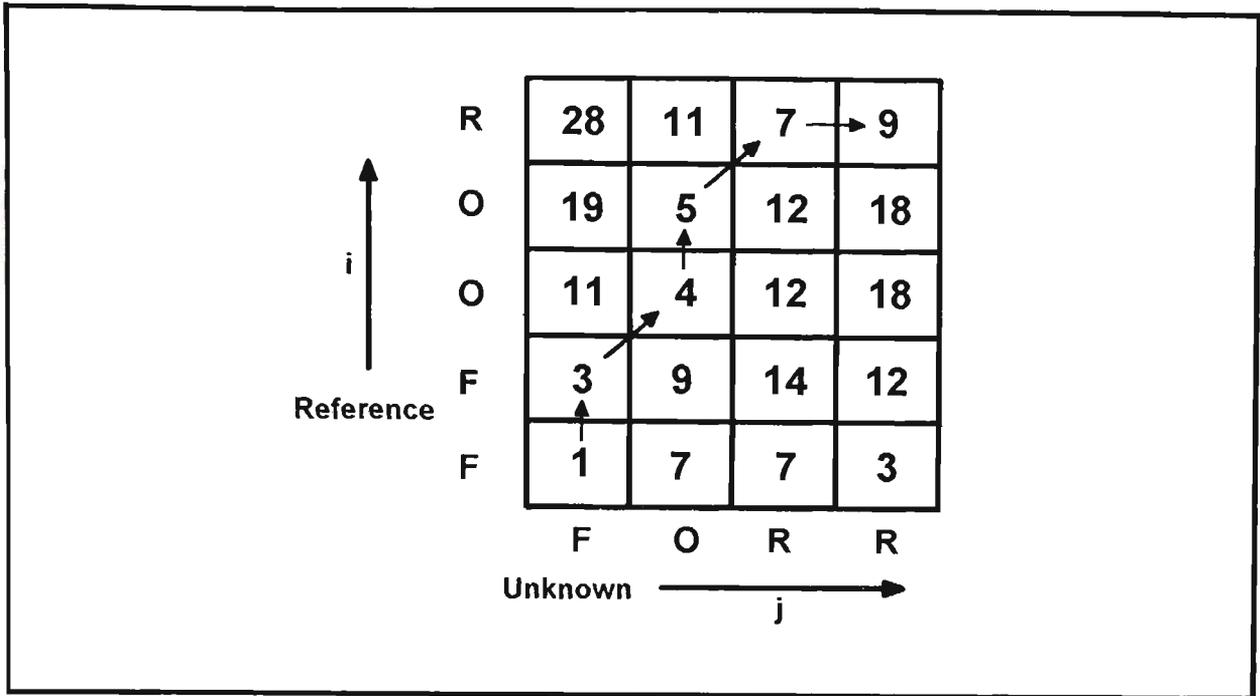


Figure 2.4 Total accumulated distance matrix.

2.4 Speech Output

Low cost speech output systems are approached in two ways, depending on the voice quality required and the allowed cost of the system. Inexpensive output systems use commercially available speech synthesis chips initially developed for the electronic games industry for educational toys such as spelling teachers and talking calculators.

These low cost speech synthesis chips are based on a simple digital to analogue converter and amplifier reproducing a string of phonemes constructed from stored digital patterns in a read only memory. Words are generated by outputting the phonemes in a suitable order using a simple address sequence generator or store. The output of such chips is typically characterised by a mechanical tonality and poor pronunciation, making the result difficult to understand and unpleasant to listen to for any length of time. The design of such a system is shown in Figure 2.5.

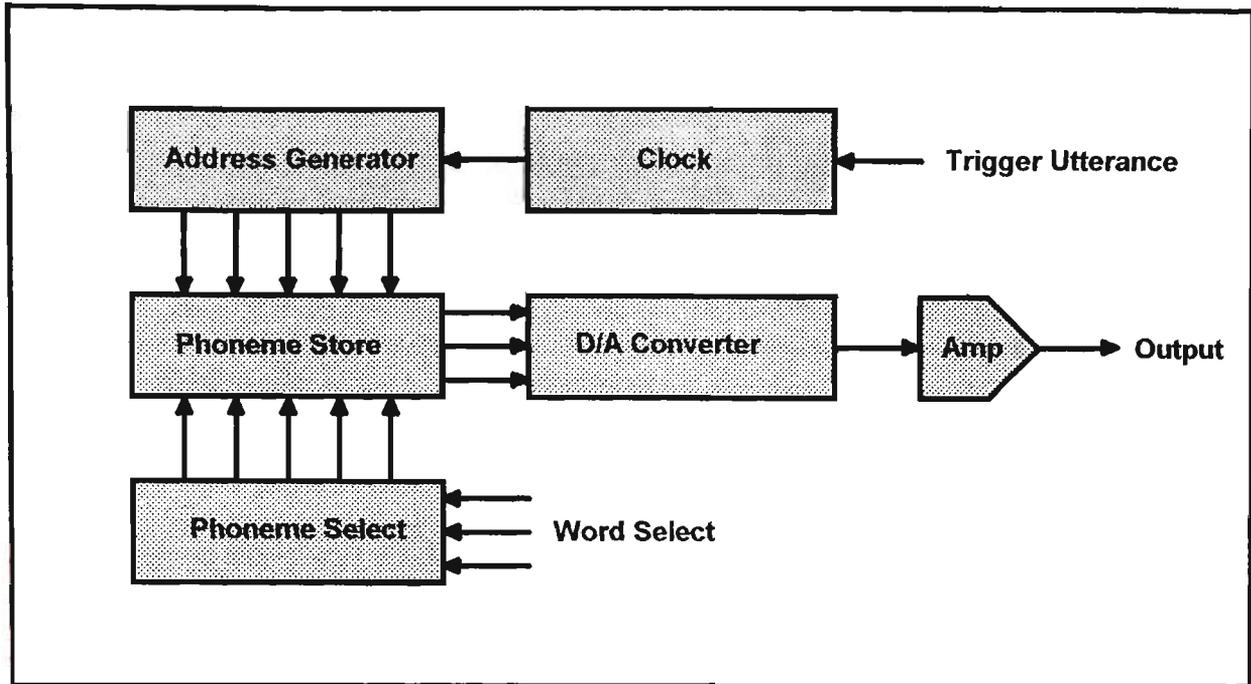


Figure 2.5 A simple speech synthesiser

An alternative approach, used in the SAR-10, is to store sampled human speech in the form of words or phrases. The stored speech can be output as required by addressing the start of the word or phrase and then incrementing through the addresses, passing the digital samples to a digital to analogue converter and amplifier. The resulting speech quality is as high as the number of bits stored per sample allows. The sound and pronunciation are completely natural, being those of the speaker who trained the system. Data compression saves storage, which can become a limitation if a large vocabulary is required.

The major trade-off between speech synthesis and speech recording is in the amount of memory required. A synthesiser only needs to store a relatively small number of phonemes, used over and over to construct a very large vocabulary. A speech recording device must store its entire vocabulary intact. Even if compressed, this represents a large amount of data. If the trained words are well chosen, combining the basic vocabulary items in different ways allows generation of a larger repertoire of phrases and sentences.

2.5 The Speech Control and Audio Response System

The hardware to implement the speech recognition and audio response interface is provided by a SAR-10 Speech Control and Audio Response System, manufactured by the NEC Corporation. The SAR-10 is based on an NEC DSP chipset and is produced as an add on card for the IBM Personal Computer.

2.5.1 The SAR-10 System

The SAR-10 speech recognition facility is based on a filter bank. Speech energy in a number of frequency bands is measured and compared with templates previously stored during a training phase. Templates are normalised using dynamic time warping, and template matching is performed using the dynamic programming algorithm.

High quality voice output is achieved in the audio response mode by storing the operator's voice in memory, using Adaptive Differential Pulse Code Modulation (ADPCM) [CARL86]. This ensures maximum intelligibility of speech output. Speech input can be from a microphone or a tape recorder, selected by setting a jumper on the card. The output is suitable for driving a 4 or 8 ohm loudspeaker to a level of 1 watt of audio power. Input and output connectors and an output level adjusting potentiometer are available on a panel at the rear of the card, while microphone and tape input levels can be adjusted on the card itself. The SAR-10 system appears as a pair of ports to the personal computer. The status and data port addresses and the interrupt level used are configured by setting a DIP switch and a jumper on the card.

Sufficient random access memory is provided on the SAR-10 card to hold up to 250 trained voice recognition templates and audio response patterns for up to 87 seconds of

speech. It has a facility to upload and download templates and patterns to the personal computer.

The specifications of the SAR-10 Speech Recognition and Audio Response System [NEC85] are shown in Table 2.1.

<p><u>Speech Recognition Specifications:</u></p> <p>Recognition vocabulary size: Up to 250 words or phrases. Recognition accuracy: Over 98%. Reject threshold: User programmable. Recognition clusters: Up to 250 clusters. Voice control capability: Microphone on/off, Change cluster. Utterance duration: 0.2 to 2 seconds. Pause between words: Minimum 250 milliseconds. Recognition response time: Within 0.25 to 0.5 seconds. Pattern matching: Dynamic programming matching. Number of training passes: User selectable. Input audio bandwidth: 200 to 5000 Hz.</p> <p><u>Audio Response Specifications:</u></p> <p>Coding method: ADPCM coding with compression coding for silent duration. Bit rate: 24/28/32 kbps, selectable. Response vocabulary size: Up to 250 words or phrases. Total response duration: Maximum of 87 seconds at 24 kbps. Maximum of 74 seconds at 28 kbps. Maximum of 65 seconds at 32 kbps. Output audio bandwidth: 250 to 4000 Hz.</p>
--

Table 2.1 SAR-10 Speech Recognition and Audio Response Specifications

2.5.2 SAR-10 Software

Programs provided with the SAR-10 card enable the system to be used as a stand alone speech recognition system without requiring any modifications to the user's applications. There is a utility program named VOICE PLUS for training, recognition and audio

response, a transparent keyboard handler program, and programs for installation and configuration.

The transparent handler program is a Terminate and Stay Resident [TSR] device driver which accepts recognised voice commands from the SAR-10 card under interrupt control and outputs pre-programmed ASCII strings to the application program as if they came from the keyboard. Using this handler program any application program becomes controllable by voice command. The VOICE PLUS utility program is used to train the SAR-10 to recognise the required voice utterances and provide the appropriate ASCII strings for each recognised utterance to emulate the appropriate keyboard responses. The transparent handler can then be activated, and the application program run.

While this is simple and convenient, it is limited to emulating the user interface provided by the application program. It is not ideal for use under exclusive voice control. If optimal voice control is desired, then it is necessary to control the speech recognition and audio response hardware directly from the application program. Such an approach is made possible by using SAR-10 commands.

2.5.3 SAR-10 Command Structure

The SAR-10 provides a set of commands for controlling all aspects of its operation. These can be called by user written application programs, and are summarised in Tables 2.2, 2.3 and 2.4.

These commands provide all of the facilities needed to gain complete control over the higher level aspects of the speech recognition and audio response processes, such as administration of sets of vocabularies and reference patterns, switching from one application environment to another, and tailoring the system for different users. Some

optimisation of lower level recognition performance can also be carried out by control of parameters such as the reject threshold. However the basic operation of the SAR-10 is fixed and not accessible to user manipulation.

The implementation of these commands into a library of functions suitable for calling from an application program is described in Chapter 8.

<ul style="list-style-type: none">Train a new reference patternUpdate a specified reference patternRecognise best speech candidateRecognise best and next best speech candidateStart recognition modeChange recognition clusterChange recognition reject thresholdInquire about recognition reject thresholdDelete all reference patternsDelete reference pattern(s) of a specified wordDelete last trained reference pattern of a specified wordDelete specified reference pattern of a specified wordUpload vocabulary table and reference patterns to PC (ASCII format)Upload vocabulary table and reference patterns to PC (binary format)Download vocabulary table and reference patterns from PC (ASCII format)Download vocabulary table and reference patterns from PC (binary format)Download vocabulary pattern from PCChange recognition parametersInquire about recognition parametersChange recognition flagsInquire about recognition flagsInquire about recognition status

Table 2.2 SAR-10 Speech Recognition Commands

Digitise input speech
 Record input speech
 Output audio response word(s)
 Output audio response macro(s)
 Define an audio response macro
 Inquire about an audio response macro
 Delete all audio response macro definitions
 Delete all speech patterns
 Delete speech patterns for a specified word
 Upload speech patterns to the PC (ASCII format)
 Upload speech patterns to the PC (binary format)
 Download speech patterns to the PC (ASCII format)
 Download speech patterns to the PC (binary format)
 Change audio response parameters
 Inquire about audio response parameters
 Inquire about audio response status

Table 2.3 SAR-10 Audio Response Commands

Inquire about error status
 Change format of response from SAR-10 to PC
 Inquire about format of response from SAR-10 to PC
 Change memory contents
 Dump memory data
 Test work memory
 Test recognition reference pattern memory
 Test audio response speech pattern memory
 Initialise the SAR-10
 Cancel command execution
 Pause in data transmission to PC
 Beep the SAR-10

Table 2.4 SAR-10 Control and Testing Commands

2.5.4 Communication Between the SAR-10 and the PC

- **Ports:** The SAR-10 appears to the personal computer as two 8 bit ports, a status port and a data port. Both of these ports can be read from and written to in order to send data and commands to the SAR-10 and receive back data and status

information. The status port provides the signals needed for handshaking. The data and status ports can be seen in Figure 2.6.

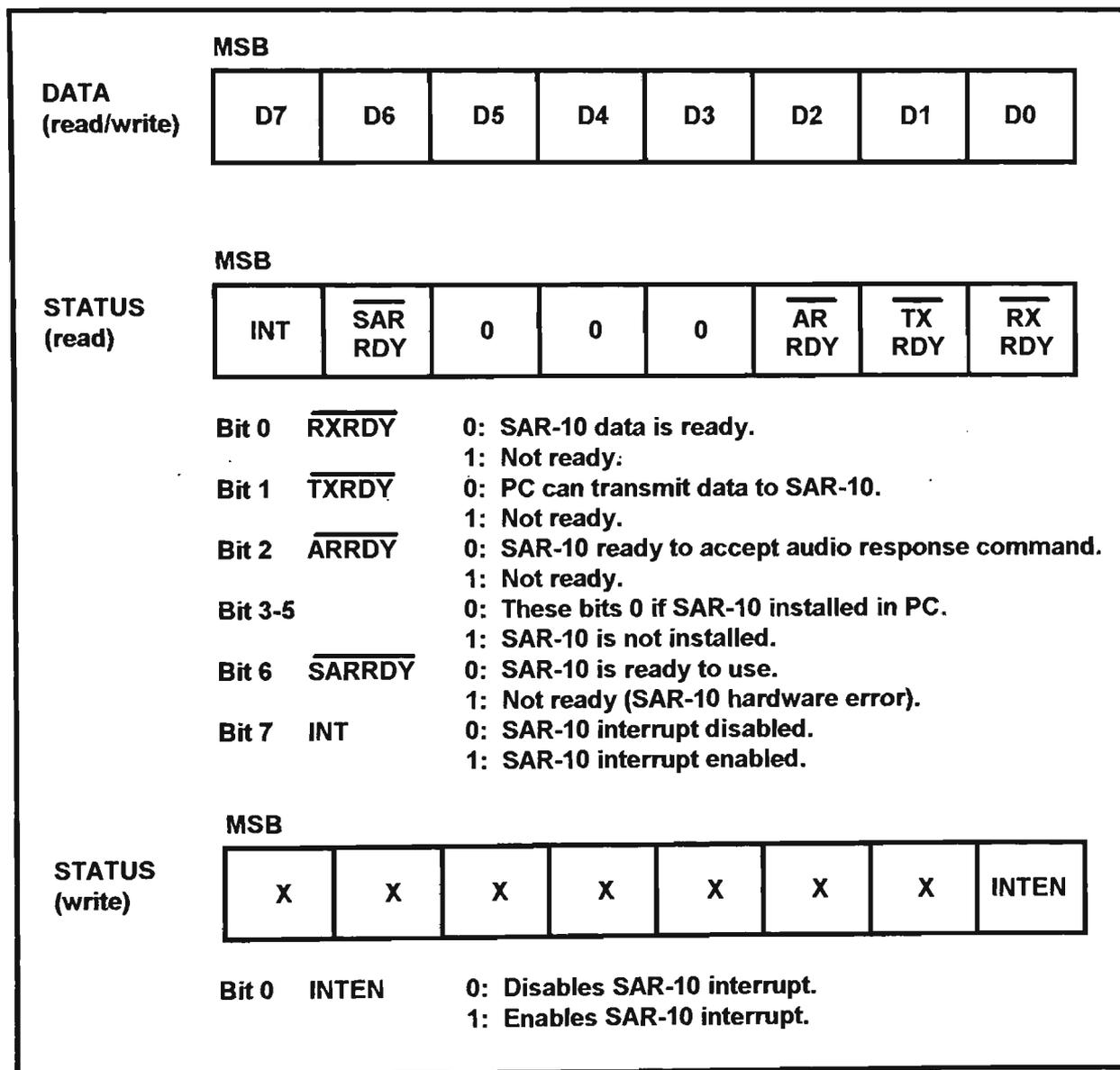


Figure 2.6 SAR-10 Data and Status Ports.

- **Communication Protocol** The communication protocol which used to write data to the SAR-10 is shown in Figure 2.7. Reading data from the SAR-10 is performed similarly.

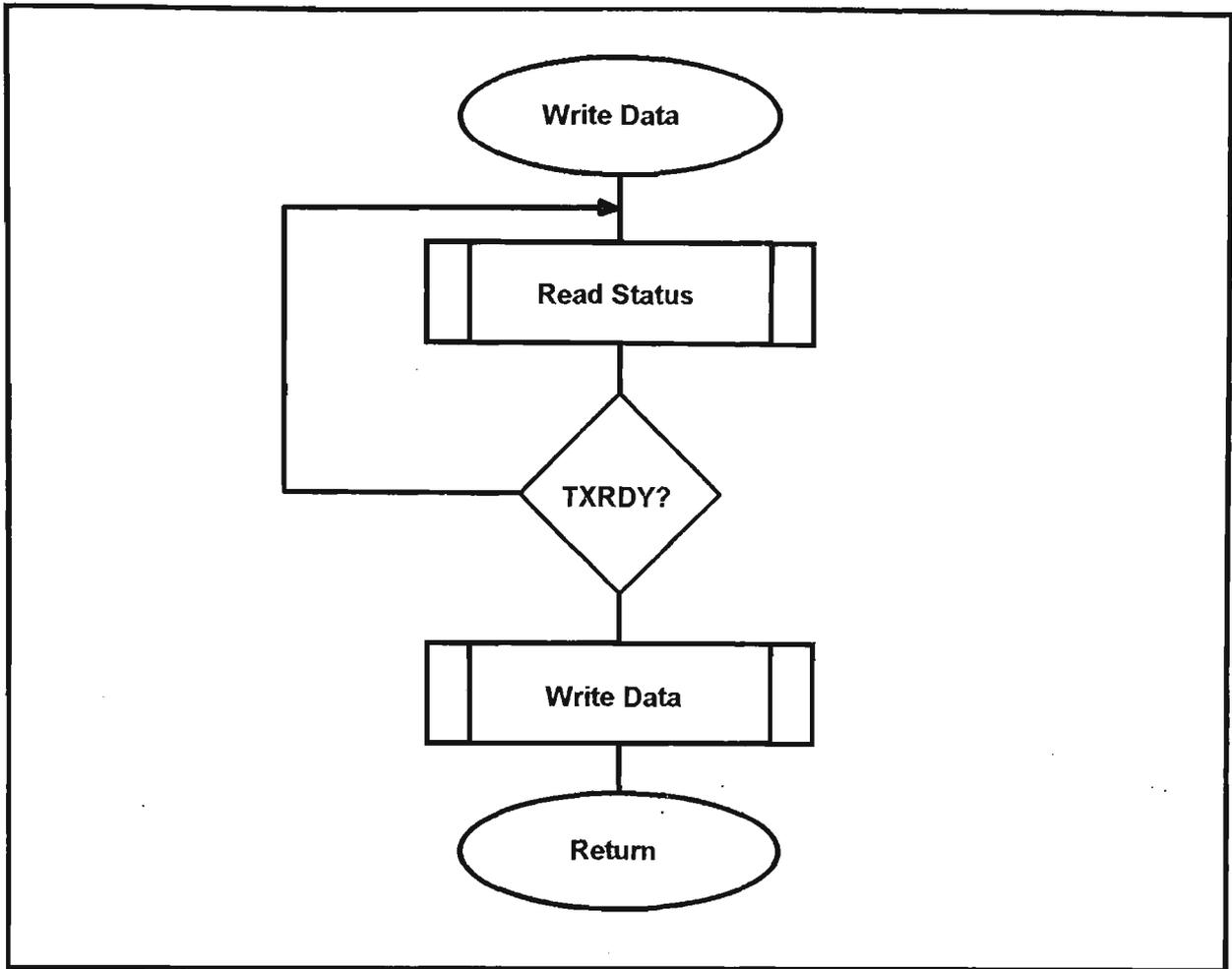


Figure 2.7 Writing data to the SAR-10.

Data may also be transferred from the SAR-10 to the PC under interrupt control. If the SAR-10 interrupt is enabled, the SAR-10 turns on the interrupt signal when output data is ready. After the PC reads the data, the SAR-10 turns the interrupt signal off again.

2.5.5 The SAR-10 Speech Recognition Function

Before the SAR-10 is able to recognise its vocabulary, it must be trained. During the training phase speech inputs are digitised and stored, and features of each utterance, called *reference patterns*, are extracted to form a template by which the utterance is

described. These templates are stored along with user supplied ASCII string descriptions to form a vocabulary table.

When the SAR-10 is recognising speech, the input is digitised and its features extracted. Then the reference patterns are compared with the trained reference patterns to find a suitable match. The vocabulary table entry of the pattern which gives the lowest recognition score, a measure of the *distance* between two patterns, is sent to the PC. The application program can then accept this as a suitable match, if its recognition score is low enough, or reject it.

At least one recognition pattern is needed for each vocabulary table entry. When it is trained with more than one speaker the SAR-10 achieves considerable speaker independence. The training can be carried out by a number of different people, and the SAR-10 will attempt to recognise them all. Even if only one person is to use the SAR-10, it is desirable to train each utterance several times to allow for variations of the user's voice.

There is a compromise inherent in this process. As the SAR-10 stores 250 reference patterns, if N words are to be trained then each word can be trained up to $250/N$ times. More training passes results in a smaller possible vocabulary. Another undesired result is the effect of increasing the number of reference patterns which are potential matches for each utterance. This decreases the recognition accuracy of the SAR-10. This problem will be considered later.

2.5.6 Speech Recognition Vocabulary Table

The SAR-10's speech recognition vocabulary tables contain, for each word, the following items of information:

- **Training message:** This is a message describing the utterance to be trained. It is written on the screen as a prompt so that the operator can know the content of the current vocabulary item being trained.
- **Recognition output code:** A character string or hexadecimal code which, when the SAR-10 is being used in keyboard transparent mode, is to be sent to the application program. It can consist of up to 32 of any of the ASCII character set.
- **Recognition cluster:** Vocabulary table entries can be grouped together into what are called *clusters*. Each cluster can be enabled or disabled under program control. This is useful for overcoming some of the limitations of the SAR-10's recognition performance, as will be described in Chapters 5 and 6. Up to 250 clusters are possible.
- **Reject threshold:** This is a value between 0 and 250 for the recognition score above which the utterance will be rejected as unrecognised.
- **Voice control code:** Control codes can be used to control certain functions of the SAR-10 by voice command. These include the ability to set, reset and change clusters and turn the microphone on and off.

2.5.7 Recognition Clusters

Speech recognition vocabulary entries can be grouped into clusters. The only entries checked against an input speech pattern during recognition are those in enabled clusters. It is desirable for recognition accuracy to have as few recognition *targets* as possible, so unused clusters can be switched off when not required, enabling a small number of active entries to serve for a larger actual vocabulary.

As clusters can be enabled and disabled under voice control as well as by program control, the clustering facility is advantageous even when using the SAR-10 in transparent keyboard mode with an unmodified application program. Appropriate voice

commands must be provided to switch in and out the required clusters. Cluster zero is always active, regardless of which other clusters are being used. This means that by putting the most fundamental and frequently needed commands into cluster zero the user can avoid the possibility of locking the system out completely. Up to six clusters can be selected at any time.

2.5.8 Speech Recognition Parameters and Flags

The SAR-10 has a number of parameters and flags which can be set to adjust recognition accuracy in different conditions and to control the functions of the card in speech recognition mode. They are as follows:

- **TH0 - Word beginning point detection threshold:** This is the level of input signal at which the SAR-10 decides that a new word has begun and starts digitising.
- **TH1 - Word boundary detection point threshold:** This is the level of input signal below which the SAR-10 decides that the end of an utterance has been reached and ceases digitising. TH0 and TH1 must be carefully selected to suit the prevailing noise conditions. The recommended values are given in Section 7.5.1.
- **THU - Matching score threshold for update:** When the SAR-10 is being trained in update mode (i.e. existing trained patterns are being improved rather than new patterns created), if the matching recognition score between the incoming speech and the old reference patterns already trained is greater than THU, the pattern which gives the worst match is overwritten by the incoming pattern.
- **LF - Self-learning flag:** This flag turns the self-learning function on or off. If self-learning is on, then the SAR-10 automatically updates reference patterns during recognition. The reference pattern is renewed by the incoming speech if its

recognition score is less than the self-learning threshold (THL). A small value of THL is recommended (10-20) to avoid too many cases of existing patterns with low recognition scores being overwritten by new patterns with larger recognition scores.

- **THL - Matching score threshold for self-learning:** The recognition threshold value below which new reference patterns will replace old reference patterns when the self-learning function is turned on.
- **VCF - Voice control flag:** This flag enables or disables the voice control function.
- **RJBZF - Reject buzzer flag for recognition:** This flag enables or disables the reject buzzer. This buzzer sounds whenever speech input is rejected as unrecognisable.
- **ECHBF - Echo back flag:** This flag enables or disables the echo back function. If this function is enabled then whenever the SAR-10 recognises an utterance it uses the audio response function to output an audio response. The response chosen is the vocabulary entry in the audio response vocabulary table which has the same entry number as the recognised speech recognition vocabulary table entry. This is useful while training to verify correct recognition while under voice control. If the echo back flag is disabled recognition proceeds without any automatic audio response.

2.5.9 The SAR-10 Audio Response Function

The SAR-10 is capable of recording speech, music, or any other sound, up to a maximum duration of 87 seconds, depending on the selected sampling rate. This total duration can be divided up into a maximum of 250 separate parts, each of varying length depending upon requirements. Once the sound has been recorded in the SAR-10 it can be output again, either in individual parts, or in combinations of parts. These are called response macros.

Recording of speech or other sounds can be carried out using two different commands, "digitise" or "record". The digitise command causes the SAR-10 to start recording when

speech is detected. When a silence of more than 0.5 to 2.0 seconds (set by the TEND parameter) is detected, recording stops. The record command records speech until it is commanded to stop. This command does not detect pauses in sentences.

Digitised speech is associated with a prompt string and an entry number in an audio response vocabulary table, similar to that used for speech recognition.

2.5.10 Audio Response Parameters

- **FS - Sampling frequency for digitising/recording:** The audio response sampling frequency can be set to 6, 7 or 8 kHz. The higher sampling frequency gives better audio output quality but shorter total response duration.
- **TEND - End detection time for digitising:** 0.5 to 2.0 seconds silence duration can be selected as follows:
 - TEND: 1 Time: 0.5 seconds
 - TEND: 2 Time: 1.0 seconds
 - TEND: 3 Time: 1.5 seconds
 - TEND: 4 Time: 2.0 seconds

It is recommended that the shorter detection times be used for short words or phrases and longer detection times for longer sentences.

3. SIMPLE APPLICATIONS OF THE SAR-10 SYSTEM

To investigate the practicalities of using the SAR-10 in transparent mode, its use with six popular application programs and utilities was explored. These were a word processor, a spreadsheet, a DOS shell, a memory resident keyboard macro processor, a programming environment, and an expert system shell. Most of these would be useful to the majority of disabled computer users, but the investigation into a voice controlled programming environment was carried out to illustrate the potential for this technology to open up, for a disabled person, the possibility of a career in programming.

3.1 Application Program Interfacing

The simplest way to use the SAR-10 Speech Recognition and Audio Response system is through its transparent keyboard handler. This can be used with any existing application program which expects its user input to come from the keyboard of the PC. Examples of such applications which could benefit from this approach are word processors, spreadsheets, database systems, CAD/CAM packages, educational programs, programming environments, network control systems, electronic mail and other communications programs.

The major advantage of using a transparent keyboard handler is not needing access to the source code of the application program. This means that commercial packages, for which source code is not generally available, can be used in association with voice control. Until practical and efficient voice control becomes commonplace, this is likely to be the most frequent mode of use of such systems. The alternative is that all applications will have to be written for a particular voice control system, or source code be supplied so that voice control facilities can be incorporated and the application recompiled. Neither option is particularly attractive.

One solution is to develop a standard protocol for the interaction between voice control systems and application programs, as has been done for video controllers, MIDI (Musical Instrument Digital Interface) interfaces [DEFU89], and other computer peripherals. A good example of this is the Windows environment, where standard device drivers are provided for peripherals independently of the particular application program being used. At the present experimental stage of voice control systems this is hardly a practical proposition, so most users will need to opt for transparent keyboard handlers.

An alternative approach is to incorporate voice control facilities into the operating system itself. Again, this requires a standard protocol to be developed, and programs which are to make use of the facilities would have their user interfaces designed accordingly. Such an approach does, however, remove the need for the application programmer to be concerned with the low level intricacies of communication with, and control of, a voice system. This alternative is explored in Chapters 7 and 8.

3.2 Word Processing

The operation of a word processor presents the user with three major activities:

- Entry of text.
- Navigation of the cursor within a file, and recognition of command sequences.
- Execution of recognised commands.

The use of voice control introduces an additional problem of distinguishing which of these activities is intended by the user at any time and keeping them separate. An investigation into the possibilities and problems of controlling the Micropro Word Star [MICR87] word processing package was carried out.

3.2.1 Entry of Text

The fast, accurate and convenient entry of text into a file is one of the most basic functions required of a word processing program. This immediately presents the user of a voice control system with a problem. The number of possible words which could be entered by the user is potentially enormous. In actual fact, though, the average person uses quite a small vocabulary in normal speech, perhaps 500 words, and a somewhat larger one for written text. However, in both speaking and writing the vocabulary far exceeds the capacity of the type of speech recognition system under consideration here.

A partial solution to the problem of vocabulary size is to train only the most common words used in typical text, and rely on the user spelling out any other more unusual words required. This is not a complete solution for two reasons. The first is that the set of common words needed will vary depending on the writing style of the user and the subject being written about. The second is that any resort to spelling out words immediately causes great inconvenience to the writer.

It is possible to alleviate the first problem by having different vocabulary sets available for different subjects and writing styles. The most appropriate vocabulary can be loaded by a utility program before the application program is run. Standard suggested vocabularies, such as Basic English [OGDE68], exist which cover the words typically used in relatively unsophisticated English text. One of these could be used, if the user is willing to put up with some cramping of freedom in his or her writing style.

Spelling words out is tedious. It is difficult to maintain creativity at any reasonable level when one has to consciously say each letter, perhaps repeating it when it is not immediately recognised, sometimes correcting mistakes when the recognition is inaccurate. All of this is quite apart from the need for accurate oral spelling, something

many people find difficult. In addition, after the last letter of each word is spoken and received, the word itself must then be verified and entered into the system.

The other inconvenience is that the user must remember which words are in the system's vocabulary and which words must be spelled out. This results in having to continually switch back and forth between thinking in complete words and in separate letters.

Using a transparent keyboard handler with a commercial word processor presents a particular problem of word verification before entry. If the user is to verify the correctness of a word visually, this can only be done by reading the text as it appears on the word processor screen. At this point the text has already been entered into the application program. Any necessary corrections can then only be carried out by using the word processor's editing commands. Spelling and style checkers can help in the finding of errors once the text has been entered. This is no different to the problem of typing errors, but correction is more difficult using voice control. It is possible to verify recognition aurally by using the SAR-10's echo back facility. If the audio response vocabulary is constructed and trained so as to correspond to the speech recognition vocabulary, then each utterance recognised can be repeated by the audio response facility. Unfortunately, if the word is entered letter by letter then the audio verification will also be letter by letter. There is no facility for the system to combine the letters into a word and verify that it is correct. As well as this, there is still the problem, as for visual verification, that the text has already been entered into the system, and cannot be easily altered.

The lower reliability of the SAR-10 when recognising short utterances, such as single letters, especially with some voices, requires some form of phonetic alphabet to be provided. This phonetic alphabet may be a standard one, as commonly used in radio telecommunications. On the other hand, there is no guarantee that a standard alphabet will work reliably with all speakers, and special phonetics may need to be devised for

particular users. Apart from the inconvenience of needing to learn a special set of phonetics, spelling words in this way is even more tedious than using the common letter names. If audio response is used for verification of phonetics, the system can be trained to echo the actual recognised letter, rather than its phonetic equivalent, thus providing some measure of improvement in operator convenience.

Some of the tedium could be reduced if the word processor program contained a continuous spelling checker which could automatically correct mistakes, rather than simply prompt the user to make the correction. Similarly, a style checker would greatly facilitate text entry, particularly if it was capable of completing words once it had received sufficient of its component letters to make unambiguous identification likely.

For subject matter which is sufficiently constrained in style and content, such as certain standard report and letter formats, then an isolated phrase recognition system might become quite convenient. In the case of such documents, the standard phrases could be trained in their entirety, rather than as single words. These longer utterances are often more readily recognised by the system. In addition, the speech entry becomes quicker and far more natural than uttering isolated words.

3.2.2 Navigation Within a File and Command Sequences

The second essential requirement of a word processing system is that the user should be able to move the cursor to the position in the file where it is desired to enter or edit text or carry out some other command. At first the way to achieve this might seem obvious. The system needs to be trained to recognise such utterances as *up*, *down*, *left*, *right*, *page up*, *page down*, and so on, as cursor control commands, and simply pass on the equivalent keyboard characters to the word processor.

In fact, this is very easy to achieve, and works exceptionally well. The user's pleasure at being able to merely speak and see the cursor respond, however, lasts only up until the time it is required to insert one of these words or phrases into the document as text, rather than execute it as a command.

There are a number of possible approaches to this problem of escaping the cursor control commands. These same approaches can be used to distinguish any of the other commands needed by the word processor for functions other than cursor control. The methods considered and tested in this project were as follows:

- Remember which words represent commands and spell these words when they are required in text.
- Identify each command by beginning it with some sound which never appears in text. Possibilities for consideration might include nonsense syllables or non-speech sounds such as a whistle.
- Use utterances for commands which will never appear in text. Examples might include foreign language words, or other sounds which don't appear in normal speech.
- Precede each command by a separate escape command, or alternatively, when a command is required to be entered as text precede it with an escape command.
- Issue a special unique command to toggle the system between text entry and command execution modes.
- Ensure that commands are words rarely used in normal speech. When they do occur they can be spelled out.
- Use multi-word word command sequences trained as phrases. When the phrases appear in the text they can be entered as separate words.

In practice it was found that a combination of the last three is the most useful. When a lengthy sequence of commands is needed, it is useful to be able to toggle the voice control system into command mode. The necessary commands can then be issued

relatively efficiently. Once the task is completed the system can be toggled back into text mode and text entry continued. An example of an operation where this method is advantageous is when a block of text must be located and marked and the block moved to a different location in the file.

For less command intensive operations a mixture of the last two methods listed is convenient. Unique short utterances are desirable for frequent commands such as *enter*, *shift*, *backspace*, *caps lock* and *tab*. For other commands, such as for cursor movement, block marking and manipulation, and menu operations, multi-word command sequences proved to be the most reliable. Independence of vocabulary entries (*orthogonality* - see Chapter 2) is easier to achieve with long phrases, because there is more scope for variation in their longer templates. In addition, such phrases are more natural, making them easier to remember and more satisfying to use.

The use of escape sequences, while common in keyboard oriented systems, presents particular problems in voice control. First, with operation generally tending towards the tedious, anything which increases the average number of distinct recognition operations required to execute a function is undesirable. Second, the need to initially recognise the escape command itself means that it becomes one which is extremely critical to the operation of the system. If the escape sequence should prove unreliable for a particular speaker or in certain environmental conditions, the user can soon become involved in a seemingly unending regression of failed commands.

The same objection can be raised against other schemes which require another utterance to be recognised before the command itself is issued. The use of a special command to toggle the system between two or more modes has this deficiency also, but in this case there are fewer uses of the escape sequence. Also, it may be possible to arrange matters so that fewer special critical commands are required.

One place where the use of a non-speech sound, such as a whistle, can be justified, is for the provision of a fail-safe command to facilitate the breaking out of difficult situations, such as when the voice control system fails to recognise any of the more usual commands for any reason. It might be used to reset the system. Such an arrangement has the advantage that it would make less likely an inadvertent system reset due to a misrecognised command.

3.2.3 Execution of Recognised Commands

The handling of error conditions is one area where a voice control system implemented by use of a transparent keyboard handler is at a serious disadvantage. Audio responses cannot be triggered by characters output from the application program, only in response to recognised speech input. This means that, unless the program issues a warning tone, a user who is not reading the screen does not know that the error condition exists. In any case they will not know what the error is.

This problem, along with the difficulty of verifying the correct recognition and operation of a command, limits the usefulness of such a system, particularly for a sight impaired user. The most effective solution requires the modification of either the application program or the operating system in order to provide a more sophisticated speech input and output system.

One solution is to provide a memory resident utility which monitors the calls the application program makes to the operating system for the purpose of placing text on the screen. This utility interprets this text and generates commands to the voice control system to produce suitable audio output. Similarly, some error conditions are detected by monitoring operating system calls and status, and suitable audible error messages produced.

In applying voice control to the operation of a word processor there is always the danger that a misrecognised command will cause loss of, or damage to, data. This is particularly so in the case of commands which may intentionally or unintentionally erase text. Apart from careful use of command verification where possible, it is particularly good for the user's piece of mind if the word processor has an effective *undo* command. It is advisable for this to be made as easily and reliably accessible as possible. Of course, such a facility is unlikely to be available for commands related to the file system. Responsible use of file backup facilities needs to be encouraged.

Some of the disadvantages of voice control may be overcome by the provision of specialised tools. If the word processor has facilities for external programs to be run then these can be used to simplify tasks which might be extremely laborious if carried out inside the editor. Such tools as word counters, text formatters, style checkers and file backup facilities prove useful in this way.

Another useful tool is a utility which could be called by the user if a file was accidentally erased. In many cases, if the disk is not written to subsequent to the erasure, the file can be recovered. Being able to carry this out without leaving the word processor considerably enhances the chances of the original file still being intact and recoverable.

Even more could be gained if the external tools were written especially to take full advantage of the facilities provided by the voice recognition and audio response system. For example, a blind user would benefit greatly from being able to call up a utility which could read a text file and output it as speech.

3.2.4 Controlling Command Context

The provision of a set of commands which are only available when the system has been toggled into a particular mode makes it possible to use the vocabulary clustering facility to great advantage. If the commands available in each mode are arranged in distinct clusters, and only the appropriate cluster enabled when in a particular mode, then recognition performance is considerably enhanced. Also, by this means the same set of utterances can be used to produce different results for each mode. This reduces the number of commands to be remembered and provides a more natural working environment.

As an example of this, the same cursor movement commands would normally be used whether the user is moving around in a text file, a command menu, or a help system. Since in many systems these make use of the same keyboard characters, no recognition advantage is gained by clustering. The use of clustering in this situation, however, makes it possible to link these commands to different sets of audio responses. This is of particular help to a sightless user, who does not then need to continually remember in which mode the system is currently operating. The system can provide a suitable identifying response. Even a user with keyboard skills finds such an arrangement useful.

A natural division of the command set of a word processor results from consideration of its menu system. Each menu can have its own set of commands grouped together in one or more vocabulary clusters distinct from all other commands, and the command to gain access to that particular menu can be arranged to also enable the appropriate vocabulary clusters and disable all others.

Since, after completion of a particular command, the word processor will probably revert back to text entry mode without any further opportunity for the voice system to reset the appropriate vocabulary clusters, a special command will need to be issued to achieve this.

This is a disadvantage which must be considered if such a scheme is implemented. Normally, however, the mode in which the system will finish after the completion of a command is known. In this case the speech control command is able to carry out the required cluster control by placing the appropriate commands in a command macro along with the required characters to be sent to the word processor to implement the command.

One area of operation where such a scheme fails is in the case where the word processor (or any other application program) switches to another mode without being explicitly commanded to by the user. For example, on encountering an error, such as insufficient disk space to save a file, some word processors bring up a file handling menu. A second example is the invocation of a help system when an unrecognised command is received. When this happens the system is in one mode of operation, but only has access to voice control commands appropriate to another mode.

To overcome, this careful consideration has been given to the provision of a suitable default set of commands which are available at all times, regardless of the current context. In the SAR-10 system such commands are implemented by placing them in cluster number zero. This cluster is available regardless of which other clusters are enabled or disabled. This is also the logical place to place emergency commands such as those which provide entry into the help system, program abort, system reset, and access to voice recognition retraining and updating, if provided.

3.2.5 Results

Despite the obvious limitations of isolated word and phrase recognition for the control of a word processor it was still considered desirable to implement such a system to gain

experience of its potential for a handicapped user. Word Star was chosen as it was the Department standard at the time this work was carried out.

The command vocabulary chosen consisted of each of the Word Star commands, with commands for each menu in separate clusters. In addition, the English alphabet, punctuation, numerals, and cursor control were placed in a cluster, with a standard phonetic alphabet in another. A selection of common words was also provided. The prototype system was not specifically intended for use by a sightless user, so only a few audio response outputs were included for test purposes.

Despite the simplicity of the arrangements described, the vocabulary size was close to the maximum capability of the SAR-10. A success rate of about 80% of commands recognised was attained with all vocabulary clusters enabled. This is sufficiently low to cause operation of the system to be extremely frustrating for the user. Commands for enabling and disabling vocabulary clusters were provided in order to evaluate the effect of vocabulary size on recognition performance. As expected, it was discovered that recognition accuracy improved markedly as unnecessary clusters were disabled.

With clustering in place a recognition rate of about 95% was achieved, decreasing when the user was not one of those who trained the system. This performance also depends greatly on background noise. This degree of accuracy provided far more satisfactory operation, but was offset partly by the need to remember to switch clusters when necessary. Leaving the editing commands and cursor control clusters on and switching between standard words, alphabet, numerals and punctuation, and phonetics clusters, proved to be an effective compromise. Cluster switching for different menus was handled automatically by control codes embedded in the appropriate menu select commands.

The desirability of being able to *pop up* a vocabulary retraining utility when needed became evident after considerable use. The recognition success rate varied considerably

with operator fatigue and changes in the level and nature of background noise. The ability to perform a quick training pass to update the recognition templates often successfully eliminated this performance degradation. The tedious nature of letter-by-letter text entry makes the onset of fatigue, with its attendant lapses in concentration, rapid indeed.

3.2.6 Conclusions

Except for the case of the entry of highly constrained documents, such as standardised reports, the only redeeming feature which can be found for using an isolated word and phrase recognition system in a word intensive application such as word processing is that it does make text entry possible for someone who, through handicap or environmental constraints, cannot use a keyboard. It is certainly not the method of choice if some other text entry system is available.

A continuous speech recognition system with a large vocabulary, high accuracy of recognition, and a far greater language understanding capability would make such an application very convenient and natural to use. Anything less than this makes word processing extremely laborious.

3.3 Spread Sheet

The use of an isolated word recognition system for voice control of a spreadsheet is more attractive than for operation of a word processor. A spreadsheet is generally less dependent on text than it is on numbers. Such text that is commonly used tends to be restricted to relatively short labels for the rows and columns of figures.

3.3.1 Data Entry

The entry of labels, figures and formulas into a spreadsheet are normally distinguished by a change of mode, triggered either by a particular character typed (e.g. a digit for numbers, an equals sign for a formula, and a quotation mark for text) or by a particular command. In either case the mode change can be used to select an appropriate vocabulary cluster for the new mode. This means that text oriented commands, the alphabet, and standard labels, can be placed in one cluster, digits in another, and arithmetic operators in a third. Switching between the appropriate clusters will normally be totally transparent to the user.

3.3.2 Commands

If the spreadsheet has a menu system this can be operated using cursor movement commands similar to those described for a word processor. The same considerations for command verification and error condition handling apply. The ease of segmenting the vocabulary into logical clusters, and the smaller number of commands available mean that high recognition accuracy is likely to be achieved.

Handicapped users can take advantage of the fact that for many tasks a preprogrammed standard skeleton spreadsheet might be used as a starting point, removing much of the tedious text entry involved in building a sheet from scratch. Application specific sets of standard labels can be provided, while more general text may be entered by spelling the words.

3.3.3 Results

These principles were applied to producing voice controlled Perfect Calc [WADE84] spreadsheets with considerable success. The relative ease of producing small orthogonal command sets for a spread sheet context meant that the spreadsheet packages could be simply controlled, and numerical information entered fairly rapidly. The package tested was menu based, so full advantage could be taken of command clustering.

The only real difficulty encountered was the SAR-10's difficulty in distinguishing between the utterances *five* and *nine* - possibly the worst possible problem for a numerical application such as a spreadsheet. Despite their clear vocal difference, these two utterances consistently produced very similar normalised reference patterns. For one particular user, the only effective solution was to revert to training the digits in his native language, not in English. Others needed to make some artificial distinction in the way they pronounced these two words. (Interestingly, this problem is well documented - during World War II pilots needed to pronounce these numbers as *fife* and *niner*. However, the mechanism underlying the confusion is undoubtedly different.)

3.4 DOS Shell

While an experienced PC user might find the use of a DOS shell of limited advantage, and perhaps even a liability, for the handicapped user such a utility can be extremely valuable. When voice control is being used, a shell is of even more benefit, since instead of having to cope with a relatively large set of text oriented DOS commands, the user can accomplish operating system interaction using voice controlled cursor movement commands to select menu items.

A sightless user would not find this of much advantage, as it relies far more on being able to see the position of items on the screen, and on being able to read file names. A utility which reads each label on the screen and produces appropriate speech output assists greatly here. Dependence on text input could be further reduced if some form of file name generator was provided.

3.5 Keyboard Macro Processor

The provision of a programmable keyboard macro processor can further simplify operation of a PC under voice control. Using such a utility, application programs which rely on often repeated complex command sequences can have the appropriate commands programmed into a macro. The macro can be assigned to an unused control character on the keyboard, and this character may then be used as the output from the speech recognition system on recognition of an appropriate voice command.

This scheme was tested using the SuperKey macro processor [BORL85], and was found to be a labour saving convenience when used in conjunction with Word Star or Perfect Calc. As an example, the character sequence (<insert> "|" <down arrow> <left arrow> <insert>) required to repetitively draw a vertical line down the page could be programmed in, and then a line could be produced using a voice command such as *vertical line*. This proved useful for drawing character boxes around items on the page.

A similar process can be followed if the voice control system has been programmed to output a certain character sequence in response to a certain command and, perhaps temporarily, a different character sequence is required. The macro processor can be *popped up*, assuming a suitable voice command has been assigned to its triggering character sequence, the macro programmed, and work continued in the application program. Many such utilities provide other features which have the potential to be of

great benefit to a handicapped user. These include calculators, calendars, clocks and notepads.

3.6 Programming Environment

One of the most rewarding applications of voice control via a transparent keyboard handler proved to be in the area of programming. This offers great potential for a handicapped computer user, giving such users the ability to create their own specialised utilities and application programs, and opening up many possibilities for freedom in work which would otherwise prove to be difficult. Many tools are available to assist with program development, and these make the benefits of voice controlled programming even more attainable to the handicapped user.

The capability of handicapped users to write their own programs can help them greatly in overcoming some of the disadvantages inherent in using computers and software created with only the able bodied user in view. A disabled user is in an excellent position to know at first hand what is required in an application interface, and would be able to contribute greatly to such development if a suitable programming environment is made available.

3.6.1 Turbo Pascal

The approach taken to the development of a voice controlled programming environment was to build upon the discoveries made and lessons learned through experience with word processing, spreadsheets and the use of DOS shells and keyboard macro processors. The programming language chosen was Borland's Turbo Pascal [BORL88b]. This provides a program editor, a compiler, a debugger and an effective help system, all integrated into a menu driven programming environment. Provision is also made for the

execution of programs external to the environment by exiting temporarily into a DOS command interpreter.

Considerable assistance is given to the user by the operation of the command system. For example, if the programmer attempts to leave the system without saving a file which has been modified, he or she will be prompted as to whether this is intentional or not. Unfortunately this prompting is only visual, and so is of no use to a blind user.

A further useful feature for our application is provided by the help system. In addition to containing a comprehensive reference manual for the language and the environment, it also provides a cut and paste capability which allows text from the help system to be exported to the user's program. As the help system also provides example code illustrating the use of each function and procedure of the language, this can be used to advantage by cutting and pasting the lines of code into the program. By this means the amount of tedious *voice typing* required to write a program can be reduced.

The manner in which the Turbo Pascal compiler deals with errors is particularly advantageous to its use in a voice controlled system. The detection of an error during compilation of a program or a unit causes the compiler to halt and a suitable error message to be displayed. In addition, the file which contains the error is loaded, even if it is not currently the file being edited, and the cursor placed at the position of the error. This allows the programmer to immediately rectify the problem. If further assistance is needed a single keystroke will invoke the context sensitive help system, which will make available further information on the nature of the error.

This scheme might be preferred by inexperienced programmers, who find one error enough to cope with at a time, whereas seasoned programmers might prefer to have the compiler present them with a number of errors at once, so that they can fix the problems without needing to recompile the program after each one is dealt with. For the user of a

voice control system, however, the behaviour of Turbo Pascal is a distinct advantage, since it means that the system automatically carries out the location of the error and the movement of the cursor to the correct location. The saving in voice commands is well worth the inconvenience of having to deal with errors on an individual basis. The Turbo Pascal compiler is fast enough for this not to be a problem for small programs. For large multi-file programs the automatic loading of the correct file outweighs the enforced waiting. Borland C++ [BORL92a] implements both schemes and is even more suitable in this respect.

3.6.2 Reusable Code

For the efficient carrying out of any task using voice control it is desirable to reduce the need for text entry. The concept of reusable code can take the user a long way towards this goal.

Turbo Pascal provides an extensive set of code libraries, called *units*, which provide hundreds of functions and procedures not available in standard Pascal. These also reduce the amount of work needed to develop a program. Users can easily create their own units, and by this means the habit of writing reusable code is encouraged. Many toolboxes containing units for the carrying out of many tasks are commercially available. Once a set of general purpose units for such tasks as menu construction, file manipulation, input and output device control, searching sorting and storage of data and text, numerical calculation and graphics routines, is available to the user, application programs for many different tasks can be produced relatively quickly.

Great assistance to the programmer is provided by a unit, called SARLIB, which implements the necessary routines for manipulation of the voice control system and the

integration of its input and output facilities into the user's program. The construction of SARLIB is described in Chapter 8.

3.6.3 Vocabulary Selection and Programming Technique

The selection of vocabulary entries such as the alphabet, cursor control characters and menu commands is similar to those for application programs. For a programming language there are additional considerations. One of these is the need to provide a set of reserved words for the language. The names of routines in units which the user desires to use will also be needed. Another consideration is the need for the user to be able to produce suitable words for the naming of variables, constants, types, functions and procedures.

To the user there is no difference between reserved words and the names of library routines. Taken together they form a very large set of names, each of which must appear as a vocabulary entry. The version of Turbo Pascal used in this study contained 48 reserved words. The main units supplied with the system contain about 220 routine names and a considerable number of identifiers used for types, constants and variables.

When alphanumeric characters, operators and punctuation symbols are included, it is clear that the total is greater than the number of entries the SAR-10 can accommodate. Some limitation will need to be placed on the number of routines made available to the programmer.

The first approach taken was to determine which subset of the provided routines are frequently used, and to place these in the vocabulary, using a separate cluster for each unit. This does not rule out the use of other routines. They can still be accessed by

spelling the appropriate identifier out. The advantage of placing routine names from different units into separate clusters is that only the required clusters need to be enabled.

The routines in a unit are only available to the compiler if the name of that unit appears in a *uses* statement at the head of the program or unit being developed. This can be used by the programmer to discover which clusters need to be enabled. To enhance recognition accuracy it is desirable that the user disable all clusters which are not necessary for the immediate segment of code being written.

Nevertheless, the number of items needed in the vocabulary at any time is still considerable. This means that great care must be taken in the selection of suitable utterances for each identifier if reasonable vocabulary orthogonality is to be achieved. It is often necessary to give many items names which are not identical to those defined by the language or the library routines. This is an unavoidable price for using such unsophisticated speech recognition techniques.

Some advantages can be gained from the use of a speech system. It is possible to program entire program structures into the system so that they can be reproduced by a single command. For example, instead of needing to build a Pascal block by placing a *Begin*, writing the statements, then placing an *End* and perhaps a semicolon, the utterance *make a new block* might be used to produce the code shown in Figure 3.1.

```
Begin  
  
End;
```

Figure 3.1 Turbo Pascal Block Skeleton

The statements can then be included within the block. This simple process can be taken further; *make a repeat loop* might produce the code of Figure 3.2.

```
Repeat  
  
Until ( );
```

Figure 3.2 Turbo Pascal Repeat Loop Skeleton

The command to *begin a new unit* could output the text shown in Figure 3.3:

```
Unit      ;  
{ A unit to ... }  
{ _____ }  
Interface  
Uses      ;  
Const     ;  
Type      ;  
Var       ;  
{ _____ }  
Implementation  
  
{ _____ }  
{ Initialisation }  
Begin  
  
End. { Unit ... }
```

Figure 3.3 Turbo Pascal Unit Skeleton

This process saves much typing and simplifies the process of producing well structured code.

Another way of reducing the amount of text entry is for a number of *standard* variable names to be provided. Such generally useful identifiers as *LoopCounter*, *InitialValue*, *FinalValue*, *Finished*, *Found*, *NewValue*, *OldValue*, and so on, could be produced with a single utterance. These identifiers can then be extended by the simple addition of a digit by the user, to produce a large number of ready to use names. For a handicapped programmer, although the resulting identifiers may not be as meaningful as desired, and the program may be harder to debug, these disadvantages may be outweighed by the reduced interaction with the text editor.

Any assistance which can be given to the programmer through the provision of automatic layout should be considered. For example, it proved possible to partially automate indentation by arranging for the insertion and removal of margin tabs to be triggered by the *Begin Block* and *End Block* commands.

3.6.4 Programming Aids

Clearly, the use of a keyboard macro processor would simplify many of the above operations. More satisfactory solutions can be arrived at by the use of various programming tools.

If one was to construct a compiler specifically intended for control by voice, advantage could be taken of the fact that the compiler "knows" the context of the code on which it is currently operating. For example, on encountering the identifier *while* the compiler will then expect to find a Boolean expression followed by a statement (simple or compound). A voice system under the control of a compiler could have its current vocabulary

optimised at any time according to the possible identifiers to be expected. In addition, the editor associated with such a compiler could use its "knowledge" of the language to perform automatic layout and production of complex program constructs.

Some experience of such an intelligent editor, under voice control, was gained with Alice: The Personal Pascal [SOFT85]. This system goes a long way to automating the writing of a Pascal program, providing the automatic layout control and construct production mentioned above. Such an environment, with voice control and audio response integrated into the system, rather than added on by use of a transparent keyboard handler as was necessary with Alice, would approach the ideal for a handicapped programmer.

Many other tools exist which can assist greatly in the production of programs by voice control. Among the most useful of these are formatting tools and code generators. If, in the interest of saving time and laborious text entry, code is written in a style which leaves something to be desired, it can be subsequently passed through a formatting tool in order to transform it into a more pleasing style in terms of indentation and identifier format. Existing source code, written by various people, can also be converted into a uniform style for ease of use in one's own environment.

The greatest saving of time can be gained by not writing code at all. Rather, a specification for a program, in the form of a screen or form description or a language grammar, or similar, can be produced. This description is then passed to a code generator, which produces the required program, or program fragment, to interpret the description.

Good examples of these tools are available in the form of screen and form generators for the production of code which implements effective user interfaces. A more powerful example is the parser generator. Later in this work it will be shown how a parser

generator, principally intended for the production of compilers, can be used for the implementation of a natural language interface. Typical parser generators, however, can be used in far more versatile ways than the production of language interpreters. For example, the tool used in this project, LALR [MANN87], and other examples such as LEX [LESK75], YACC [JOHN75] and Bison [RUBI86], are capable of generating code for any process which can be described by means of a simple grammar. This includes the production of compilers, interpreters, translators, text oriented user interfaces, calculators, syntax and style checkers, and text and program formatters.

These tools are of advantage to the voice control user because a grammar to describe a process is generally much smaller than the code required to implement the process. In addition, the resulting code is usually smaller, faster, and more free of errors than code produced by hand.

3.7 MicroExpert Expert System Shell

MicroExpert [THOM85], a simple expert system shell proved to be an excellent application for the SAR-10 speech recognition and audio response system. The main reason for this is that all of the responses to the expert system's diagnostic questions are well defined. They are also defined by the person who sets up the database for the system, and so it is easy to select them for their suitability for inclusion in a vocabulary. If necessary, the responses to each question could be placed in separate clusters. The routines in the expert system which prompt the user can also be used to switch the vocabulary clusters.

Another advantage of the design of an expert system shell is that the replies the system makes to the user, whether to prompt for information or to output the results of an inference, are well defined and thus suitable for SAR-10 audio responses.

The combination of an expert system shell and a speech recognition and audio response system proved to be an excellent development and training system for artificial intelligence applications. Having the source code of MicroExpert available enabled the two systems to be optimised to each other, and extended to form a very useful and easy to use tool, with a very wide range of applications.

3.8 Use With Mouse Based Applications

Since the introduction of Microsoft Windows, the mode of use for IBM style personal computers has moved from being heavily keyboard based to a more mouse oriented style. This has advantages and disadvantages for the use of a speech recognition system using a transparent keyboard handler.

As most software is now oriented around a mouse driven menu interface, the number of commands which need to be explicitly recognised by the speech system is far smaller. It is often only necessary to be able to navigate the cursor around the screen and to indicate when a menu item is to be selected. These few commands can easily be made reliably and unambiguously recognisable.

The difficulty arises out of the sheer tedious nature of such operation under voice control. Having to move the cursor by repeatedly saying, *Up, up, up, up, ... up, left, left, ... left, select*, is unacceptable. To provide distance counts such as *Up nine, left five, select*, is an improvement, but it can be difficult to judge the distance to be moved. This is particularly tedious when moving the cursor through text in a word processing application.

For voice control to be effective in this environment, the nature of the menu system needs to be taken advantage of, rather than being allowed to force us back into an

unacceptable mode of operation. The advantage of a menu system is inherent context sensitivity. Once the cursor is within a particular menu only the relevant vocabulary cluster needs to be enabled. The *hot-keys* are programmed to be the output of the transparent keyboard handler upon recognition of an utterance representing the name of a menu, or of any item in a menu. One cluster has the name of each menu, and sub-clusters have the items contained within each menu.

In this manner, while the mouse commands are not taken advantage of, the provision of the menus by the application software greatly enhances the operation of the speech recognition system. The usual provision of context sensitive help is also a great advantage, as the utterance of a command such as *help* will immediately bring to the screen information relevant to the current operation, without it having to be searched for.

Microsoft Windows is especially suitable for such operations since it has a highly standardised menu structure, with many menus appearing identically in different applications, and other menus tending to be of a predictable format.

3.9 Conclusion

Experience with the application of voice control to application programs by means of a transparent keyboard handler has shown that such a technique is useful as an inexpensive way of producing a voice controlled application, but is not an ideal approach. Limited ability to control vocabulary clusters and to produce audio output responsive to the state of the application, along with inadequate control of error conditions proves frustrating. On the other hand, if a handicapped user has a need to use a particular application program, and is willing to persevere with these limitations, the results can be rewarding indeed. With the use of imagination, much can be done to assist such a user in the more efficient use of such a system.

4. APPLICATION: A VOICE CONTROLLED ROBOT

To explore the feasibility of using voice commands to control a manipulator, in 1986 a system was constructed around a low cost industrial robot, the SAR-10 card and an IBM PC/XT [NARA86a]. Such a system forms a suitable test bed on which some of the practical aspects of using voice control for a disabled person's aid can be investigated. Experience of the problems which arise in such an application can be gained without causing any danger to the potential user of such a system.

4.1 The Robot and its Controller

The robot used was a Rhino XR-2, primarily designed for training in robotic techniques [RHIN82]. It is a six axis arm using DC servo motors and two phase optical chopper position and rate feedback. The robot controller is based on an Intel 8748 microprocessor, and communication with the PC is via three wire RS232c. The controller has the capability to control eight motors and monitor six interrupts from the robot. The instructions available for each axis are:

- Start a motor.
- Stop a motor.
- Determine motor position.
- Determine the status of a microswitch.

The robot controller can only accept a maximum of 127 optical encoder counts, so large movements of any joint must be carried out in smaller stages to avoid exceeding this count.

4.2 The Robot Control Program

Clearly, these robot controller capabilities provide only a very crude interface between the robot and application programs. To improve this situation, a more sophisticated interface program was developed by Dr. na Ranong [NARA86b]. This is capable of providing translation between the simultaneous joint movements required by any realistic application and the crude controller instructions available. The instruction set implemented by the interface program on the PC is shown in Figure 4.1.

Main Sequence Commands		Point Related Commands	
ADJ	Adjust point no. 0	Arm Out/In	
BRA	Set branch selector	BR n (n = branch no.)	
BYE	Exit to PC	Carousel Left/Right	
DEL nn	Delete point no. nn	DO n (1=main, 2=branch, 3=offset)	
DUP	Duplicate a point	Flap Up/ Down (wrist)	
GO=aaa	Go to point named aaa	Go In/Out (slide base)	
INS nn	Insert point after point nn	Hand Close/Open	
GTO nn	Move robot to point nn	IS (micro-switch status)	
LIB	List branching selectors	LO (lower arm)	
LIS	List programmed points	LI (lift arm)	
LOD	Load programmed points	N=xxxxxxxx (naming point)	
MOD nn	Modify point no. nn	Offset Delete	
NEW	Start a new program	Offset Insert	
NST	Nest the robot	Offset List	
RUN nn	Cycle nn times, varying branch selector 1..5	Offset Move	
SAV	Save programmed points	Pause +/-	
WRM	Warm start, defining preset point	Rotate Left/Right (wrist)	
???	List main selection commands	Turn Left/Right (waist)	
		=In/Out Up/Down Left/Right nn (inches) (XYZ control)	
		EXit to sequence command level	
		??? List point related commands	

Figure 4.1 Robot Controller Command Language

In contrast to the applications considered in Chapter 3, the source code for the robot interface program was available. It was decided to take advantage of this rather than use the SAR-10 in keyboard transparent mode. By inserting SAR-10 instructions directly into

the program structure full control of the speech recognition system became available. In addition, the speech response capabilities were able to be used, thus giving the robot the ability to talk.

4.3 The Robot Control Language

Based on the robot interface instruction set, a vocabulary of about 25 commands was developed. The commands provided were of a more intuitive nature than those provided by the robot interface language. They are shown in Figure 4.2. In addition, questions could be asked of the robot which elicited pre-programmed audio responses. These are not listed, as they are highly dependent on the application context.

Cluster 0 (Common commands):					
Zero	Five	Ten	Fourty	Ninety	Wake up
One	Six	Fifteen	Fifty	Hundred	Sequence commands
Two	Seven	Twenty	Sixty	Stop	Point commands
Three	Eight	Twenty-five	Seventy	Go	Help
Four	Nine	Thirty	Eighty	Go to sleep	
Cluster 1 (Sequence commands):					
Adjust point number zero		Go to point named ...		New program	
Delete point number ...		List programmed points		Nest the robot	
Duplicate point number ...		Load programmed points		Run a number of cycles ...	
Modify point number ...		Save programmed points		Warm start the robot	
Insert point after point number ...		List branching selectors		List sequence commands	
Move robot to point number ...		Set branch selector			
Cluster 2 (Point commands):					
Move the arm in ...		Rotate the waist left ...		Perform an offset	
Move the arm out ...		Rotate the waist right ...		Delete an offset	
Raise the arm ...		Rotate carousel left ...		Insert an offset	
Lower the arm ...		Rotate carousel right ...		Move an offset	
Move the slide base in ...		Open the hand		List the offsets	
Move the slide base out ...		Close the hand		Name the point	
Move the wrist up ...		Is the sensor switch open?		Perform main sequence	
Move the wrist down ...		Is the sensor switch closed?		Pause	
Rotate the wrist left ...		Branch number ...		Resume	
Rotate the wrist right ...		Perform a branch		List point commands	

Figure 4.2 Voice Controlled Robot Commands

4.3.1 Vocabulary Orthogonality

While developing and testing a suitable set of voice commands, it was soon discovered that the choice of words and phrases to be recognised is extremely critical. This is brought about by the need for the vocabulary space to be orthogonal.

For a vector space to be orthogonal it must possess a set of basis vectors which define the space, and these vectors must be linearly independent of each other. This means that they have no components in common with each other; no basis vector can be constructed by combining any other basis vectors.

An *orthogonal vocabulary space* is one where any entry in the vocabulary is independent of any other entry. Expressing this in terms of the feature templates stored for each recognised utterance, no two dissimilar utterances will produce a set of feature numbers - a feature vector - which will differ from any other feature vector by less than the currently assigned error value, the recognition threshold.

For reliable recognition performance, it is desirable to select a vocabulary which ensures that the feature vector for each utterance differs by as great an amount as possible and in as many of its features as possible from the feature vectors of all other vocabulary entries. If words and phrases could always be selected which were independent in this way, then recognition could be made completely error free, an ideal but unlikely situation.

In practice, what generally tends to happen is that the words and phrases which appear to be the most natural choices for an application, produce reference patterns which are far from orthogonal. Because of the method by which features are extracted from an utterance during the acoustic analysis process, it is not enough that the words sound quite different to the human listener to guarantee that they will produce different values in the feature vector. Extensive testing and revision of the vocabulary is required before

acceptable performance is achieved, resulting in a command set quite different from the initial more obvious choices. However, by perseverance and imaginative choice of phrases, a reliable, functional, and aesthetically acceptable vocabulary can usually be developed.

Commands which caused considerable difficulty in the context of the robot control language were: *left* and *lift*; *in* and *on*; *five* and *nine*; *listen* and *ignore*. Some of these result from the words sounding similar, such as *left* and *lift*, differing only slightly in the vowel sound. This is illustrated by the hypothetical *difference* matrices and the resulting *accumulated distance* matrices in Figure 4.3. Both utterances result in the same minimum path length through the matrix. As this is used to determine whether the utterance matches the reference template, the two words will be considered to be identical.

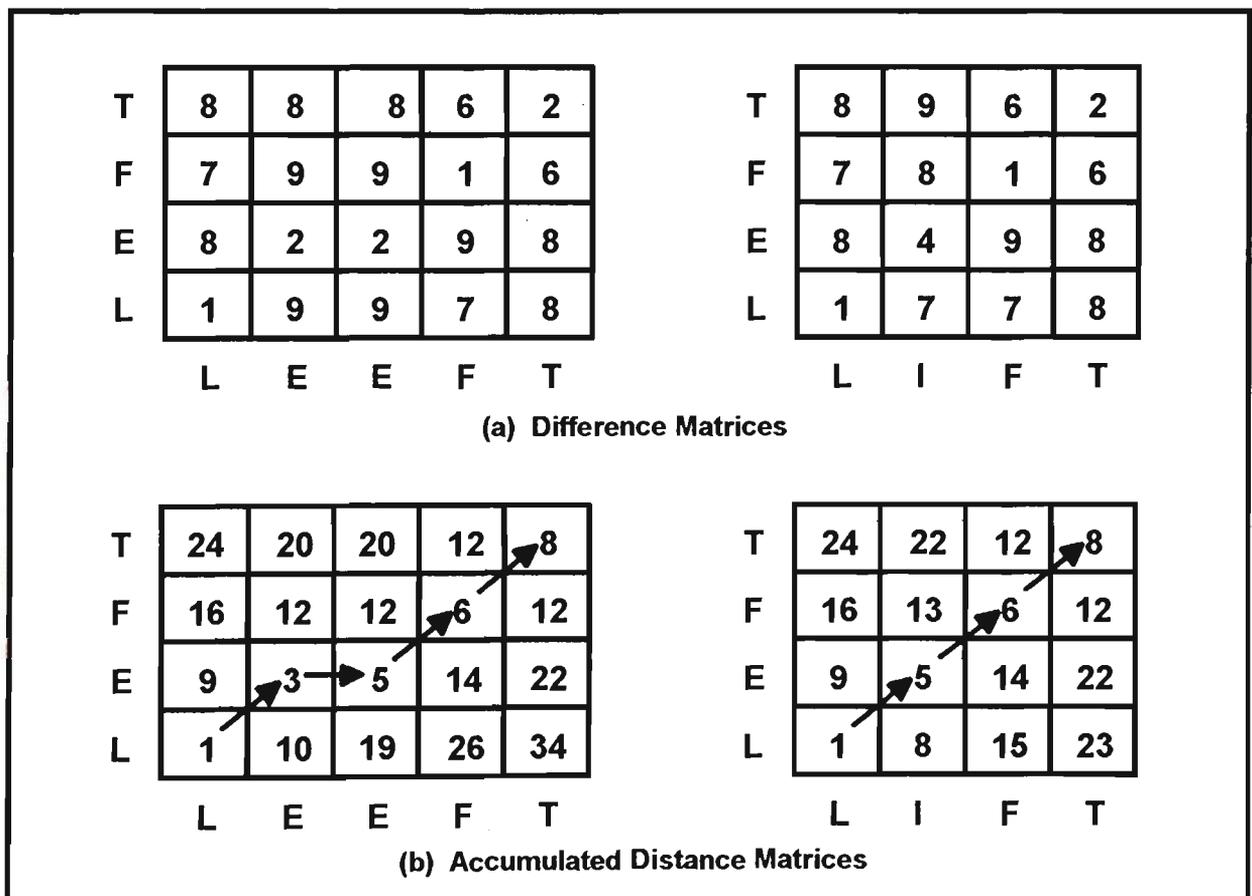


Figure 4.3 Difference and Accumulated Distance Matrices

This example also illustrates that, with algorithms like dynamic time warping in operation, even a difference in the length of two utterances may not be enough to ensure orthogonality. As can be seen in Figure 4.3, the *accumulated distance* from the reference pattern of the utterances *left* and *lift* is eight in both cases.

The utterances *listen* and *ignore* sound quite different to the listener, and produce quite different patterns. However, because their distance from the reference pattern is ultimately determined by a single number, the *accumulated distance*, it is common for these also to be impossible to separate. The accumulated distance may be the sum of different numbers, but still have the same value in each case, or a value which differs by less than the recognition threshold.

Utterances such as these pose a particular problem, as they are clearly intended to be the inverse of each other. In the case of the robot controller, *ignore* was used to turn off the microphone, and *listen* was intended to turn it on again. As was mentioned in Chapter 3 and will be explored more fully in Chapter 5, the ability to place commands into clusters can be used to resolve such recognition problems. This may not be feasible in the case of commands which are obviously closely related as these commands may need to be placed in the same cluster. In such circumstances there is no alternative to replacing one or both of the commands with a different word or words.

The words *ignore* and *listen* also illustrate the SAR-10 system's particular sensitivity to utterances which differ only in implosives with little energy content. One solution to this problem proved to be restricting the use of short words such as *ignore*, using instead longer phrases such as *ignore me*. Another way around such difficulties is to require speakers to emphasise vowel differences between words such as *left* and *lift*. This, however, introduces the disadvantage of requiring users to learn to speak in an unnatural manner. The requirement to speak in isolated words or phrases is restrictive enough without introducing this additional restriction on pronunciation. In addition such solutions

also tend to reduce speaker independence, because some people have greater difficulty remembering the less natural way they had to speak when training the problem phrases.

4.3.2 Speaker Independence

In order to test the speaker independence of the SAR-10 system a subset of 25 commands from the robot control vocabulary and various system housekeeping functions, was placed in a single cluster.

The first test involved three training passes with a single male speaker in a quiet environment. The vocabulary was modified until the orthogonality conditions were achieved, following which a recognition accuracy of 98% was attained.

The second test was again performed in a quiet environment, this time using four male speakers who had roughly the same voice pitch as the original trainer. The recognition accuracy dropped to about 85%.

Under the same conditions, two female speakers and a male speaker with a higher pitched voice all achieved about 75% accuracy.

The system was then retrained, using two passes each from all of the above speakers. Using these speakers at random an accuracy of 95% was consistently attained. Two other speakers who had not trained the system also achieved 95% accuracy. This performance could be usefully described as being 95% speaker independent.

4.4 Performance of the Robot Control System

4.4.1 Operation in a Noisy Environment

An Institute open day demonstration in 1986 was chosen as the opportunity for a rigorous test of the voice controlled robot system. To provide the illusion of intelligent behaviour, and to encourage the public to participate, it was decided to provide the robot with a rudimentary English language understanding capability. As the demonstration immediately preceded a Victorian Football League grand final, the robot was dressed up in the guise of a very vocal and biased football supporter. It was given a suitable flag to wave and a bell to ring.

Careful selection of the recognition vocabulary and the audio responses enabled the robot to carry out a simple conversation on the merits or otherwise of the participating football teams, and display an appropriate amount of disdain for the umpires to easily win the heart of the average supporter. A degree of randomness was given to its selection of suitable responses to speakers' questions and comments to maintain the illusion of intelligence (an illusion probably being more in keeping with the character than real intelligence in any case).

The language understanding feature was achieved with a simple parser, using a technique described by Lea as *sequences of isolated words, using linguistic constraints* [LEA80]. The grammar consisted of a set of 15 nouns, either preceded by a member of one of a set of 8 verbs, or followed by one of 19 verbs from a different set. The operation of the parser was based on a simple finite state machine.

During laboratory tests prior to the public demonstration, this system performed extremely well, gaining some publicity from the local press. Similar recognition accuracy and speaker independence were achieved as in the earlier tests, despite the fact that the

vocabulary size had risen to 50 words and clustering was still not being used. These results are consistent with those obtained by Flanagan, et. al. [FLAN80].

Performance during the public tests did not live up to expectations, however. The display was poorly sited, and due to the proximity of video-taped demonstrations and considerable crowd noise, it was often difficult to obtain any response at all from the speech recognition system. The major cause of failure was put down to the high level of background noise preventing the system from detecting the pauses between words. Another problem was the use of a hand held microphone to facilitate public participation. This resulted in greater variation of voice levels and increased pickup of environmental noise.

Despite these difficulties, when the system did operate correctly for any length of time, single speaker control of the robot remained at a recognition accuracy of between 80% and 90%. Speaker independence varied between 50% and 70%, enforcing frequent voice retraining. Such performance levels are not adequate for any serious application, but did allow some appreciation of the potential, and also the problems, of such a system to be gained.

4.4.2 Operation Under Stress

Operating the voice controlled robot, even in quiet laboratory conditions, can still result in some unexpected difficulties. One particular problem was revealed during attempts to control the robot in real time.

Normally the robot arm was moved by telling it which direction to move and how far to go. In the absence of the robot having a knowledge of its world model, or limit switches

to prevent collisions with objects within its reach, such operation relies on the operator not giving a command which will cause such a mishap.

An alternative method of control is for the operator to start the arm moving in the required direction, and tell it to stop once it has arrived at the desired position. This is very convenient, since the operator does not have to measure or estimate the distance from the initial position to the final position of the arm. If the operator has control of the speed of movement the arm can be slowed down as it approaches its target position, resulting in very fine control. At the time the tests were carried out the robot in use did not have a speed control capability.

Such a method of control worked well when the destination of the arm was not close to any other object. The movement could be halted quite precisely at will. This was not the case when the robot was required to stop just before hitting an object. Invariably, at the critical moment the voice recognition system completely failed to respond, resulting in many collisions with walls and other objects, occasionally causing some damage.

Clearly, if this system was intended to lead the way to a prototype of a voice controlled machine to assist a disabled person to perform such tasks as drinking a cup of tea, the cause of the 100% failure rate in avoiding collisions with target objects under real time control had to be found. The answer was soon discovered to be operator stress. As the robot arm nears the object at the destination point and a collision becomes imminent, tension is generated in the operator, intent on avoiding the collision but still arriving at the correct point. This produces sufficient change in the operator's voice pitch and pronunciation to cause the system to cease recognising commands.

No degree of concentration on the part of the speaker seems to be sufficient to overcome this problem sufficiently to allow such a mode of operation to be considered practical or safe.

The conclusion which has been drawn from these tests is that a voice controlled general purpose robot, at the present stage of development, is not a suitable machine to be used in close proximity to a person who, owing to physical handicap, is unable to protect themselves when the voice control system fails to respond. This particularly applies to the case where the robot is in motion and its path needs to be altered by voice control, but it is probably equally applicable to the case of incremental control. Too much responsibility is placed upon the operator's ability to predict the result of any command once it is activated.

4.4.3 Command Verification

Entry of commands by the use of a keyboard provides a built in verification capability that the command is correct before the enter key is pressed. The operator can see what was typed on the screen and mistakes are easily corrected. Martin and Welch [MART76] point out the need for command verification and correction capabilities in voice control systems. Three types of mistake are possible. The system may:

- not recognise the command,
- receive a command which is meaningless in the given context,
- mistake the command for another command.

An unrecognised command, should generally be harmless as the speech recognition system would not transmit it to the application program and nothing would happen. An exception must be made in the case of real time control, where the reception of the command at a particular time may be critical to the prevention of some mishap.

A meaningless command should generally be taken care of by the normal error handling procedures of a well designed system. However, in some circumstances, such as in real

time control applications, it could also produce the same unfortunate results as an unrecognised command.

The third type of mistake, the substitution of one command for another, holds the greatest potential for disaster. If the application is a relatively benign one, such as word processing, the result will be at worst some inconvenience to the user, or the loss of data. However, if the voice system is part of a process control system, for example, the results of receiving an unintentional command could be extremely dangerous. The result may well be damage, injury, or even loss of life.

When using the voice control system in keyboard transparent mode a simple form of command verification was implemented by echoing each recognised command on the screen and not activating the command until a suitable response is given, such as the word *enter*. Vocal backspace and erase commands enabled corrections to be made to incorrect commands. This method works well, but negates much of the convenience of voice control. The normally slow operation of giving commands as isolated words is hampered even further by having to give a second command to activate the system once the first command has been verified. The operator must still watch the screen, and prevent the transmission of improper commands by either erasing them and trying again, or attempting to correct the command by backspacing and spelling out the corrections. There is still the danger that another command will be wrongly interpreted as the *enter* command, although this generally results in an invalid command due to the original command and the new command being concatenated.

It is possible, under difficult operating conditions, to get into a regressive situation where attempts to correct a mistaken command are also misinterpreted, and where even attempting to enter a correctly recognised command will be mistaken. In these circumstances the pending command gets more and more mutilated with every attempt to correct it. To take account of this situation some form of *panic* command, as phonetically

distinct as possible, and which causes the system to seek some fail-safe position, should also be provided. Words which people instinctively try in an emergency when all other commands have failed or forgotten - words such as *help*, *quit*, *stop* and *bye* - should also be included. When all else fails it should be possible to resort to keyboard control.

Audio verification can be used to counter the common temptation to ignore the screen. Confirming commands in this way is slow, even though it would seem to be the more natural approach for a voice control system.

When operating in non-transparent mode, audio verification is easier to do, since the full audio response facilities of the system are then under the control of the application program. The system does not then have to depend on preprogrammed responses linked with each trained command and output using the echo-back facility. Full advantage can be taken of audio response macros to build complex responses from simpler vocabulary entries.

When an audio response is being produced, for command verification or for any other purpose, it is necessary to first turn off the microphone to stop the system trying to recognise its own responses. The microphone then needs to be turned on again prior to the next command. This is necessarily controlled by the program, not the operator, and means that the current status of the process being controlled must be known by the program. It may be helpful for the operator to be able to see whether the microphone is enabled or disabled via an indicator on the screen.

The ability of the SAR-10 to return the utterance which produced the second best match as well as the best match allows this second candidate to be presented if the first is not verified. There is a high probability that the intended command was the second best match. If the second command is correct and verified then the operator is relieved of the tedium of repeating the utterance in an attempt to have it recognised correctly.

5. OPTIMISING THE SAR-10's PERFORMANCE

The SAR-10 speech control system has limitations which are inherent to the design of its hardware and software. As these limitations are unable to be removed, attempts were made to make the best use of the facilities provided. Four areas of optimisation were tackled:

- audio pre-processing.
 - microphone choice.
 - electronic speech processing.
 - program control of microphone on/off.
- vocabulary and reference pattern control.
- parameter control, to make SAR-10 adaptive.
- user interface.

5.1 Audio Pre-processing

Three avenues of audio pre-processing listed above were investigated in an attempt to provide the SAR-10 with speech signals with the highest possible signal to noise ratio, and to minimise the effect of non-speech sounds.

5.1.1 Microphone Choice

The choice of microphone was found to have a great effect on the satisfactory operation of the SAR-10 system. While any type of microphone of reasonable quality works in controlled laboratory conditions, when the system is in general use, or the operator is working in less than ideal circumstances, closer attention to microphone performance was necessary.

The most satisfactory performance was obtained using either a headset microphone, or a throat microphone. These ensure that the microphone provides the best ratio of signal to background noise. In addition, they leave the operator's hands free for other tasks, such as typing or using a mouse. For a disabled operator the advantages are even more obvious. It is highly desirable that the microphone should be of a noise cancelling/noise masking variety, and suitable for close talking without distortion.

5.1.2 Electronic Speech Processing

The speech input bandwidth of the SAR-10 is restricted to a range of 200 to 5000 Hz, and little advantage could be expected from using further passive filtering, except possibly in a very noisy environment. In fact, in normally quiet environments, further restriction of the speech bandwidth results in a deterioration of recognition performance. On the other hand, the application of adaptive filtering, tailoring the filter shape and bandwidth dynamically to the spectrum of the speech signal, can realise a useful improvement in the signal to noise ratio. An adaptive filtering system can also incorporate a noise cancelling function.

Such a filter and noise canceller was constructed [DOW86] based on a Texas Instruments TMS32010 DSP microprocessor, as shown in Figure 5.1. The analogue to digital conversion functions were carried out using 8 bit TP3051 Codec chips, which necessitated linearisation of their logarithmic response before filtering could be carried out, but which allowed wide dynamic range to be achieved at low cost. This linearisation was performed by software in the TMS32010. Since the data bus is 16 bits wide, both codecs could be read simultaneously, simplifying both the hardware and the software. A sampling rate of 8 KHz was used, with the DSP chip clocked at 15 MHz.

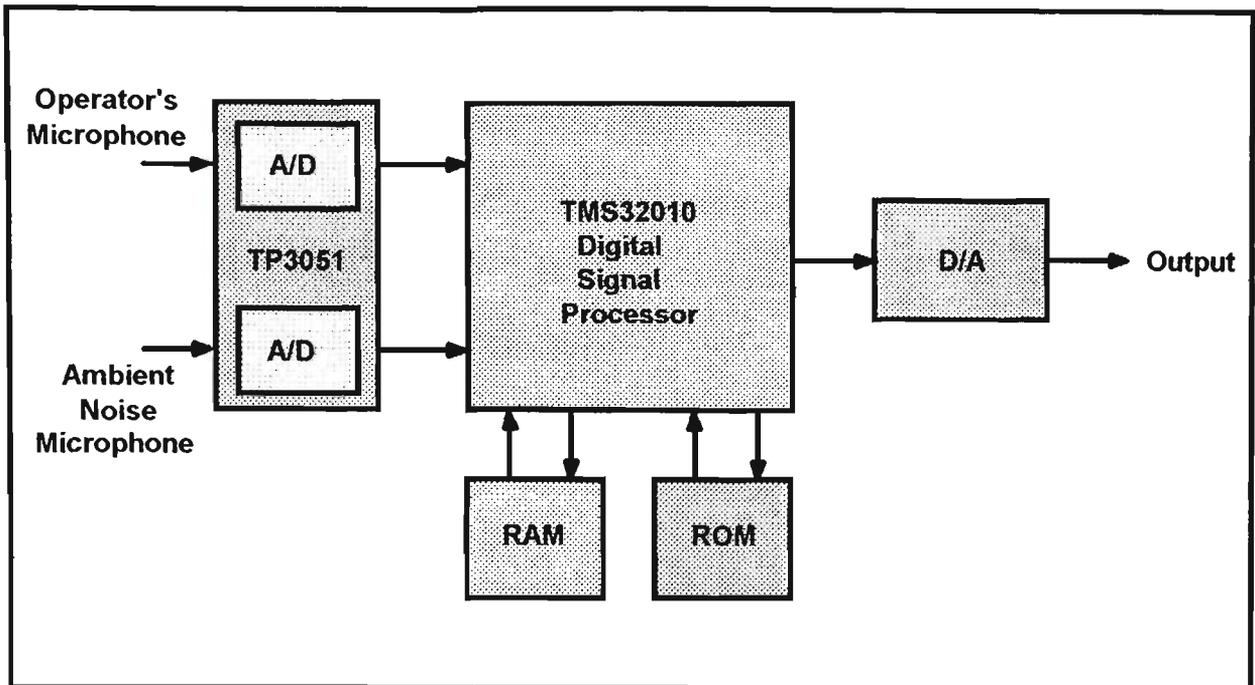


Figure 5.1 Adaptive audio filter and noise canceller

Various algorithms were investigated to produce Finite Impulse Response (FIR) adaptive filters, and to carry out noise cancellation by means of Fourier and Inverse Fourier transforms. Such adaptive filters have been useful in environments such as aircraft cockpits and on noisy factory floors. Cancellation of echoes and background noise can enable operation in environments previously considered impossible. No improvement could be measured in quiet laboratory conditions, where performance was already adequate. A small performance gain was noticed in the presence of moderate noise, and operation in the presence of significant noise was still impossible. Overall, little was gained considering the amount of effort involved.

Some form of noise gating, where the signal level must exceed a certain threshold before any speech is passed on to the recognition system, proved to be of benefit in reducing the incidence of false triggering due to background noise. However, when the operator speaks and the noise gate opens, any background noise present will still interfere with the recognition process. A combination of a noise gate, gentle application of an automatic

gain system, and a close talking headset microphone, provided the best results. Ideally the automatic gain system should have fast attack and very slow decay. It is an advantage if the gain is latched to the average level of the operator's voice, only changing if the speech level changed significantly over time.

5.1.3 Program Control of Microphone On/Off

One of the most useful facilities provided by the SAR-10 to assist the user in coping with a noisy environment where the onset of the noise can be anticipated is the control of the microphone on/off by program or by voice command.

In order to properly test the usefulness of this facility, the environment of a music studio was chosen. The SAR-10 was used to provide voice control to compose music, using Band-in-a-Box for Windows [PGMU91] and Musicator GS for Windows [BROD92], two PC based music composition and performance programs, and two music keyboard instruments: an ENSONIQ KS-32 weighted action MIDI studio [TRAC92] and a Roland JV-30 16 part multi timbral synthesiser [ROLA92]. MIDI is an acronym for Musical Instrument Digital Interface, the industry standard method of allowing electronic musical instruments and computers to communicate with each other.

During music composition using computers and synthesisers there is considerable noise while the music is being played, and silence while editing of the score is carried out. When using voice control, it is obvious that the microphone must be disabled during the performance of the music. This can be achieved automatically by embedding the SAR-10 *Mic Off* code into any command which causes music to be played.

The difficulty still remains of enabling the microphone again once the noise has stopped. When operating with a transparent keyboard handler, once the microphone has been

disabled by a *Mic Off* command, the only command the system will then respond to is *Mic On*. Provided none of the sounds made by the music system can be falsely recognised as the *Mic On* command, all is well. Of course, the chances of a true *Mic On* command being recognised while the music is playing are extremely remote, which means that some form of fail safe system may need to be provided - perhaps a chin switch or similar for a disabled operator. Otherwise, the operator will have to wait until the noise stops, if it does!

These programs greatly enhance the ability of a disabled user to compose and perform music. Band-in-a-Box, when given a chord progression, can automatically produce multiple instrument accompaniments in a large number of musical styles. The result can be exported to a standard MIDI file [DEFU89]. This file can then be read by Musicator, which provides the facilities to edit the music, enhance it by the addition of other instrumental parts, perform it, and print it out in manuscript form.

The results achieved using this system were gratifying, although manuscript editing displayed the same difficulties and tedium already discovered with text editors. It proved possible, with patience, to complete the cycle of creating, playing, editing, re-playing and re-editing required for music composition. Performance and printing of existing music scores was quite simple.

5.2 Vocabulary and Reference Pattern Control

5.2.1 Cluster Control

The use of the SAR-10 vocabulary clustering capability has already been mentioned. This provides considerable scope for the optimisation of the performance of the speech recognition system in any given application. By careful design of the clusters, and

selection of the vocabularies contained in them, many potential problems of ambiguity and false triggering can be circumvented.

The important principles in cluster design are:

- Minimise as far as possible the number of items in any cluster. This increases the likelihood of entries being orthogonal.
- Avoid having items duplicated in clusters which will be activated simultaneously. This helps to reduce the number of active entries, and avoids the difficulty of the recognition system having to distinguish between such very closely related vocabulary entries.
- Only put items in a cluster that relate to a single context. Other items present will reduce the recognition accuracy unnecessarily.
- Complex contexts can be covered by combining together a number of clusters designed for sub-contexts, or other similar contexts. The SAR-10 allows up to six clusters to be active at once, not including cluster zero which is always active.

5.2.2 Retraining and Updating Reference Patterns

The SAR-10 provides the capability of accessing the accuracy of reference pattern matching, and retraining and updating reference patterns under program control. If the recognition is not performing satisfactorily the user has the option of detecting this and retraining any desired vocabulary items. This sometimes becomes necessary if the environment changes, the user grows tired, is under stress, or even has a cold. Changes in the environment may be an alteration in the background noise, or simply a change in the acoustic characteristics of the room due to the presence of other people or equipment or furniture being moved.

The SAR-10 device driver produced for this project made use of these facilities to provide reference pattern retraining while still under voice control. An investigation of the possibility of providing automatic adaptive retraining was carried out also. When combined with adaptive control of the recognition parameters, as described below, this makes the task of the operator much simpler.

5.3 Parameter Control

Some improvement in recognition accuracy can be achieved by careful tuning of the speech recognition parameters of the SAR-10. Parameters available for adjustment are:

- Word beginning detection threshold.
- Word boundary detection threshold.
- Update score threshold.
- Self-learning flag.
- Self-learning score threshold.

Adjustment of these parameters under voice control has been provided.

The SAR-10 has a self-learning function, which can be enabled to allow it to automatically update reference patterns during recognition. The reference pattern is renewed by incoming speech only if its recognition score is lower than the current setting of the self-learning threshold. An attempt was carried out to make the SAR-10 more adaptive by accessing the value of the recognition accuracy parameter and using it to adjust all of the speech recognition parameters, including the self-learning threshold parameter. This is laborious if done manually. A heuristic which causes such adaption to operate effectively, would allow a significant performance improvement, and be especially useful for initial adaption to a new working environment. This will be the subject of further work.

Combined with adaptive retraining of reference patterns, adaptive parameter adjustment removes from the operator the need to determine empirically the direction and distance the parameters need to be changed in order to effect the desired improvement in recognition accuracy.

5.4 User Interface Optimisation

The one remaining avenue of performance improvement is via optimisation of the user interface. If we can reduce the possibility of user stress and fatigue, then we will go a long way towards improving the recognition accuracy and the performance of the entire man/machine system.

An important principle is that the interface language must be simple to learn and easy to use. It must not place a burden on the memory of the user; rather all necessary information should be immediately presented in its context. The choice of commands must be consistent throughout the interface, as must be their effect when executed. The user must be presented with no surprises which produce uncertainty as to what to do next.

Another important provision is that of an effective and reliable escape route from any situation, one which minimises damage to or loss of any work which may have been completed but left in a vulnerable state by a failure of command recognition.

A useful facility towards this end, and one which would be appreciated by a disabled user, would be a time out function. If a command has not been received for a certain adjustable period of time, then the *mic off* function is automatically activated. That the recognition system has been thus disabled should be clearly displayed on the screen, and by an audible prompt, to avoid the frustration of trying to command a system which is no longer

listening. This type of facility is of particular use to a disabled user, because when all else fails, the system can be rendered safe simply by sitting quietly back and doing nothing.

In summary, the requirements of an effective voice controlled user interface include:

- careful selection of vocabulary entries.
- clustering of related vocabulary items.
- simple, easy to learn and use interface language.
- commands presented in their context.
- consistent choice and effect of commands.
- reliable and effective escape route from any situation.
- fail-safe time-out operation.

6. EXPANDING THE VOCABULARY

The major limitation of the SAR-10 system in any substantial application is the small size of the vocabulary which can be loaded into the speech processing hardware. This is fixed to a maximum of 250 words or short phrases. In addition to this, if the system is trained to recognise a large number of words or phrases, then recognition accuracy drops owing to the difficulty of ensuring that all of the reference patterns are orthogonal.

6.1 Vocabulary Context Swapping from Disk

While careful use of clustering partially overcome the orthogonality problem, vocabulary size is still limited. A possible solution is to arrange for alternative, previously trained vocabulary reference patterns to be downloaded from disk or computer memory. In this way the entire context within which the SAR-10 is working can be swapped to suit a different application, or different contexts within a single application.

Similarly, the system could be trained for different speakers, each of whom can have their own set of reference patterns. This overcomes the problem of reduced recognition accuracy due to poorer reference pattern orthogonality, and smaller vocabulary size because of the larger number of training passes needed to achieve reasonable speaker independence. It does mean, though, that only one person, or smaller sets of people, can use the system simultaneously. When a different user is identified the appropriate patterns are downloaded, and those for the previous user, or group of users, are discarded.

Voice controlled user identification can be achieved in two ways. The first is for each user to leave the system in a user identification mode, where the vocabulary and reference patterns loaded have been trained by all of the users, using a personal

identification phrase. On recognising the user, that person's preferred operating environment and applications could be set up ready for use. This user identification mode would ideally be the mode into which the system enters when it is booted up, or when a user indicates he or she is finished.

The second way user identification can be achieved is by searching through the different users' reference patterns whenever the system starts consistently to fail to recognise the current speaker. Once a set of reference patterns is found the environment for that user is loaded.

Such a scheme also provides the basis of a security system based on voice recognition. If any speaker proves to be difficult to recognise, they could be prompted to identify themselves by their previously trained identification phrase. If recognition of this fails more than a set number of times the system could disable itself until reactivated by a valid user.

6.2 Determining and Controlling the Context

6.2.1 Context Determination

While the simplest and most reliable method of determining the most appropriate context for the system is for the user to make the decision, what may not be so simple is remembering the numerous contexts for which vocabularies are available, and remembering their contents in order to decide which ones are the most relevant to the current situation.

An efficiently designed help system is useful here. If the user decides to change context, and issues a voice command to that effect, and yet hesitates in naming the context

desired, then the help system can present information about the vocabularies available. The type of information desired would be the name of the vocabulary, the contexts for which it is designed, and the class of words contained - whether commands, cursor control, numerals, general words, etcetera.

If the application is menu driven or similar, then the help system could be selective in its presentation of data, only providing information about those vocabularies likely to be useful in the current context. Further requests for help would elicit more detailed information about the presented items, or information about the vocabularies not presented at the first request.

6.2.2 Manual Context Control

Manual context control requires the user to issue an explicit voice control command to change to a different context, naming the context desired. The application program, or the device driver in the case of transparent keyboard handler operation, then locates the appropriate vocabulary and reference patterns on disk or in memory and downloads them to the SAR-10.

This method is simple to implement, but does place a considerable burden on the user to know a great deal about the application. As one of our aims is to reduce the necessity of having to communicate in a machine dependent way, it should be considered as a last resort when all else fails. Unfortunately the present unreliable performance of low cost speech recognition and control systems make provision of manual control necessary as a backup for the following more automatic systems to be described.

6.2.3 Automatic Context Control

A logical extension of the facility to switch clusters by voice control is to use spoken commands to change entire vocabularies as well. As this is not available in the SAR-10 system, the simplest possibility is to provide it within the application program. However, when using existing applications by means of a transparent keyboard handler, it may not be possible to alter the application source code in this way.

A solution is to build a Terminate and Stay Resident (TSR) program to be loaded before the application program is executed. The TSR program is designed to respond to control keys, function keys and alternate keys which are not used in the present context of the application program itself. If these keys are used as "voice control" codes embedded in the vocabulary entries, they can cause the TSR program to perform the necessary downloading of the required vocabularies and reference patterns to the SAR-10.

Another approach is to incorporate the vocabulary context switching capability into the transparent keyboard handler itself. In either case, each application program would then be accompanied by its own resident program or programs and sets of vocabulary and reference pattern tables.

Of course, once embedded "voice control" commands for vocabulary context switching have been made available to the application program by means of a suitable TSR program, the user is no better off than for the manual control method unless the burden of determining the required context is also removed. The ability to do this will depend to a large extent on the design of the user interface of the particular application program. If the application's interface is based on a well designed menu system, then the voice commands for selecting each menu or menu item can now be made to contain the "hot-keys" for vocabulary context switching as well as for cluster control. For application programs with a different or less well designed interface, the extent of context control

available will vary, but in all cases some useful enhancement of the programs usability should be possible.

6.2.4 Programmed Context Control

When speech control is being incorporated into a new application at the time it is being written, or into an older application program for which source code is available, the task of providing context switching vocabularies is greatly simplified. Calls to the SARLIB library of SAR-10 control and operation routines, to be described in Chapter 8, may be incorporated at appropriate places in the user interface routines of the application program. In this way the speech control facilities can be made as simple or as sophisticated as the application builder desires. In addition, all of the retraining functions, and the audio response facilities can be put to use in the program.

The user interface may thus be simply adapted to account for the presence of speech control and response capabilities, or, as is preferable, the interface may be completely redesigned to make full use of the speech recognition system. Some of the traditional keyboard interface techniques are not the best way to approach speech control, and so, if the interface is rebuilt with speech in mind as the primary communication mode with the computer, life can be made much simpler, particularly for a disabled user.

The use of the programmed context switching capability to enhance a user interface will be subject to the same constraints as discussed for automatic context switching in the previous section, except that now the application designer has the option of ensuring that the interface is made suitable for voice control, instead of being restricted to the arrangement provided for keyboard or mouse control.

6.2.5 Artificial Intelligence and Context Control

Some applications present a particular difficulty when used with speech control. When working with an application such as a word processor, the context of most menus is fairly limited and require only a small vocabulary. However, once in the editing mode, and text is being entered into a document, the vocabulary required is far larger, most probably a great deal larger than the SAR-10 can handle. In this case, some understanding of the meaning of what is being entered is needed in order for the vocabulary to track this meaning as much as possible. A TSR natural language understanding system might be able to assist in this situation.

Even if the required vocabulary is not too large for the SAR-10, there is still the problem of recognising the editing commands in the presence of so many other words and phrases.

If the user restricted himself or herself to correct sentences, then the language understanding system could detect these and, as well as trying to determine the semantic context in order to adjust the vocabulary to suit, it could sensitise the system to editing commands between the input of sentences. If a non-editing command was then uttered and recognition of it therefore failed, the system could switch back to the currently selected general vocabulary and try again to recognise the word or phrase, on the reasonable assumption that the phrase was not a command.

Of course, in such a case, it would be necessary to provide a means of forcing the system to accept an utterance as a command rather than input text. This could be done in a manner analogous to escape sequences in a keyboard oriented environment. An easily recognised "escape" command, which is unlikely to appear in text, could be used to place the system in command acceptance mode. This would be especially useful for such

operations as formatting text or changing a font. Once the command has been carried out, the system can be put back into text entry mode using a suitable command. (See section 3.2.)

Such an arrangement could also form the basis of an editor which automatically assists the user with grammar and style. This would be of undoubted benefit to a disabled user.

The design of a suitable natural language system will be described later.

6.3 Storage and Transfer Considerations

If vocabulary context switching is implemented in a speech recognition system then the requirement for entire vocabularies to be downloaded from disk between utterances places some severe timing constraints on file transfer speed. There are several approaches which can be taken to provide the necessary performance.

The most obvious, but also one of the most expensive, is to provide a very fast hard disk, preferably with caching. In 1986, when this research program was started, such a solution was not affordable, but more recently the cost of high speed systems has fallen to the point where disk transfer speed is no longer a problem.

Similarly, the use of RAM disk emulators can provide extremely high transfer speed. As most modern systems have enough memory to be able to implement RAM disks, this also is a feasible solution. In fact, if memory is limited, a combination of a small RAM disk and an intelligent caching system which tries to determine from the current context which vocabularies might be needed next, and keeps them loaded down to the RAM disk, might be a good approach. Such a system could provide satisfactory performance with even an older, slower hard disk.

One way to improve file transfer rate is to reduce the size of the vocabulary and reference pattern files. This can be achieved in several ways.

The first way is to ensure that all vocabularies are as small as possible. This is a good general principle, assisting in maintaining recognition accuracy, but it also reduces vocabulary transfer time by trading off transfer speed against the need to swap vocabularies more frequently.

Another method is to compress the vocabulary and reference pattern files. This also reduces the need for disk storage space, but must be balanced carefully against the time needed to decompress the files after transfer to memory.

The third method, which is also a form of compression, is to take advantage of the ability of the SAR-10 to upload and download vocabularies and reference pattern tables in either ASCII or binary form. During development the tables can be used in ASCII form, where they are easier to read and edit. They can then be translated into binary form so that they occupy less disk space and can be transferred to the SAR-10 in approximately half the time taken to transfer an ASCII file.

The ability of the SAR-10 to handle both types of files could even be used as a "compiler" for vocabulary data, although this transformation can also be easily produced by a program. There is probably merit in using only the binary format to store data, and to provide for the transformation in the vocabulary editing utility.

7. VOICEDOS: A VOICE CONTROLLED OPERATING SYSTEM EXTENSION

The optimal solution to the provision of speech control and response facilities in a general purpose computer is to incorporate into the operating system the necessary software to control and use the speech hardware. If the speech interface is an integral part of the system in this way, and applications software is produced to run under that operating system, then the application programs need not be concerned with the nature of the speech system hardware. Rather, they simply make calls to the provided operating system interface routines. Even if application programs are not designed with speech control in mind, provided the operating system extensions are carefully integrated into the existing input/output routines, then some form of speech control will be available to the user.

As we have seen for application packages, effective incorporation of speech control into an operating system requires a standard speech control and audio response protocol to be developed, and programs which are to make use of the facilities will need to have their user interfaces designed accordingly. This removes the need for the application programmer to be concerned with the low level intricacies of communication with, and control of, a voice system.

7.1 Methods of Extending an Operating System

Voice control can be incorporated into an operating system in three main ways. The first is to write an entirely new operating system with speech as its primary input and output medium. This is a formidable task, but should be tackled once the development of voice control systems settles into some kind of generally accepted standard.

The second method is to replace the existing PC BIOS (Basic Input Output System) with a new one incorporating the required speech facilities. This might be the preferred solution for a computer which had the necessary voice control built in as standard equipment. The new BIOS could be placed in ROM.

The third method is to produce a memory resident program which intercepts the operating system calls from an application program and diverts them to appropriate voice control and audio response routines. These routines may then, if necessary, pass the call on to the original operating system function. This is a simpler task, allowing for easier experimentation and development, and is the method adopted for this project.

7.2 The VOICEDOS Control Program

The SAR-10's device driver allows the speech system to operate in parallel with the keyboard, called keyboard transparent recognition by NEC [NEC85]. It takes the form of a TSR program. A speech recognition and response training utility is also provided.

This software is not easy for an unsophisticated computer user to operate. Also, if while using an application program the user finds that recognition accuracy is decreasing, perhaps due to voice fatigue, then the application must be aborted before retraining of the speech system can be carried out. The small sizes of the recognition and response vocabularies are a limitation in any substantial application involving more than one environmental context.

To circumvent these problems a new control program has been designed, incorporating many of the improvements and extensions discussed in previous chapters. This program, called VOICEDOS, is presently implemented as a command processing shell around MS-DOS, but the intention is to develop it into an operating system optimised for applications

where speech is the normal means of communication, and keyboard use limited or entirely absent. The use of VOICEDOS is illustrated in Figure 7.1.

VOICEDOS is implemented in several layers, and is written in Turbo Pascal. The first layer is the memory resident command interpreter. Then there is SARLIB, a library of routines which interface the SAR-10 hardware with the command interpreter and application programs. The third layer consists of the speech system utilities.

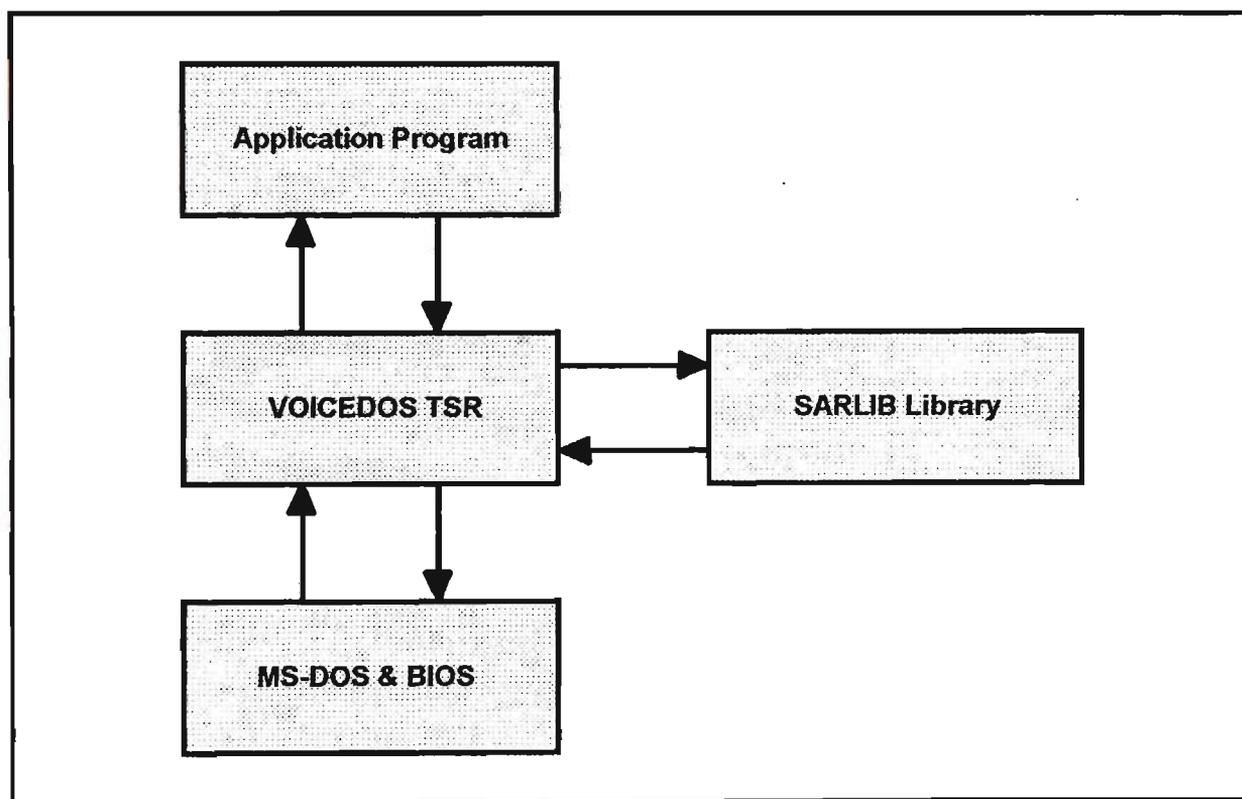


Figure 7.1 Use of the VOICEDOS Command Interpreter

7.2.1 The VOICEDOS Command Interpreter

The core of VOICEDOS is a memory resident command interpreter, designed to interface the numerous speech utilities and application programs with MS-DOS. This is patched

into the MS DOS keyboard interrupt, so that the voice command system operates in parallel with the keyboard. This routine, when summoned by a voice command, opens a window in the screen of the currently running application program, whilst preserving the context, and then allows full access to the voice control utilities.

VOICEDOS was initially implemented as a transparent keyboard handler similar to the one supplied by NEC with the SAR-10. The reason for this was that the NEC software proved to be inoperable in any PC with a clock speed faster than 4.7 MHz. It was deduced that the likely problem was in the order in which handshaking between the SAR-10 and the PC was carried out. It is suspected that the SAR-10 sets its data ready flag before it puts the data in the register, instead of after. If this is the case, it probably worked on a slower PC because its attempt to read the data was late enough for the data to be ready when needed. As soon as a faster machine was used, even a 6MHz IBM AT or an 8 MHz Turbo XT, it failed and returned a "Communication Protocol Error" message.

Rather than attempt to pursue this further with the manufacturer, and as the source code was not available, it was decided to construct a new keyboard handler and take the opportunity to incorporate some extensions such as vocabulary context switching, and monitoring of input/output interrupts in order to add audio response capability as well as reference pattern retraining and user identification capability.

7.2.2 The SARLIB Interface Routines

The interface to the SAR-10 is implemented via the command structure provided by NEC, and is transparent to the user. It is implemented as a library of Turbo Pascal routines, called SARLIB. This library is fully described in a technical report [DOW94a], but its essential features will be described here. They include the ability to invoke all the

primitives provided by the SAR-10 hardware as if they were Pascal functions and procedures. These routines are then used to build the more complex library routines.

The main facilities provided are - training and updating speech recognition patterns; digitising or recording audio responses; audio response output; performing speech recognition; recognition cluster control; setting and inquiring about parameters, flags and error status; deleting, uploading and downloading speech recognition or audio response patterns and vocabularies; and various control and memory housekeeping functions. This library of interface routines will be described in detail in Chapter 8.

7.2.3 Speech Control Utilities

The speech control utilities include facilities for updating speech patterns, retraining them if necessary, vocabulary extension or replacement, and testing of recognition performance. The audio response facility can similarly be trained and tested. These utilities are built up from the SARLIB primitives to provide a friendly environment for the user. Extensive use is made of pop-up windows and audible prompting, so that the user is never left in any doubt about what action is required or the commands that are available for use.

At all points in the design of this software, care has been taken to ensure that intelligent action is taken on the occurrence of any system error. A physically handicapped person will have even less time than other users for a system which might respond to a command to answer the telephone with "Error reading drive A - Abort, Retry or Ignore?".

The speech utilities are built around two main program modules. The first is a vocabulary file system, designed to enlarge the restricted vocabulary of the SAR-10 speech recognition and audio response unit, without reducing accuracy of recognition or speed

of response to an utterance. This file system is indexed by context numbers for fast access, and maintains as many disjoint speech contexts as are necessary for an application.

The second module comprises the routines needed to implement training and retraining of speech recognition patterns and audio responses.

7.3 Implementing Memory Resident Utilities

A major limitation of MS-DOS (partially overcome now by Microsoft Windows) is its lack of multitasking facilities. In voice controlled applications the ability to temporarily interrupt a running program in order to perform voice retraining is mandatory for reliable and convenient operation. In addition, for a program to intercept interrupts to the operating system and react to them in order to modify the behaviour of the operating system as seen by the application program, it is necessary for that program to be resident in memory.

7.3.1 Requirements Of Memory Resident Programs

The main requirement of a memory resident routine is that it is able to be loaded into memory by another program and then invoked when needed, either by its parent program, or directly by the user, without corrupting in any way the running environment of an already resident program. If a number of different routines are potentially resident, they must be removable and the memory occupied reallocatable to the operating system.

Since neither MS-DOS or the BIOS is re-entrant, care must be taken to prevent such routines calling each other or themselves recursively. This can occur particularly when different routines use the same BIOS facilities to perform I/O. Nor can non re-entrant

operating system facilities be used, since they may be already being used by the program whose execution was interrupted when the memory resident program was invoked.

7.3.2 Construction of Memory Resident Programs

VOICEDOS was implemented in the form of a TSR (Terminate and Stay Resident) program, but owing to difficulties in using the MS-DOS Terminate and Stay Resident interrupt (INT 27H) with .EXE format programs the MS-DOS Keep Process function was used (INT 21H, Code 31H), as recommended by Microsoft [ANGE86].

7.4 Making VOICEDOS Utilities Memory Resident

7.4.1 The Installation Program

Upon boot-up the program VoiceDosInstall is installed in memory as a filter for the hardware keyboard interrupt. Pressing the <Alt => key, or issuing a voice command which has been programmed to simulate the <Alt => key, invokes VoiceDosInstall, whose function is to load and execute another program from disk into a reserved memory area and execute it, then return to the original application without loss of context. VoiceDosInstall can remove itself from memory if the user desires by using it to invoke the utility VoiceDosRemove.

To filter the interrupts, VoiceDosInstall places the original interrupt vectors into interrupt vectors reserved for user programs. Interrupt #69 is used for the keyboard interrupt (#09). If these interrupts conflict with current use, they are easily changed by a utility provided. During the running of a resident routine the interrupts are restored to normal to prevent the routine from being able to invoke itself recursively, and so cause problems

with DOS non re-entrancy. An outline of the design of VoiceDosInstall is shown in Figure 7.2.

```
Program VoiceDosInstall
Begin
  Save current program status for later restoration.
  Open a window to display messages during installation.
  Check to see if VoiceDosInstall is already installed by looking at
    the identification field in the ISR control block (if it is there).
  If VoiceDosInstall already installed then
    Inform user and exit.
  else if VoiceDosInstall can be safely installed then
    Place vector #09 into #69 so that once #09 is changed
      the previous ISR can still be invoked.
    Prompt the user and wait for a key press.
    Install the new interrupt service routine.
    Remove the message window to conserve heap space
      for the application routine windows.
    Exit and remain resident.
End
```

Figure 7.2 VoiceDosInstall memory resident routine design

The keyboard Interrupt Service Routine (ISR) needs a local stack of sufficient size to handle any variables defined in its procedures. The positioning of the ISR stack above the stack segment base allows for the window routines to allocate needed space for screen images on the heap. Consequently, an amount of space must be allocated for stack/heap space during compilation, and that figure passed to the MS-DOS "Terminate and stay resident" function when terminating. The application program routines use their own windows. The windows are displayed when needed but removed when no longer required.

The routine in VoiceDosInstall which installs the ISR first installs the address of the ISR Control Block into the specified interrupt vector. The first bytes of the ISR Control Block make a call to the ISR dispatcher, placing the address of the ISR Control Block

information on the stack. Thus the dispatcher can access the information in the ISR Control Block. This routine also returns the previous vector so that when necessary it may be restored. Enough information must be passed to the routine to enable initialisation of the ISR Control Block.

Removal of a memory resident routine has been accomplished by a process which is largely the reverse of the installation process. This will not always work if other memory resident programs have been installed prior to the use of VoiceDosInstall as DOS is not always able to free the memory, but by keeping only one routine in memory at a time as this system does, no problems have been encountered.

7.4.2 The Interrupt Service Routine

Installing a memory resident routine is carried out by the procedure VoiceDosIsr. This interrupt service routine filters the hardware keyboard interrupt (interrupt #09). Every time an interrupt is generated by the keyboard or voice system this routine gains control. It immediately invokes the previous keyboard ISR whose address has been moved to vector #69. Upon return it checks to see if the key or extended key sequence is significant to the application. If so, it activates the application. An outline of the design of VoiceDosIsr is shown in Figure 7.3.

Procedure VoiceDosIsr

Begin

Save the CPU flags and issue interrupt #69 in place of the intercepted keyboard service routine.

On return from the keyboard service routine restore the original flags.

If a keystroke is ready **then**

If it is one we are interested **then**

Save the cursor status then remove the keystroke from the buffer.

Save the current keyboard interrupt vector.

Disable interrupts while resetting the interrupt vector

If the character corresponds with a particular application program **then**

Load and execute the desired application program.

Disable interrupt vectors and reset the interrupt vectors and cursor status to their original values.

End

Figure 7.3 VoiceDosIsr interrupt service routine design

7.5 VOICEDOS Utilities

Although all of the voice control facilities provided by SARLIB are available for use in application programs, it is desirable that several be provided as memory resident utilities to be invoked during normal system use. The most important of these are:

- train or update speech patterns
- upload or download vocabularies or speech patterns
- examine and/or change speech recognition parameters.

The SAR-10 system is relatively speaker independent once it has been trained using several different voices. However, best performance is obtained if the speech patterns have been provided only by the person currently using the system. Hence the ability to change quickly from one user's patterns to those of another person is a great advantage.

The ability to alter recognition parameters may aid an experienced operator to overcome the problems of noise or voice fatigue without resorting to the retraining of speech patterns. The parameters that this utility affects are:

- word beginning detection threshold (level)
- word boundary detection threshold
- update score threshold (an arbitrary measure of word recognisability)
- self-learning flag (the SAR-10 has a certain amount of self-training ability)
- self-learning score threshold

7.5.1 Setting the SAR-10 Recognition Parameters

This utility can be popped up over an application program to allow various parameters and flags of the SAR-10 to be adjusted. This is useful if the recognition performance begins to deteriorate, perhaps due to operator fatigue or changed environmental conditions.

Among the speech recognition parameters which are able to be set are the word beginning point detection threshold, and the word boundary point detection threshold. Careful adjusting of these two parameters can aid in achieving successful recognition in different background noise conditions. The values recommended by the manufacturer for these thresholds are - in quiet conditions: 10 to 30; in normal conditions: 30 to 50; and in noisy conditions: 50 to 80.

The matching score threshold for update, the self learning flag, the matching score threshold for self learning, the voice control flag, the reject buzzer flag, and the echo back flag are also adjustable. The audio response parameters which can be adjusted are the sampling frequency for digitising or recording and the end detection time for digitising.

7.5.2 Training the SAR-10 for Speech Recognition

The utility for training the SAR-10 to recognise words or phrases, when invoked, allows the user to either build a new vocabulary, edit an existing vocabulary, or to train or update all or any of the entries for recognition. In addition reference patterns can be uploaded from the SAR-10 for storage on disk, and download to the SAR-10 for updating, testing or use.

The information needed for each vocabulary entry is entered using a simple editor and consists of the following:

- The training message - this is the message displayed on the screen, telling the user what word or phrase he or she should speak when the SAR-10 signals them to do so by beeping. This utterance will be digitised and transformed into reference patterns, and used for later comparison during the recognition mode.
- Recognition output - this is the character string or hexadecimal code to be output to the application program upon successful recognition of the word or phrase being trained.
- Cluster number - this is the number of the cluster to which the current word or phrase being trained will be allocated.
- Recognition rejection threshold - this is the threshold below which recognition accuracy must be before the word or phrase will be accepted as a match with a stored recognition pattern.
- Voice control code - this is a code to specify either a set of up to six cluster numbers to enable, reset the active cluster number to that cluster specified by the Start Recognition command, or turn the microphone on or off.

The process of training a vocabulary entry is simple. The user is prompted for the file name which does or will contain the vocabulary and reference patterns. Then the user must select whether all words are to be trained, some words, or one particular word. The

system will accept "all words", "some words", or "word number ..." followed by the number of a word. If all words are selected then the training process will be carried out in sequence for each word, beginning with the first word. If some words are to be trained then the user will be prompted for the numbers of the first and last words of the group of words to be trained.

Once words have been selected, the number of training passes to be carried out must be entered, and whether the patterns are to be initially trained or updated. If initial training is selected then any existing patterns in the SAR-10 or the file are cleared. Before this occurs the user is asked to verify the command.

The process of training each word involves each word number and its training message being displayed on the screen. If the echo back function is enabled then the system will output the associated audio response message and ask the user to repeat it after the beep. Otherwise the user just speaks the displayed training message after the beep. When this has been repeated for the required number of passes the system moves on to the next word to be trained.

If updating of patterns is in progress, then if the new patterns do not achieve a recognition score less than the updating threshold, the system displays the minimum and maximum scores and does not replace the old patterns with the new ones. It then asks if the user wishes to continue trying.

Once training or updating has been completed the user may save the vocabulary and reference patterns to disk. At any time during the use of these utilities the user can request a display of the contents of the vocabulary as a reminder.

7.5.3 Testing the SAR-10's Recognition

If the recognition testing utility is called the system displays the vocabulary file name and each of the speech recognition flags, giving an opportunity for them to be altered. Then it, both audibly if echo back is enabled, and on the screen, asks the user to speak any word in the current vocabulary.

The utility tries to recognise the word, and displays its best two attempts along with their recognition scores and whether or not the best match was accepted as a valid word. If echo back is enabled then the audio response associated with a successfully recognised word is output.

This process continues until the user says to stop. At any time the effect of changing recognition parameters can be tested.

7.5.4 Training the SAR-10 for Audio Output

Training the SAR-10 for audio response is similar to training it for recognition. The only item to be edited in the vocabulary table is the word or sentence to be output. Alternatively the training messages from a speech recognition vocabulary can be copied directly into an audio response vocabulary.

The audio response parameters, sampling time and end detection time, and whether the trained audio response will be echoed back during training can be set, as can the name of

the file to contain the vocabulary. The user can select to train or update the audio responses, and which words will be trained. The system then steps through each of these words, asking the user to speak them after the beep. If the user selected audio output to be echoed back then the input speech will be output as an audio response after each utterance is trained. If the result is not satisfactory the user can elect to redo it before going on to the next word.

Another function of the audio response utility is that of macro building. Up to eight macros, which are a sequence of audio responses separated by short periods of silence, can be loaded into the SAR-10 or saved in a macro file. If macro building is selected then a simple editor will appear. This editor allows up to eight macros to be constructed on screen. Each macro consists of up to fifteen word numbers from the current audio response vocabulary, and a pause value of between 10 mSec and 2.5 Sec. The pause will be automatically inserted between each word when the macros are saved.

Once macros have been defined they can be tested in a similar manner to audio responses, and can be downloaded to the SAR-10 or saved on disk.

8. INTEGRATING VOICE CONTROL INTO A NEW APPLICATION PROGRAM

Chapter 8 describes the use of the SARLIB library in the construction of the VOICEDOS utilities. This library also forms the basis of the speech control and audio output facilities designed to be incorporated into new application programs.

8.1 The SARLIB Interface Routines

The interface to the SAR-10 is implemented via the command structure provided by NEC, but is transparent to the user of the system. It is implemented as a library of Turbo Pascal routines. The primitives provided by the SAR-10 hardware may be invoked as Pascal functions and procedures incorporated into the SARLIB library.

The construction of these routines uses the fact that the SAR-10 appears to the IBM PC as two parallel ports. Commands, data and status are transferred between the two systems using a simple handshaking protocol. Low level Pascal routines were written to send a command or data string to the speech processor, and to monitor the status port for command completion or an error return. Data is returned to the PC in a similar fashion. These routines are then employed to build the more complex library routines. The main facilities provided are shown in Appendix K.

These routines are implemented in a Turbo Pascal unit called *Sar10*, and may be grouped into three categories as follows:

- Control and test routines,
- Speech recognition routines,
- Audio response routines.

The routines provided are all listed in Appendix K. The complete SARLIB library is described in a technical report, "SARLIB: A Library of SAR-10 Speech Recognition and Audio Response Interface Routines" [DOW94a].

8.1.1 An Example SARLIB Routine

As an example of the design of these routines the algorithm for the Recognise First Speech Reference Pattern Candidate routine is shown in Figure 8.1.

```
Procedure SarRecogOne
Begin
  Command = 'RA'.
  Form a list of the active clusters. Append to the command.
  Send the command to the SAR-10.
  Get the SAR-10 response.
  If the response string begins with '>' then
    Get word number from response string.
    Get recognition score from response string.
    If word number = 999 then
      Recognition failed.
      Return failure code.
    else
      Recognition succeeded.
      Return word number.
  else
    Return error code.
End
```

Figure 8.1 Recognise First Speech Reference Pattern Candidate algorithm

8.1.2 SARLIB Utility Routines

The SARLIB utility routines are the main utilities for performing speech recognition and audio response, training for speech recognition and audio response, and testing speech recognition. They are constructed from the more primitive routines described above:

- SarRecogniseSpeech -- perform speech recognition
- SarAudioResponse -- output an audio response
- SarTrainRecognition -- train SAR-10 speech recognition
- SarTrainResponse -- train SAR-10 audio response
- SarTestRecognition -- test SAR-10 speech recognition

8.1.3 SARLIB Error Handling

When the system does not behave as expected error codes are returned. These codes indicate whether a SAR-10 error or a system error occurred, and what was the nature of the error. For example, the SAR-10 hardware memory may be full, or a vocabulary upload command may have been unable to access a disk drive. All of the error codes are listed in Appendix K.

8.1.4 SARLIB Data Structures

Information is transferred to and from the SARLIB routines by means of a standardised set of data structures, described in the Pascal types below.

Each cluster in a vocabulary is identified by a number, and the 32 possible clusters are grouped together in an array:

ClusterNoArray = Array[1..32] of Byte;

ClusterRec = Record
NoOfClusters: Byte;
ClusterNos: ClusterNoArray
End;

Speech recognition reference tables are stored in blocks of memory, allocated dynamically as required, and accessed through the following structure:

```
SRPatternRec = Record  
    NoOfBytes: Integer;    { No of bytes in pattern }  
    PatternPtr: Pointer    { Pointer to these bytes }  
End;
```

```
SRPatternTable = Array[1..250] of SRPatternRec;
```

Words in a vocabulary are identified in the SAR-10 by a number between 1 and 250, consistent with their position in a vocabulary table in the PC. The words themselves are not stored in the SAR-10, so their position number enables them to be identified. Each vocabulary entry is stored in a dynamically allocated record structure, and a complete vocabulary is stored by maintaining an array of 250 pointers to these structures, plus other information such as cluster number, reject threshold, and the voice control code and output code to be output upon recognition of this word.

```
VocabRec = Record  
    SRWordNo: Byte;  
    ClusterNo: Byte;  
    RejectThresh: Byte;  
    VoiceControlCode: Byte;  
    OutputCode: String32  
End;
```

```
VocabRecPtr = ^VocabRec;
```

```
VocabTable = Array[1..250] of VocabRecPtr;
```

Global speech recognition parameters and flag values are stored in static structures as follows:

```
SRParamRec = Record  
  WordBeginThresh: Byte;  
  WordBoundThresh: Byte;  
  UpdateScoreThresh: Byte;  
  SelfLearnFlag: Boolean;  
  SelfLearnThresh: Byte  
End;
```

```
RecogFlagsRec = Record  
  VoiceControlFlag: Boolean;  
  RejectBuzzerFlag: Boolean;  
  EchoBackFlag: Boolean  
End;
```

Audio response speech patterns and parameters are stored in similar structures as those for speech recognition. The structures for an audio output word or macro are variant records, the interpretation of their contents depending upon whether the entry is an output utterance or a period of silence.

```
ARWordRec = Record  
  Case WordNotPause: Boolean of  
    True : (WordNo: Byte);  
    False: (PauseLen: Word)  
End;
```

```
ARMacroRec = Record  
  Case MacroNotPause: Boolean of  
    True : (MacroNo: Byte);  
    False: (PauseLen: Word)  
End;
```

Audio output word numbers and macro numbers are stored in arrays of these structures. Any one output can be constructed from up to 32 concatenated single digitised words or phrases, or macros, separated by pauses from 10 mSec to 2.5 Sec.

A macro is a combination of up to 15 words or phrases, each separated by the same length of pause, adjustable from 10 mSec to 2.5 Sec. The SAR-10 can store up to 8 macros in its memory at one time, and these can be composed of any of the 250 words or phrases stored in the SAR-10 memory.

The audio response structures are defined as follows:

ARWordNoArray = Array[1..32] of ARWordRec;

ARMacroNoArray = Array[1..32] of ARMacroRec;

ARMacroDef = Record
 MacroNo: Byte;
 NoOfWords: Byte;
 WordNos: Array[1..15] of Byte;
 PauseLen: Word
End;

ARPatternRec = Record
 NoOfBytes: Integer; { No of bytes in pattern }
 PatternPtr: Pointer { Pointer to these bytes }
End;

ARPatternTable = Array[1..250] of ARPatternRec;

ARParamRec = Record
 SamplingFreq: Byte;
 EndDetectTime: Byte
End;

ARStatusRec = Record
 NoARWords: Byte;
 TotalTime: Word;
 TimeUsed: Word
End;

Detailed descriptions of the operation of each of the SAR-10 commands is contained in the SAR-10 User's Manual [NEC85], and will not be elaborated on here.

8.2 Using SARLIB in an Application Program

Using the SARLIB utility routines as a basis, more application specific routines can be constructed, analogous to more conventional input and output routines provided by

most languages. For example, the SARLIB rough equivalents to Pascal's *Readln* and *Writeln* are:

Function ListenPhrase(Var Phrase: String): Boolean;

Procedure SpeakPhrase(PhraseNum: Byte);

Procedure SpeakMacro(MacroNum: Byte);

The ListenPhrase function returns a Boolean result to indicate whether or not a phrase was successfully recognised. The returned parameter is in the form of a string rather than a word or phrase number. This is more convenient for programs which also incorporate a parallel keyboard input facility. The translation between word numbers and strings is easily accomplished by means of a lookup table, and helps to make programs easier to read. The same process could also be carried out for the audio output routines if desired.

It was also found desirable to expand the SAR-10 audio response macro capability to allow more complex audio responses to be constructed from a basic audio output vocabulary. It is likely that a single output vocabulary of 250 basic words could be then used for almost any application program. This was done by paralleling the SAR-10 hardware macro facility with a software version based on storing sequences of output vocabulary entry numbers in arrays and storing them in files. This gave the additional advantage of allowing the time period between components to be made variable, rather than constant as in macros. Using this approach the number and length of macros is determined by the storage available instead of the SAR-10 hardware.

A number of general purpose input and output routines were also constructed, using the SARLIB primitives, to perform all of the interface requirements of an application program. These include:

- Prompting a user, both audibly and via the screen, to input a value, either audibly or via the keyboard. A number of such routines were constructed, to accommodate the different inputs possible, such as strings, integers, floating point numbers, yes/no responses, etcetera.
- Input in a variety of window formats, with error checking, editing and correcting facilities.
- Output values, both audibly and visually, in a variety of formats.
- Output help files, both audibly and visually.

If a help file is composed of a small set of words it is a simple matter to train those words into an audio response vocabulary. When the help facility is invoked, the current audio response vocabulary and reference patterns can be swapped for those applicable to the help context, and the SAR-10 can be made to "read" the help file by finding the number of each word in a lookup table file and outputting it as an audio response.

This approach can be taken for more general applications apart from help files. Given a suitable vocabulary any text file can be "read" out, enabling talking programs to be produced.

Part II

Natural Language Understanding

9. NATURAL LANGUAGE UNDERSTANDING SYSTEMS

One method by which the use of a simple speech recognition system such as the SAR-10 can be made easier to use is to remove, as far as possible, any requirement on the user to speak in a certain manner. This is particularly beneficial if the restrictions that are removed are ones which force the user to adopt unnatural modes of speaking.

Because the SAR-10 is an isolated word or phrase recogniser there is nothing that can be done to remove the need to leave silences between trained words or phrases. What can be done, however, is to take advantage of the past few decades of research in the fields of theoretical and computational linguistics and implement a natural language understanding system.

9.1 Natural Language Interfaces

During the 1950s, and even the late 1940s [WEAV49], the desire for efficient computer translation from one natural language to another spurred much research into the development of language analysis programs. The difficulties involved in achieving this goal had been underestimated, and gradually funding was curtailed. Once it was realised that little could be achieved in translation without the accompanying ability to "understand" the material, more realistic aims were set [SLOC85]. More recently, effort has been applied to achieving machine assisted translation, automated information retrieval from textual material, and natural language interfaces for application programs [SNEL79] [PERR86].

9.1.1 Approaches to Language Understanding Systems

Researchers have approached natural language understanding from a number of directions. Initially a great deal of work was carried out on theoretical linguistics, and as computers became powerful enough the field of computational linguistics developed. Part of this is an attempt to describe language in a rigorous enough way that it can be analysed by algorithms based largely on formal logic [GRIS86].

The greatest difficulties faced by computational linguists are the ambiguities involved in natural language, and the reluctance of human languages, which are essentially infinite in structure, to confine themselves to strictly finite systems of analysis. In addition, much of the early work of theoretical and computational linguists was aimed at determining whether or not a string of words is a valid sentence, and at constructing sentences from a set of rules - a task where ambiguity does not prove to be a major problem. The task facing the constructor of a natural language system is the reverse of this - the unambiguous breaking down of a sentence into its components so that its meaning can be extracted and acted upon.

Rather than taking a theoretical approach, in order to construct a successful natural language interface, the task was approached from the point of view of an engineer. Elegance and rigour must take second place to a more pragmatic attitude that says, in effect, if it works then use it.

9.1.2 Processes of Language Analysis

A possible approach to language analysis is to consider a text or dialogue in terms of a sequence of sentences. We can concentrate on analysing a sentence as a unit. Then, once the sentences are understood, their meanings can be considered together in order

to try to establish the meaning of the complete text. This is an oversimplification, since any sentence in a dialogue will almost certainly only be entirely explicable when considered in conjunction with the other sentences. However, for the purposes of making progress, such a segmented approach is convenient [GRIS86].

Sentence analysis may be broken into two parts: *syntax analysis* and *semantic analysis*. Syntax analysis is concerned with discovering the grammatical rules which bind the components of the sentence together, while semantic analysis tries to uncover the meaning which results from combining the components in such a way.

As with sentence analysis and dialogue analysis, it is convenient to be able to separate the analysis of the syntax from that of the semantics of a sentence. Such modularity simplifies the construction of computer programs to carry out the analysis. Dividing the task in such a way does ignore the fact that grammatical structure and meaning are closely related. Because of this, much effort is being put into discovering how to make use of semantic information to aid in syntactic analysis [SCHA75] [RIES75] [RIES78] [HEND77a] [HEND77b] [CATE82] [KING83], especially disambiguation [HIRS87].

Once the sentences have been analysed and their individual meanings extracted and codified, then the meaning of the overall text must be determined and stored in some form. This brings in the fields of discourse structure and knowledge representation [GRIS86] [HOEY83] [WINT82].

Another area, related to but separate from language analysis, is that of language generation. This is necessary in any practical natural language interface which intends to provide anything more than the most rudimentary form of speech output in addition to speech input [GRIS86].

9.2 Syntax Analysis

There have been a variety of approaches to the analysis of the syntax of a sentence, including phrase structure grammar [HOPC79], linguistic string theory [SAGE81], generalised phrase structure grammar [GAZD85], word grammar [HUDS84], transformational-generative grammar [CHOM57], and many others.

Rather than following the traditional path of trying to accommodate the results of theoretical linguistics to the limitations of computer systems, some computational linguists have come from the opposite direction. They have taken the well defined grammars and theoretical structures developed for finite computer languages and tried to expand them to take into account the peculiarities of human language. By restricting themselves to a subset of a natural language, considerable progress has been possible.

Such an approach has a singular advantage in that it can put to use the powerful computational tools which have been developed to ease the task of producing compilers for computer languages. This is the method used to produce the parsers used in this project, as will be seen in Chapter 11. In addition, a goal of the present work is to adapt such tools so that they can be used more effectively in the construction of natural language analysers by being less constrained by the limitations of context free grammars. This is discussed more in the concluding chapter.

Computer languages generally use a context-free grammar [NAUR63] [AHO77], where the function of a word in a sentence does not depend on the functions of the words surrounding it. On the other hand, human languages are context-sensitive. Thus the description of a natural language using a context-free grammar must of necessity be incomplete and approximate, so most natural language interfaces only attempt to tackle a subset of a language. As natural languages are so large and difficult to reduce to a

formal grammar anyway, it is likely that using any method at all will result only in a subset of the language being understood, at least with the present state of the art.

One part of this project consists of an attempt to modify the tools and techniques used for computer language parsing so that they can take into account some of the context-sensitivity, ambiguity and variability of human languages.

A popular method of describing a natural language is to begin with Recursive Transition Networks (RTN), which can describe context-free languages, and to augment them with extra rules and conditions to accommodate special features of the natural language. Such a structure is called an Augmented Transition Network (ATN). This is the method followed in the present project, and uses the form of ATN described by Terry Winograd [WINO83], which will be described in Chapter 10.

9.3 Semantic Analysis

The analysis of the meaning of the sentence is an even more challenging task than syntax analysis. Researchers have used various approaches to date, with many of them revolving around an attempt to describe the sentence semantics in some kind of formal logic language or semantic network, or describing it in the form of a standardised script or frame of information [WINS84]. The position of the semantic analysis within the overall parsing scheme also varies. Margaret King [KING83] describes the three approaches as sentence-final [WOOD73], homogeneous [RIES75], and interleaved [WINO72].

Semantic analysis is complicated by the fact that many assumptions are made by a speaker about the context in which the sentence is uttered, beliefs and emotions, the world knowledge and world view of the listener, and many other factors. The later

stages of discourse analysis and knowledge representation can be of assistance in resolving some of these problems, which means that it is difficult to separate these functions of a language understanding system into discrete components.

In the present project the semantic analysis component is rudimentary, and probably can best be described in terms of a semantic network in which each action, object and subject are described in terms of a simple set of standard forms. The sentence-final method is used, that is, syntactic analysis is completed before semantic analysis is attempted. Further details are given in Chapter 14.

9.4 Discourse Analysis and Knowledge Representation

Discourse analysis is the attempt to understand the inter-relationships between the sentences and their actions, objects and subjects, so that the overall context and meaning of the text can be determined. One approach is to maintain a frame of information about the discourse, and to fill it in with any information gained during the syntax and semantic analysis phases. This information can be used during these phases to help to resolve some of their difficulties. After a final pass through the entire set of semantic networks produced by the semantic analysis, any gaps in the frame may be filled in.

When the context in which the system is working is known, as is the case in many application programs to which speech control is being added, much of the general information in the frame can be completed in advance. It then only remains for the language analysis to provide the specific objects, subjects and actions to be carried out. This form of simple system has been implemented in the present project.

10. SYNTAX ANALYSIS AND GRAMMARS FOR NATURAL LANGUAGE SUBSETS

This chapter describes the method of syntax analysis used in this project, the development of the grammar used and the augmented transition networks from which it was derived.

10.1 Words and Word Categories

The syntactic analysis of text or speech requires a categorisation of each word encountered in the input stream. Words in the English language can be categorised in a number of ways. The classification chosen here is as follows: *adjective, adverb, conjunction, interjection, noun, prefix, preposition, pronoun, verb, word element*.

This differs somewhat from those chosen by Terry Winograd [WINO83], whose grammar was used as the basis of the parser to be described. Winograd uses the following word categories: *adjective, complementiser, determiner, noun, preposition, pronoun, proper noun, relative pronoun, verb*. However, as each of the more general word types in the dictionary used in this project has associated with it a number of properties, it is not difficult to derive Winograd's restricted set of categories from those provided. This arrangement also permits the derivation of any other word category, facilitating ready expansion of the grammar.

The large number of features and properties associated with each word will be described in Chapter 12, where the development of the dictionary is discussed. The word categories used in this project are fully described in Appendix A: Parts of Speech for English.

10.2 Context-free Grammars and Parsing

10.2.1 Context-free Grammars

In order for text in a particular language to be analysed for its syntactic structure a specification of the rules and structure of the language must be available. The most common method for specifying a context-free language such as a programming language, or any other machine input language, is via a context-free grammar. The use of context-free grammars was originated by the theoretical linguist Noam Chomsky [CHOM56] [CHOM59].

Context-free grammars are often described using a notation called BNF (Backus-Naur Form), developed by Naur to describe the programming language ALGOL 60 [NAUR63]. The notation used in this report is a simplified BNF which is easier to read but retains all the expressive power of BNF.

Some language constructs are most naturally defined recursively. For example, we might want to derive a grammar rule to express the statement:

If *NP* is a NounPhrase and *Det* is a Determiner, then

***Det NP* is a NounPhrase.**

If we use the syntactic category *NounPhrase* to denote the class of noun phrases and *Determiner* to denote the class of determiners, then this can be expressed by the recursive rewriting rule or production

NounPhrase -> Determiner NounPhrase

Another way to read this is: "One way to form a noun phrase is to concatenate a determiner and another noun phrase".

A second example:

**If $A_1, A_2, A_3, \dots, A_n$ are Adjectives and NP is a NounPhrase, then
 $A_1 A_2 A_3 \dots A_n NP$ is a NounPhrase.**

Here, we could write:

NounPhrase \rightarrow Adjective Adjective Adjective ... Adjective NounPhrase

but the use of ellipses (...) would create problems when we attempt to define translations based on this description. Each rewriting rule or production must have a known number of symbols, with no ellipses permitted. To express this statement by rewriting rules, we can introduce a new syntactic category *AdjectiveList*, denoting any sequence of adjectives separated by spaces. The production then becomes:

NounPhrase \rightarrow AdjectiveList NounPhrase

AdjectiveList \rightarrow Adjective

\rightarrow Adjective AdjectiveList

The production for *AdjectiveList* can be read as: "An adjective list is either an adjective or an adjective followed by an adjective list", or, alternatively: "Any sequence of adjectives separated by spaces is an adjective list".

Sets of rules like the above constitute a grammar. In general, a grammar involves four quantities: terminals, nonterminals, a start symbol, and productions.

The basic symbols of which strings in the language are composed are called terminals. For example, English language words and punctuation symbols are terminals. The word *token* is often used as a synonym for *terminal*.

Nonterminals are special symbols that denote sets of strings. The expression *syntactic category*, used above, is a synonym for *nonterminal*. In the above examples, the syntactic categories *Adjective*, *AdjectiveList*, *Determiner* and *NounPhrase* are nonterminals.

One nonterminal is selected as the start symbol, and it denotes the language in which we are truly interested. The other nonterminals are used to define other sets of strings, and to help to define the language. They also help to provide a hierarchical structure for the language. In the grammars constructed for this project the starting symbol is *Sentence*.

The productions (rewriting rules) define the ways in which the syntactic categories may be built up from one another and from the terminals. Each production consists of a nonterminal, followed by an arrow, followed by a string of nonterminals and terminals, as can be seen in the above examples.

The productions may be applied repeatedly in any order to expand the nonterminals in a string of nonterminals and terminals, until eventually only terminals are left, and the expression has been parsed.

A graphical representation of the process of parsing a sentence is the *parse tree*. It makes explicit the hierarchical syntactic structure of sentences that is implied by the grammar. Each interior node of the parse tree is labelled by some nonterminal, say A, and the children of that node are labelled, from left to right, by the symbols in the right side of the production by which A was replaced in the derivation.

For example, if at some step in the parse we use the production

$A \rightarrow X Y Z,$

then the parse tree for that derivation will have the subtree shown in Figure 10.1.

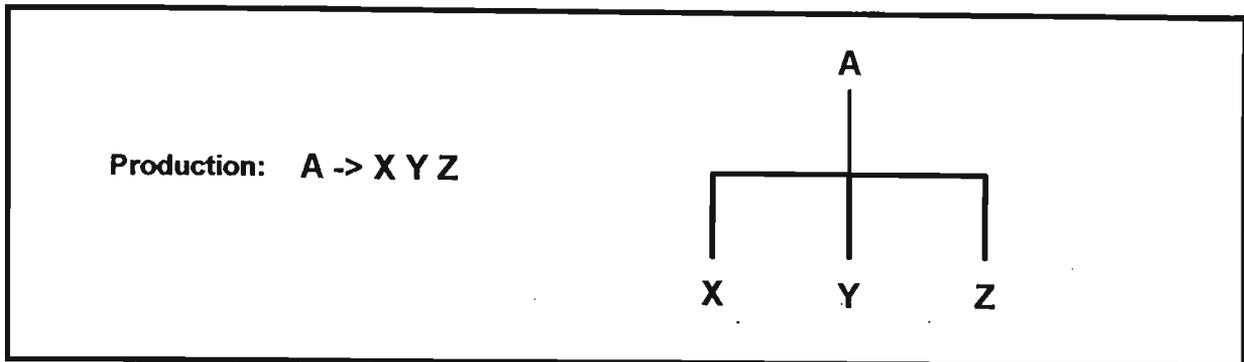


Figure 10.1 Subtree for parsing the production $A \rightarrow X Y Z$

For example, consider the grammar for a simple English subset shown in Figure 10.2.

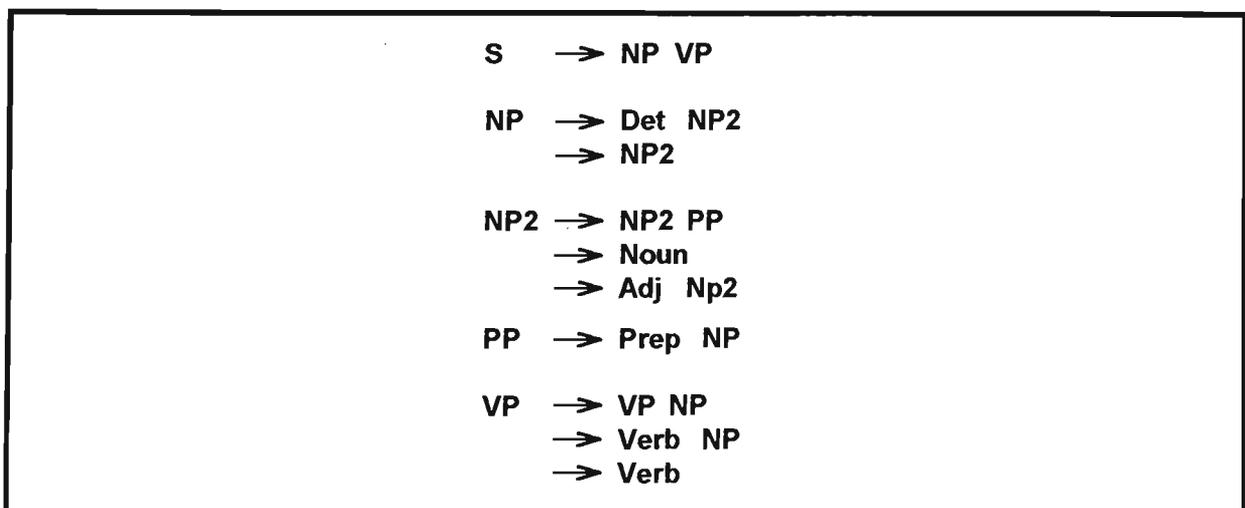


Figure 10.2 Context-free Grammar for a Subset of English

program. The scanner also removes all information which is not useful to the parser, such as spaces, tabs, line feeds and carriage returns (white space). The output of the scanner is a token, usually a number, representing the class of the symbol, and the symbol itself. In the case of a computer language, the symbol may also be placed in a symbol table. For natural language parsing systems the symbol table takes the form of a dictionary. The dictionary is searched for each symbol is encountered, and the features of the symbol retrieved.

Each token produced by the scanner is passed on to the parser which is used to identify the forms present in the input, and to call the relevant action routines to do the processing and possibly generate output. So, the total system might appear as in Figure 10.4.

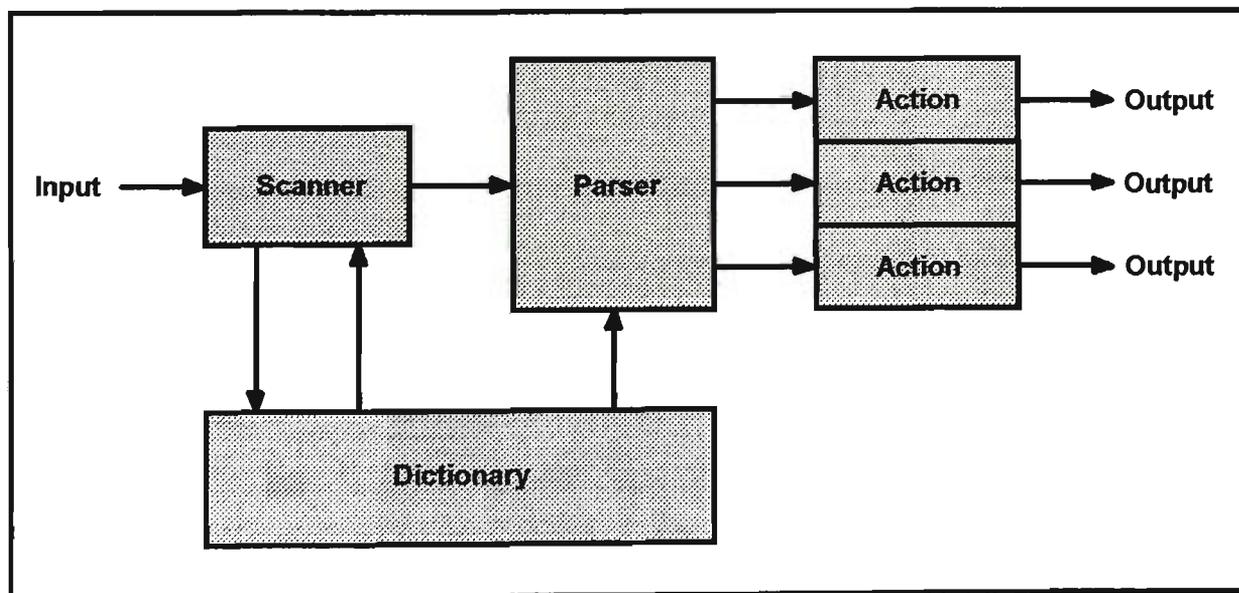


Figure 10.4 Block Diagram of a Parser

There are two ways in which a parser can be produced. It can either be coded by hand, or produced automatically from a description of the language to be parsed. One of the

most popular hand coding methods is that of *recursive descent*. While this method is easy to use by hand, it is not easy to automate. Two more recent methods of compiler production are called LL parsing and LR parsing. These parsers are difficult to produce by hand, but extremely suitable for automatic parser production. The most important of these is LR parsing. These parsers are called LR parsers because they scan the input from left-to-right and construct a rightmost derivation in reverse [AHO77]. The first parser constructed for this project used recursive descent, and although it was later replaced by an LR parser, it will be described, as it is considered desirable to include it in the computational linguistics toolkit which will be produced as a result of this work (see Chapter 15).

10.2.3 Recursive Descent Parsing

The grammar rules which are used to describe computer language or natural language grammars are often based on Recursive Transition Networks (RTNs) [WINO83], a graphical notation which clearly illustrates the recursive nature of such grammars. Because of this recursive nature, an obvious approach to constructing a parser is to write routines which call each other recursively. In the field of parser construction this is called *recursive descent*. It was a popular method of parser design, before the advent of automatic parser generators. The great bulk of compilers which existed in the early 1970s were constructed using this method.

Recursive descent does have one serious deficiency. Recursive descent can produce some unexpected results, particularly when backtracking is used to assist in the parsing process. It is possible, if care is not taken, for the parser to recognise inputs which are not in the language.

As an example of the application of recursive descent, consider again the very simple context-free grammar for a small subset of English, shown earlier in Figure 10.2. A recursive descent parser to analyse a sentence written using this grammar could be designed as outlined in Figure 10.5.

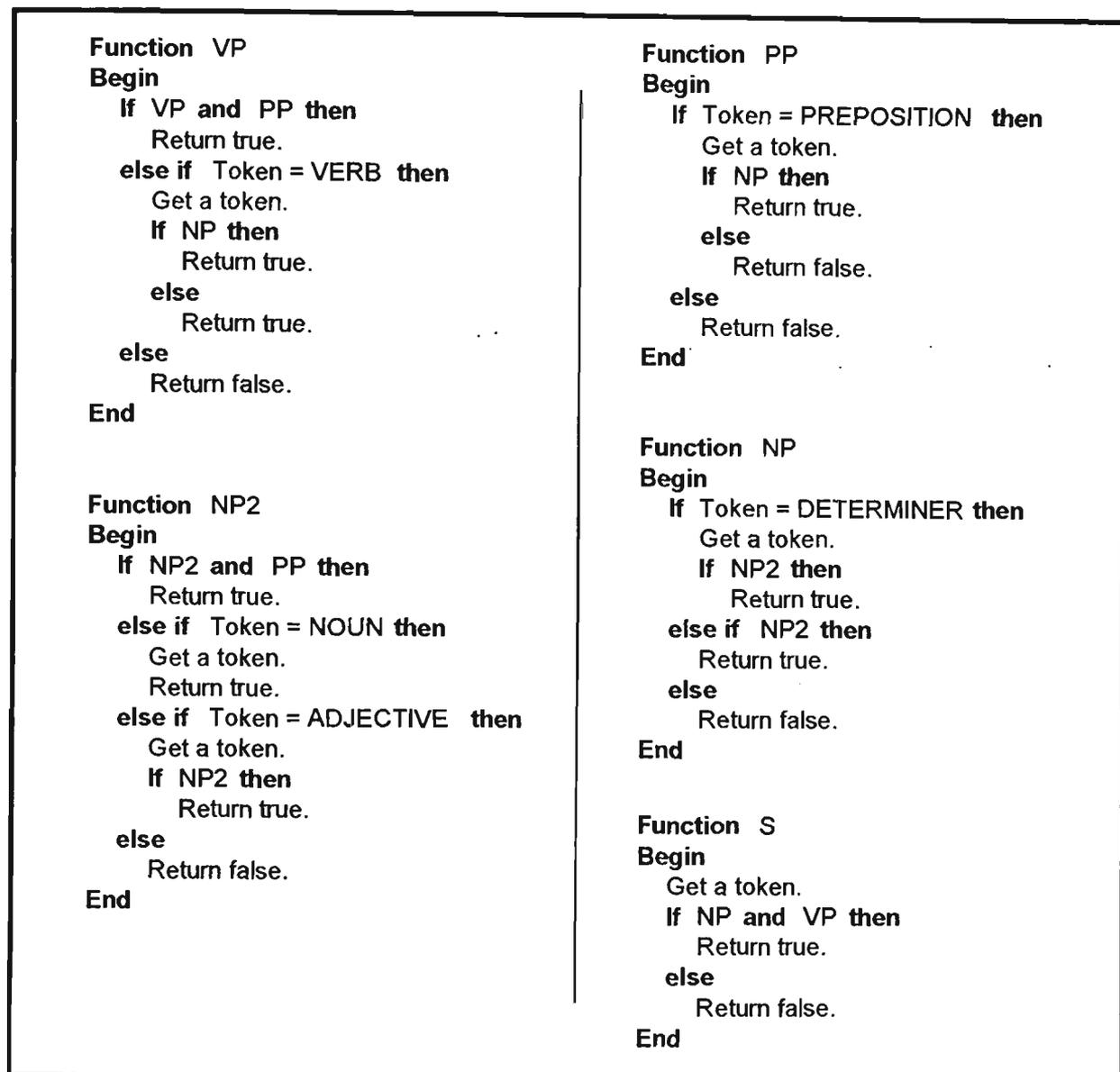


Figure 10.5 A Recursive Descent Parser

It is a relatively simple task to add calls to output action routines to such a parser, in order to cause it to fulfil a more useful role than that of a mere recogniser.

10.2.4 LALR(1) Parsing

LR parsing is the preferred method of parsing because it handles a larger class of grammars than the previous method. LR recognisers are small and fast, and can be generated automatically from a grammar. No backtracking is necessary, and syntax errors can be detected as soon as it becomes possible to do so on a left-to-right scan of the input. Their only disadvantage is that a parser generator is necessary for their implementation. But, of course, once a parser generator is available, this becomes an advantage.

Although LR(k) recognisers were first proposed by Knuth in 1965 [KNUT65], at the time they were not considered practical. DeRemer proposed a practical LR(k) recogniser in 1969 [DERE69] and LALR in 1971 [DERE71], and an efficient method for computing the LALR(1) look-ahead sets was presented in 1979, by DeRemer and Pennello [DERE79].

There are several varieties of LR parser. The first, sometimes called Simple LR (SLR), is easy to implement, but may not work for some grammars which the other types will handle. Canonical LR is the second method. This is the most powerful but is expensive to implement. Look Ahead LR (LALR) has a recognising power in between that of SLR and LR and will work with most languages, while not being too difficult to implement [AHO77].

The symbol k in LR(k) or LALR(k) represents the number of input symbols which the parser can examine at any one time to help to resolve potentially ambiguous situations as it parses the input. Usually 0 or 1 are sufficient [AHO77].

LALR(1) grammars are a subset of LR(1) grammars. LALR(1) parser tables are generated by first generating the LR(0) parser tables and then computing look-ahead sets

from the tables with some sort of graph traversal algorithm. LR(1) parser tables are generated by first generating the canonical LR(1) parser tables and then merging similar states.

The recognition power of an LALR(1) parser is only slightly less than that of an LR(1) parser, and most languages which can be recognised by an LR(1) parser can also be redefined in such a way that an LALR(1) parser would also handle them.

The following description of the operation of an LALR(1) parser is adapted from that of Aho and Ullman [AHO77].

An LR parser consists of two functional parts, a parsing table or set of parsing tables, and a driver routine. An LR parser generator is really a program which reads the grammar of a language and produces a parsing table or tables that represent the possible states of that grammar. These tables are combined with the driver routines, which may also be output by the parser generator, to produce a complete parser for the language described by the grammar, as illustrated in Figure 10.6. The tables and the driver routine together form a state machine.

The parser has an input, a stack, and a parsing table. The input is read from left to right, one symbol at a time. The stack contains a string of the form $s_0X_1s_1X_2s_2 \dots X_ms_m$, where s_m is on top. Each X_i is a grammar symbol, and each s_i is a symbol called a *state*. Each state symbol summarises the information contained in the stack below it, and is used to guide the shift-reduce decision (the meaning of which we will see shortly). The parsing table consists of two parts, a parsing action function *ACTION* and a goto function *GOTO*. This is shown in Figure 10.7.

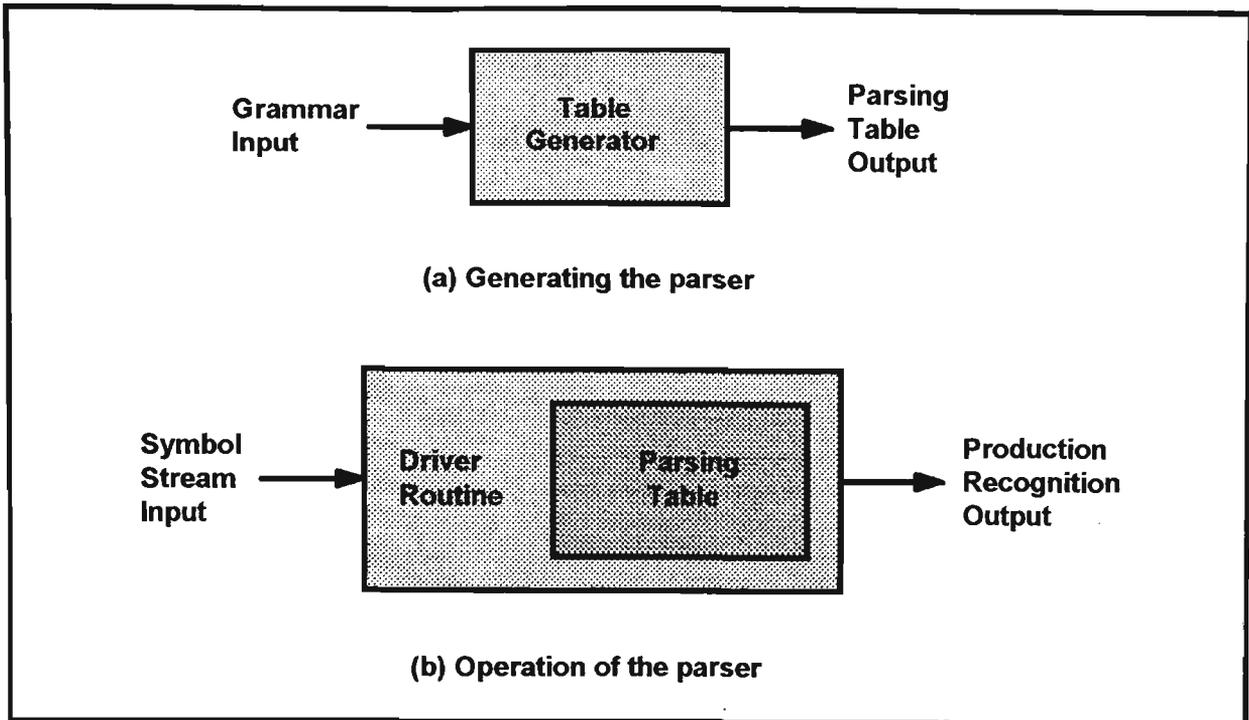


Figure 10.6 Building a State Machine Parser

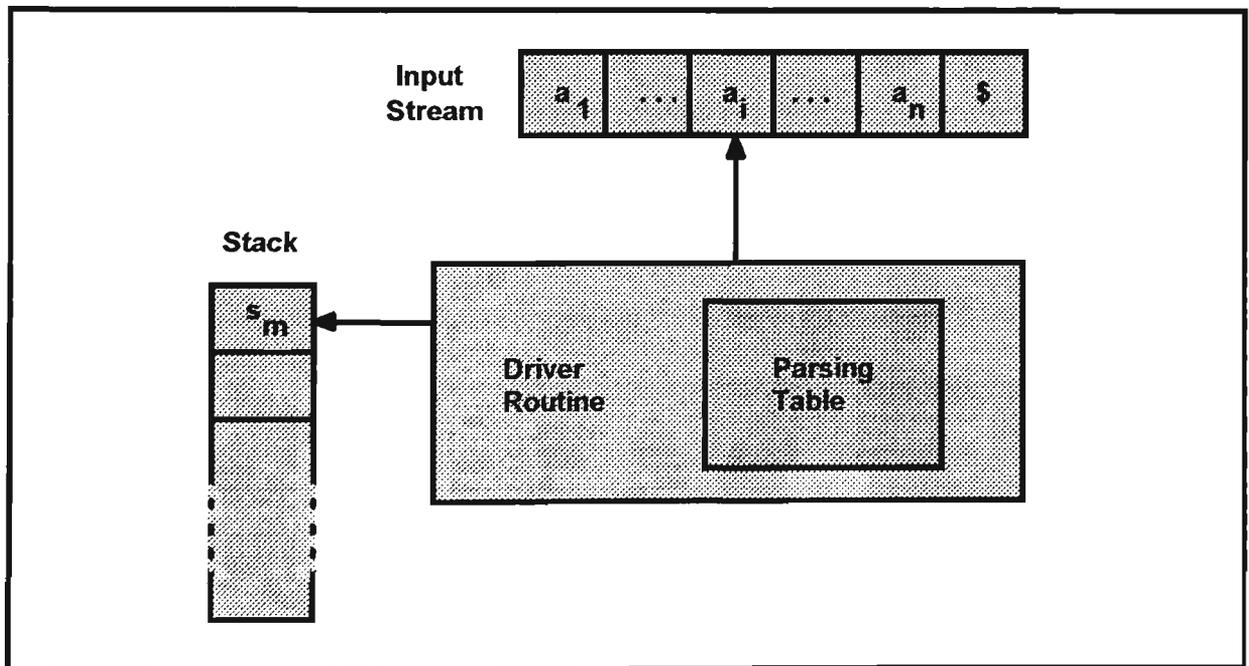


Figure 10.7 Operation of an LR parser

The program driving the LR parser behaves as follows. It determines s_m , the state currently on top of the stack, and a_j , the current input symbol. It then consults $ACTION[s_m, a_j]$, the parsing action table entry for state s_m and input a_j . The entry $ACTION[s_m, a_j]$ can have one of four values:

1. *shift s*
2. *reduce A -> B*
3. *accept*
4. *error*

The function *GOTO* takes a state and grammar symbol as arguments and produces a state. It is essentially the transition table of a deterministic finite automaton whose input symbols are the terminals and nonterminals of the grammar.

The four possible values that *ACTION* returns cause the following processes to be carried out:

1. If $ACTION[s_m, a_j] = \textit{shift s}$, the parser executes a shift move, in other words, the current input symbol a_j and the next state $s = GOTO[s_m, a_j]$ are pushed (shifted) onto the stack.
2. If $ACTION[s_m, a_j] = \textit{reduce A -> B}$, then the parser executes a reduce move. Here the parser first pops $2r$ symbols off the stack, r state symbols and r grammar symbols (where r is the length of B), exposing state s_{m-r} to the top of the stack. The current input symbol is not changed in a reduce move.
3. If $ACTION[s_m, a_j] = \textit{accept}$, the parsing is complete.
4. If $ACTION[s_m, a_j] = \textit{error}$, the parser has discovered an error and calls an error recovery routine.

The LR parsing algorithm is very simple. Initially the LR parser is in the configuration where s_0 , a designated initial state, is on the stack, and all input symbols $a_1 a_2 \dots a_n$ are waiting to be parsed. Then the parser executes moves until an accept or an error action is

encountered. All LR parsers behave in this fashion. The only difference between one LR parser and another is the information in the parsing action table and goto fields of the parsing table.

As well as the simple actions involved in the above description, the driver routine must also be able to resolve conflicts which arise when it is possible to go to more than one state from the present state. It is here that the ability to lookahead in the input stream is useful. If this does not resolve the conflict, arbitrary rules such as the first production of those in dispute will be accepted, or it may attempt to produce the longest possible language construct.

A *parser generator* is a program which, when given a grammar as input, automatically constructs the above tables as data structures, and the driver routines required to make a complete parser for that grammar.

10.3 Augmented Transition Networks

The context-free grammars described so far are equivalent to grammars which can be described by means of recursive transition networks. A *Recursive Transition Network* (RTN) is a set of nodes, or states, joined by arcs. To pass from one state to another an arc must be followed. This is illustrated in Figure 10.8, taken from Winograd [WINO83]. RTNs and context-free grammars have equivalent descriptive power. The RTN makes it easier to visualise the operation of a grammar, whereas the context-free grammar rules are more amenable to program construction and automatic parser generation.

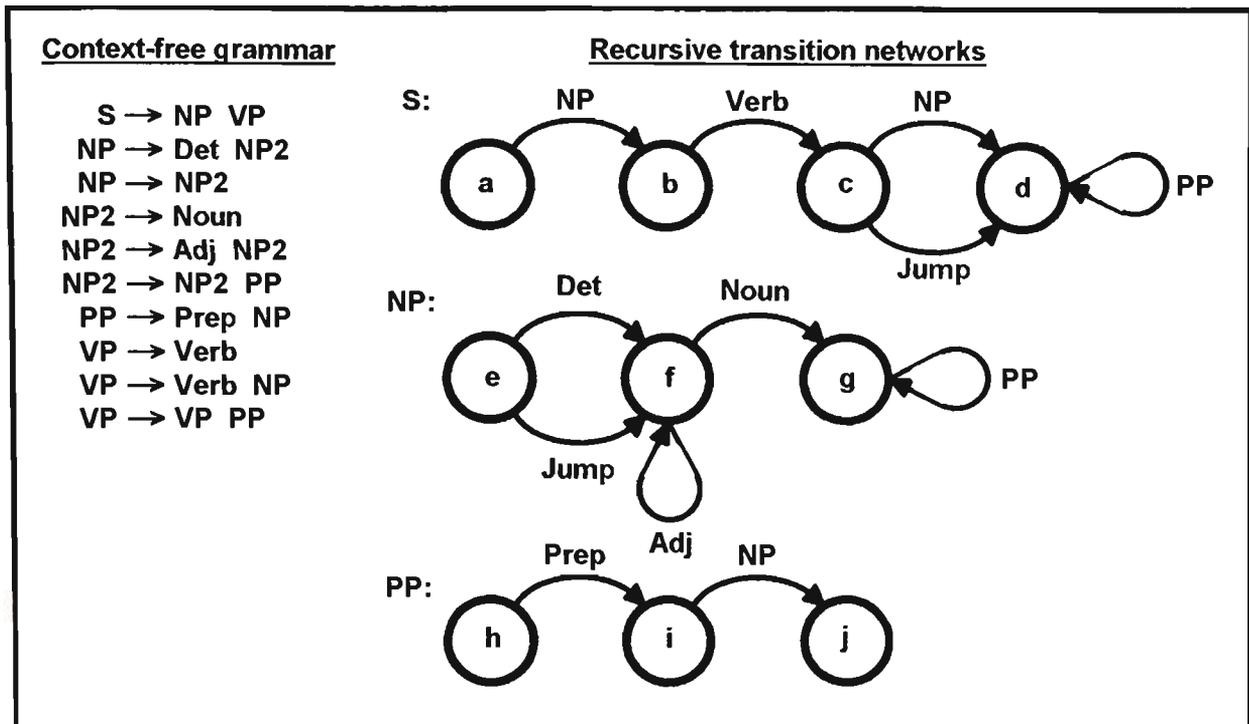


Figure 10.8 Equivalent Context Free and RTN Grammars

In order to try to accommodate some of the complexities of the English language, including the fact the English is not context-free, it is desirable to augment the arcs of an RTN with extra information. This may take the form of conditions which must be fulfilled before an arc can be taken, actions which are carried out if the arc is taken, and initialisations which are performed before an arc is taken. The resulting network is called an *Augmented Transition Network (ATN)*.

The ATN from which the grammar for this project was derived is based on that by Winograd [WINO83], with amendment to some of the augmentations to accommodate the different word categories used, and to correct a minor error in the original formulation. The complete ATN is shown in Figures 10.9 to 10.11.

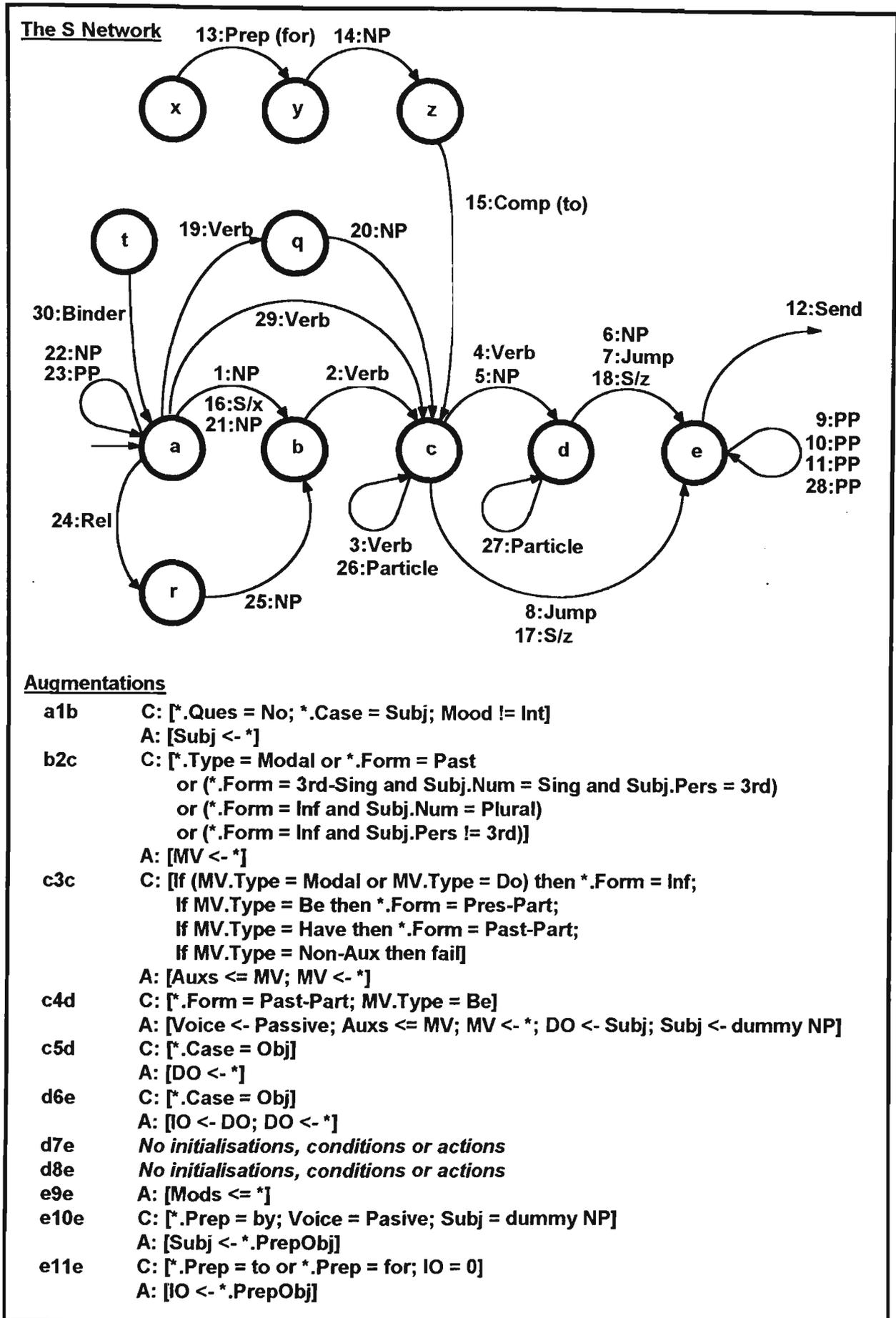


Figure 10.9 An ATN for a Sentence

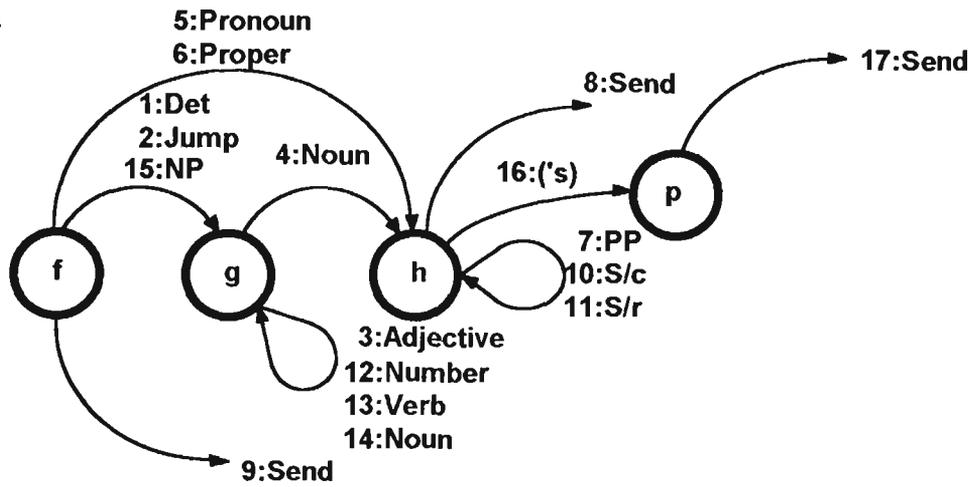
Augmentations (continued)

e12	C: [If (Mood = Int or Mood = WhRel) then Hold = 0; If IO != 0 then MV.Transitivity = Bitransitive; If (IO = 0 and DO != 0) then MV.Transitivity = Transitive; If (DO = 0 and IO = 0) then MV.Transitivity = Intransitive]
x13y	C: [* = for]
y14z	C: [*.Case = Obj] A: [Subj <- *]
z15c	C: [* = to] A: [MV <- dummy Verb; MV.Type = Modal]
a16b	A: [Subj <- *]
c17e	I: [Subj <- ^.Subj] A: [DO <- *]
d18e	I: [Subj <- ^.DO] A: [IO <- DO; DO <- *]
a19q	C: [*.Type != Non-Aux; Mood = Decl or Mood = Int] A: [MV <- *; Mood <- Int]
q20c	C: [*.Ques = No; *.Case = Subj; MV.Type = Modal or MV.Form = Past or (MV.Form = 3rd-Sing and *.Num = Sing and *.Pers = 3rd) or (MV.Form = Inf and *.Num = Plural) or (MV.Form = Inf and *.Pers != 3rd)] A: [Subj <- *]
a21b	C: [*.Ques = Yes; *.Case = Subj; Mood = Decl] A: [Subj <- *; QE <- *]
a22a	C: [*.Ques = Yes; Mood = Decl] A: [QE <- *; Hold <- *; Mood <- Int]
r24a	<i>No initialisations, conditions or actions</i>
r25b	A: [Subj <- *]
c26c	C: [(MV + *) @Dict] A: [MV <- Dict]
d27d	C: [(MV + *) @Dict] A: [MV <- Dict]
e28e	C: [(MV + *.Prep) @Dict; DO = 0] A: [MV <- Dict; DO <- *.PrepObj]
a29c	C: [*.Form = Inf; Mood = Decl] A: [Subj <- dummy NP; Subj.Head <- you; MV <- *; Mood <- Imper]
t30a	A: [Binder <- *; Mood <- Bound]

Figure 10.9 (cont.) An ATN for a Sentence

In order to use the grammar described by this ATN with an automatic parser generator, a method was devised by which augmentations could be added to a context-free grammar in such a way that the grammar is still acceptable to a parser generator, so that conditions, actions and initialisations contained in the augmentations are incorporated into the parser produced. The method developed is explained in Chapter 11.

The NP Network

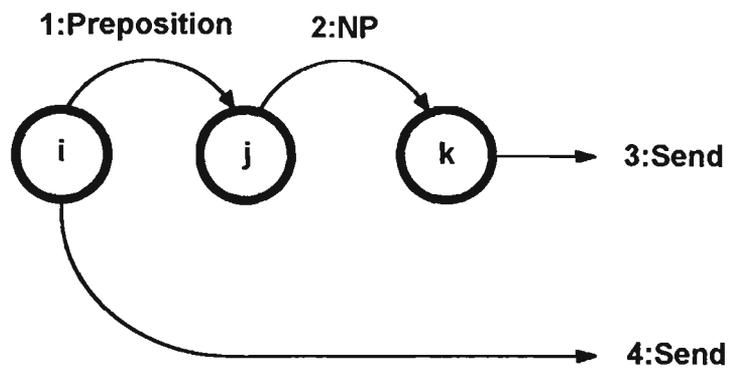


Augmentations

- f1g A: [Num <- *.Num; Ques <- *.Ques; Det <- *]
- f2g *No initialisations, conditions or actions*
- g3g A: [Desc <= *]
- g4h C: [* .Num = Num or Num = 0]
A: [Num <- *.Num; Head <- *]
- f5h A: [Num <- *.Num; Per <- *.Per; Ques <- *.Ques; Head <- *]
- f6h A: [Num <- *.Num; Head <- *]
- h7h A: [Qual <= *]
- h8 A: [Case <- Head.Case]
- f9 C: [Hold = NP]
A: [Hold <- Empty; Return Hold]
- h10h I: [Subj <- *COPY*; Mood <- Rel; MV <- dummy Verb; MV.Type <- Be]
A: [Qual <= *]
- h11h I: [Hold <- *COPY*; Mood <- WhRel]
A: [Qual <= *]
- g12g A: [Desc <= *]
- g13g C: [* .Form = Pres-Part or *.Form = Past-Part]
A: [Desc <= *]
- g14g C: [* .Num = Sing]
A: [Desc <= *]
- f15g C: [* .Case = Poss]
A: [Det <- *]
- h16p C: [Head.Cat != Pronoun]
- p17 A: [Case <- Poss]

Figure 10.10 An ATN for a Noun Phrase

The PP Network



Augmentations

- i1j A: [Prep <- *]
- j2k C: [* .Case = Obj]
 A: [PrepObj <- *]
- k3 *No initialisations, conditions or actions*
- i4 C: [Hold = PP]
 A: [Hold <- Empty; Return Hold]

Figure 10.11 An ATN for a Preposition Phrase

11. A NATURAL LANGUAGE PARSING SYSTEM

11.1 A Recursive Descent Parser

Because the ATNs which are used to describe the natural language grammar are based on recursive transition networks (RTNs), a natural approach to constructing a parser is to use recursive descent. This was attempted for the ATNs for English described by Terry Winograd [WINO83]. The size of such a task soon became evident. It was simple enough to build the recursive descent routines for the networks, but adding the initialisations, conditions and actions with which the RTNs were augmented to produce the ATNs proved to be tedious.

Much of the difficulty was overcome by devising a prototype for a typical recursive routine for a state in the ATN. This took the form of the code skeleton shown in Figure 11.1. These skeletons were filled out with the appropriate information for each possible state in the ATN, and a complete parser produced.

With the large number of states, and especially augmentations, involved in the ATN, the recursive descent parser proved to be large, rather slow, and difficult to modify as the grammar rules evolved. When the LALR Parser Generator [MANN87] became available, and its advantages for such a project became obvious, further development of the recursive descent parser was stopped. However, because of its usefulness as a teaching tool, owing to its human readability, it is planned to later produce an automatic generator of recursive descent parsers, based on this skeleton approach, and incorporate it in the computational linguistics toolbox to be developed (see Chapter 15).

```

Function State0( Var S: Sentence; Var WordList: WordArray;
                  WordNo, NoOfWords: Integer; Var Position: Integer;
                  Var CurPhrase: PhraseNodePtr): Boolean;

Var
  TempS: Sentence; TempPhrase: PhraseNodePtr;
  TempWordList: WordArray; TempWordNo: Integer;
Begin { State0 }
  If TempWordNo > MaxWords then { Over sentence length }
    Begin { Failure }
      State0 := Failure;
      Exit
    End;
  InitState; { Initialise state }
  { **** 1:Noun **** }
  If Noun(TempWordList[TempWordNo]) then
    If ( ... ) then { Special conditions }
      Begin
        ... { Special actions }
        If State1(TempS, TempWordList, TempWordNo + 1,
                  NoOfWords, Position, TempPhrase) then
          Begin { Success }
            State0 := Success;
            Exit
          End
        End;
      End;
    { **** 2:Noun Phrase **** }
    If NP(TempS, TempWordList, TempWordNo,
          NoOfWords, Position, TempPhrase) then
      If ( ... ) then { Special conditions }
        Begin
          ... { Special actions }
          If State2(TempS, TempWordList, TempWordNo + 1,
                    NoOfWords, Position, TempPhrase) then
            Begin
              State0 := Success;
              Exit
            End
          End;
        End;
      { **** 3:Jump **** }
      If ( ... ) then { Special conditions }
        Begin
          ...
          If State2(TempS, TempWordList, TempWordNo + 1,
                    NoOfWords, Position, TempPhrase) then
            Begin { Success }
              State0 := Success;
              Exit
            End
          End;
        End;
      { **** 4:Send **** }
      If ( ... ) then { Special conditions }
        Begin
          ... { Special actions }
          State0 := Success; { Success }
          Exit
        End;
      State0 := Failure { Failure }
    End; { State0 }

```

Figure 11.1 Recursive Descent Parser State Skeleton

11.2 The LALR Compiler Generator

For many years people who write system code for the UNIX operating system have been making use of two tools provided to make their task easier. These tools are LEX, a LEXical analyser generator [LESK75], and YACC, which stands for Yet Another Compiler Compiler [JOHN75]. These tools are designed to be used together. LEX produces a scanner (lexical analyser) whose output is compatible with a parser (syntax analyser) produced by YACC. Actions incorporated into the source code produced by YACC are executed whenever a language construct is recognised.

In 1987 Paul Mann, of LALR Research, produced LALR, an LALR(1) parser generator [MANN87] similar in performance to YACC, but considerably easier to use because it accepts the more modern style of grammar notation used in Chapter 10. This is more readable than the BNF (Backus-Naur Form) used by YACC.

LALR reads a user supplied grammar for a context-free language and outputs a complete working parser, in source code or a binary format. This parser accepts valid input in the language of the supplied grammar, and performs the actions specified in that grammar when it recognises a production rule. If it encounters a syntax error it detects the source of the error and attempts to perform error recovery by either substituting a correct symbol for the incorrect one, deleting an incorrect symbol, or inserting the symbol it expected to find. The decision as to which method is used is based on which will allow the parser to continue furthest before failing again.

LALR is comparable in speed, power and memory efficiency to the better known YACC parser generator. It can produce a parser for the Ada programming language in 28 seconds on an 8 MHz IBM PC, or a Pascal parser in 16 seconds. The parsers it produces are efficient in terms of memory space. The parser tables produced for Ada occupy about 15k bytes, those for Pascal only 9k bytes. This is very compact when you consider that a

complete C compiler, of which these tables would form a significant part, occupies 150k bytes. The parsers produced by LALR are also quite efficient in terms of speed. A C syntax checker built using LALR can perform both scanning and parsing at a rate of 10,000 lines per minute on an 8 MHz PC.

The LALR parser generator was originally designed to produce code for the Borland Turbo C compiler, but it can easily be adapted for other dialects of C. With a bit more work in producing suitable skeleton files, which guide the generation of source code, it can be made to produce output code in any other language as well.

11.2.1 An LALR Skeleton Using Turbo Pascal

As this project uses the Turbo Pascal language and the skeletons provided with LALR only produce C code, once it was decided to attempt to use LALR to produce a syntax analyser for a subset of English, a Turbo Pascal skeleton file needed to be developed. This essentially involved the design by hand, apart from the contents of the parser tables, of a full parser in Pascal. By this means the author became familiar with the intricacies of a full error detecting and correcting LALR(1) parser, which opened up the possibilities of extending and improving the parser design to handle some of the problems involved in natural language syntax analysis.

The design of this first LALR(1) parser is based on the C parser provided with LALR [MANN87]. Mann's original parser relied heavily on pointer manipulation, with dynamic memory allocation and a considerable amount of multiply-indirect addressing. As the size of all data structures is known before the parser is ready to be compiled, this was changed to static allocation. Pascal's very readable method of constant array initialisation was used to construct the parser tables, and the numerous *goto* statements were replaced

with properly designed Pascal control structures. Some of the original intersecting loop structures proved so difficult to unravel that those sections of the code were completely redesigned. The resulting parser is produced as a Turbo Pascal unit, as is the hand built scanner.

The result is a parser design that is much easier to understand, and has proved to be more adaptable to testing extensions to the LALR(1) parsing method to cope with syntactical ambiguity and other problems of human language parsing. Later, a parallel parser was developed, and the LALR skeleton for this is provided in Appendix B. The simpler state machine parser will be described first.

11.3 A State Machine Parser Using LALR

The first LALR parser to be constructed was a sequential design which tries to find one complete parse of the input sentence. If at any point it encounters a state from which it cannot proceed, it assumes an error has occurred and attempts to correct the error in order to resume parsing.

The basic error recovery system is based on the assumption that an error might be caused by three things: either a symbol has been omitted, a symbol is incorrect, or an extra unwanted symbol is present. Given these possibilities, when an error is encountered the parser tries out three possibilities: inserting the symbol it expected to find, substituting the expected symbol for the offending symbol, or deleting the offending symbol. The parser attempts to continue parsing with each of these alterations, and settles for the one which allows processing of the language to continue furthest, hopefully to completion. If it is not able to complete the parse then it aborts. The design is shown in Figure 11.2. The Recover routine carries out the attempt to correct any error encountered, and, if

successful, parsing continues. The Recover routine is itself a complete shift-reduce parser. The basic design of the parser is as shown in Figure 11.3.

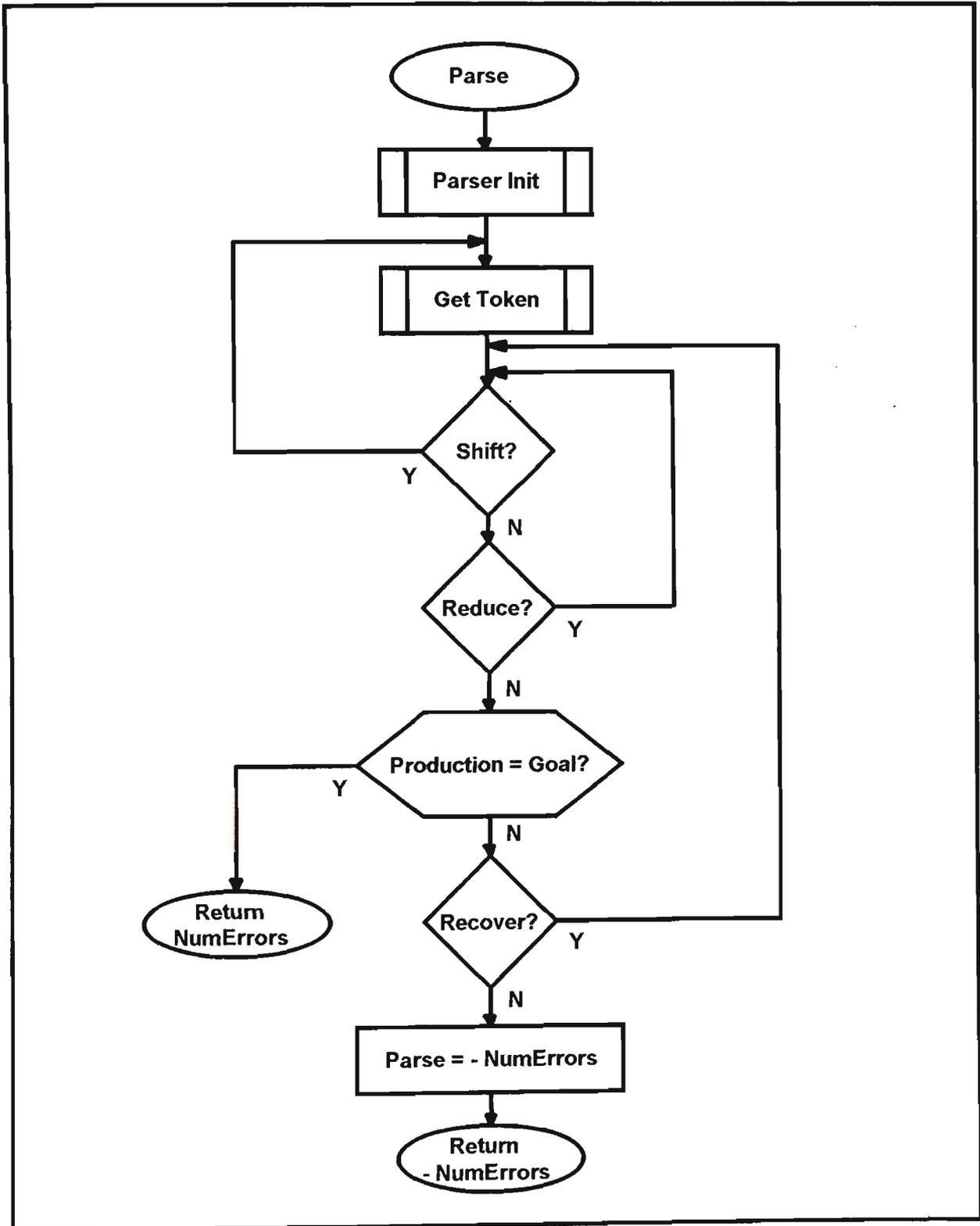


Figure 11.2 Error Correcting Parser

```

Function Parse: Integer;
Var
    Shifted: Boolean;
Begin { Parse }
    ParserInit;           { Initialise parser and get token }
    GetToken;

    Repeat
        Repeat
            Repeat           { Keep shifting until shift fails }
                Shifted := Shift;
                If Shifted then
                    GetToken
            Until not Shifted   { When shift fails, reduce }
        Until not Reduce;      { then try shifting again }

        If Production = Goal then
            Begin
                Parse := NumErrors; { Return success }
                Exit
            End
        Until not Recover;     { When shift-reduce fails, do error recovery }

    Parse := - NumErrors      { If can't recover, return failure }
End; { Parse }

```

Figure 11.3 Error Correcting Parser Algorithm

11.4 A Parallel Parser

While evaluating the sequential parser just described, several problems became evident. The first involved the decision to use error detection and correction, similar to that used in compiler generators. When applied to computer languages, with their relative freedom from syntactic ambiguity, such a scheme works well. With a natural language, where there is almost always more than one way to parse a sentence, and where there are a great many partial parses which look promising but cannot be completed, such a scheme is not satisfactory. The input might be a valid sentence, but the correct parse is not found because none of the assumptions used by the error correcting scheme are able to bring the parse to completion. Thus the parse is abandoned, when it should not have been.

Clearly, a parser which evaluates all possible parses is required. Initially this was attempted by making a recursive call to the parser when a fork in the sentence structure was reached, such as when a word could be both a noun and a verb. The existing parse continued with the noun, while a recursive call to the parser was made with the word as a verb. This scheme showed promise, but was abandoned when a better method was discovered.

Masaru Tomita constructed a parser which could handle ambiguous sentences by introducing the concept of a *graph-structured stack* to an otherwise standard LR parsing algorithm [TOMI87]. This allows an LR shift-reduce parser to maintain multiple parses without parsing any part of the input twice in the same way. Owing to the use of precomputed LR tables, as in the author's parser, Tomita's algorithm proved to be five to ten times faster than Earley's context-free parsing algorithm [EARL70].

Rather than duplicate Tomita's method, the recursion in the first design was abandoned and a control structure which simplified the design considerably, was developed. This uses Boolean flags to record the current state of the shift-reduce cycle, and a switching scheme to evaluate the current state of the parse and direct it into the correct next state. The basic shift-reduce parsing is carried out by a routine called DoParse, shown in Figure 11.4.

This scheme works well. It was realised that, if the complete context, such as the state variables, stack and current symbols, for each parse was stored in separate memory locations, then it would be a simple matter to extend the switching scheme to implement a parallel, or concurrent, parser, which evaluates every possible parse of a sentence simultaneously. In effect, a time sharing arrangement was used between the parses, with each parse completing one phase of its processing and handing on to the next parse.

```

Type
  ActionType = (NoAction, ShiftAction, ReduceAction, RecoverAction, GetNextToken);

Procedure DoParse;
Var
  NextAction: ActionType;
  Finished: Boolean;
Begin { DoParse }
  Finished := false;
  NextAction := ShiftAction;
  While not Finished do
    Begin
      Case NextAction of
        NoAction:
          Finished := true;
        ShiftAction:
          If Shift then
            NextAction := GetNextToken
          else
            NextAction := ReduceAction;
        ReduceAction:
          If Reduce then
            NextAction := ShiftAction
          else if Production = GOAL then
            Begin
              DoParse := NumErrors;
              NextAction := NoAction
            End
          else
            NextAction := RecoverAction;
        RecoverAction:
          If Recover then
            NextAction := ShiftAction
          else
            Begin
              DoParse := - NumErrors;
              NextAction := NoAction
            End;
        GetNextToken:
          Begin
            GetToken;
            NextAction := ShiftAction
          End
      End { Case NextAction }
    End { While not Finished }
  End; { DoParse }

```

Figure 11.4 Switching Shift-Reduce Parsing Algorithm

When a parse reaches a point where a decision occurs in the direction to be taken, then a new context is created and a new parse begun in parallel with the existing parse. This branching occurs when a word can occupy more than one part of speech.

In the parsing routine to be described next, the variable *ContextP* is a pointer to the current context values. A context consists of a record structure of the form shown in Figure 11.5, organised into a context stack. The record is shown here to illustrate the amount of information which needs to be maintained for each fork in the parse.

```

Type
Context = Record
  State: Integer;           { Current state number }
  Terminal: Integer;       { Current terminal number }
  Head: Integer;           { Current production head }
  Production: Integer;     { Current production number }
  Token: CharPtr;          { Ptr to current scanner text position }
  Input: CharPtr;          { Ptr to next scanner text position }
  TokenBeg: CharPtr;       { Ptr to start of current token }
  TokenEnd: CharPtr;       { Ptr to end of current token }
  NumNextShiftStates: Integer; { Number of next shift states }
                               { for a state/terminal }
  NextShiftStates: StateArray; { Next shift states for a state/terminal }
  NumNextRednStates: Integer; { Number of next reduction states }
                               { for a state/nonterminal }
  NextRednStates: StateArray { Next redn states for a state/nonterminal }
  LineNum: Integer;        { Current text line number }
  LineStart: CharPtr;      { Current text line start }
  DictData: DictRec;       { Returned word data from dictionary search }
  PartsOfSpeech: WordTypeSet; { Set of word types for this terminal }
  ParseStackTop: ParseStackPtr; { Ptr to top of parse stack }
  RednStackTop: RednStackPtr; { Ptr to top of reduction stack }
  NumErrors: Integer;      { Number of errors from this parse }
  PrevContext: ContextPtr; { Link to previous context in context stack }
  NextContext: ContextPtr  { Link to next context in context stack }
End; { Context record }

```

Figure 11.5 Context Record for a Parallel Parser

Each new context has such a record on the stack to contain a complete description of its current state during the parse. Each time a routine such as *GetTokens*, *Shift*, or *Reduce* is called, it is passed a pointer to the context record for the current context in which it is operating. In effect, each of the routines is time shared among all of the contexts currently in existence; that is, each of the current partial parses. The routine for the parallel parser is defined as shown in Figure 11.6.

This *DoParse* routine is called from the *Parse* function, shown in Figure 11.7, which initialises the context stack, begins the parsing process, and controls the stepping through the contexts that are created and placed on the stack as parsing proceeds. When *Parse* terminates, all possible parses of the input sentence will have been traversed.

When any of the partial parses reaches an error condition, instead of attempting error correction the parse is simply abandoned and its context storage space released back to the operating system. By removing the need to carry out multiple parses in the error correction phase, it was found that, on average, it was quicker to try to find all possible parses than to try to produce a single correct parse via error detection and correction. The parser itself is also considerably smaller and simpler than the sequential version, although it does need more memory in which to run. It also has the enormous advantage that if the sentence is valid then at least one correct parse will be found. As already explained, this could not be relied on with the previous design.

```

Type
  ActionType = (NoAction, ShiftAction, ReduceAction, GetNextTokens);

Procedure DoParse(ContextP: ContextPtr);
Var
  NextAction: ActionType;
  Finished: Boolean;
Begin { DoParse }
  With ContextP^ do
    Begin
      Finished := false;
      NextAction := ShiftAction;
      While not Finished do
        Begin
          Case NextAction of
            NoAction:
              Finished := true;
            ShiftAction:
              If Shift(ContextP) then
                NextAction := GetNextToken
              else
                NextAction := ReduceAction;
            ReduceAction:
              If Reduce(ContextP) then
                NextAction := ShiftAction
              else
                NextAction := NoAction;
            GetNextTokens:
              Begin
                GetTokens(ContextP);
                NextAction := ShiftAction
              End
          End { Case NextAction }
        End; { While not Finished }
      If Production = GOAL then
        Begin
          With ContextHead do
            NumSuccessfulParses := NumSuccessfulParses + 1;
          Exit
        End
      else
        Begin
          DeleteContext(ContextP);
          Exit
        End
      End { With ContextP^ do }
    End; { DoParse }

```

Figure 11.6 Context Switching Parallel Parser Algorithm

```

{ Parse -- find all possible LALR parses }
Function Parse: Integer;
Var
  ContextP: ContextPtr;
  AllDone: Boolean;
Begin { Parse }
  With ContextHead do
    Begin
      ParserInit(ContextP);           { Initialise the parser }
      GetTokens(ContextP);           { Scanner gets the first terminal }
      Repeat
        AllDone := true;             { Tell parser it is finished }
        ContextP := ContextStackTop; { Step through the contexts }
        While ContextP <> nil do
          Begin
            DoParse(ContextP);       { Perform the parse }
            If not ContextP^.Finished then
              AllDone := false;      { If any parse not finished tell parser }
              ContextP := ContextP^.NextContext { Get the next context on the stack }
          End
        Until AllDone;             { Parsing is finished }
      Parse := NumSuccessfulParses
    End { With ContextHead do }
End; { Parse }

```

Figure 11.7 A Parallel Parser

If the sentence has more than one possible parse, then it is desirable, as happens with this parser design, to have all of these parses available. Further processing may then be carried out, perhaps using semantic information, to determine which parse is the most appropriate to the context of the sentence. A parser which satisfies itself with only the first example it finds of the possible parses, may return the least appropriate parse in the context of the sentence. The user will have no way of knowing this, or even of knowing that there may have been alternative parses which were more satisfactory.

The source code for the parallel parser is provided in Appendix C.

11.5 Adapting LALR for Use with Augmented Grammars

Two possibilities have been explored for extending the parser produced by LALR to accommodate the initialisations, special conditions and special actions (called *augmentations*) which Winograd incorporated into his ATN for the English language. The first is the tedious task of editing the parser itself, or its skeleton before producing the parser, adding the necessary code to provide the desired augmentation. This is not a satisfactory solution, however, owing to the varied nature and large number of augmentations.

The most suitable way to extend the parser is to incorporate the necessary information into the grammar itself in such a way that the initialisations, special conditions, and special actions are inserted into tables by the parser generator, and to modify the parser skeleton to provide code which makes use of these tables.

LALR was not designed with such augmentation in mind, but it has proved to be possible to press into use a facility it does have, called *descriptors*, which were intended for a different purpose. By this means the concepts have been proved, and a future task will be to build a new parser generator which incorporates these features in a more convenient form.

To describe this method of augmenting the grammar it is necessary to understand some of the syntax of LALR's input.

11.5.1 The Grammar for LALR

A grammar for LALR must have the following order of definitions [MANN87]:

- **Terminal definitions**
- **Goal definition**
- **Nonterminal definitions**

Terminal definitions have one of the forms:

Terminal Descriptor?

Terminal Descriptor? => Input-processor

Terminal Descriptor? => Input-processor Input-argument

Terminal is any grammar symbol. Descriptor is optional, indicated by the question-mark, and may be a string or a number. Input-processor is the name of a routine for processing the input terminal, and may have an argument. Terminal definitions are used to describe the terminals to the scanner, and to specify any input processing which needs to be carried out. For example, a symbol table may need to be searched to check if the input terminal is a keyword of the language being processed.

There will always be one goal definition, and it is the first production of the grammar. It has the form:

Goal -> Symbol <eof>

where Goal, Symbol and <eof> may be any grammar symbol. It specifies the goal of the parser, in this case a sentence.

Nonterminal definitions have one of the following forms for their first production:

Head Descriptor? ->

Head Descriptor? -> Tails

Head Descriptor? -> Tails => Output-processor

Head Descriptor? -> Tails => Output-processor Output-arg

and one of these forms for subsequent productions:

-> Tails

-> Tails => Output-processor

-> Tails => Output-processor Output-arg

The symbol '|' can also be used instead of the arrow '->'.

Head may be any grammar symbol. Descriptor is optional, and may be a string or number. Tails may be any number of grammar symbols. Output-processor, and its optional argument specify an output processing routine to be called if the production is recognised by the parser.

11.5.2 Grammar Augmentation Using Descriptors

The reason for the provision of descriptors in the grammar is not made clear in the LALR manual, but they would clearly allow the insertion of comments or other information about terminal or nonterminal symbols, which would then be accessible to the parser routines for output if necessary, maybe as part of an error description or user help system.

The feature of descriptors which is useful in the present application is that any string placed in the grammar as a descriptor in a nonterminal production will become part of an array of such descriptors in the output of the parser generator, along with indexing information to link it with the nonterminal definition from which it came. If we insert in the descriptor a string of information which describes the initialisations, special conditions, and special actions relevant to that nonterminal definition, then these can be accessed by the parser which is generated.

Unfortunately, a descriptor can only be a single string or number. As the information which needs to be placed here generally amounts to a number of different items, a string is necessary. This string can be parsed as part of the parser initialisation process, and the information converted into a more useful form. The alternative is for the string to be interpreted during the running of the parser, but this would considerably slow the operation of the parser, so compiling the string during initialisation is considered to be better.

A language had to be devised by which the descriptor strings may be used to represent the desired items of information. The following scheme, based on the notation used by Winograd in his ATN descriptions [WINO83], has been adopted:

- **A: represents an action**
- **C: represents a condition**
- **I: represents an initialisation**

Each of these is followed by a list of items in brackets [...]. The list items use the notation described in Figure 11.8.

Referencing:	
C.R	The R of C
R.last	The last member of R
COPY	A copy based on ^
dummy X(xxx)	A dummy X with word = xxx
Initialisations:	
R1 ← ^.R2	Initialise R1 to the R2 of ^
Actions:	
R1 ← R2	Set R1 to R2
R1 ← R2	Append R2 to R1
Conditions:	
C.R = X	The R of C is X
C.R != X	The R of C is not X
R = 0	R is empty
R != 0	R is not empty
R = xxx	The word in R is xxx
(R1 + R2) @ Dict	R1 and R2 share a dictionary entry
<p>In the above, R stands for a register (possibly *, the ATN node most recently parsed); C for a constituent; X for a feature or constituent; and ^ for the node from which a recursive call was made.</p>	

Figure 11.8 Augmentation List Notation

A complete description of the ATN, with all of the initialisations, conditions and actions, is given in Appendix D.

The system used to produce the augmented grammar from the ATN was to segment the network into self-contained parts which are common to several different ways of traversing the network, write productions for these sub-networks, and then to add productions to complete the grammar. The segmenting of the sentence network is shown in Figure 11.9.

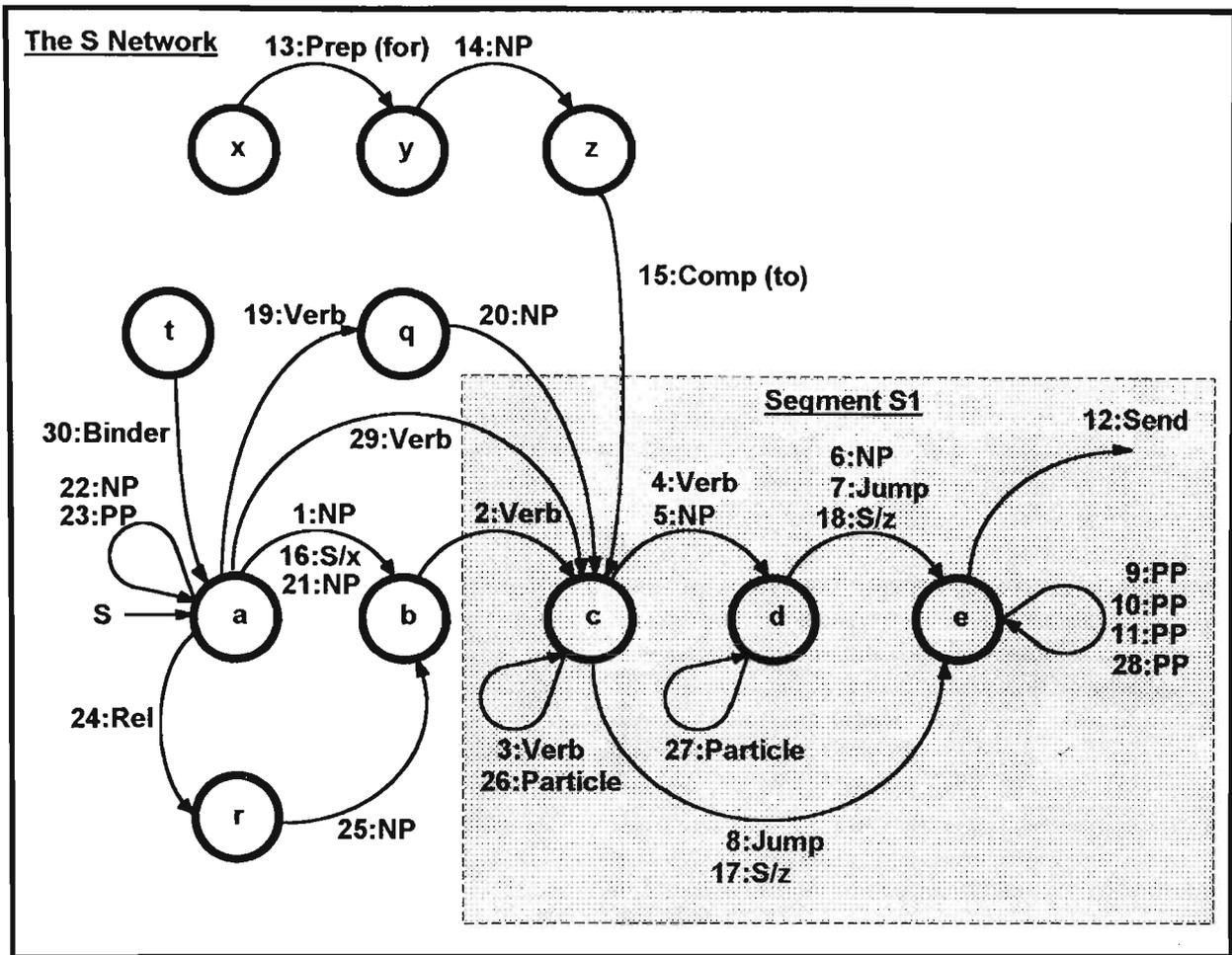


Figure 11.9 Segmenting the Sentence ATN

The head of each production is named for the ATN states which the arc joins together, separated by the augmentation number relevant to that arc. A sample of some of the augmented grammar rules can be seen in Figure 11.10. A complete listing of the augmented grammar may be seen in Appendix E.

<u>Goal Production:</u>	<u>Arc Productions:</u>
Sentence → S <eof>	Saa → → Sa22a → Sa23a
<u>Sentence Network Alternate Entry Points:</u>	Sab → Sa1b → Sa16b → Sa21b
S-c → S1	Sac → Sa29c . . .
S-r → Sra S → Srb Sbc S1	Sa1b "C:[*.Ques=No;*.Case=Subj; Mood!=Int] A:[Subj<-*]" → NP
S-t → Sta S	Sb2c "C:[*.Type=Modal or *.Form=Past or (*Form=3rd-Sing and Subj.Num=Sing and Subj.Pers=3rd) or (*.Form=Inf and Subj.Num=Plural) or (*.Form=Inf and Subj.Pers!=3rd)] A:[MV<-*]" → Verb . . .
S-x → Sxy Syz S1	
S-z → Szc S1	
<u>Sentence network:</u>	
S → Sac S1 → Saa Sab Sbc S1 → Saq Sqc S1	
<u>S1 Segment:</u>	
S1 → Scc Scd Sdd Sde See Se → Sce See Se	

Figure 11.10 Some Augmented Grammar Rules

11.5.3 Building an Augmented Parser

Because with the present parser generator the augmentation information is in the form of a string which needs to be parsed in order to produce suitable code for the conditions, actions and initialisations, producing a parser for an augmented grammar becomes a two pass process. The alternative is to interpret the augmentation string at run time, and it is felt that the reduction in parsing speed this would entail is unacceptable.

The first pass in generating the parser is with LALR, using the augmented grammar as input. The parser skeleton has been modified to produce an array of augmentation strings. It also places calls to functions and procedures which will carry out the intent of

these strings into the appropriate places in the parser code. These functions and procedures are given names consisting of the number of the relevant production appended to the words "Action", "Condition" or "Initialisation". For example, the names of the action, condition and initialisation routines for production number 15 would appear as Action15, Condition15 and Initialisation15 respectively.

At the same time as LALR produces these function and procedure names and calls, it also produces another file, called ACTCONIN.GRM, containing the routine names and their corresponding augmentation strings. LALR puts a Uses statement in the parser file for the unit, called ACTCONIN, to be produced from this file in the second pass described below.

The second pass in the parser construction process is to parse the ACTCONIN.GRM file, using another parser produced by LALR. This produces a file called ACTCONIN.PAS consisting of the code for a Turbo Pascal unit containing the augmentation routines and their interface to the main parser.

The condition strings produce functions which return a Boolean, indicating if the condition is true or false. The action and initialisation strings produce procedures. The functions and procedures all take as a parameter the current node being parsed.

The parser for the augmentation strings has been constructed using LALR and a grammar for the augmentation language described in the previous section. This grammar is shown in Appendix F.

12. THE DICTIONARY

At the heart of any natural language understanding system is a lexicon, or dictionary. As a sentence is parsed, the part of speech represented by each word and the values of certain features associated with each word need to be known. These feature values are stored along with the word entry in the dictionary. Later, during the semantic analysis phase, information about the meanings associated with each word also needs to be obtainable from the dictionary.

12.1 The Choice of Words for the Dictionary

Owing to three constraints - adequate vocabulary size, memory size and retrieval time - the choice of words to be incorporated in the dictionary is necessarily a compromise. The naturalness and flexibility of conversation attainable only when sufficient words are available must be balanced against the need to reserve sufficient memory for the other parts of the system, and against the longer retrieval time inherent in searching a larger dictionary.

12.2 A Small Dictionary

The initial approach taken was to construct the smallest dictionary which was likely to be useful. This was constructed based on information from three principal sources. The first source was a basic book of English grammar [MALL44]. From this was obtained the more commonly used pronouns, irregular verbs, auxiliary verbs, prepositions and conjunctions. This book also provided the framework for the feature information associated with each part of speech.

It is commonly known that much of everyday speech is carried out using a vocabulary of as few as 500 words. The 'Good News Bible' [BIBL76] is primarily intended for use by those who do not have a high level of reading ability. For this reason its text is largely restricted to a vocabulary of about 500 words. By performing an analysis of parts of the text of this Bible about 300 additional words were found for the dictionary.

The third source was Dolch's Basic Sight Vocabulary [MYER76], a list of 219 words commonly used by children. This is frequently used to teach literacy to children who are intellectually handicapped. The combination of these three sources, along with other words which became evident during testing of this dictionary, resulted in a total vocabulary of about 1000 words.

A short period of testing demonstrated the inadequacy of this dictionary. While many basic concepts were represented by the words available, they were generally present as only one part of speech. For example, the word *help* was in the dictionary, but *helped*, *helping*, *helper* and *helps* were not. Another problem was that while most of the words used primarily to give a sentence a correct grammatical structure were present, the nouns and verbs which are more closely related to the context of the sentence were missing.

Missing parts of speech can be tackled in two ways. Extra words can be inserted into the dictionary, or generated from words already in the dictionary. The second method was chosen to prevent the dictionary growing too large. After ensuring that the dictionary contained one basic form of a word, often the verb or noun, rules for the addition of prefixes and suffixes were used to produce additional words from these roots.

What was required was not the generation of words by the addition of prefixes and suffixes, but the reverse of this process. A word is presented for testing. If the word itself is not found in the dictionary, then the possibility of the word having been derived from

another word by the addition of prefixes and/or suffixes must be considered. To do this the word must have any possible affixes identified and removed. The presence of the resulting simpler word in the dictionary is then checked.

The only solution to the lack of contextual variety is to add words to the dictionary. This was done in two ways: by incorporating the 3000 familiar words identified by Dale in readability studies [DALE48], and by scanning through the Macquarie Dictionary [MACQ81], selecting the simpler forms of words which seemed likely to occur in normal conversation. The dictionary was expanded to approximately 4000 entries.

12.2.1 Prefix Processing

It is difficult to include all desired words in a small dictionary. Instead, the basic forms of words, or roots, are stored and additional parts of speech are handled by the removal of prefixes and suffixes in order to obtain the root of the word. The motivation for this approach was taken from an examination of a spelling checker program, *Perfect Speller* [GOOD84], [DOW87b].

To allow for a wide range of words the possibility of up to two prefixes and two suffixes is taken into account. The processing of prefixes is quite simple. The initial letters of each word are scanned to find a match with one of the prefixes provided. This process is speeded up by only checking those prefixes which begin with the same letter as the first letter of the word, for example: **un** + **finished** -> **unfinished**

Table 12.1 shows the set of prefixes used to modify root words in the small dictionary.

a	ab	abs	ad	after	air
all	ana	ant	ante	anti	aqua
arch	astro	auto	bio	bi	by
co	centi	deci	dis	en	ex
electro	fore	hyper	intra	inter	in
iso	kilo	magneto	mega	meta	mono
micro	milli	multi	mis	non	out
over	pre	pro	poly	post	photo
pseudo	radio	re	sub	self	semi
super	stereo	tele	thermo	un	ultra
under	vice				

Table 12.1 Prefixes used to modify root words in the small dictionary

12.2.2 Suffix Processing

A special notation is used to describe the rules used to describe the process of adding a suffix to a word. This is adapted and expanded from that used in the manual for the *Perfect Speller* program. The rules are as follows:

- A plus (+) means that the following letters are added to the end of the word. For example, *+ing* means add *ing* to the end of the word, so: **build -> building**
- A minus (-) means the letters are removed. For example, *-e+ing* means remove the final *e* and add *ing*, so: **age -> aging**
- A capital *C* means that the final consonant must be doubled. For example, *+C+ing* means double the final consonant and add *ing*, so: **fit -> fitting**
- A lower case letter with no sign means that the indicated letter must already be present for this rule to be used. For example, *a+n* means that if *a* is present add *n*, so: **Australia -> Australian**

Suffix removal is similar to prefix removal, but additional complications arise. The addition of a suffix to a word very often requires changes to the root word. As an example, if the word ends in a consonant, the addition of *ing* often requires the doubling of that consonant, so: **hop + ing -> hopping**

This situation is complicated by the fact that there are exceptions to this rule. An example of this is where the word already ends in a double consonant, so:

add + ing -> adding

Other rules require the removal of certain letters before the suffix is added, so:

hope + ing -> hoping

Still other rules require the substitution of some letters for others before the addition of the suffix, so: **happy + er -> happier**

In this case the transformation could be approached in two ways. The first is to change the *y* to *i*, then to add the *er*. This would be expressed by a sequence of two rules, *-y+i* and *+er*. A problem with this approach is that the intermediate word, *happi*, will not be found in the dictionary. Although the rule *+er* is extremely useful, it is unlikely that the rule *-y+i* would be of general usefulness. A better approach is to carry out the transformation in one step: remove the *y* and add *ier*. This is expressed in the rule *-y+ier*.

The method used to find which suffix ends a word is a simple test of each suffix which ends with same letters as the word, accepting the first suffix which matches the final letters of the word.

Because many suffixes end with the same letter or letters, it was found necessary to test for the presence of suffixes in a certain order. If this is not done it is possible for some

rules never to be used because an earlier rule which matches the word is accepted, when a later rule would have carried out a more 'powerful' transformation. For example, if the rule *+an* precedes *+ian*, the second rule will never be used because every word which ends in *ian* also ends in *an*. This dictates that the suffixes be ordered with the longer ones placed ahead of the shorter.

Another factor to be taken into account is the different effect the same suffix will have on different words. As was seen in the above examples, the addition of *ing* to a word requires a different rule, depending on how the root word ends. In fact, three different rules are needed to handle the suffix *ing*. These are *+ing*, *+C+ing* and *-e+ing*.

Some suffixes require even more rules. For example, the simple addition of *er* to a word may occur in at least four ways:

- **+er** **E.g. kill +er -> killer**
- **+C+er** **E.g. dig +er -> digger**
- **-e+er** **E.g. file +er -> filer**
- **-ry+er** **E.g. angry +er -> anger**

These situations cannot be handled by a simple ordering of the application of the rules. Remembering that we are applying the rules in reverse (e.g. *anger -> angry*), then certainly, the case of the double consonant *+C+er* can be discriminated from the other three rules because we are actually searching for a double consonant followed by *er*. Thus, as long as *+C+er* appears before the other rules, the following rules will still be found once it is determined that the *er* is not preceded by a double consonant. However, the other rules present a more difficult problem, because each of them can only depend on the same information - the presence of *er*.

The solution adopted was to partition the rules into three sets, placing each of the conflicting rules into a different set than the others. These sets are tested for separately,

and following a match in any set the dictionary is checked for the presence of the resulting word. If the word is not found the next set of rules is tried.

Partitioning of the rules in this way also helps with some of the words which break the more general rules. For example, the word *adding* presents a problem - its root *add* ends with a double consonant. Applying *+C+ing* produces *ad*, a non-word. The system would conclude that *adding* is not a real word, unless it was included in the dictionary intact. If, when partitioning the suffix rules, *+C+ing* and *+ing* are placed in different sets, this problem is overcome. After failing with *+C+ing*, the rule *+ing* will be tested and the resulting word *add* found in the dictionary.

Suffix rules used to modify root words in the small dictionary are shown in Table 12.2.

Set 1					
+s'	+'	+ac	-y+ic	+C+ard	-y+iad
-y+ied	+C+ed	-e+ed	-y+ihood	+d	-y+iable
+C+able	-e+able	+C+ible	-e+ible	-e+ade	+age
+C+ence	+ence	+C+ance	+ance	-an+ane	+like
+ative	-e+ive	-e+ize	-e+ise	-ce+se	+re
+ee	-t+ce	+C+ing	-e+ing	+full	+ful
-y+ial	+al	-e+ism	-y+ication	-y+ation	-e+tion
-e+ion	+ian	+an	+ren	+en	+ship
-e+our	-y+ier	-le+ular	+C+ar	-e+ar	+C+er
-e+er	-e+or	-y+iness	+ness	-y+iless	+less
-e+y+ous	-y+ious	-x+ces	-y+ies	-fe+ves	+es
+s	+s	+ment	+ent	+ant	+C+art
-y+iest	+C+est	-e+est	-y+ist	-y+ieth	+th
-cy+t	+bility	-le+ility	-e+ity	-y+ily	-e+ify
+ly	+ary	-te+cy	+y		
Set 2					
+ic	+ed	-t+d	+able	+ible	+ade
-e+ee	+ing	-e+al	-y+ian	+ation	+ion
a+n	e+n	+our	+ar	+er	+or
-f+ves	-e+ant	-e+ent	-e+ist	+est	-d+t
+ity					
Set 3					
-e+ic	-ry+er	-ism+ist	-se+t	-er+ry	-e+y

Table 12.2 Suffixes used to modify root words in the small dictionary

12.3 A Larger Dictionary

The small dictionary performed adequately for simple specialised tasks, such as recognition of operating system or application program commands, but users frequently used words and phrases which it could not accommodate, producing considerable frustration. It was felt that little user freedom had been gained because the system imposed the need to adapt one's language to its requirements rather than allowing the user to use more natural language. Evidently, a larger vocabulary was desirable.

Analysis of the roots of words in the Macquarie Dictionary which begin with the letter 'a', and extrapolating the word count to include the whole alphabet (by noting the number of pages out of the total dictionary size devoted to the 'a' words), a decision was made to aim at a total of 10000 to 15000 basic words. This number is further enhanced by the application of prefixes and suffixes to form more complex words and other parts of speech.

A combination of listing all prefix and suffix forms listed in the Macquarie Dictionary and the Concise Oxford Dictionary, and the experimental development of other prefixes and suffixes, revealed about 450 prefixes and 650 suffixes. For the small dictionary, the approach to affix processing had been to code all the prefixes and suffixes into a set of large procedures based on Pascal case statements. These selected appropriate processing statements on the basis of a match between the first characters of the input word and a prefix, or the last character of the word and a suffix. With only about 50 each of prefixes and suffixes this method was simple and efficient. With the number of affixes increasing to approximately 1100, and with the likelihood of considerable experimentation being required to optimise the choice and ordering of these affixes, such an approach was undesirable.

The approach adopted was to read the prefix and suffix rules from files into memory, and to provide a pair of routines which interpret the rules and apply them to the input words. Care was needed to develop a processing algorithm which could operate in this more general manner without losing any of the speed of the hard coded scheme used in the small dictionary. The method used to rapidly access words from the dictionary also proved ideal to handle the prefixes and suffixes. The lists of prefix and suffix rules used appear in Appendices I and J respectively.

The result of using common methods for retrieving both words and affixes, and of reading in all lexicographical data from text files, is that the resulting dictionary system is completely user programmable. Words and word features can be added, altered, or removed from the dictionary, as can prefixes and suffix processing rules, by editing the files. Thus the system can be adapted to different English language subsets, or even other languages. In the case of some languages, it may be desirable to also alter the rules for processing affixes, which will be the subject of future work. The dictionary entries are described in a technical report, "A Machine Readable Dictionary for Natural Language Understanding Systems" [DOW94b].

12.3.1 Word Features

The dictionary stores, as *Word Type*, the following information about the parts of speech of the English language: *prefix, word element, noun, pronoun, verb, adjective, adverb, preposition, conjunction, interjection*. For each word type, provision is made for the storage of its features. Feature values are either true or false, present or absent. The features provided are shown in Figure 12.1.

Prefix:	No features
Word Element:	No features
Noun:	
Class:	<i>common, proper</i>
Group:	<i>abstract, concrete, collective</i>
Person:	<i>first, second, third</i>
Number:	<i>singular, plural</i>
Gender:	<i>masculine, feminine, neuter, common</i>
Case:	<i>nominative, possessive, objective</i>
Pronoun:	
Class:	<i>personal, relative, interrogative, demonstrative, indefinite</i>
Person:	<i>first, second, third</i>
Number:	<i>singular, plural</i>
Gender:	<i>masculine, feminine, neuter, common</i>
Case:	<i>nominative, possessive, objective</i>
Verb:	
Kind:	<i>transitive, intransitive, bitransitive, principal, auxiliary, regular, irregular</i>
Voice:	<i>active, passive</i>
Mood:	<i>indicative, subjunctive, imperative</i>
Tense:	<i>present, past, past-participle, perfect, past-perfect, future-perfect</i>
Person:	<i>first, second, third</i>
Number:	<i>singular, plural</i>
Form:	<i>be, can, do, have, may, must, shall, will</i>
Adjective:	
Class:	<i>descriptive, limiting, proper</i>
Kind:	<i>regular, irregular, numeral, article</i>
Degree:	<i>positive, comparative, superlative, non-comparable</i>
Type:	<i>cardinal, ordinal, definite, indefinite</i>
Adverb:	
Meaning:	<i>time, place, manner, degree, cause, purpose, number</i>
Type:	<i>affirmative, negative, interrogative, relative, conjunctive, comparison</i>
Kind:	<i>regular, irregular</i>
Degree:	<i>positive, comparative, superlative, non-comparative</i>
Preposition:	No features
Conjunction:	
Group:	<i>coordinating, subordinating, correlative</i>
Interjection:	No features

Figure 12.1 Word Feature Values

The prefix and suffix processing introduces an additional complication into establishing the value of each of the features of a word. The root word (from which a more complex word is derived) has well established feature values. These are obtainable from the dictionary, or determined by the provision of default values. However, when an affix rule

is applied the value of features may change. In the case of a rule which produces a word of an entirely different part of speech from the root word, a whole new set of features will become appropriate. This is particularly so for the addition of a suffix. Placing a prefix ahead of a word often has little more effect than changing the meaning of the word; its part of speech and feature values are rarely affected.

To accommodate the changes in word type and feature values each suffix rule must have associated with it one or more transformation rules to describe its effect on the root word. As in the case of the suffix rule itself, what is described is the effect of applying the transformation to the root word. During processing, what actually happens is the reverse - the root word is recovered from the compound word. This means that the inverse of the transformation needs to be applied.

To illustrate the varied effects suffix rules can have on the part of speech which a word represents, some of the suffixes and their transformations are presented here. These have been compiled from entries in the Macquarie and Concise Oxford dictionaries [MACQ81][FOWL70]. A complete listing of the suffixes, their meanings and associated transformations can be found in a technical report [DOW94b].

- **ed** - suffix forming past-tense; the past-participle; participial adjectives indicating a condition or quality resulting from the action of the verb; adjectives from nouns. For example:

+ed	add -> added	[v -> v]
	beard -> bearded	[n,v -> v,adj]
	recess -> recessed	[n,v -> v]
+C+ed	bog -> bogged	[v -> v]
-e+ed	hope -> hoped	[v -> v]
-y+ied	ready -> readied	[v -> v,adj]

This example illustrates that a suffix may transform a word from a variety of parts of speech into the same or different parts of speech. In such a case it may be difficult to decide the effect of the reverse transformation. For instance, transforming *beard* to *bearded* may result in either a verb or an adjective. When parsing, it is often enough to determine the possibility of the resulting word being a certain type, without having to know if it is actually of that type. On the other hand, if the meaning of the sentence is being determined, the part of speech represented by a word is important. The uncertainty might often be resolved by the fact that the sentence is only syntactically correct if the word is of a certain type.

- **able** - suffix used to form adjectives, esp. from verbs, to denote ability, liability, tendency, worthiness, or likelihood, but also attached to other parts of speech (esp. nouns), and even verb phrases. Many of these adjectives - e.g. *durable*, *tolerable*, have been borrowed directly from Latin or French, in which language they were already compounded. However, -able is attached freely (now usually with passive force) to stems of any origin. For example:

+able	teach -> teachable	[v -> adj]
	peace -> peaceable	[v -> adj]
	objection -> objectionable	[n -> adj]
	act -> actable	[v,n -> adj]
+C+able	club -> clubbable	[n,v -> adj]
	chop -> choppable	[v -> adj]
-e+able	sale -> salable	[n -> adj]
	debate -> debatable	[n,v -> adj]
	receive -> receivable	[v -> adj]
-ate+able	navigate -> navigable	[v -> adj]
+isable	actual -> actualisable	[adj -> adj]

- **iable** - form adjective, esp. from verb to denote ability, liability, tendency, worthiness, likelihood. For example:

-y+iable	pity -> pitiable	[n -> adj]
-----------------	----------------------------	----------------------

- **ible** - variant of *-able*, occurring in words taken from the Latin or modelled on the Latin type. For example:

+ible	add -> addible	[v -> adj]
	percept -> perceptible	[n -> adj]
-e+ible	reduce -> reducible	[v -> adj]
-t+sible	revert -> reversible	[v -> adj]
-ge+sible	submerge -> submersible	[v -> adj]

These examples, *-able*, *-iable* and *-ible*, illustrate that many suffixes reliably transform a variety of different word types into one particular part of speech, in this case an adjective.

- **ing** - suffix of nouns formed from verbs, expressing the action of the verb or its result, product, material, etc. It is also used to form nouns from words other than verbs. Verbal nouns ending in *-ing* are often used attributively - e.g. *the printing trade*, and in composition - e.g. *drinking song*. It is also a suffix forming the present participle of verbs, such participles often being used as adjectives (participial adjectives) - e.g. *warring faction*. For example:

+ing	sew -> sewing	[v -> adj,v]
	build -> building	[v -> adj,n,v]
	off -> offing	[prep -> n]
	shirt -> shirting	[n -> n]
	print -> printing	[n,v -> adj,v]
+C+ing	war -> warring	[n -> adj,v]
	wad -> wadding	[n,v -> n,v]
-e+ing	receive -> receiving	[v -> v,adj]
	actualise -> actualising	[v -> v,adj]
+ising	actual -> actualising	[adj -> v,adj]
+ualising	concept -> conceptualising	[n -> v,adj]
	act -> actualising	[n,v -> v,adj]
-e+ating	active -> activating	[adj -> v,adj]

The example of *-ing* reveals that a suffix may predominantly produce one particular part of speech, yet allow for exceptions which prevent a general rule from being applied. In

addition, by observing the several ways that the word *actualising* may be derived, it may be seen that there is considerable scope for efforts designed to optimise the set of suffix rules provided in a completed system. Intractable cases of word transformation may be resolved by a judicious choice between shorter, more general suffix rules and longer, more specific rules. If this fails to bring satisfactory performance then the final resort of placing the complete word into the dictionary is always available.

- **al** - adjective suffix meaning *of or pertaining to, connected with, of the nature of, like, befitting*, etc., occurring in many adjectives and nouns of adjectival origin; forms nouns of action from verbs; suffix indicating that a compound includes an alcohol or aldehyde group.

+al	economic -> economical	[n -> adj]
	poetic -> poetical	[adj -> adj]
	regiment -> regimental	[n -> adj]
	betroth -> betrothal	[v -> n]
-e+al	refuse -> refusal	[v -> n]
	suicide -> suicidal	[n -> adj]
-a+al	lingua -> lingual	[n -> adj]
-us+al	virus -> viral	[n -> adj]
-y+ial	deny -> denial	[v -> n]
	actuary -> actuarial	[n -> adj]
+C+ial	centen- -> centennial	[prefix -> adj]
-ary+C+ial	centenary -> centennial	[n -> adj]
-e+ential	preside -> presidential	[v -> adj]
+ional	concept -> conceptual	[n -> adj]
	except -> exceptional	[v -> adj]
	recess -> recessional	[n,v -> adj]
-ed+dural	proceed -> procedural	[v -> adj]
+ual	concept -> conceptual	[n -> adj]
+eval	long -> longeval	[adj -> adj]

- **ical** - compound suffix forming adjectives from nouns; providing synonyms to words ending in *-ic*; providing an adjective with additional meanings to those in the *-ic* form.

+ical	class -> classical	[n -> adj]
	dynam- -> dynamical	[prefix -> adj]
-e+ical	cone -> conical	[n -> adj]
y+ical	history -> historical	[n -> adj]
cy+tical	policy -> political	[n -> adj]
r+tical	grammar -> grammatical	[n -> adj]
atical	problem -> problematical	[n -> adj]
ic+istical	logic -> logistical	[n -> adj]
a+istical	lingua -> linguistical	[n -> adj]
e+istical	state -> statistical	[n -> adj]

The suffixes *-al* and *-ical* illustrate all of the above points. In particular, the example of *-al* shows that a suffixal ending may have a well defined effect on a word. In this case the suffix itself has a meaning. The suffix *-ical* provides examples of all four of the different suffix rule types - +sss, -sss+ttt, +C+sss and -sss+C+ttt.

- **ly** - normal adverbial suffix, added to almost any descriptive adjective; adverbial suffix applied to units of time, meaning *per*; adjective suffix meaning *like*.

+ly	forceful -> forcefully	[adj -> adv]
	active -> actively	[adj -> adv]
	saint -> saintly	[n -> adj]
	scholar -> scholarly	[n -> adj]
	hour -> hourly	[n,adj -> adv]
-e+ly	feeble -> feebly	[adj -> adv]
-le+ly	supple -> supply	[adj -> adv]
	multiple -> multiply	[adv,n -> adv,v]
+ively	*decept -> deceptively	[element -> adv]
	percept -> perceptively	[n -> adv]
-y+ily	happy -> happily	[adj -> adv]
	day -> daily	[n,adj -> adv]
-e+ingly	receive -> receivingly	[v -> adv]
+ably	action -> actionably	[n -> adv]
-e+ably	conceive -> conceivably	[v -> adv]
-able+ably	conceivable -> conceivably	[adj -> adv]
+ibly	percept -> perceptibly	[n -> adv]
-ible+ibly	terrible -> terribly	[adj -> adv]
-y+ially	actuary -> actuarially	[n -> adv]
-ial+ially	actuarial -> actuarially	[adj -> adv]
+ually	concept -> conceptually	[n -> adv]
-ual+ually	conceptual -> conceptually	[adj -> adv]

One feature to be noticed in this example is the use of a non-word **decept*. More will be said about this in the next section. In addition, this example demonstrates some of the variety of forms in which a simple suffix such as *-ly* may appear. One fact, which does not come out in the examples, but which does cause some difficulty, is that such a combination may also appear at the end of a word without it representing a suffix. An example of this is the word *belly*. A naive English understanding system could apply a suffix rule such as *-y+ly* to the legitimate word *bell*, transforming it into *belly*, also a common English word. This word could then be assumed to be an adjective meaning *like a bell*. However, I suspect that this would be a mistaken assumption. Fortunately, in this case, the more common transformation for English adjectives ending in *-ly* is via the rule *-y+ily*, and this would obviously fail to extract a root from the dictionary.

- **iatry** - combining form meaning *medical care*.

+iatry **psych-** -> **psychiatry** **[prefix -> n]**

- **olatriy** - word element meaning *worship of*.

+olatriy **demon** -> **demonolatriy** **[n -> n]**
-ol+olatriy **idol** -> **idolatriy** **[n -> n]**

- **atriy** - see *olatriy*.

+atriy **idol** -> **idolatriy** **[n -> n]**

- **metry** - word element denoting the process of measuring, abstract for *-meter*.

+metry **anthropo-** -> **anthropometry** **[prefix -> n]**

These final examples show even more clearly that a suffix may have a literal meaning, as well as producing a certain transformation from one part of speech to another. Suffixes such as those shown here are in fact derived from words, often Latin or Greek. It would perhaps be more correct to describe *metry* as a word element rather than a suffix, although its use here is clearly suffixal. Sometimes such word elements may appear at the beginning of, or embedded within, a word. Frequently their use as a suffix requires some slight change in their form.

12.3.2 Suffixes and Intermediate Forms

As mentioned earlier, a particular transformation can often be achieved in a number of ways. For example, *idolatry* can be obtained by appending the suffix *-atry*. In addition, it can be formed by removing the final letters *ol* and appending *-olatry*. The second may be preferred, although more complex, because *-olatry* has a defined meaning - *worship of*. Thus, it can be seen that the shortest transformation may not always be the most desirable.

Examine the entry for *+ively*. Here the non-word **decept* is transformed into *deceptively* (the *** is used in linguistics to indicate a word or sentence which is not grammatical). This assumes that *decept* appears in the dictionary.

The reason for the incorporation of such non-words into the dictionary may be seen by examining the following word table with the accompanying rules to produce each word from the word at the head of the table. The first part of the word table is shown in Table 12.3:

Each of the rules used could reasonably be expected to appear in the set of suffix rules. However, there are a large number of other words which are derivable from these same

root words, but which do not have such a simple derivation. Consider the next part of the word table in Table 12.4:

receive	deceive	conceive	perceive	Rule
received	deceived	conceived	perceived	-e+ed
receiving	deceiving	conceiving	perceiving	-e+ing
receivingly	deceivingly	conceivingly	perceivingly	-e+ingly
receiver	deceiver	conceiver	perceiver	-e+er
receivable	deceivable	conceivable	perceivable	-e+able
receivableness	deceivableness	conceivableness	perceivableness	-e+ableness
receivability	deceivability	conceivability		-e+ability
receivably	deceivably	conceivably		-e+ably
receipt				-ve+pt
	deceit			-ve+t
	deceitful			-ve+tful
	deceitfully			-ve+tfully
	deceitfulness			-ve+fulness

Table 12.3 Word Derivation Table - Part 1

receive	deceive	conceive	perceive	Rule
		concept	percept	-ive+pt
reception	deception	conception	perception	-ive+ption
receptionist	deceptionist	conceptionist	perceptionist	-ive+ptionist
		conceptual	perceptual	-ive+ptual
receptive	deceptive	conceptive	perceptive	-ive+ptive
receptively	deceptively	conceptively	perceptively	-ive+ptively
receptiveness	deceptiveness	conceptiveness	perceptiveness	-ive+ptiveness
receptor	deceptor	conceptor	perceptor	-ive+ptor
receptacle				-ive+ptacle
		conceptual	perceptual	-ive+ptual
		conceptually	perceptually	-ive+ptually
		conceptualise	perceptualise	-ive+ptualise
		conceptualised	perceptualised	-ive+ptualised
		conceptualising	perceptualising	-ive+ptualising
		conceptualism	perceptualism	-ive+ptualism
		conceptualist	perceptualist	-ive+ptualist
		conceptualistic	perceptualistic	-ive+ptualistic
			perceptible	-ive+ptible
			perceptibleness	-ive+ptibleness
			perceptibility	-ive+ptibility
			perceptibly	-ive+ptibly
recipient			percipient	-eive+ipient
recipience			percipience	-eive+ipience

Table 12.4 Word Derivation Table - Part 2

Clearly, the suffix rules are undesirably complex. More importantly, these rules are probably not used for many, if any, other words. This situation can be improved in two ways. The first is to rely on the application of two suffix rules in succession. Application of the rule -ive+pt produces the results shown in Table 12.5:

receive	deceive	conceive	perceive	Rule
*recept	*decept	concept	percept	-ive+pt

Table 12.5 Word Derivation Table - Part 3

The words *concept* and *percept* are real English words whereas **recept* and **decept* are not. This is of no importance if we are going to transform them by the application of the second suffix rule, hopefully to produce *receive*, *deceive*, *conceive* and *perceive*. However, the use of two suffixes on one word results in considerable processing overhead. A more satisfactory solution is to place the intermediate words and non-words **recept*, **decept*, *concept* and *percept* in the dictionary. Once this is done the second group of words originally derived from *receive*, *deceive*, *conceive* and *perceive* are able to be reached via simpler suffix rules. Also, these are likely to be rules which will be required for many other word derivations, and so will already be available, as can be seen from the word table in Table 12.6:

The judicious selection of dictionary entries can also be seen to advantage when the list of words which could conceivably be built upon the root word *act* is considered, along with the rules required to do so, using both complex rules without the use of intermediate words and simple rules with the use of intermediate words. The intermediate words which seem to be the most satisfactory choice become clear in such a

table, and consequently the table, shown in Table 12.7, has been partitioned to make them easily recognisable.

*recept	*decept	concept	percept	Rule
reception	deception	conception	perception	+ion
receptionist	deceptionist	conceptionist	perceptionist	+ionist
		conceptual	perceptual	+ual
		conceptually	perceptually	+ually
		conceptualise	perceptualise	+ualise
		conceptualised	perceptualised	+ualised
		conceptualising	perceptualising	+ualising
		conceptualism	perceptualism	+ualism
		conceptualist	perceptualist	+ualist
		conceptualistic	perceptualistic	+ualistic
			perceptible	+ible
			perceptibleness	+ibleness
			perceptibility	+ibility
			perceptibly	+ibly
recipient			percipient	-ept+ipient
recipience			percipience	-ept+ipience

Table 12.6 Word Derivation Table - Part 4

Word	Complex Rule (based on 'act')	Simple Rule (based on intermediate word)
act		
acts	+s	+s
acted	+ed	+ed
acting	+ing	+ing
actable	+able	+able
actables	+ables	+ables
actability	+ability	+ability
actabilities	+abilities	+abilities
active	+ive	
activity	+ivity	-e+ity
activities	+ivities	-e+ities
activate	+ivate	-e+ate
activates	+ivates	-e+ates
activated	+ivated	-e+ated
activating	+ivating	-e+ating
activation	+ivation	-e+ation
activations	+ivations	-e+ations
activator	+ivator	-e+ator
activators	+ivators	-e+ators
actively	+ively	-e+ly
activeness	+iveness	+ness
activist	+ivist	-e+ist
activists	+ivists	-e+ists
activism	+ivism	-e+ism
action	+ion	
actions	+ions	+s
actionable	+ionable	+able
actionables	+ionables	+ables
actionability	+ionability	+ability
actionabilities	+ionabilities	+abilities
actionless	+ionless	+less
actionist	+ionist	+ist
actor	+or	
actors	+ors	+s
actress	+ress	-or+ress
actresses	+resses	-or+resses
actual	+ual	
actualise	+ualise	+ise
actualises	+ualises	+ises
actualised	+ualised	+ised
actualising	+ualising	+ising
actualisation	+ualisation	+isation
actualisations	+ualisations	+isations
actualisable	+ualisable	+isable
actualisables	+ualisables	+isables

Table 12.7 Complex Versus Simple Transformation Rules

Word	Complex Rule (based on 'act')	Simple Rule (based on intermediate word)
actualize	+ualize	+ize
actualizes	+ualizes	+izes
actualized	+ualized	+ized
actualizing	+ualizing	+izing
actualization	+ualization	+ization
actualizations	+ualizations	+izations
actualizable	+ualizable	+izable
actualizables	+ualizables	+izables
actuality	+uality	+ity
actualities	+ualities	+ities
actually	+ually	+ly
actuary	+uary	
actuaries	+uaries	-y+ies
actuarial	+uarial	-y+ial
actuarially	+uarially	-y+ially
actuate	+uate	
actuated	+uated	-e+ed
actuating	+uating	-e+ing
actuates	+uates	-e+es
actuator	+uator	-e+or
actuators	+uators	-e+ors
actuation	+uation	-e+ion
actuations	+uations	-e+ions

Table 12.7 (continued) Complex Versus Simple Transformation Rules

Considering the number of letters to be added to a word by a rule to be some measure of the time taken to search for these letters when removing them, the counts for the above table are as follows:

Number of letters (complex rules, no intermediate words): 438

Number of letters (simple rules with intermediate words): 242

Number of complex (less generally usable) rules: 69

Number of simple (more generally usable) rules : 56

Number of words which can be generated : 70

Number of dictionary entries (complex rules): 1

Number of dictionary entries (simple rules) : 7

In the system developed, the actual method used to group suffixes during search brings the two methods somewhat closer together in search time, so a smaller improvement than might be expected is achieved. The use of multiple suffixes can achieve a similar result in terms of processing operations to that of the simple rules, but at the expense of more suffix processing time.

When more than one root word is taken into consideration the use of complex rules rapidly becomes unwieldy. Separating out the pluralising function into a separate process reduces the overheads significantly, as illustrated by the following rework of the word table for *act*, shown in Table 12.8:

It can be readily seen that the simple but more generally useful rules now enjoy a considerable advantage over the more complex rules:

Number of letters (complex rules, no intermediate words): 411

Number of letters (simple rules with intermediate words): 153

Number of complex (less general) rules: 52 } plus plural

Number of simple (more general) rules : 39 } rules

Number of words which can be generated : 70

Number of dictionary entries (complex rules): 1

Number of dictionary entries (simple rules) : 7

Many more intermediate forms could be placed in the dictionary to facilitate disambiguation of rules. For example, the derivation of *solicitude* from *solace* can be carried out in one step:

solace -> solicitude (-ace+itude),

or it can be achieved in two stages by the use of the non-English **solice*:

solace -> *solice (-ace+ice) -> solicitude (-e+itude).

Word	Complex Rule (based on 'act')	Simple Rule (based on intermediate word)
act		
acts	+s	+s
acted	+ed	+ed
acting	+ing	+ing
actable	+able	+able
actability	+ability	+ability
active	+ive	
activity	+ivity	-e+ity
activate	+ivate	-e+ate
activates	+ivates	-e+ates
activated	+ivated	-e+ated
activating	+ivating	-e+ating
activation	+ivation	-e+ation
activator	+ivator	-e+ator
actively	+ively	-e+ly
activeness	+iveness	+ness
activist	+ivist	-e+ist
activism	+ivism	-e+ism
action	+ion	
actionable	+ionable	+able
actionability	+ionability	+ability
actionless	+ionless	+less
actionist	+ionist	+ist
actor	+or	
actress	+ress	-or+ress
actual	+ual	
actualise	+ualise	+ise
actualises	+ualises	+ises
actualised	+ualised	+ised
actualising	+ualising	+ising
actualisation	+ualisation	+isation
actualisable	+ualisable	+isable
actualize	+ualize	+ize
actualizes	+ualizes	+izes
actualized	+ualized	+ized
actualizing	+ualizing	+izing
actualization	+ualization	+ization
actualizable	+ualizable	+izable
actuality	+uality	+ity
actually	+ually	+ly
actuary	+uary	
actuarial	+uarial	-y+ial
actuarially	+uarially	-y+ially
actuate	+uate	
actuated	+uated	-e+ed
actuating	+uating	-e+ing
actuates	+uates	-e+es
actuator	+uator	-e+or
actuation	+uation	-e+ion

Table 12.8 Singular Transformation Rules

This intermediate form can also be taken advantage of to produce the word *solicitous*. At first sight it might seem to make more sense to derive *solicitude* and *solicitous* from the word *solicit*, which would be in the dictionary, and from which would also be derived *solicitation*, *solicitor*, *solicited*, *soliciting*, etc. Tempting as this might be, examination of the semantic content of the words reveals that *solicit* and *solicitude* are not as clearly related to each other in meaning as are *solace* and *solicitude*.

The task of identifying suitable intermediate forms is not easy. Much time and experimentation will be required to produce an optimal balance between root forms and affix rules. Rather than incorporate such non-word intermediate forms into the dictionary as a new word type, they have been included with the existing word element type.

12.4 Problems with this Dictionary Design

While the transformation of one word into another by the addition of one or more affixes is a relatively simple process, the reverse of this - removing affixes - is not so easily achieved. Modifying a word, particularly by the addition of a suffix, often results in some loss of information about the root word. This causes the possibility of ambiguity when the root needs to be rediscovered.

For example, consider the word *sentence*. The application of the rule *+ence* will result in the root *sent*, obviously incorrect. Yet *sent* may be found in the dictionary, or, if not, then by the further application of *-d+t*, its root *send* will be found, also incorrect. The simplest solution to this problem is to ensure that *sentence* itself appears in the dictionary. As it is not formed from a root by the application of affixes, then it ought to have an entry of its own, but as not all words can be included in a small dictionary, it may

have been omitted. In this case an incorrect root would have been returned rather than the more useful information that the word was unknown.

In a similar fashion, a word such as *repented* may appear to have a prefix, when in fact it has not. The present system, on not finding the whole word in the dictionary, first removes possible prefixes. It will not find the resulting *pent* in the dictionary, so eventually it will remove the suffix *ed*. The word *pent* will be found and incorrectly reported as the root of *repented*.

The most satisfactory solution to this problem would be to arrange the affix processing to incorporate single suffix removal before that of mixed prefixes and suffixes. In fact, this has been done in the present design, but in whatever order the testing is carried out, an incorrect root may be arrived at before the correct root.

Another possibility which should be kept in mind is that of the interaction between multiple affixes. The present dictionary design allows for up to two prefixes and two suffixes to be removed from a word. It is conceivable, however, that the application of an incorrect first suffix rule to a word may still result in a form with a recognisable second suffix. In this case removal of the second suffix may result in an incorrect root result.

The suffix rules used by *Perfect Speller* [GOOD84] did not need to take into account the resulting word type. If possible it would be best to try to standardise on a progression of word types, for example, nouns transform to verbs, and so on. At present, the transformation may go either way, depending on which rule is invoked. Determining which word type should be in the dictionary is a compromise between selecting the most basic form of the word, and, if more than one basic form is available, selecting the one which contains the most useful information in its feature set.

The prefixes present little difficulty. They are simply appended to existing words and so can be as easily removed. There may be a case for more complex prefix rules - e.g. *at* is equivalent to *ad* before a *t*; e.g. *attend* -> (*at*) *tend*. This could be expressed as $+at?t$, meaning add *at* if next letter is *t*. It would be removed by checking if the first three letters are *att* and if so removing the *at*.

Words which begin with the same letters as a prefix but which do not use them as a prefix need to be placed in the dictionary. For example, *re* is not a prefix in the English word *remedy*, although in its original Latin derivation *remedium* it might be looked upon as a prefix to the stem *med*.

Words which end in a double consonant and can take *-ing* might need to be in dictionary (e.g. *adding*), especially if the result of applying the suffix rule $+C+ing$ still produces a valid word. A similar warning applies to many of the rules.

Much of the interaction of suffix rules can be avoided by the order in which the rules are tried. In the small dictionary it was necessary to separate the rules into three distinct sets to handle cases where the rule searched for the same string. The sets are tried sequentially until the resulting word is found in the dictionary. This means that a dictionary check needs to be carried out following each rule application, not necessary for prefixes.

The larger dictionary implements an exhaustive test of all applicable suffix rules, again stopping with the first rule which produces a word found in the dictionary. In both the small and the larger dictionaries this means that a word could be attributed to the wrong root because that root was discovered first, even though the correct solution is potentially available. One way to circumvent this problem would be to pass in from the application program, information about the part of speech required for the word being searched for. This information will be available in the case of a natural language parser

attempting to prove the truth or falsity of a production rule. If the word obtained does not match the needed type, the system could continue its search through the suffix rules for another word.

An aid to the process of filtering out incorrect words produced by the application of suffix rules would be to make a reject table of words generated by the rules which are wrong, for example *ad* from *adding* via the rule *+C+ing*. The overheads involved in this would mean it would be best to restrict its use to overcoming intractable cases involving important words, and then only if the problem cannot be resolved by reordering the rule applications or rearranging the partitioning of the rules.

12.5 Dictionary Enhancements

A desirable enhancement to this dictionary would be the provision of special purpose sub-dictionaries containing words needed in a particular context or subject field. This concept is easily implemented, and could be extended by partitioning the entire dictionary into sections determined by word popularity.

Another enhancement would be the provision of the ability to automatically learn new words and correct improperly derived words. With this facility enabled, the detection of an incorrect or unknown word by either the application program, the user, or the dictionary system could prompt the user for information about the spelling of the word, its parts of speech, its semantic content, and its possible derivation from simpler roots. This information could then be used to update the dictionary word and affix files.

A further enhancement would be the provision of a compiler for the word and affix files. This would reduce the disk space occupied by the files, shorten the loading time, and increase the security of the data, which could also be compressed if desired. Ideally the

dictionary system would provide routines for reading both text and compiled files, writing both types of files, and translation between one type and the other.

12.6 Semantic Information

The use of this dictionary in a natural language understanding system requires its extension to incorporate semantic information for each dictionary entry. The derivation of many of the words from more basic forms by the use of prefixes and suffixes complicates this process considerably. The effect of an affix on the meaning of a word needs to be defined and the information incorporated into the affix rules. While this complexity is regrettable, the alternative of constructing a complete dictionary becomes even more formidable when the addition of semantic information to each entry is contemplated.

The subject of designing a suitable meaning representation scheme and its incorporation into the dictionary will be considered in Chapter 14.

13. DICTIONARY IMPLEMENTATION

This chapter describes the implementation details of the dictionary. The primary requirements of very fast access and efficient use of memory required innovative adaptations of the more usual database techniques to be employed.

In order to reduce the amount of memory required for storage of dictionary information without restricting the number of words in the vocabulary, it was decided to use a dictionary of root words and enhance this by using prefix and suffix transformations to attempt to reduce more complex words to these root words.

13.1 The Structure of the Dictionary

13.1.1 String Storage

The dictionary for the present system is a stable database with permanent data being added as it is developed. Experiments show it is to have a capacity of 10-15,000 entries, giving a size of approximately 75,000 bytes, assuming an average word length of 5 characters. Each word is stored in a dynamic string. Pointers to this and the features belonging to the word are stored as a record linked into a list structure, with indexed access using the first two or three characters of the word. This allows each entry to be accessed in minimum time. The data remains largely unaltered during the life of the system, except for minor changes to rectify mistakes or improve performance. The only likelihood of needing to regularly add words to the dictionary would arise if it was desired to make the speech understanding system able to learn new words while it is being used - a future development.

Instead of using Turbo Pascal strings, with their overhead of wasted memory, it was decided to construct dynamic strings. When a string is to be stored in memory its length

is determined and one more memory byte than this number is allocated at run time. The length of the string is stored in the first byte, as in Turbo Pascal, and the string characters are placed in the bytes following this. This provides efficient use of memory while retaining the ability to quickly determine the string length, a feature useful in the application of prefix and suffix rules and in providing a fast test for the possible inequality of two strings. Null-terminated strings do not have this advantage.

As the structure in memory of the dynamic string matches that of a Turbo Pascal string, it is possible to access the string in one assignment statement by type casting it to a suitable static string type. Even if the string type is longer than the dynamic string the extra characters copied which do not belong to the string will be ignored by string manipulation routines which use the length byte. Copying a string into dynamic memory using this technique would be more hazardous, however, as either the string itself or other data would certainly be corrupted unless the string type used in the cast were exactly the right size.

Addition of a length byte increases character storage to 90,000 bytes (15,000 x 6). This does not take into account the need to access these dynamic strings in some way. This can be done in numerous ways. A linked list of all entries would occupy at least 150,000 bytes, allowing 4 bytes per pointer, a 100% overhead in storage space over that required for the characters alone. Sequential access would also be too slow. Instead a dynamic indexing scheme was devised, and will be described once the method of storing each entry has been detailed.

13.1.2 Dictionary Entry Storage

Space must also be allocated for storage of the feature values associated with each word entry. The simplest and most efficient method of describing the features of words is to

use items which are simply present or absent, true or false. Consequently, each word type has associated with it a collection of Boolean valued features, as described in Chapter 12. In Pascal, the most convenient method of storing such a collection is in a *set* type.

As the smallest amount of memory that can be allocated to an item is one byte, or eight bits, the memory occupied by the feature set for each word type is as shown in Table 13.1. The average feature set size, taking into account the frequency of the different parts of speech in the dictionary, is approximately 3 bytes.

Word Type	Bits in Set	Bytes Occupied
Adjective	15	2
Adverb	19	3
Conjunction	3	1
Interjection	1	1
Noun	17	3
Prefix	1	1
Preposition	1	1
Pronoun	17	3
Verb	31	4
Word Element	1	1

Table 13.1 Memory Occupied by Word Feature Sets

The simplest way of accommodating data items of varying sizes is to use a variant record. This is not very economical of space, since each item then occupies the same amount of storage as the largest item, in this case 4 bytes for a verb feature set. Instead, a dynamic data structure again gives the most economical solution. This is easy to implement because the size of the set for each word type is known in advance. As the word type also needs to be stored along with the feature values, the size of the set to be retrieved is simply determined. Figure 13.1 illustrates the dynamic data storage for each

entry. Addition of the feature values and the word type increases the memory required for each entry to 14 bytes, equating to 210,000 bytes in total for 15,000 entries.

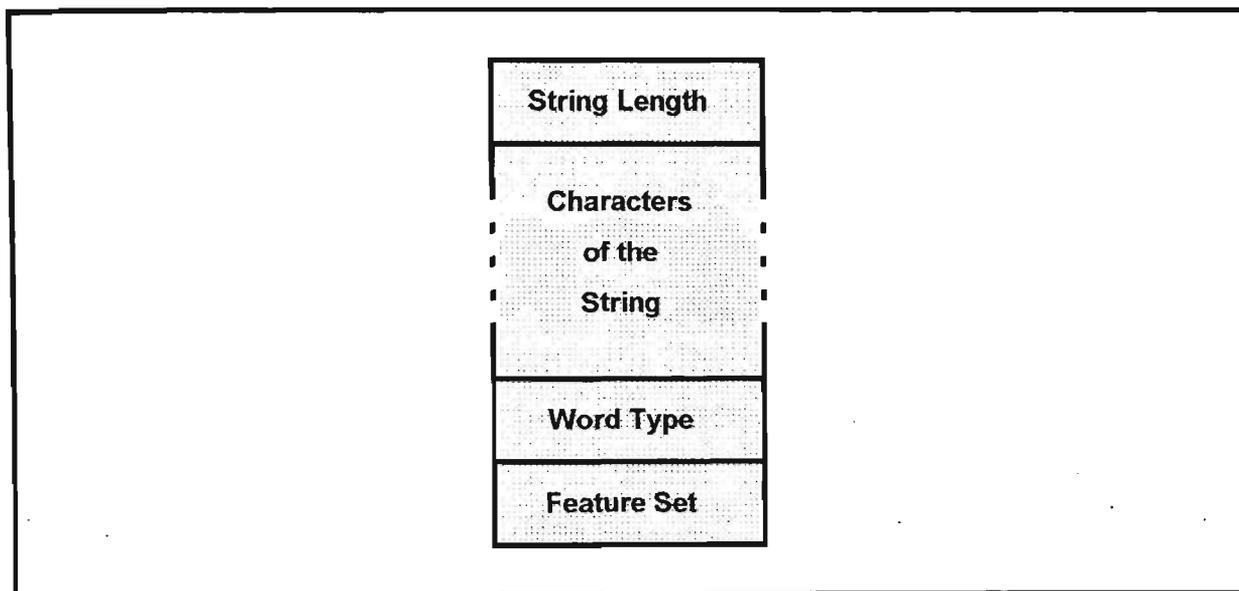


Figure 13.1 Dynamic Data Storage Structure for a Dictionary Entry

For each entry in the dictionary, a Pascal record is created containing two pointers. One pointer contains the address of the dynamic data entry described above, the other forms the link to the next entry record, enabling a linked list to be formed, as shown in Figure 13.2:

An improvement would be to incorporate the data fields into the linked list nodes, removing the need for an extra four byte pointer per entry. This will be carried out in a future revision of the software. The removal of this redundant pointer would decrease the total storage of the scheme as described so far by 60,000 bytes, from a total of 270,000 bytes to 210,000 bytes for 15,000 entries.

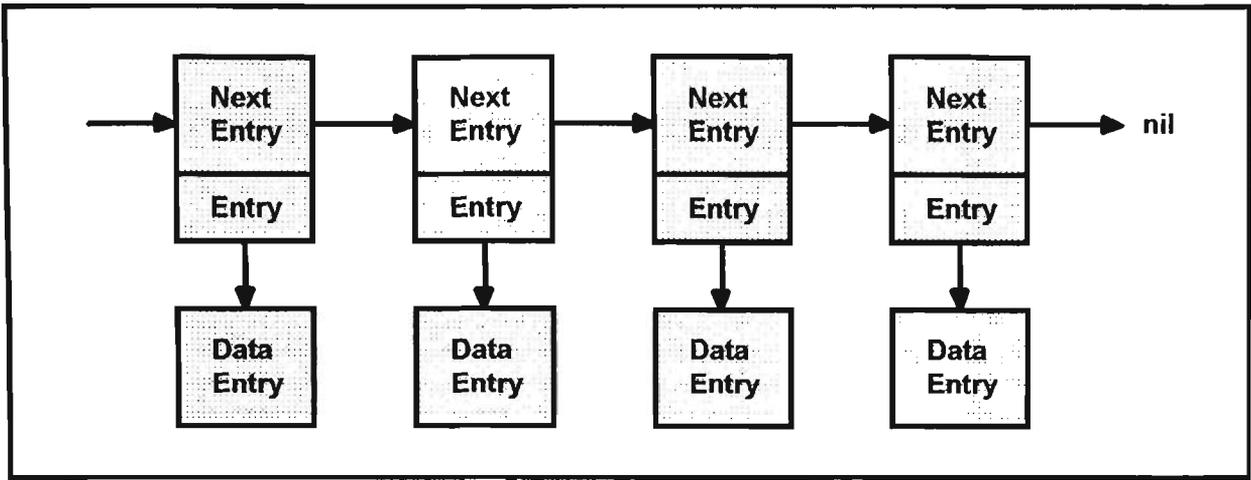


Figure 13.2 Linked List Dictionary Structure

13.1.3 Static Two Character Indexing

An alternative to linked lists is a linear byte array, storing the size of each entry along with the entries. The array could be traversed by using the size of the entries to locate the next entry. This would be economical in the use of storage, but the time taken to search for an entry, owing to the arithmetic required to locate each entry from the information about the size of the entry before it, would be too great. Also, the memory allocation routines on a PC can only allocate a maximum of less than 64k bytes to any one item, requiring the dictionary data to be partitioned into three or more components. A third problem is the difficulty of altering the data once it is loaded into memory.

As the total amount of memory required was not yet prohibitive, and as retrieval time is a very important factor in a dictionary system, it was decided to retain linked lists. The problem then became one of organising the list in such a way that access time was minimised. First an index array, indexed by the first letter of each word, was constructed. The array entries consisted of a pointer to the first entry in a linked list of words beginning with that letter, as shown in Figure 13.3.

While the dictionary was small this scheme worked well, but as the number of entries increased the linked lists grew too long, resulting in an unacceptable access time.

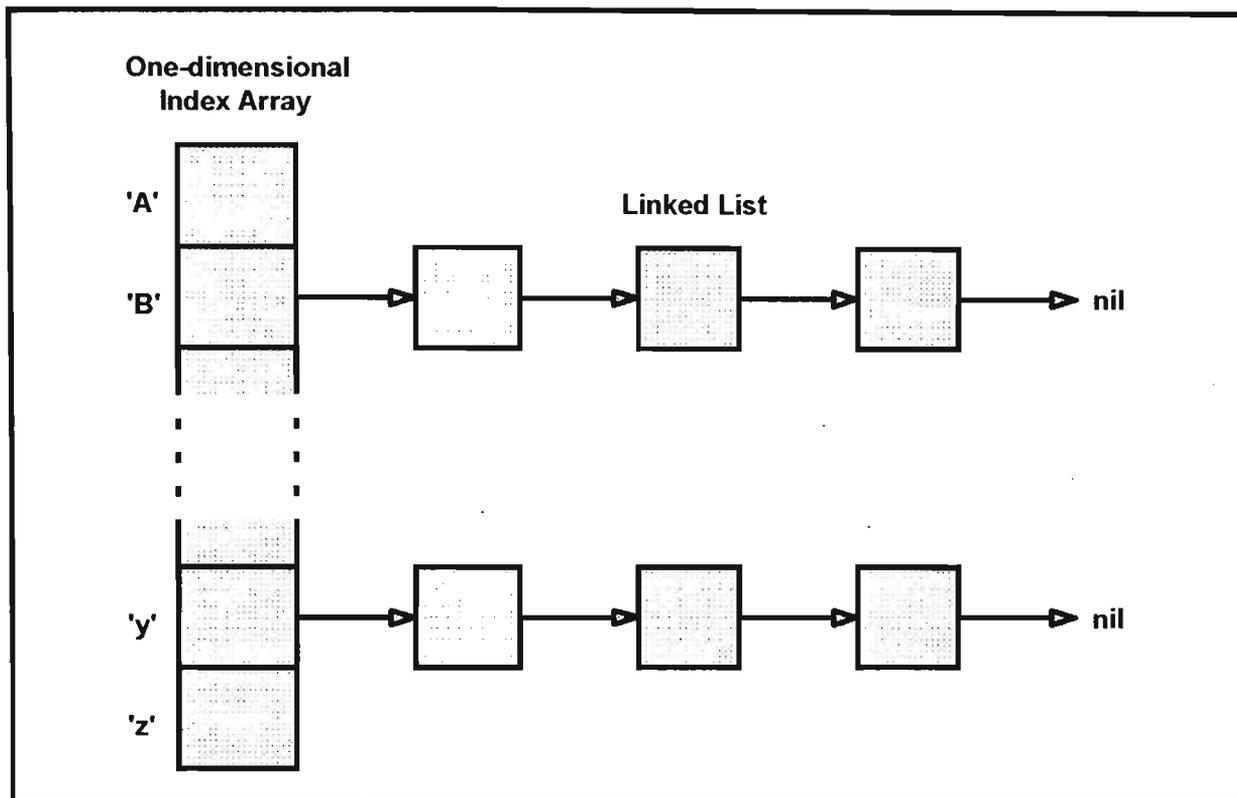


Figure 13.3 Static Single Character Indexing

The solution adopted was to change the single linear index array to a two dimensional array, indexed by the first two letters of the word, as illustrated by Figure 13.4. This, in fact, is an array of 53 x 30 pointers, a 1590 byte overhead. For a small dictionary this would be a large price to pay, but as the dictionary size increases the index array becomes insignificant, amounting to only 0.59% of the memory requirements for 15,000 entries. In fact, this overhead is exaggerated, since the pointers it contains have already been accounted for in the calculations for memory size if linked lists are to be used.

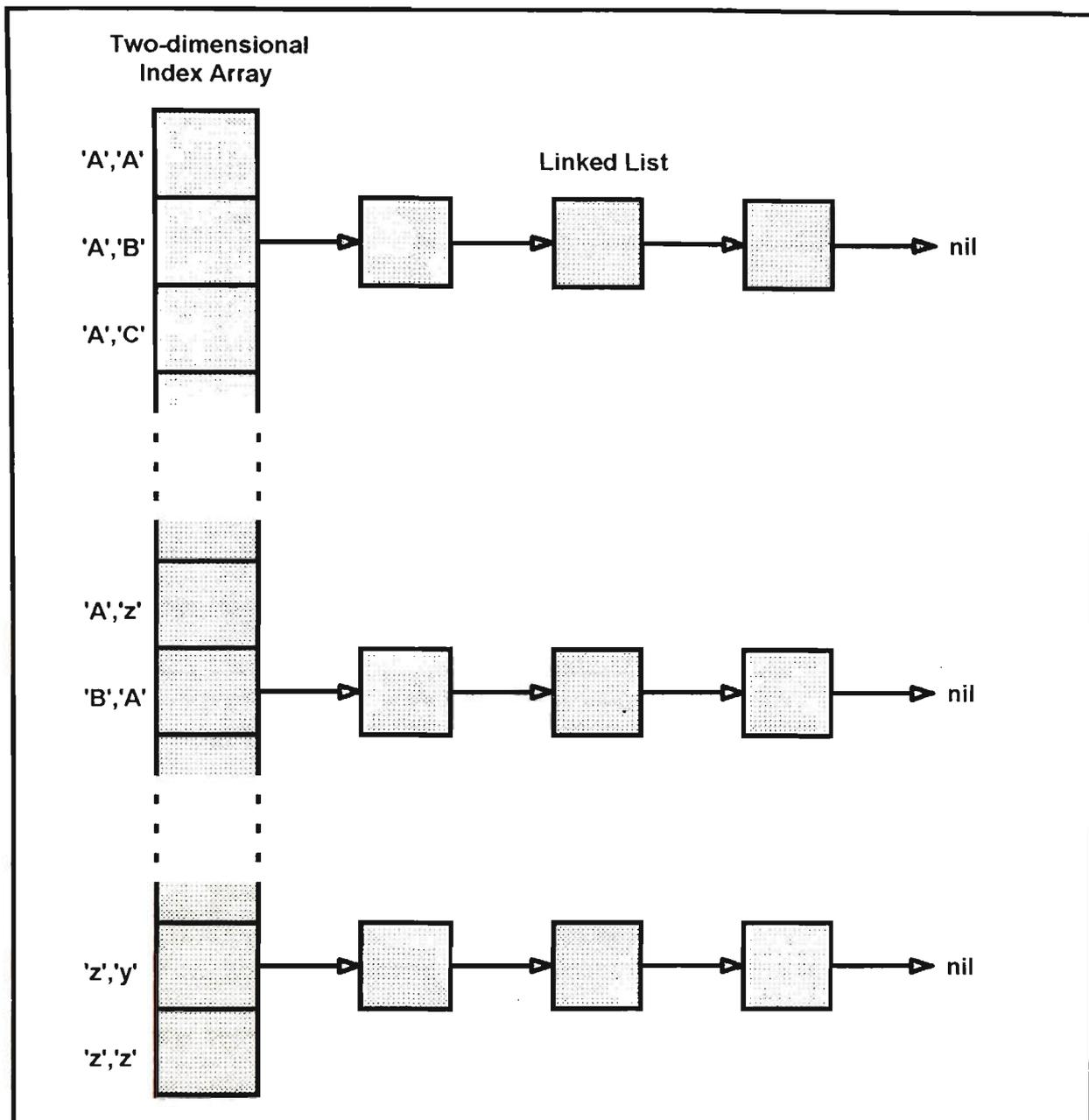


Figure 13.4 Static Two Character Indexing

The 53 x 30 array allows for an apostrophe and for proper nouns to begin with a capital letter. It was thought desirable to retain case sensitivity in order to retain the possibility of discriminating between ordinary nouns and proper nouns when they consist of the same letters but have different meanings. This is not possible with spoken input, but broadens the scope of applications for the dictionary. The index array is quite sparse, especially as many two letter combinations never begin a word.

The apostrophe is included as a first letter because the same type of array structure is used to control the storage of prefix and suffix rules. Suffixes are indexed by the last and second-last characters, and it is possible for the last character to be an apostrophe. The use of the same construction simplified the programming task. The second index has 30 values, to handle the extra characters - space, apostrophe, and plus and minus signs. The space character is necessary to facilitate the handling of words, prefixes and suffixes which consist of only one character.

Figure 13.5 shows an example of an array indexed by the first and second letters of 469 English prefixes. It gives the counts of all entries pointed to by each array element. The sparsity of the array can be seen. There are only 142 non-zero elements, out of a total of 1590. The figure of 142 two-letter combinations is typical of the start of English language words, so a 15,000 word dictionary has an average linked list length of 106 entries. This was found experimentally, by measuring access speed, to be longer than desired. (In fact, some lists would be considerably longer than this.) If all two-letter combinations were used then the average list length is just over 9. The actual figure will be somewhere between these two extremes. By inserting suitable counters into the routines which load the dictionary files actual list lengths were determined, along with a measure of the memory space taken up by the components of the dictionary.

When the dictionary had been expanded to the point of including all desired words beginning with the letter 'a' and 'b', a total of 3809 entries, the following results were reported. The word array contained 225 non-empty entries and the longest list, that of the words starting with 'bu...' contained 211 entries. These figures give an average list length of 16.93 entries. The average word length was 5.36 characters and the average feature set size was 2.83 bytes. So, with a total of 31,196 bytes of useful information, each byte of information requires around two bytes of storage in order to make it easily and rapidly accessible.

List Length	0-1	2-4	5-9	10-19	20-49	50-99	100-199	200+
Number of Lists	35	55	40	51	26	11	5	2
Number of Words	35	156	264	718	694	765	758	419
% of Words So Far	0.9	5.0	11.9	30.8	49.0	69.1	89.0	100.0

Figure 13.6 Linked List Length Distribution

These figures reveal that 40% of all lists are less than 5 entries long, 58% are less than 10 entries long and 81% are less than 20 entries long. Such a distribution should give satisfactory retrieval speed, unless the longer lists happen to contain a preponderance of very popular words. On examining the figures for the number of words contained in each category of lists the latter is found to be likely, since the longer lists, of course, contain more words than the shorter ones. Only 5% of all words lie in lists with less than 5 entries, 12% in lists with less than 10 entries, 31% in lists with less than 20 entries, and 49% in lists with less than 50 entries.

The use of the two letter indexing makes it unnecessary to store the first two characters of a word. This results in a saving of almost 30,000 bytes for a 15,000 entry dictionary. The complexity of adding and removing two characters from each word did not slow the system down. Not having to read as many characters from dynamic storage and reassemble them into a string actually more than compensates for this extra computation time.

The eight byte granularity of the Turbo Pascal memory allocation scheme nullifies some of the gains made in this area. Any item whose size is not an exact multiple of 8 bytes wastes some memory. To indicate the magnitude of the effect of granularity, memory usage was measured. For a dictionary size of 3809 entries the memory requested was

61,692 bytes, whereas the memory actually allocated was 73,936 bytes. The difference of almost 12k bytes of memory represents a wastage of 16.6% of the allocated memory. The figures also indicate that the average size of an entry is 16 bytes. If space becomes a premium a solution is to construct one's own memory allocation routines.

13.1.4 Dynamic Three Character Indexing

During use of the dictionary with static two character indexing, when a word such as *buzz*, which was stored in one of the longer lists, was being retrieved, a considerable degradation in access speed was noticeable. Some of the lists were of the order of 200 entries long. Clearly, some means of reducing the length of these lists was needed.

The indexing was extended to use three characters. A three dimensional static array requires $53 \times 30 \times 30 = 47,700$ four byte pointers, occupying 190,800 bytes. This is a prohibitive amount of memory to use simply for indexing, and the structure is too large for a Turbo Pascal static variable. Most of the entries would never be used. It is capable of individually addressing approximately 3 to 5 times as many words as the dictionary is likely to contain. Even so, as there may be many words which begin with the same three letters, linked lists will still be needed for many entries.

It was decided to extend the existing two dimensional indexing array, with a maximum allowed length for a list addressed by a two letter combination. Once a list exceeded this length, it is broken up into smaller lists, each distinguished by the third letter in the word. These lists were accessed by pointers contained in a one dimensional array indexed by this third letter. Importantly, it was a dynamic variable, created only when it was required to break a long list up into smaller lists. A pointer to this dynamic array was placed into the original index array instead of the pointer to the original long list. In addition, a flag was added to each two dimensional index array entry to indicate whether its pointer

pointed to a two character indexed list or a set of three character indexed lists. The structure is illustrated in Figure 13.7.

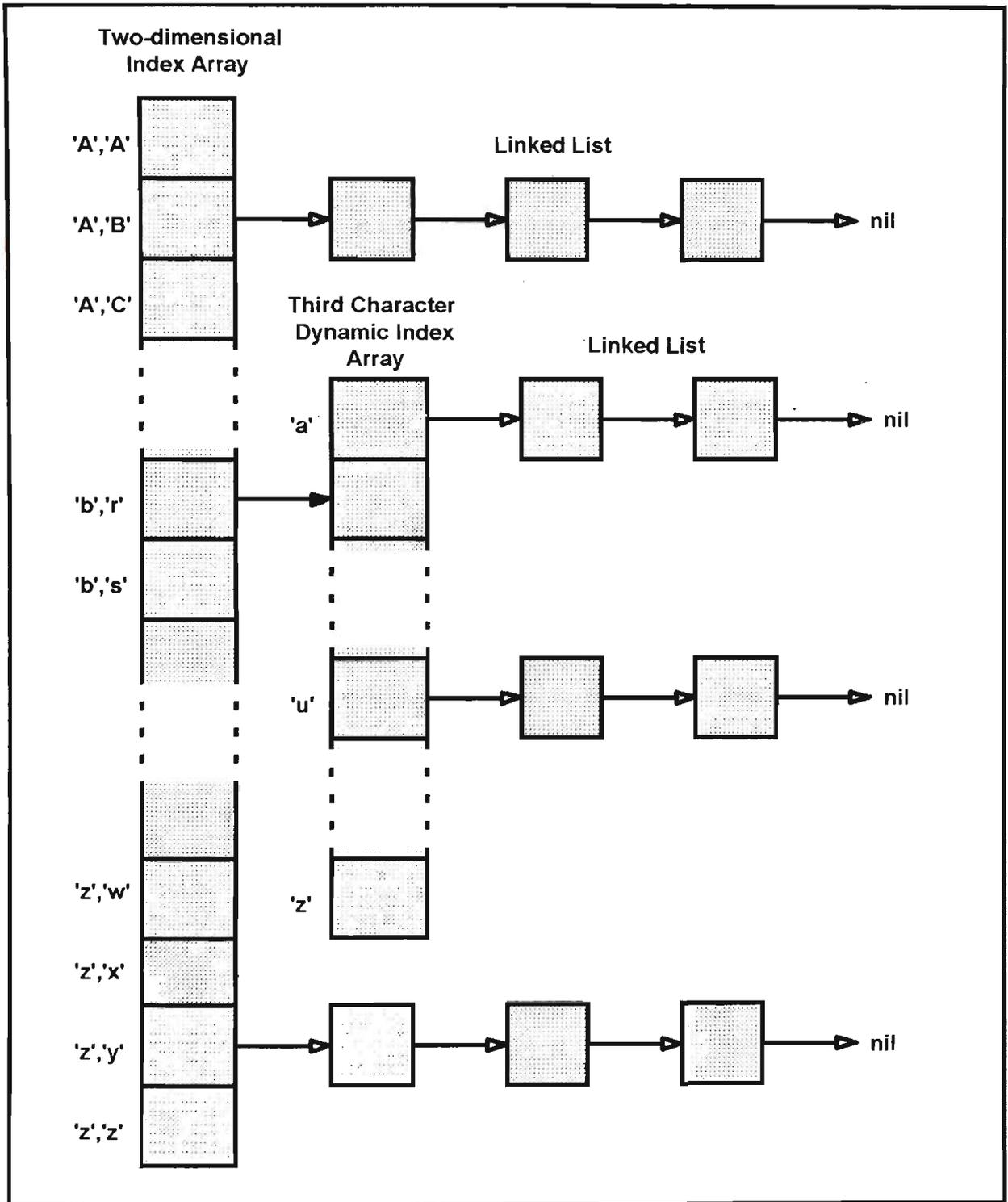


Figure 13.7 Dynamic Three Character Indexing

This scheme provides a considerable improvement in access times for the poorly behaved lists, i.e. lists with words beginning with letters common to many other words. To determine the optimum length to set as a maximum for a two character indexed list, and to examine the memory requirements of this scheme, the analysis code was modified to traverse the complete data structure. The results obtained are shown in Table 13.2.

Total number of entries = 3809 Number of index array entries = 225 Average word length = 5.36 Average feature set size = 2.83 bytes												
Max list length	5	7	10	15	20	35	50	75	100	150	200	250
Average list length	4.36	4.52	4.77	5.50	6.47	8.14	8.66	10.95	11.94	13.56	15.61	16.93
Num 2-char indexed	102	117	134	164	184	204	207	216	218	221	223	225
Num 3-char indexed	771	725	664	529	405	264	233	132	101	60	21	0
Total num indexed	873	842	798	693	589	468	440	348	319	281	244	225
Longest list length	53	53	53	53	53	53	53	75	95	149	185	211
Longest list chars.	'bra'	'bra'	'bra'	'bra'	'bra'	'bra'	'bra'	'bi'	'al'	'be'	'ba'	'bu'
Memory requested	76452	74652	72612	69012	66612	64212	63852	62772	62532	62172	61932	61692
Memory allocated	88696	86896	84856	81256	78856	76456	76096	75016	74776	74416	74176	73936

Table 13.2 List Length Data for Static 2-Character Indexing

The data in Table 13.3 illustrates the effect of setting different maximum list lengths on the distribution of lists and words through the data structure. The entries represent the number of lists in each length group, with the percentage of words in lists shorter or equal to this length in parentheses.

Max List Length	Actual List length							
	1	2-4	5-9	10-19	20-49	50-99	100-200	201+
5	243(6.40)	400(35.2)	153(60.7)	49(77.9)	26(97.3)	2(100)	0(100)	0(100)
7	223(5.90)	376(33.0)	166(60.7)	49(77.9)	26(97.3)	2(100)	0(100)	0(100)
10	205(5.40)	340(30.2)	172(59.6)	53(77.9)	26(97.3)	2(100)	0(100)	0(100)
15	157(4.10)	278(24.2)	148(49.8)	82(77.9)	26(97.3)	2(100)	0(100)	0(100)
20	123(3.20)	208(18.4)	128(40.5)	99(76.3)	29(97.3)	2(100)	0(100)	0(100)
35	83(2.20)	142(12.4)	99(29.5)	95(64.0)	47(97.3)	2(100)	0(100)	0(100)
50	77(2.00)	125(11.1)	94(27.3)	92(60.8)	50(97.3)	2(100)	0(100)	0(100)
75	54(1.40)	94(8.40)	68(20.2)	75(48.0)	46(82.0)	11(100)	0(100)	0(100)
100	48(1.30)	83(7.30)	59(17.6)	71(43.9)	45(77.2)	13(100)	0(100)	0(100)
150	43(1.10)	71(6.30)	52(15.4)	60(37.8)	39(67.0)	13(89.8)	3(100)	0(100)
200	36(0.90)	59(5.30)	44(13.0)	54(32.9)	33(57.3)	13(80.1)	5(100)	0(100)
250	35(0.90)	55(5.00)	40(11.9)	51(30.8)	26(49.0)	11(69.1)	5(89.0)	2(100)

Table 13.3 Effect of Maximum List Length on Actual List Length Distribution

13.2 Affix Processing

The affix transformation table is loaded into memory. A word which is not found in the dictionary is checked to see if applying transformations produces other possible words. If any new words are produced by this process, the dictionary is again checked for their presence. If a word is successfully transformed to one which is in the dictionary then the meaning and part of speech of the root word is altered according to which transformations have been applied to it, and possibly producing syntactic and semantic information which will be correct for the derived word. The affixes and their transforms are described in detail in a technical report, "Affix Transforms for a Machine Readable Dictionary for Natural Language Understanding Systems" [DOW94c].

13.2.1 Affix Transformations:

If an affix transformation is applied to a word to find its root form, the root is likely to form a different part of speech than that of the complex word. In addition, the meaning of the word will usually have changed, sometimes only in emphasis, but often to a related

but quite different meaning. Consequently, adjustment of both the word type and the meaning returned by the dictionary is needed.

By laboriously examining the effect of all of the transformations which were candidates for the dictionary, a set of transformations which seemed to be stable and reliable was identified. Many transformations were so diverse in their effects on different words that it was not possible to discover a single effect to associate with them. These transformations were omitted from the dictionary and the words they affected placed in the dictionary intact. However, if a transformation had the same effect on a large number of words, then even if there were also a large number of exceptions to this rule, the transformation was included. The exceptions were accounted for by including those words in the dictionary. Also, some words which, although derivable, are common roots of other words and are likely to be accessed frequently, were also placed into the dictionary to reduce processing overhead.

It may be more efficient to omit some transformations which don't produce many commonly used words, but a case can also be made for deriving these words and not having to carry them in the dictionary.

Prefixes, unlike many suffixes, don't usually produce complex transformations. Rather, they usually result in the juxtaposition of two meanings, or the negation or emphasis of the root word meaning. For this reason prefixes have been handled by looking up the prefix itself in the dictionary and treating it as a separate word which modifies the root word. Any case which is greatly different from this is handled by including the complete word in the dictionary. A complete list of the prefixes used is given in Appendix G.

Each suffix transformation identified was given a name which described both the effect it had on the part of speech of the word, and the semantic effect of the suffix. Thus a suffix which takes an adjective, noun or verb and produces a noun, and at the same time

conveys a sense of producing an action or describing a condition related to the root word, would be named

AdjNounVerb_Noun_ActionOrCondition.

This transform is illustrated by the examples of the *+ion* suffix, shown in Table 13.4.

Suffix	Example	Transformation	Suffix	Example	Transformation								
+ion	assert → assertion	[v → n]	-d+sion	extend → extension	[v → n]								
	flex → flexion	[v → n]		-ce+sion	pretence → pretension	[n → n]							
	reflect → reflection	[v → n]			-el+ulsion	compel → compulsion	[v → n]						
	concept → conception	[n → n]				-e+sion	precise → precision	[adj → n]					
	recess → recession	[n,v → n]					+tion	incise → incision	[v → n]				
	intersect → intersection	[n,v → n]						+tation	prevent → prevention	[v → n]			
-e+ion	private → privation	[adj → n]	-ive+ption						actual → actuation	[adj → n]			
	converse → conversion	[n → n]		+ation					conceive → conception	[v → n]			
	commune → communion	[v → n]			-e+ation				consider → consideration	[v → n]			
	opine → opinion	[v → n]				-ect+icion			deprive → deprivation	[v → n]			
	agitate → agitation	[v → n]					-de+sion		decline → declination	[v → n]			
	contrite → contrition	[adj → n]						-se+sion	starve → starvation	[v → n]			
	pollute → pollution	[v → n]	-t+sion						immunise → immunisation	[v → n]			
	infuse → infusion	[v → n]		-ge+sion					preserve → preservation	[v → n]			
-ite+ion	unite → union	[v → n]			-it+ission				active → activation	[adj,n → n]			
	-ect+icion	suspect → suspicion				[v → n]			-y+ication	realise → realisation	[v → n]		
-de+sion		allude → allusion				[v → n]	-a+ation			saliva → salivation	[n → n]		
	provide → provision	[v → n]				+uation		act → actuation		[n,v → n]			
	explode → explosion	[v → n]	-ine+ension					pacify → pacification		[v → n]			
-se+sion	converse → conversion	[n → n]		-it+ission				remit → remission		[v → n]			
	pervise → perversion	[adj,n → n]			-ine+ension			decline → declension		[v → n]			
-t+sion	convert → conversion	[v → n]						-ge+sion	submerge → submersion	[v → n]			
	-ge+sion	submerge → submersion					[v → n]		-it+ission	remit → remission	[v → n]		
-it+ission		remit → remission				[v → n]	-ine+ension			decline → declension	[v → n]		
	-ine+ension	decline → declension	[v → n]			-a+ation				saliva → salivation	[n → n]		
-a+ation		saliva → salivation	[n → n]	+uation						act → actuation	[n,v → n]		
	+uation	act → actuation	[n,v → n]		-y+ication					pacify → pacification	[v → n]		
		-y+ication	pacify → pacification					[v → n]		-ine+ension	decline → declension	[v → n]	
			-ine+ension					decline → declension	[v → n]		-ge+sion	submerge → submersion	[v → n]
							-ge+sion	submerge → submersion	[v → n]			-it+ission	remit → remission
						-it+ission		remit → remission	[v → n]				-ine+ension
-ine+ension				decline → declension				[v → n]	-a+ation				

Table 13.4 The Suffix *+ion* and Some of its Variants

Some transformations have alternate effects, depending on the part of speech of the original word. For example,

Noun_Adj_Pertaining__Verb_Noun_Agent

transforms a noun into an adjective, with the meaning of 'pertaining to' whatever the root word means, while if it is applied to a verb it produces a noun which describes the agent which produces the action of the original verb. Consider the example of the *+ar* suffix, in Table 13.5.

<u>Suffix</u>	<u>Example</u>	<u>Transformation</u>	<u>Suffix</u>	<u>Example</u>	<u>Transformation</u>
+ar	line → linear	[n → adj]	-le+ular	corpuscle → porpuscular	[n → adj]
+C+ar	beg → beggar	[v → n]		muscle → muscular	[n → adj]
-e+ar	tube → tubular	[n → adj]		single → singular	[adj → adj]
	lie → liar	[v → n]	-ule+ular	molecule → molecular	[n → adj]
-al+iar	pedal → pedlar	[v → n]		granule → granular	[n → adj]

Table 13.5 The Suffix *+ar* and Some of its Variants

Again, some transformations are even more complex. The transformation

AdjNounPfxVerb_AdjNoun_FunctionLocationOrRelation

transforms an adjective, noun, prefix or verb into either an adjective or a noun, the resulting part of speech depending on the original word. The present system is unable to decide which it will be, so it is left for the parser to try and make this decision. The meaning of the resulting word is that of a function, location or relation, again depending on the original word, and also on the particular suffix removed. This transform is illustrated by the example of the suffix *+ary*, in Table 13.6.

<u>Suffix</u>	<u>Example</u>	<u>Transformation</u>	<u>Suffix</u>	<u>Example</u>	<u>Transformation</u>
+ary	diction → dictionary	[n → adj,n]	-ample+emplary	example → exemplary	[n → adj]
	function → functionary	[n,v → n]			
	secret → secretary	[adj,n → n]	-our+orary	honour → honorary	[n → adj]
	centen- → centenary	[pfx → n]			
	contr- → contrary	[pfx → adj]	-ain+anary	grain → granary	[n → n]
-e+ary	adverse → adversary	[adj → n]	-an+ary	apian → apiary	[adj → n]
	prime → primary	[adj → adj]			
-eer+ary	volunteer → voluntary	[n,v → adj]	-nial+ary	centennial → centenary	[adj → n]
-ant+ary	militant → military	[adj → adj,n]	-ate+ary	arbitrate → arbitrary	[v → adj]
				primate → primary	[n → adj]
				actuate → actuary	[v → n]

Table 13.6 The Suffix +ary and Some of its Variants

Some of the transforms have a very clear semantic interpretation. For example,

AdjAdvNounVerb_Noun_Proficiency

reduces the original word to a noun which describes the action at which the original word indicated a degree of proficiency. As an example, application of the suffix rule +*smanship* to *gamesmanship* results in the noun *game*, and the transform **AdjAdvNounVerb_Noun_Proficiency** shows that the original noun indicates a proficiency in games. Similarly,

NounPfx_Noun_MeasuringInstrument

clearly indicates that the original noun or prefix described an instrument for the measuring of the particular quantity indicated by the root noun. The word transformations for the suffix +*meter*, in Table 13.7(a), illustrate this. However, the transformations in Table 13.7(b) show that there are exceptions to this rule, and these words would need to be individually included in the dictionary if this transform is adopted for use in the system.

	<u>Suffix</u>	<u>Example</u>	<u>Transformation</u>
(a)	+meter	alti- → altimeter	[pfx → n]
		baro- → barometer	[pfx → n]
		water → watermeter	[n → n]
		gas → gasmeter	[n → n]
	+ometer	gas → gasometer	[n → n]
-p+C+meter	amp → ammeter	[n → n]	
<hr/>			
(b)	+meter	penta_ → pentameter	[pfx → n]
		tri- → trimeter	[pfx → n]

Table 13.7 The Suffix *+meter* and Some of its Variants

A total of 188 such suffix transforms were finally selected for inclusion in the dictionary system, and these are listed in Appendix H. A list of each transform and its effect is in Appendix I, while the transform associated with each suffix is given in Appendix J.

The transform appropriate to each suffix is stored along with the suffix and applied to a word at the same time the suffix is applied. In addition, the transform is passed along with the resulting root word in order for the meaning change to be taken into account when the word is used. By producing the transforms as a table of constant values, only one extra byte of storage is needed for each suffix to incorporate the syntactic and semantic information encoded by the transform. It can be seen that, even though the dictionary entries in the dictionary so far provide only rudimentary semantic information, the application of these suffix transforms provides considerable information about the meaning of derived words.

13.2.2 Affix Transformation Storage

Because the scheme for efficient storage and fast access of dictionary entries proved so successful, it was used again for the information required for prefix and suffix

transformation rules. To determine which affix rules result in a root word which is in the dictionary, many transformations may need to be tested. Hence, high access speed is essential. The number of items of information for affixes is far smaller than the number of words in the dictionary, so the simpler two dimensional indexing scheme has proved to be fast enough.

13.2.3 Affix Removal

The removal of affixes from a word presents a number of problems. When the original parser, which only tried to find one correct parse, was in use, the process was relatively simple. If a word was not found in the dictionary then successive attempts were made to remove prefixes and suffixes in a predetermined order, until a word which was in the dictionary was discovered. This word was passed on to the parser. If it represented the part of speech which the parser was expecting, parsing would proceed, otherwise the parse failed.

This process overlooked the fact that different numbers of prefixes and suffixes can be removed from many words in a number of different orders, to produce a variety of words which will have different syntactic and semantic values. Any one of these may be the one which the parser requires in order to continue parsing. Altering the parser to allow it to try each of these words was not a simple task. In addition, if more than one word was acceptable, there was no way to determine which would lead to a correct parse when combined with later words in the sentence.

Changing the parser to one which attempted all possible parses in parallel provided the opportunity to also try every possible word and affix combination. Of course, this increased the number of possible parses, with a corresponding possibility of much longer time taken to find the correct parse or parses. However, in practice, many of the possible

words produce partial parses which quickly fail and can be abandoned. Given the speed of dictionary access, and provided the parser is written efficiently, and run on a fast processor, the extra work required is worth the guarantee that if a correct parse is possible it will be found.

To facilitate the finding of all possible derivations of a word the routine which removes affixes returns a data structure which has room for the original word, and for a large number of derived words along with the prefix and suffix rules used to derive that word. The structure, *AlteredWordsRec*, takes the form shown in Figure 13.8.

```
Type
  AffixOrders = (N, P, S, PP, PS, SP, SS, PPS, PSP, PSS,
                SPP, SPS, SSP, PPSS, PSSP, SPPS, SSPP);
  AlteredWordEntry = Record
    NumPrefixes: Byte;
    NumSuffixes: Byte;
    Prefix1: String10;
    Prefix2: String10;
    Suffix1: String10;
    Suffix2: String10;
    SufTfm1: SuffixTransforms;
    SufTfm2: SuffixTransforms;
    AlteredWord: String30;
    InDictionary: Boolean;
    AffixOrder: AffixOrders
  End;
  AlteredWordArray = Array[1..MaxAlteredWords] of AlteredWordEntry;
  AlteredWordsRec = Record
    UnalteredWord: String30;
    NumAlteredWords: Byte;
    Num1stPrefixes: Byte;
    Num2ndPrefixes: Byte;
    Num1stSuffixes: Byte;
    Num2ndSuffixes: Byte;
    LastEntryChecked: Byte;
    Entries: AlteredWordArray
  End;
```

Figure 13.8 Data Structure to Hold Altered Words Information

The routine *RemoveAffixes* returns one of these structures containing all of the root word derivations it was able to find for the complex word passed to it. For developmental purposes a parameter is provided which indicates how many prefixes and suffixes will be tried and in which order. A limit of two prefixes and two suffixes was placed on any word, but this still allows for nineteen different orderings, as follows:

- No affixes
- One prefix (P)
- One suffix (S)
- Two prefixes (PP)
- One prefix followed by one suffix (PS)
- One suffix followed by one prefix (SP)
- Two suffixes (SS)
- Two prefixes followed by one suffix (PPS)
- One prefix followed by one suffix followed by one prefix (PSP)
- One prefix followed by two suffixes (PSS)
- One suffix followed by two prefixes (SPP)
- One suffix followed by one prefix followed by one suffix (SPS)
- Two suffixes followed by one prefix (SSP)
- Two prefixes followed by two suffixes (PPSS)
- One prefix followed by one suffix followed by one prefix followed by one suffix (PSPS)
- One prefix followed by two suffixes followed by one prefix (PSSP)
- One suffix followed by two prefixes followed by one suffix (SPPS)
- One suffix followed by one prefix followed by one suffix followed by one prefix (SPSP)
- Two suffixes followed by two prefixes (SSPP)

It is likely that trying all of these orderings might slow processing down too much, so only one of these is passed to the affix processing routine. Each combination which includes the same number of prefixes and suffixes should produce the same derivations. However, the order in which they are tried will affect the number of false derivations which must be looked for in the dictionary, and so greatly influence the speed of finding the root words. Experience will enable a judgment to be made as to which orderings are the most desirable.

By careful design of the affix processing routine, the number of times the dictionary needs to be checked can be minimised. For example, if the ordering is to be PPS, the first step is to check if the word is in the dictionary. If it is, the dictionary entry is returned. If the word is not in the dictionary then an attempt is made to remove one prefix from the word. This may generate a number of candidates, since one prefix may also be part of another prefix. This can be seen in the case of the prefix *abs*, which begins with the prefix *ab*. Either the root word might be prefixed by *abs*, or it could begin with *s* and be prefixed by *ab*. Each candidate generated is placed in the data structure, along with the prefix rule which was applied to it.

Looking the words up in the dictionary may reveal some possible roots, in which case the *InDictionary* flag is set for these words. The candidates which were not found in the dictionary cannot be discarded yet, as they may form the basis of a more complex derivation. The second prefix is then removed from each of the successful and unsuccessful candidates, forming more possible words, which are added to the data structure. The dictionary check is repeated, and then the resulting words are similarly tested for the presence of a suffix.

Once the two prefixes and the suffix have been removed, any words which were not found in the dictionary may be discarded, and the remainder returned to the calling program.

Ideally, if the affix rules and the dictionary entries are optimal, only one word will result from this process, however, this is extremely unlikely. There may be no words, one word, or a large number of words returned.

Clearly, using this approach causes words to be produced which are incorrect for the particular sentence being parsed. The parser rejects any which are syntactically incorrect. In line with the philosophy of generating all possible parses of the sentence, it is felt that it is better to produce too many candidate words than to miss the one which will enable parsing to succeed. If more than one correct sentence results, then other means using semantic and/or pragmatic analysis, will have to be used to make the final choice. Alternatively, all of the syntactically correct sentences might be accepted, perhaps with different weight placed on their veracity. They may yield additional insight into resolving ambiguities and deciding on the meaning of the text.

With a suitably fast processor, such as an Intel i486, the extra processing time appears to be acceptable. The memory required for the word candidate data structures is more of a problem, sometimes causing stack overflow with the present arrangement of passing them as function parameters. This was overcome by placing the data in heap memory and passing pointers to it.

13.3 Semantic Information Storage

The semantic information incorporated into the present dictionary is only rudimentary - sufficient to provide a test bed for the development of ideas. Five avenues have been provided for the storage of semantic information in the dictionary, three directly in the dictionary entry, while the last two are indirect methods.

13.3.1 Auxiliary Verbs

Auxiliary verbs have inherent semantic information. Each such verb has a field labelled *form* which can take the values *be, can, do, have, may, must, shall* and *will*.

13.3.2 Affix Transforms

The second method has already been described above: the incorporation of word transforms for handling suffixes and prefixes. Many of these transforms specifically include information about the meaning of the resulting word. The meaning of the word can be deduced by combining the meaning of its root, obtained from the dictionary, with the effect of the transform.

For example, the transformation

AdjNounPfxVerb_AdjNoun_FunctionLocationOrRelation

as well as transforming an adjective, noun, prefix or verb into either an adjective or a noun, also shows that the meaning of the resulting word is that of a function, location or relation, depending on the original word and the particular suffix removed. This transform is illustrated by the *+ary* transformation shown in Table 13.8.

Some of the transforms have a very clear semantic interpretation. For example,

AdjAdvNounVerb_Noun_Proficiency

reduces the original word to a noun which describes the action at which the original word indicated a degree of proficiency. As an example, application of the transform

+*smanship* to the word *gamesmanship*, as shown in Table 13.9, results in the noun *game*, and the transform **AdjAdvNounVerb_Noun_Proficiency** shows that the original noun indicates a proficiency in games.

<u>Suffix</u>	<u>Example</u>	<u>Transformation</u>
+ary	diction → dictionary	[n → adj,n]
-e+ary	prime → primary	[adj → adj]
-eer+ary	volunteer → voluntary	[n,v → adj]
-ant+ary	militant → military	[adj → adj,n]
-ample+emplary	example → exemplary	[n → adj]
-our+orary	honour → honorary	[n → adj]
-ain+anary	grain → granary	[n → n]
-an+ary	apian → apiary	[adj → n]
-nial+ary	centennial → centenary	[adj → n]
-ate+ary	arbitrate → arbitrary	[v → adj]
	primate → primary	[n → adj]
	actuate → actuary	[v → n]

Table 13.8 Examples of the +*ary* Transformation

<u>Suffix</u>	<u>Example</u>	<u>Transformation</u>
+smanship	game → gamesmanship	[n → n]

Table 13.9 Example of the +*smanship* Transformation

Similarly,

NounPfx_Noun_MeasuringInstrument

clearly indicates that the original noun or prefix described an instrument for the

measuring of the particular quantity indicated by the root noun or word prefix. As an example, consider the *+meter* suffix, shown in Table 13.10.

<u>Suffix</u>	<u>Example</u>	<u>Transformation</u>
+meter	alti- → altimeter	[pfx → n]
	baro- → barometer	[pfx → n]
	water → watermeter	[n → n]
	gas → gasmeter	[n → n]
+ometer	gas → gasometer	[n → n]
-p+C+meter	amp → ammeter	[n → n]

Table 13.10 Examples of the *+meter* Transformation

13.3.3 Meaning Field

The third method provides a specific meaning field for the dictionary entry. The range of values which this field may take is the subject of a considerable research effort at present. The significance of this field is described in Section 14.1 when *canonical primitive acts* are discussed.

13.3.4 Inter-word Links

A link field with each entry is provided to cross-reference synonyms in the dictionary. If such a link is encountered it can be assumed that whatever semantic information is provided for the word in the link field can be used for the current word. This information may take any of the other forms, including another link.

13.3.5 Semantic Reference Field

The fifth way in which semantic information has been incorporated into the dictionary is via a semantic reference field which may be appended to each entry. This field consists of an index into a separate file of semantic strings. These strings take the form of a textual description of the meaning, or meanings, of the word. In this way detailed, and even vague or ambiguous, descriptions can be accommodated. Such information might be difficult to codify in any more precise form. It is left up to the application programmer to decide what use might be made of such information.

13.4 Dictionary Files

13.4.1 Dictionary File Format

Figure 13.9 shows a sample of dictionary entries. A few of the entries have been given tentative semantic descriptors.

The entries are contained in a text file, in the compact format shown in Figure 13.10, which has the advantage of being only 53% of the size of a crude text file while retaining the convenience of being able to be edited with any text editor.

<u>Word</u>	<u>Type</u>	<u>Features</u>
agree	verb	tense:pres transitivity:trans,intrans
agri	element	
agro	element	
ah	interjection	
ahead	adverb	
ahem	interjection	
ahoy	interjection	
aid	noun	number:sing
aid	verb	tense:pres transitivity:trans,intrans
ail	verb	tense:pres transitivity:trans,intrans
aim	noun	number:sing
aim	verb	tense:pres transitivity:trans,intrans
ain't	verb	kind:aux person:first number:sing form:be tense:pres mood:indic
air	noun	number:sing mean:obj,phys,nonlive,nat,simp,gas
air	element	
air	verb	tense:pres transitivity:trans mean:show
aisle	noun	number:sing
aitch	noun	number:sing mean:obj,abs,symb,letter

Figure 13.9 Dictionary entries

agree/verb/tns:pres/trans:trns,intrns/
agri/elt/
agro/elt/
ah/intrj/
ahead/advb/
ahem/intrj/
ahoy/intrj/
aid/noun/num:sing/
aid/verb/tns:pres/trans:trns,intrns/
ail/verb/tns:pres/trans:trns,intrns/
aim/noun/num:sing/
aim/verb/tns:pres/trans:trns,intrns/
ain't/verb/kind:aux/pers:frst/num:sing/form:be/tns:pres/mood:indictv/
air/noun/num:sing/mean:obj,phys,nonlive,nat,simp,gas/
air/elt/
air/verb/tns:pres/trans:trns/mean:show/
aisle/noun/num:sing/
aitch/noun/num:sing/mean:obj,abs,symb,letter/

Figure 13.10 Dictionary entries in compact format

13.4.2 Compiling the Dictionary

The dictionary is a large file, and it takes longer to load when the system is started than might be desired. Consequently it was decided to compile the dictionary into an even more compact format.

A compiler was written which reads the compact form of the dictionary, removes all of the separators, and replaces each word type, feature type, and feature value with a unique single character token. These tokens were chosen from those which would never appear in a dictionary word entry to simplify the task of recognising them when the compiled dictionary is read into memory during initialisation of the language understanding system.

After compilation a 4009 entry dictionary, which in its expanded form occupied 174538 bytes, and in its compact form occupied 92851 bytes of disk space, was reduced to only 41721 bytes. This is a reduction to 24% of its former expanded size, and to 45% of the compact form. The resulting file loads in considerably less time.

A complementary decompiler was written so that compiled dictionary files could be expanded to their less compact form to facilitate editing and debugging. This also provided a useful integrity check for the compiled dictionary. If a dictionary was compiled and then decompiled, and the result was identical to the original, then it can be said with some confidence that the compiled dictionary is a true representation of the original, and that it also obeys the syntax rules of the dictionary format.

14. SEMANTIC ANALYSIS AND KNOWLEDGE REPRESENTATION

In order to test a natural language understanding system in any realistic context it is necessary to give consideration to the provision of some form of semantic analysis. Semantic analysis is the subject of a great deal of the research effort in computational linguistics at present. Only a rudimentary implementation of some form of meaning analysis was provided for the purpose of testing the current system and defining future research goals.

14.1 Canonical Primitives

14.1.1 Verbs

One of the forms of semantic information provided in the dictionary is the semantic field. The system used is an adaptation of Yorick Wilks' 'primitives' [WILK73] [WILK75] and Roger Schank's *canonical primitive acts* [SCHA73][SCHA84][WILK72][WINS84]. These acts provide a concise way of describing the actions carried out by verbs.

Schank's primitives consisted of the following:

MOVE-BODY-PART	HEAR
MOVE-OBJECT	SMELL
EXPEL	FEEL
INGEST	MOVE-POSSESSION
PROPEL	MOVE-CONCEPT
SPEAK	THINK-ABOUT
SEE	CONCLUDE

They provide a basic set of meanings for verbs, and can describe acts in the physical, perceptual, mental and social worlds.

The original idea for these primitives came from the developers of Basic English [OGDE68]. They proposed that people could communicate effectively with a vocabulary of only about 1000 words by relying on the verbs:

COME	PUT
GET	TAKE
GIVE	HAVE
GO	SAY
KEEP	SEE
LET	SEND
MAKE	

It is likely that the verbs of English can be expressed by a surprisingly small number of combinations of these and similar primitives. The definition of auxiliary verbs already described is a similar concept to that of primitive acts.

14.1.2 Nouns and Adjectives

There is a need now to extend this set of primitives to include other types of words apart from verbs. The child's game *Animal, Vegetable and Mineral* points toward a way of incorporating nouns and adjectives into the scheme. Figure 14.1 shows the beginning of a tree of descriptors for objects. This is similar in style to the *type hierarchies* described by James Allen [ALLE87]. A combination of such descriptors is able to delineate the meaning of many nouns and adjectives.

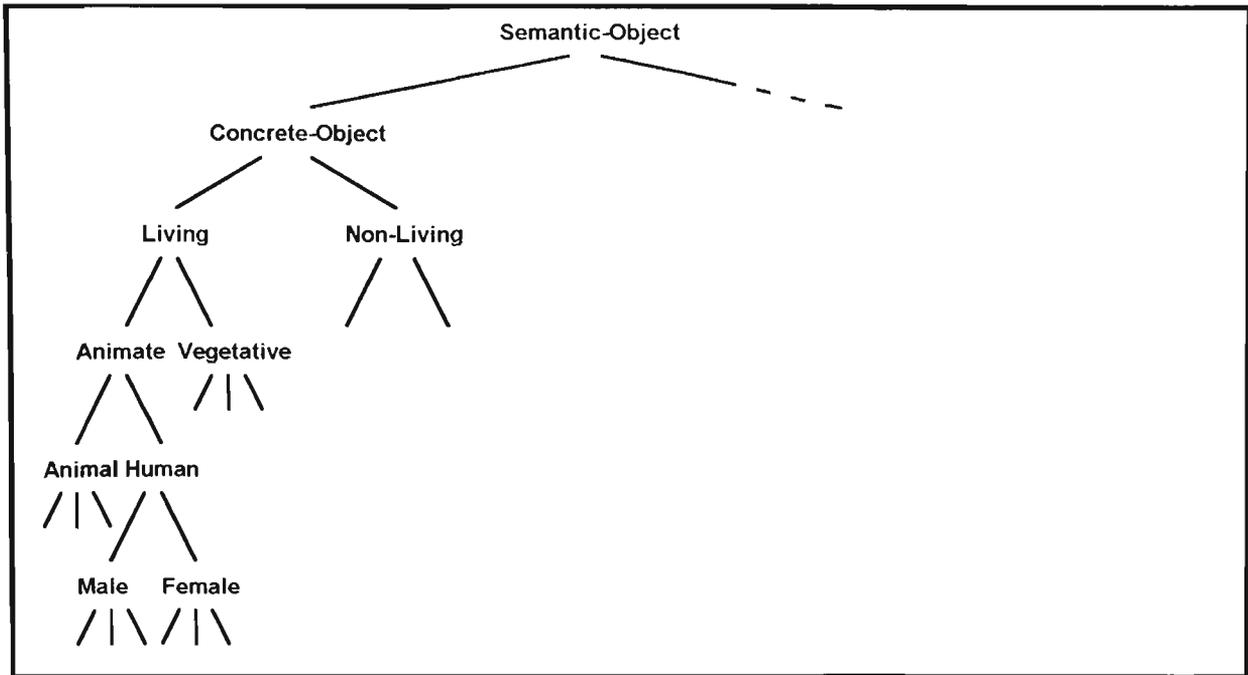


Figure 14.2 Alternative Treatment for Concrete Objects

Such anomalies are bound to occur all through any attempt to classify the complexity of reality in such a simple structure. For the present purposes of this project it was decided to follow a pragmatic approach - whatever works is right.

14.2 Object Oriented Semantic Actions and Descriptors

The above figures, when compared with the object hierarchies for object oriented programming libraries, provide the motivation for a compact and efficient approach to implementing in a program the data structures and action code required for a specific application using such semantic descriptors. This approach is to implement the descriptors and actions as the data fields and methods of a hierarchy of program objects in an object oriented language. The language chosen, for compatibility with the code already developed for this project, was Turbo Pascal with Objects [BORL92].

More specifically, the noun and adjective descriptors become the data fields of an object instance, while the actions carried out by the verbs are implemented as methods (functions) in the object. The property of inheritance can be used to build a hierarchy of such semantic objects in a library. The top of the tree, a Semantic-Object in Figure 14.1, would contain descriptors and methods basic to all other semantic objects, and each semantic object further down the tree would be based on alterations and additions to the properties of its parent object. Figure 14.3 illustrates the top level of such a library.

When a sentence is parsed, a register structure is produced, and values placed in the registers to describe the properties of the constituent parts of the sentence. The register structure produced is identical to that used by Winograd [WINO83], and an example taken from Winograd is shown in Figure 14.4. This illustrates the final register structure for the sentence: "*We have been given a firm deadline by the secretary*".

In the present system the semantic objects are added to the register structure after parsing is complete. It is an advantage to construct these objects as parsing proceeds, so that the information can be made available to guide the disambiguation of sentences.

Once a set of semantic objects has been constructed from a sentence, it is a fairly simple matter to carry out the indicated actions on the objects described. For testing purposes, each object can be passed a message which causes it to describe itself. In this way a simple and concise description of the meaning of the sentence can be quickly displayed.

Type

```
SemanticObject = Object
{ Descriptors (Data Fields) }
Name: NameString;
Position: Coordinate;
"      "

"      "
{ Actions (Methods) }
Procedure DescribeObject;
"      "

"      "
Function IsConcrete: Boolean;
Function IsAbstract: Boolean;
Procedure MoveBodyPart(Part: SemanticObject; Distance: Number;
                      Direction: Vector);
Procedure MoveObject(Distance: Number; Direction: Vector);
Procedure Expel(ObjectToExpel: SemanticObject);
"      "
"      "
End;
```

```
ConcreteObject = SemanticObject
{ Descriptors }
"      "

"      "
{ Actions }
Function IsLiving: Boolean;
Function IsNonLiving: Boolean;
"      "
"      "
End;
```

```
AbstractObject = SemanticObject
{ Descriptors }
Value: AbstractValue;
"      "

"      "
{ Actions }
Function IsNumber: Boolean;
Function IsTime: Boolean;
Function IsConcept: Boolean;
Function IsSymbol: Boolean;
Function IsLocation: Boolean;
Function IsQuality: Boolean;
"      "
"      "
End;
```

```
"      "
"      "
```

Figure 14.3 Semantic Object Library Implementation

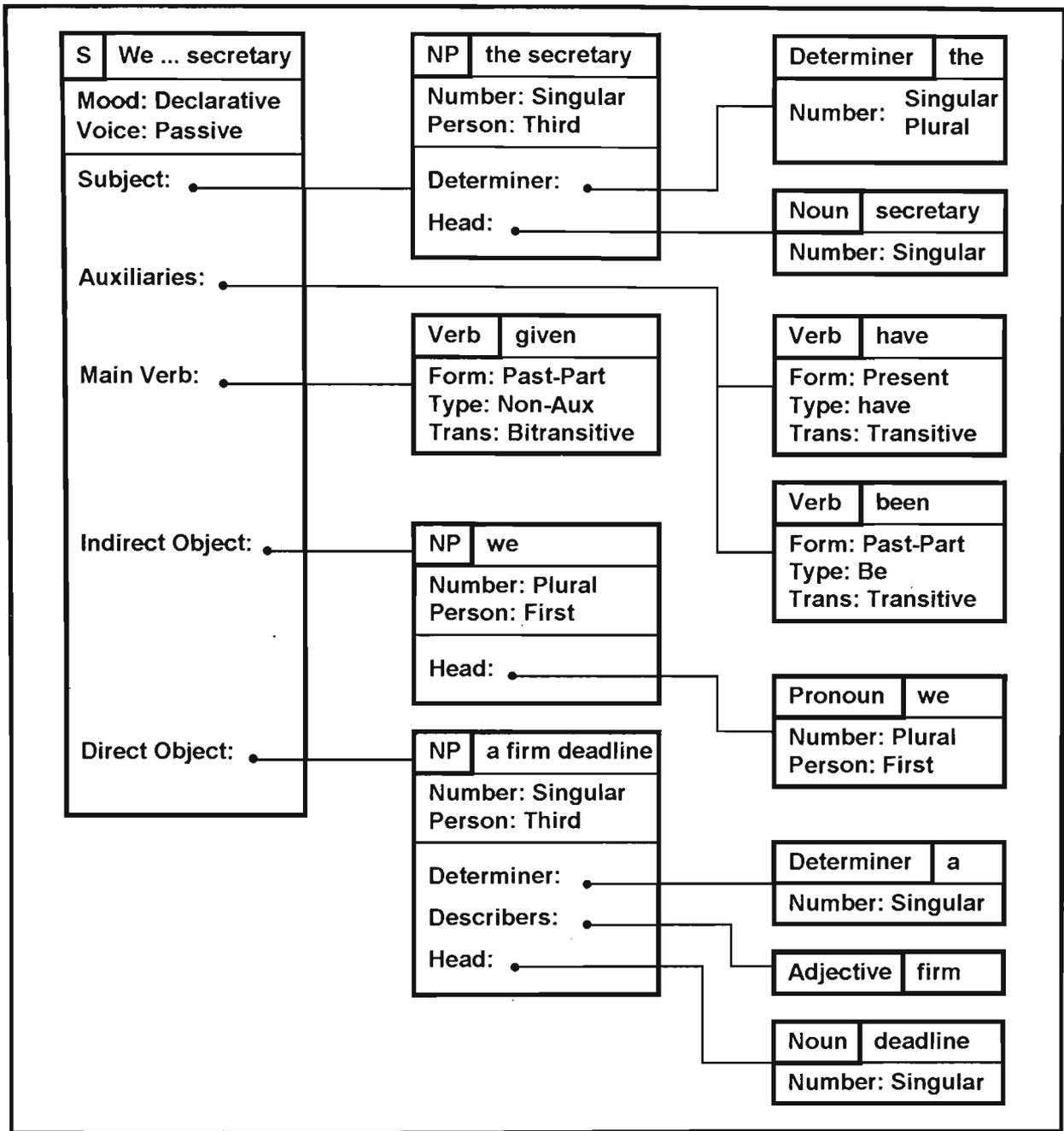


Figure 14.4 Register Structure for a Sample Sentence

For the purposes of controlling the SAR-10 system, the actions of the objects discovered can include code to load the most appropriate vocabulary for the context in which those objects normally operate.

The method chosen to augment such a structure with semantic information was to add a field to each of the resulting constituents of the main sentence register. These added fields contain the index number of a semantic object which defines the entity and the actions which it can perform or which may be performed on it. The register structure for the sample sentence, augmented by its semantic objects, can be seen in Figure 14.5.

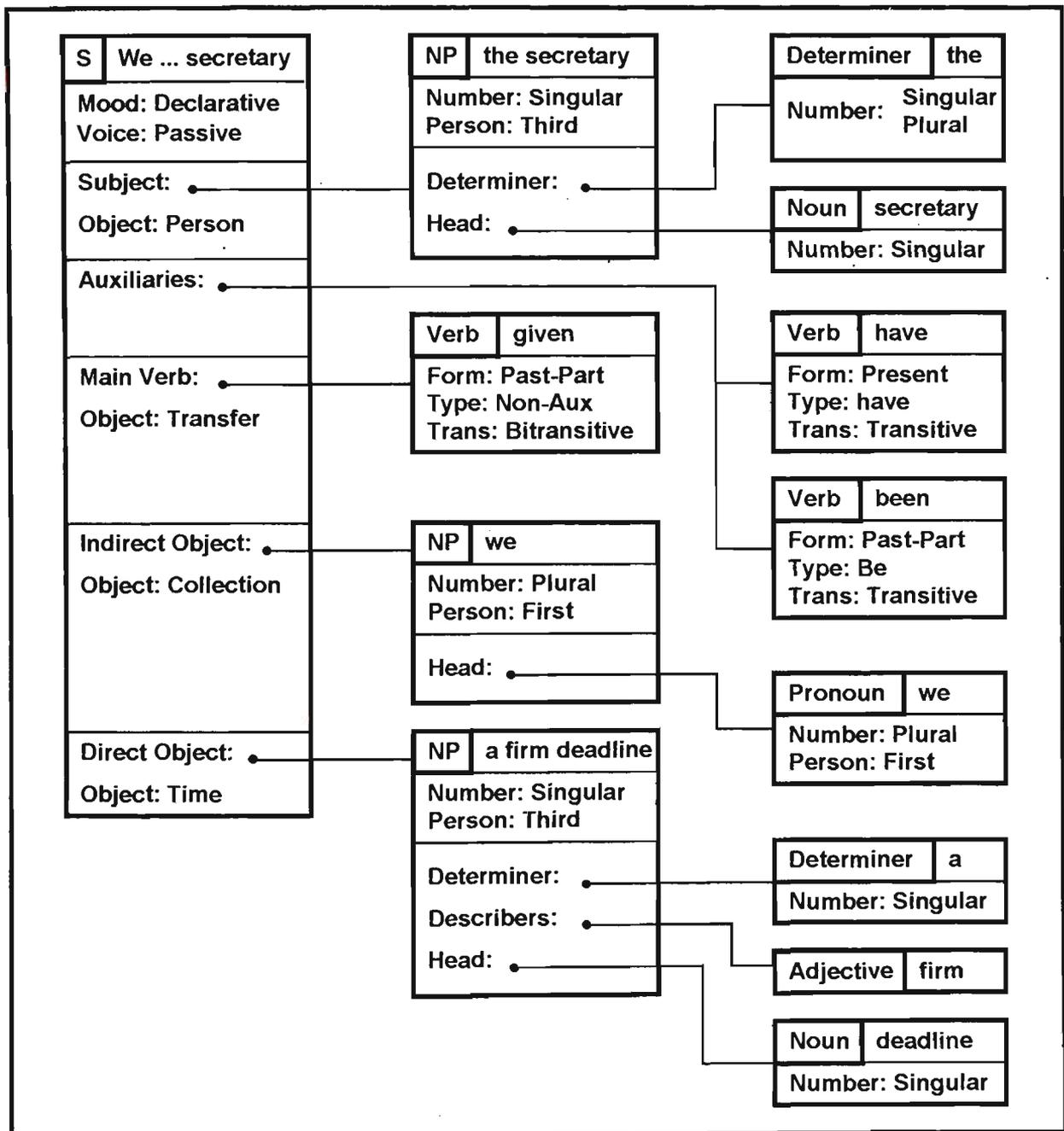


Figure 14.5 Register Structure Augmented by Semantic Objects

When the parser is being constructed, a list of names of the semantic objects is included with the grammar. During generation of the parser, a lookup table indexed by the numbers associated with the semantic object names, is constructed. Pointers to the actual semantic objects are placed in the table. By this means each register structure has access to the semantic object which describes the entity represented by the register.

Each semantic object can point to further semantic objects. For example, the indirect object of this sentence, *we*, is a **Collection** object. This will need to be further defined in terms of what objects the collection consists of (**Person**), and the number of such objects in the collection (**IndefinitePluralNumber**). Further information on the objects can also be gleaned from the syntactic information stored in the register structure.

Part III

Conclusion

15. FINDINGS AND FUTURE DIRECTIONS

15.1 Summary of the Final Results

The results of this project take two forms. The first is the information and experience gained in applying a relatively simple example of speech control and audio response systems to the kind of tasks most computer users require, and in incorporating natural language understanding capabilities into the system in an attempt to enhance its performance and user friendliness. In particular, the use of voice control as an aid to physically disabled users, or other users whose ability to use a keyboard is constrained by environmental factors, has been investigated, and found to be beneficial to such a user. On the other hand, little advantage was gained for able-bodied or physically unconstrained users because of the tedious nature of such ways of working. Many ideas for future work have been generated, and some of the pitfalls, and even dangers particularly in the use of speech control with robots, identified.

The second outcome of the project is the production of a set of tools which allow for the relatively easy incorporation of speech control and natural language understanding into existing and new applications. These fall into two areas. The first area relates to speech control:

- Utilities to control the SAR-10 Speech Control and Audio Output system,
- The *VOICEDOS* extensions to the DOS operating system to allow for voice control,
- The *SARLIB* library of speech control and audio output routines to use in new application programs.

The second relates to natural language understanding:

- An efficient dictionary,
- A parallel parser for natural language applications,
- Improved techniques for using the LALR compiler generator in context sensitive applications,
- The beginning of a promising approach to the handling of semantic information using object oriented techniques,
- Tools to facilitate the construction and testing of natural language understanding systems.

A prototype system has been constructed, consisting of a general purpose computer with the SAR-10 speech control and audio response system, a MIDI interface and music synthesiser, and an interface to a Rhino robot. It is equipped with software for carrying out general tasks such as word processing, spreadsheet calculations and database operations, as well as more specialised music composition and performance programs and a voice controlled programming environment. Some of this software consists of well known commercial packages, while some is built specifically with speech control and audio response in mind. In addition, this computer contains the growing computational linguists toolkit, which allows experimentation and development work to be carried out on voice control and natural language understanding, much of it under voice control. Multi media equipment is currently being installed in the system. Testing with disabled users is in its very early stages yet, but the author has gained considerable experience and insight using it, which should assist its application to this more specialised area.

15.2 Future Enhancements

There are several areas in which this system can be improved. The most obvious is to replace the SAR-10 isolated phrase recognition system with a continuous speech recognition system. This would provide a more natural voice command system, and would greatly improve the performance of the system in applications requiring a large amount of text entry. It is envisaged that this will be possible in the near future, as such systems are just beginning to become available at reasonable cost to microcomputer users.

The relatively new field of neural networks, with its promise of adaptable pattern recognition, shows promise for speech recognition, either by using networks simulated in software, or by producing neural network chips dedicated to the task.

The recent explosion of interest in multimedia computing has made available high performance and low cost audio input and output hardware in the form of sound cards, such as the Sound Blaster 16 ASP, made by Creative Labs. Software for speech recognition using these cards, such as Voice Assist, is available for use in Windows 3.1 systems. Development of a system based on these would overcome the limitations of the present system based on the SAR-10.

An attractive approach to attacking the task of continuous speech recognition would be to use a highly parallel system comprising processing paths which simultaneously perform acoustic, syntactic and semantic analysis, and at all times sharing and correcting the knowledge obtained at each stage. Modern digital signal processing and floating point transputers might make a low cost continuous speech recogniser based on this model feasible. The available parallelism could be used to process the incoming speech simultaneously through a large number of identical analysers, all in parallel but separated

slightly in time. Cross correlation of the extracted speech parameters while the analysis proceeds might provide sufficient information to allow identification of individual words in a continuous speech stream.

The entire source code for the system is presently being translated into an object oriented language - Borland Pascal with Objects [BORL92b], enabling each component of the system to become an object in a class library, similar to that used for the semantic objects (see Chapter 14). This translation is being carried out with the aid of LALR, for which a Turbo Pascal with Objects template is being produced.

In parallel with the production of an object oriented version is its transformation into a Windows 3.1 application. This will allow for a far more powerful user interface, but requires the *SARLIB* library to become a Dynamic Link Library (DLL) and a Windows device driver to be constructed for the SAR-10 to replace the *VOICEDOS* driver. Once this is done it will be a relatively simple matter to incorporate speech control and a natural language interface into Windows applications.

VOICEDOS may be enhanced by incorporating some of the frequently used utilities, such as a calculator, calendar, editor or notepad, modem controller and file transfer facility, particularly as it is unlikely that standard utilities will be optimised for use with a speech control system. Incorporating these into *VOICEDOS* would make use of the system easier, especially for a disabled user.

The performance of the dictionary may be improved, particularly in the processing of affixes. Little effort has been put into optimisation of the order of application of the affix rules, but it is believed that some improvement in processing speed could be gained if this were done.

The design of suitable prefix rules presents little difficulty. They are simply appended to existing words and so can be as easily removed. However, there may be a case for more complex prefix rules. For example, *at* is equivalent to *ad* before a *t*, as in: *attend* -> *(at) tend*. This could be expressed as the rule *+at?t*, meaning add *at* if next letter is *t*. It would be removed by checking if the first three letters are *att* and if so removing the *at*.

A possible aid to the process of filtering out incorrect words produced by the application of suffix rules would be to make a reject table of words generated by the rules which are wrong, for example *ad* from *adding* via the rule *+C+ing*. The overheads involved in this would mean it would be best to restrict its use to overcoming intractable cases involving important words, and then only if the problem cannot be resolved by reordering the rule applications or rearranging the partitioning of the rules.

Although the parallel parser was implemented on a single processor machine, its design lends itself naturally to implementation on a parallel processor. As the desirability of using such technology for natural language understanding systems has already been mentioned, this only reinforces the idea that such a concept should be explored. If enough processors were available, it would be a natural approach to use hierarchical processor allocation, i.e. allocate a processor to each partial parse as it commences, releasing the processors for reuse when any of these parses fails. A single processor could be used to control the scanning of the input, the allocation of resources during parsing, and the presentation of the resulting parses to the next stage of processing. Parsing based on parallel processors would provide a natural complement to the parallel speech analysis system described above

15.3 Construction of Natural Language Programming Tools

The system as described so far in effect forms a set of relatively independent resources which could be used to add speech control and/or natural language understanding capabilities to almost any application program. In order to test the feasibility of using the various components of this system in an integrated form, a suitable application was needed.

As the author's major interest in applying these techniques is in the development of more powerful natural language interfaces, the next logical step is to construct a speech control and natural language programming toolkit and testbench. Such a system forms the next stage in this research project. It might best be described as a computational linguist's assistant.

The facilities provided by the toolkit and workbench will include:

- speech analysis,
- speech control and audio response vocabulary production and testing,
- dictionary construction and testing,
- natural language scanner and parser construction and testing.

It will run in Windows 3.1 and be controlled as far as possible using speech input and output. Graphical displays of speech waveforms, parse trees and semantic trees will be provided. Device drivers for various speech input, output and control devices will be included, with the ability to add drivers for new hardware as it becomes available.

A parser generator similar to *LALR*, but adapted to handle context sensitive grammars with embedded semantic descriptors and produce an object oriented scanner and parallel parser capable of accessing semantic information as it carries out syntax analysis, is being developed and will be incorporated in the toolkit. This parser generator will be able to

take, as input, a grammar containing syntactic augmentations, semantic descriptors and semantic object references in a more convenient form than that adopted at present.

In addition, as mentioned in Chapter 11, a recursive descent parser generator might be provided, based on the use of user supplied skeleton code which will be filled out from an augmented grammar, similar to the way the LALR(1) generator works. The readability of the code of such a parser is useful for teaching purposes.

A feature of multimedia computer systems is the CD-ROM drive. This makes available excellent voice analysis software tools, and also extensive dictionaries which may assist in the production of natural language systems. The next step in this project will be to construct a machine readable dictionary, based on the Complete Oxford Dictionary on CD-ROM. This dictionary package has tools which enable the user to extract information in various forms. Using the existing natural language system to interpret this information, and the dictionary building tools already produced, it is felt that a more complex machine readable dictionary than the present one can be produced. The existing system will, in effect, *bootstrap* the next generation system.

15.4 Conclusions

This work has demonstrated that development of voice controlled aids to assist a disabled person to use a computer, and hence any equipment which can be controlled by that computer, is feasible. Only limited usability is available with present low cost speech recognition systems, but as continuous speech recognition becomes available this situation will improve.

There are dangers inherent in applying speech control to moving mechanical devices such as robot arms if those devices are to operate in close proximity to a handicapped user, and in such a way that safe operation depends on accurate response to the user's voice commands.

The reliability and ease of use of speech recognition systems is enhanced by the provision of suitable training, optimising and operating utilities, especially if these are also capable of being used under speech control.

The provision of operating system extensions and a library of routines designed to interface with the speech recognition system makes it possible to incorporate voice control and audio response into both existing and new application software.

The efficient and convenient use of low cost speech recognition systems can be enhanced considerably by using natural language understanding techniques to free the user from some of the constraints imposed by the language requirements of the speech interface.

Unconventional use of tools and techniques designed primarily for development of context-free computer language parsers makes possible their use for the production of context-sensitive natural language parsers, and provides a basis from which grammars and parser generators specifically designed for natural languages may be produced.

The success of the music performance and composition system described in Chapter 5 adequately demonstrates some of the possibilities voice control of suitable software can provide to a disabled user - possibilities perhaps unachievable in any other way. The potential of such a system opens up a large number of employment and enjoyment opportunities to a disabled person. In addition to music composition, such areas as film, video and audio production and editing, using modern largely automatic equipment, could be considered.

BIBLIOGRAPHY

[AINS76] Ainsworth, W.A., "Mechanisms of Speech Recognition", Pergamon Press 1976.

[AHO77] Aho, Alfred V. and Ullman, Jeffrey D., "Principles of Compiler Design", Addison-Wesley 1977.

[ALLE87] Allen, James, "Natural Language Parsing", Benjamin/Cummings 1987.

[ANGE86] Angermeyer, John and Jaeger, Kevin, "MS-DOS Developer's Guide", Howard W. Sams 1986.

[BEES86] Beesley, K. R. (1986), "Machine Assisted Translation with a Human Face", *Data-processing*, vol 28 no 5 June 1986.

[BIBL76] The Bible Societies, "Good News Bible: Today's English Version", The Bible Societies/Collins/Fontana 1976.

[BIER83] Biermann, A.; Rodman, R.; Ballard, B.; Betancourt, T.; Bilbro, G.; Deas, H.; Fineman, L.; Fink, P.; Gilbert, K.; Gregory, D. and Heidlage, F., "Interactive Natural Language Problem Solving: A Pragmatic Approach", in *Proceedings of the Conference on Applied Natural Language Processing*, ACL, Santa Monica, California 1983.

[BORL85] Borland International, "Superkey Macro Processor, Version 1.03A", Borland International, Scotts Valley, California 1985.

[BORL88a] Borland International, "Turbo C, Version 2.0", Borland International, Scotts Valley, California 1988.

[BORL88b] Borland International, "Turbo Pascal, Version 5.0", Borland International, Scotts Valley, California 1988.

[BORL92a] Borland International, "Borland C++, Version 3.1: User's Guide", Borland International, Scotts Valley, California 1992.

[BORL92b] Borland International, "Turbo Pascal with Objects, Version 7.0: User's Guide", Borland International, Scotts Valley, California 1992.

[BROD92] Brodtkorb, Jo; TanGaard, Karsten and Blix, Brynjulf, "Musicator GS for Windows: Owner's Guide", Musicator A/S, Oslo, Norway 1992.

[BROW82] Brown, E.K. and Miller, J.E., "Syntax: Generative Grammar", Hutchinson 1982.

[BUTL85] Butler, Christopher S., "Systemic Linguistics: Theory and Applications", Batsford Academic and Educational 1985.

[CARL86] Carlson, A. Bruce, "Communication systems: an introduction to signals and noise in electrical communication", 3ed., p452, McGraw-Hill Book Co, Singapore 1986.

[CATE82] Cater, Arthur William Sebright, "Request Based Parsing with Low Level Syntactic Recognition", in Sparck Jones and Wilks 1983.

[CHOM56] Chomsky, Noam, "Three Models for the Description of Language", *IRE Trans. on Information Theory* 2:3, pp113-124 1956.

[CHOM57] Chomsky, Noam, "Syntactic Structures", Mouton, The Hague 1957.

[CHOM59] Chomsky, Noam, "On Certain Formal Properties of Grammars", *Information and Control* 2:2, 137-167 1959.

[CHOM65] Chomsky, Noam, "Aspects of the Theory of Syntax", MIT Press 1965.

[DALE48] Dale, E. and Chale, J.S., "A Formula for Predicting Readability: Instructions", in *Education Research Bulletin*, February 18, 1948.

[DERE69] DeRemer, F.L., "Practical Translators for LR(k) Languages", PhD dissertation, MIT, Cambridge, Mass 1969.

[DEFU89] De Furia, Steve and Scacciaferro, Joe, "The MIDI Programmer's Handbook", M&T Publishing Inc., Redwood City, California 1989.

[DERE71] DeRemer, F.L., "Simple LR(k) Grammars", *Comm. ACM* 14:7, 453-460 1971.

[DOW86] Dow, Malcolm J. and Stevens, Mark, "Adaptive Audio Noise Cancelling Unit", Footscray Institute of Technology, Department of Electrical and Electronic Engineering, unpublished project report, Footscray 1986.

[DOW87a] Dow, Malcolm J. and na Ranong, Chula, "An Interrupt Driven Memory Resident Control Program for a General Purpose Speech Recognition System", *Conference on Computing Systems and Information Technology*, The Institution of Engineers, Australia, Brisbane 1987.

[DOW87b] Dow, Malcolm J. and na Ranong, Chula, "Natural Language Interface for a Disabled Person's Aid" in *Proceedings of the Australian Colleges of Advanced*

Education 18th Annual Computer Conference, South Australian Institute of Technology, Adelaide 1987.

[DOW94a] Dow, Malcolm J., "SARLIB: A Library of SAR-10 Speech Recognition and Audio Response Interface Routines", Technical Report 1, Computer Application Software and Hardware Group, Department of Electrical and Electronic Engineering, Victoria University of Technology, Footscray, Victoria 1994.

[DOW94b] Dow, Malcolm J., "A Machine Readable Dictionary for Natural Language Understanding Systems", Technical Report 2, Computer Application Software and Hardware Group, Department of Electrical and Electronic Engineering, Victoria University of Technology, Footscray, Victoria 1994.

[DOW94c] Dow, Malcolm J., "Affix Transforms for a Machine Readable Dictionary for Natural Language Understanding Systems", Technical Report 3, Computer Application Software and Hardware Group, Department of Electrical and Electronic Engineering, Victoria University of Technology, Footscray, Victoria 1994.

[EARL70] Earley, J., "An Efficient Context Free Parsing Algorithm", *Communications of the ACM*, August 1970.

[FLAN80] Flanagan, J.L., Levinson, S.E., Rabiner, L.R. and Rosenberg, A.E., "Techniques for Expanding the Capabilities of Practical Speech Recognizers", in *Trends in Speech Recognition*, ed. Lea W.A., Prentice-Hall, 1980.

[FOWL70] Fowler, H.W. and Fowler, F.G. (eds.), "The Concise Oxford Dictionary", Book Club Associates/Oxford University press, Oxford 1970.

[GAZD85] Gazdar, Gerald; Klein, Ewan; Pullum, Geoffrey and Sag, Ivan, "Generalised Phrase Structure Grammar", Basil Blackwell 1985.

[GOOD84] Goodall, R.; Harrang, J. and Lindsey, D., "Perfect Writer with Perfect Speller and Perfect Thesaurus", Perfect Software Inc. 1984.

[GRIS86] Grishman, Ralph, "Computational Linguistics: An Introduction", *ACL Studies in Natural Language Processing*, Cambridge University Press 1986.

[GUEN78] Guenther, F. and Guenther-Reutter, M. (eds.), "Meaning and Translation: Philosophic and Linguistic Approaches", Duckworth 1978.

[HEND77a] Hendrix, Gary Grant, "The LIFER Manual: A Guide to Building Practical Natural Language Interfaces", Technical Note 138, Artificial Intelligence Centre, SRI International, February 1977.

[HEND77b] Hendrix, Gary Grant, "Human Engineering for Applied Natural Language Research", *Proceedings of the 5th International Conference on Artificial Intelligence*, Cambridge, MA, August 1977.

[HIRS87] Hirst, Graeme, "Semantic Interpretation and the Resolution of Ambiguity", *ACL Studies in Natural Language Processing*, Cambridge University Press 1987.

[HOEY83] Hoey, Michael, "On the Surface of Discourse", George Allen and Unwin 1983.

[HOPC79] Hopcroft, J. and Ullman, J., "Introduction to Automata Theory, Languages, and Computation", Addison-Wesley 1979.

- [HUDS84] Hudson, Richard, "Word Grammar", Basil Blackwell 1984.
- [JOHN75] Johnson, S.C., "YACC - Yet Another Compiler Compiler", *CSTR 32*, Bell Laboratories, Murray Hill, NJ 1975.
- [KING83] King, Margaret (ed.), "Parsing Natural Language", Academic Press 1983.
- [KNUT65] Knuth, D.E., "On the Translation of Languages from Left to Right", *Information and Control 8:6*, 607-639 1965.
- [LEA80] Lea, W.A., "Speech Recognition: Past, Present, and Future", in *Trends in Speech Recognition*, ed. Lea W.A., Prentice-Hall, 1980.
- [LEEC81] Leech, Geoffrey N., "Semantics: The Study of Meaning", Penguin 1981.
- [LEEC87] Leech, Geoffrey N., "Meaning and the English Verb", 2 ed., Longman 1987.
- [LESK75] Lesk, M.E., "LEX - a Lexical Analyzer Generator", *CSTR 39*, Bell Laboratories, Murray Hill, NJ 1975.
- [MACQ81] Delbridge, A. (ed.) et.al., "The Macquarie Dictionary", Macquarie Library 1981.
- [MALL44] Mallery, Richard D., "Grammar, Rhetoric and Composition for Home Study", Barnes & Noble 1944.
- [MANN87] Mann, Paul, "LALR, an LALR(1) Parser Generator", LALR Research, Burnt Mill, Tustin CA, 1987.

[MARK93] Markowitz, Judith, "The power of speech", in *AI Expert, Vol 8, No 1*, pp29-33, January 1993

[MART76] Martin, T.B., "Practical Applications of Voice Input to Machines", *Proc. IEEE, Vol. 64, No. 4*, pp 487-501, April 1976.

[MELL85] Mellish, C.S., "Computer Interpretation of Natural Language Descriptions", Ellis Horwood 1985.

[MEYE88] Meyer, Bertrand, "Object-oriented Software Construction", Prentice Hall 1988.

[MICR87] Micropro International, "Wordstar Professional Release 4.0", Micropro International Corporation, 1987.

[MICR83] Microsoft Corporation, "MS-DOS Programmer's Reference", 1983.

[MILL82] Mills, Helen, "Connecting and Combining in Sentence and Paragraph Writing", Scott Foresman and Company 1982.

[MOOR82] Moore, Terence and Christine Carling, "Understanding Language: Towards a Post-Chomskyan Linguistics", Macmillan 1982.

[MOYN85] Moyne, John A., "Understanding Language: Man or Machine", Plenum Press 1985.

[MYER76] Myers, Patricia I. and Hammill, Donald D., "Methods for Learning Disorders", John Wiley and Sons 1976.

[NARA86a] na Ranong, C. and Dow, M.J., "A Voice Controlled Robot", *Robots in Australia's Future Conference*, Perth, Western Australia 1986.

[NARA86b] na Ranong, C., "A Robot Control Program", FIT Centre for Automation Technology Internal Report, Footscray Institute of Technology, Department of Electrical and Electronic Engineering, Footscray, Victoria 1986.

[NAUR63] Naur, P. (ed), "Revised Report on the Algorithmic Language ALGOL 60", *Comm. ACM* 6:1, 1-17 1963.

[NEC85] NEC, "Sar-10 Speech and Audio Response Unit, User's Manual". NEC Corporation 1985.

[OGDE68] Ogden, C.K., "Basic English: International Second Language", Harcourt, Brace and World, New York 1968.

[PARK78] Parker-Rhodes, A.F., "Inferential Semantics", Humanities Press 1978.

[PERR86] Perrault, C.R. & Grosz, B.J., "Natural Language Interfaces", in *Annual review of computer science* 1, 47-82, 1986.

[PGMU91] PG Music Inc., "Band-in-a-Box, IBM Version 5: User's Manual", PG Music Inc., Buffalo, New York 1991.

[RHIN82] "Hands On Introduction to Robotics, the Manual for the XR-1", Rhino Robots Inc., Champaign, Illinois 1982.

[RIES75] Riesbeck, Christopher K., "Conceptual Analysis", in Schank 1975.

[RIES78] Riesbeck, Christopher K. and Schank, Roger C., "Comprehension by Computer: Expectation-based Analysis of Sentences in Context", in Levelt, Willem J.M. and Flores D'Arcais, Giovanni (eds.), Research Report 78, Department of Computer Science, Yale University, October 1976.

[ROBI79] Robins, R.H., "A short history of linguistics", Longman, London 1979.

[ROBI80] Robins, R.H., "General linguistics: an introductory survey", Longman, London 1980.

[ROLA92] Roland Corporation, "Roland JV-30 16 Part Multi Timbral Synthesiser Owner's Manual", Roland Corporation, Osaka, Japan 1992.

[RUBI86] Rubin, Paul, "Bison Documentation", Free Software Foundation, 1986.

[SAGE81] Sager, N., "Natural Language Information Processing", Addison-Wesley, Reading, MA 1981.

[SCHA73] Schank, Roger C. and Colby, Kenneth (eds.), "Computer Models of Thought and Language", W. H. Freeman, San Francisco, CA 1973.

[SCHA75] Schank, Roger C. (ed.), "Conceptual Information Processing", *Fundamental Studies in Computer Science 3*, North Holland, Amsterdam 1975.

[SCHA84] Schank, Roger C. and Childers, Peter G., "The Cognitive Computer: On Language, Learning and Artificial Intelligence", Addison-Wesley, Reading, Massachusetts 1984.

[SILB88] Silberschatz, Abraham and James L. Peterson, "Operating System Concepts", Addison-Wesley 1988.

[SLOC85] Slocum, J., "A Survey of Machine Translation", in *Computational Linguistics* 11, 1, 1-17, 1985.

[SMIT80] Smith A.R. and Sambur M.R., "Hypothesizing and Verifying Words for Speech Recognition", in *Trends in Speech Recognition*, ed. Lea, W.A., Prentice-Hall, 1980.

[SNEL79] Snell, Barbara M., ed., "Translating and the Computer", North Holland, Amsterdam 1979.

[SOFT85] Software Channels Inc., "Alice: the Personal Pascal", Software Channels Inc., Kingswood, Texas 1985.

[SPAR83] Spark Jones, Karen and Wilks, Yorick Alexander (eds.), "Automatic Natural Language Parsing", Ellis Horwood/John Wiley, Chichester 1983.

[STOC77] Stockwell, Robert P., "Foundations of Syntactic Theory", Prentice-Hall 1977.

[THOM85] Thompson, Beverly and Thompson, William, "MicroExpert", McGraw-Hill, New York 1985.

[TOMI87] Tomita, Masaru, "An Efficient Augmented-Context-Free Parsing Algorithm", in *Computational Linguistics*, Vol. 13 Nos. 1-2 January-June 1987.

[TRAC92] Tracy, Tom; Boggia, Jim; Kuldell, Suzanne; McCutcheon, Bill; Senior, John O. and Whipple, Bill, "ENSONIQ KS-32 Weighted Action MIDI Studio Musician's Manual, Version 1.0", ENSONIQ Corp, Malvern, Pennsylvania 1992.

[VANR86] van Riemsdijk, Henk and Williams, Edwin, "Introduction to the Theory of Grammar", The MIT Press 1986.

[WADE84] Wade, Howard H., "Perfect Calc", Perfect Software Inc./Thorn EMI, Costa Mesa, California 1984.

[WALK87] Walker, Adrian; McCord, Michael; Sowa, John F. and Wilson, Walter G., "Knowledge Systems and PROLOG", Addison-Wesley 1987.

[WEAV49] Weaver, Warren, "Translation", in *Machine Translation of Languages*, W. Locke & A.D. Booth (eds.), Technology Press of M.I.T. & Wiley and Sons, New York 1949.

[WILK72] Wilks, Yorick and Charniak, Eugene, "Grammar, Meaning, and the Machine Analysis of Language", Routledge and Kegan Paul, London 1972.

[WILK73] Wilks, Yorick, "The Stanford Machine Translation Project", in *Natural Language Processing*, Rustin, R. (ed.), Algorithmics Press, New York 1973.

[WILK75] Wilks, Yorick, "Preference Semantics", in *Formal Semantics of Natural Language*, Keenan, E. (ed.), Cambridge University press, Cambridge 1975.

[WINO72] Winograd, Terry, "Understanding Natural Language", Edinburgh University Press, Edinburgh 1972.

[WINO83] Winograd, Terry, "Language as a Cognitive Process, Volume 1: Syntax", Addison-Wesley 1983.

[WINS84] Winston, Patrick Henry, "Artificial Intelligence", 2ed., Addison-Wesley, Reading, Massachusetts 1984.

[WINS92] Winston, Patrick Henry, "Artificial Intelligence", 3ed., Addison-Wesley, Reading, Massachusetts 1992.

[WINT82] Winter, Eugene O., "Towards a Contextual Grammar of English", George Allen and Unwin 1982.

[WOOD73] Woods, W.A., "An Experimental Parsing System for Transition Network Grammars", in Rustin, R. (ed.), *Natural Language Processing*, Algorithmics Press, New York 1973.

Appendices

Appendix A: Parts of Speech for English

Main Reference: Mallery, Richard D., "Grammar, rhetoric and composition for home study", Barnes & Noble 1944. [MALL44]

Noun

Classes: *common, proper*

Groups: *abstract, concrete, collective*

Properties:

gender:

masculine, feminine, neuter, common

person:

first, second, third

number:

singular, plural

A **regular noun** forms its plural by adding *s* to the singular form.

Nouns ending in *ch, s, sh, x* or *z* add *es*.

Some nouns ending in *o* add *es*, e.g. *hero* -> *heroes*.

Some nouns ending in *o* add *s*, e.g. *studio* -> *studios*.

Nouns ending in *f* or *fe* change *f* to *v* and add *es*.

Nouns ending in *y* preceded by a consonant or *qu* change *y* to *i* and add *es*.

Nouns ending in *y* preceded by a vowel add *s*.

An **irregular noun** forms its plural by retaining the older plural form of *en* or *ren*, e.g. *child* -> *children*, *ox* -> *oxen*, or by a vowel change, e.g. *man* -> *men*, *foot* -> *feet*.

A few nouns are unchanged in the plural, e.g. *sheep, swine, trout, deer*.

Some nouns have two plural forms, one regular, one irregular, e.g. *brother* -> *brothers* or *brethren*, *fish* -> *fishes* or *fish*.

Some nouns are plural in form and singular in meaning, e.g. *mathematics, politics, news*.

A noun plural in form may, if used collectively, be treated as singular, e.g. "*Ten years is a long time.*"

Most compound nouns form the plural by adding *s* to the end of the last word, e.g. *high school* -> *high schools*.

If the first member is more important, *s* may be added to this member, e.g. *father-in-law* -> *fathers-in-law*.

case:

nominative: *denotes the person or thing acting.*

possessive: *denotes the person or thing possessing.*

objective: *denotes the person or thing acted upon.*

The possessive case involves a change in the form of the noun.

If the noun is singular apostrophe *s* is added.

If the noun is plural an apostrophe is added after the *s*.

Plural nouns not ending in *s*, add *'s*.

Compound nouns, add *'s*, e.g. *sister-in-law -> sister-in-law's*.

If the singular ends in *s* and the word is short, add *'s*, e.g. *Keats's poems*.

If the singular ends in an *s*-sound and the following word begins with an *s*-sound, the apostrophe only is added.

English idioms sometimes require a double possessive, e.g. *that book of George's*.

Pronoun

Used in place of a noun, to avoid repetition of the noun to which it refers.

Classes of pronoun:

personal:

Stand directly for names of persons, places or things. E.g. *I, you, he, she, it, we, they*.

Compound forms are made by adding *self*, e.g. *himself, myself*. These are called **reflexive** if they are in the predicate of the sentence and refer back to the subject. E.g. *"He injured himself last week"*.

These are called **intensive** if they reinforce or give emphasis to another word in the same part of the sentence. E.g. *"The governor himself will be at the meeting"*.

relative:

Refer to antecedents and, at the same time, introduce independent clauses. E.g. *who, what, that, which*.

Compound relative pronouns are formed by adding *ever* or *soever*. E.g. *whoever, whatever, whatsoever, whichever*.

interrogative:

Used in asking questions. E.g. *who, which, what*.

Direct questions: E.g. *"Just what do you mean by that?" "Which will you have?"*

Indirect questions (no question mark): E.g. *"I want to know what you mean by that"*.

demonstrative:

Point definitely to persons or things to which they refer. E.g. *this, that, these, those*. E.g. *"This is my plan"*.

indefinite:

Point out persons or things, but less definitely than demonstratives. E.g. *all, any, anybody, anyone, anything, each, either, everybody, everything, few, neither, nobody, none, one, several, some*. E.g. *"Some stayed away from the meeting"*.

Properties of pronouns:

person:

first person E.g. *I, we*
second person E.g. *you*
third person E.g. *he, she, it, they*

gender:

Personal pronouns in the third person vary in form to show gender change.
He is masculine, she is feminine, it is neuter.

number:

singular or plural depending on the number of the antecedent.
E.g. singular: *I will work now*. plural: *We will work now*.
singular: *You will work now*. plural: *You will work now*.

demonstrative:

plural: *these, those*.

Indefinite pronouns:

singular: *one, someone, each*.
plural: *few, all, many*.

Interrogative and **relative** pronouns don't change form to indicate a change in number.

case:

Personal pronouns and a few **relative** pronouns show change in case by a change in form.

Nominative case:

	Singular	Plural
First person	<i>I</i>	<i>we</i>
Second person	<i>you</i>	<i>you</i>
Third person	<i>he, she, it</i>	<i>they</i>

Possessive case:

	Singular	Plural
First person	<i>my (mine)</i>	<i>our (ours)</i>
Second person	<i>your (yours)</i>	<i>your (yours)</i>
Third person	<i>his, her (hers), its</i>	<i>their (theirs)</i>

Objective case:

	Singular	Plural
First person	<i>me</i>	<i>us</i>
Second person	<i>you</i>	<i>you</i>
Third person	<i>him, her, it</i>	<i>them</i>

Relative pronouns:

Nominative case:	<i>who, that, which</i>
Possessive case:	<i>whose</i>
Objective case:	<i>whom, that, which</i>

Verb

Verbs say or assert something about a person, place or thing. They may make a statement, ask a question, or give a command. They may express action, occurrence, or mode of being.

Kinds of verbs:

- Verbs are either **transitive** or **intransitive**.

Transitive

Require an object.
E.g. *Mrs. Jones bakes wonderful pies.*

Intransitive:

Don't require an object.
E.g. *She walks rather rapidly.*

Some verbs can be both transitive and intransitive. E.g.
 The woman ran a bazaar at the fair. (transitive)
 The horse ran away. (intransitive)

- Verbs are either **principal** verbs or **auxiliary** verbs.

Principal:

Complete in themselves.

Auxiliary:

Joined to the principal verb to express the idea of the principal verb more fully.

E.g. *She speaks.* (*speaks* = principal verb) *She will speak.* (*will* = auxiliary verb)

Most common auxiliary verbs: *be, can, do, have, may, must, ought, should, shall, will, would.*

- Verbs are either **regular** or **irregular**.

Regular:

Forms past tense and past participle by adding *d* or *ed* to the present tense (or simple form of the verb).

Irregular:

Forms past tense and past participle by some internal change in the word.
 All auxiliary verbs are irregular.

	Present	Past	Past participle
Regular	<i>permit</i>	<i>permitted</i>	<i>permitted</i>
Irregular	<i>am</i>	<i>was</i>	<i>been</i>
Irregular	<i>do</i>	<i>did</i>	<i>done</i>

Properties of verbs:

Voice:

Active:

The subject of the verb performs the action.
 E.g. *He gave me the money.*

Passive:

The subject of the verb is acted upon.
 E.g. *The money was given to me.*

Mood:

Manner in which the state of being or action of the verb is regarded.

Indicative:

States a fact or asks a question.

E.g. *We are uneasy about it.* (*are* = verb)

Are we uneasy about it? They say that we are uneasy about it.
(*say, are* = verbs)

We are not uneasy about it. (*are* = verb)

Subjunctive:

Suggests that something is uncertain, imagined, desirable, undesirable, or contrary to fact.

Generally found in a dependent clause, introduced by a conjunction such as *if, though, lest, that, till, or unless*.

E.g. *If I were you, I should not worry.* (*were* = verb)

It is essential that you be on the alert. (*be* = verb)

Imperative:

Expresses a request or a command. The subject is usually omitted.

E.g. *Introduce me to your friend.* (*introduce* = verb)

Do not slam the door. (*do slam* = verb)

Tense:

The time of the action indicated.

Present: E.g. *She is home.*

Past: E.g. *She was at home yesterday.*

Future: E.g. *She will be at home all next week.*

If the action is thought of as completed (perfected) we say the tense is perfect:

Present-perfect: (or Perfect)

E.g. *She has been on her vacation.*

Past-perfect: E.g. *She had been on her vacation when I saw her last week.*

Future-perfect: E.g. *She will have been on her vacation by the time I see her next.*

Person:

A verb must agree with its subject in person.

A verb is said to be in the first, second, or third person depending upon whether it shows that the action is that of the speaker, the person spoken to, or the person spoken of.

Number:

A verb is said to be either singular or plural if the action or state expressed is that of one person or thing, or that of more than one.

	Singular	Plural
First person	<i>I buy</i>	<i>we buy</i>
Second person	<i>you buy</i>	<i>you buy</i>
Third person	<i>he buys</i>	<i>they buy</i>

Most English verbs have the same form for all persons except for third person singular.

Finite & infinite verbs:**Finite verbs:**

Possess the properties of voice, mood, tense, person, and number.

Infinite verb-forms:

Infinitive, participle, and gerund - are not limited as to number, person, and mood.

Infinitive:

Has properties of both a noun and a verb.

E.g. *To see her is to love her.* (*to see, to love* = infinitives)

The word *to* (originally a preposition) is called the sign of the infinitive. It sometimes is omitted, usually when an auxiliary verb is used.

E.g. *Make him stop.* (*stop* = infinitive)

The infinitive is used in various ways:

As subject: *To win is his main desire.*

As object: *They wanted to linger.*

As adjective: *This is a point to be noted.* (*to be noted* = inf.)

As adverb: *We went yesterday to see him.* (*to see him* = inf.)

Participle:

A "verbal adjective".

The present participle of regular verbs is formed by adding *ing* to the stem or root of the verb.

The past participle of regular verbs has the same form as the past indicative.

The past perfect participle consists of the past participle preceded by the word *having*.

Stem	Present Participle	Past Participle	Past Perfect Participle
<i>look</i>	<i>looking</i>	<i>looked</i>	<i>having looked</i>
<i>measure</i>	<i>measuring</i>	<i>measured</i>	<i>having measured</i>

The participle may be used:

as an adjective to modify a noun:

E.g. *Barking dogs seldom bite.*

The threatening letter alarmed him.

as a verb-form taking an object:

E.g. *Giving him the message, I left at once.*

as a verb-form modified by an adverb:

E.g. *Muttering incoherently, the old man walked away.*

Gerund:

a "verbal noun".

The present tense of a gerund is formed by adding *ing* to the stem and is thus identical in spelling with the present participle.

The gerund may be used in the ways a noun is used:

as subject of a verb:

E.g. *Walking is good exercise.*

as object of a verb:

E.g. *He taught swimming.*

as predicate noun:

E.g. *Seeing is believing. (believing = gerund)*

as a verb-form taking an object:

E.g. *Answering questions was her job.*

The gerund often has a subject, either in the possessive or objective case:

E.g. *Alfred's coming to town was unexpected. (Alfred = subject)*

Can you picture him winning first prize? (him = subject)

The gerund is often used in a phrase:

E.g. *After studying for six hours, he gave up (after studying).*

Auxiliary verbs:

Combine with other verbs and modify the meaning of the verb.

The auxiliary verbs are: *be, can, do, have, may, must, shall, will.*

Forms of the verb *may*:

Present:

I
he
we
you
they } **may**

Past:

I
he
we
you
they } **might**

Forms of the verb *can*:

Present:

I
he
we
you
they } **can**

Past:

I
he
we
you
they } **could**

Potential mood of various auxiliary verbs:

Present:

I
he
we
you
they } **may, can or must work, eat, agree**

Present Perfect:

I
he
we
you
they } **may, can or must** *have worked, eaten, agreed*

Past:

I
he
we
you
they } **might, could, would, or should** *work, eat, agree*

Past perfect:

I
he
we
you
they } **might, could, would, or should** *have worked, eaten, agreed*

Forms of the verb *do*:

Do is used for emphasis, interrogation, or negation.

It is used only in the present and past tenses of the indicative and subjunctive moods, and in the imperative.

Present indicative:

I **do**
he **does**
we **do**
you **do**
they **do** } *work*

Past indicative:

I **did**
he **did**
we **did**
you **did**
they **did** } *work*

Present and past subjunctive:

(If) { I **do**
 he **(does) do**
 we **do**
 you **do**
 they **do** } *work*

Imperative:

Do work

Forms of the verb *have*:

Used as an auxiliary to form perfect tenses.

Present perfect:

I	have	}	<i>worked</i>
he	has		
we	have		
you	have		
they	have		

Past perfect:

I	had	}	<i>worked</i>
he	had		
we	had		
you	had		
they	had		

Semi-auxiliary verbs:

Ought and let are called semi-auxiliaries.

Ought is used only in the present indicative and is followed by the infinitive form of the verb with which it is combined.

E.g. *I ought to work.*

Let is most often used with the infinitive (without *to*) and the objective case of the pronoun.

E.g. *Let him work.*

Use of *shall* and *will*:

To express simple futurity or expectation on the part of the person speaking:

<i>I shall do it</i>	<i>We shall do it</i>
<i>You will do it</i>	<i>You will do it</i>
<i>He will do it</i>	<i>They will do it</i>

To express determination, desire, command, threat, promise, willingness and intention on the part of the person speaking:

<i>I will do it</i>	<i>We will do it</i>
<i>You shall do it</i>	<i>You shall do it</i>
<i>He shall do it</i>	<i>They shall do it</i>

Use of *should* and *would*:

Similar to *shall* and *will*.

Adjective

Used to modify a noun or pronoun to indicate a quality or condition more exactly.

Classes of adjectives:

Descriptive:

indicate a quality or condition.

E.g. *her blue dress*
bright colours
the redecorated apartment
his smiling answer

Limiting:

indicate a number or quantity, or point out limits.

The articles *the*, *a* and *an* are also limiting adjectives.

E.g. *my only ambition*
the ninth inning
ten cents

Proper:

come from proper nouns.

E.g. *American soldiers*
Bolivian tin
a victrola needle
pasteurised milk

Degree of adjectives:

Express a greater or lesser degree of quality by using comparison.

Positive:

the regular form

E.g. *old*, *simple*, *likely*

Comparative:

E.g. *older*, *simpler*, *more likely*, *less likely*

Superlative:

E.g. *oldest*, *simplest*, *most likely*, *least likely*

Words of several syllables and also participles are almost always compared by using *more* and *most*, or *less* and *least*, rather than adding *er* or *est*.

Irregular adjectives:

have a fixed form in comparative and superlative:

Positive	Comparative	Superlative
<i>bad</i>	<i>worse</i>	<i>worst</i>
<i>far</i>	<i>farther (further)</i>	<i>farthest (furthest)</i>
<i>good (well)</i>	<i>better</i>	<i>best</i>
<i>little</i>	<i>less</i>	<i>least</i>
<i>many (much)</i>	<i>more</i>	<i>most</i>

Some adjectives cannot be compared, e.g.

<i>absolute</i>	<i>fatal</i>	<i>primary</i>
<i>basic</i>	<i>final</i>	<i>replete</i>
<i>chief</i>	<i>full</i>	<i>simultaneous</i>
<i>comparative</i>	<i>fundamental</i>	<i>ultimate</i>
<i>complete</i>	<i>harmless</i>	<i>unanimous</i>
<i>contemporary</i>	<i>ideal</i>	<i>unendurable</i>
<i>devoid</i>	<i>meaningless</i>	<i>unique</i>
<i>empty</i>	<i>mortal</i>	<i>universal</i>
<i>entire</i>	<i>obvious</i>	<i>whole</i>
<i>essential</i>	<i>omnipotent</i>	<i>worthless</i>
<i>eternal</i>	<i>perfect</i>	
<i>everlasting</i>	<i>possible</i>	

Several adjectives beginning with **in-** belong in this group, e.g.

inadmissible, inevitable, indestructible, incessant

Numerals:

Numbering adjectives are either cardinals or ordinals.

Cardinals:

indicate number absolutely.

E.g. *one, two, three*

Ordinals:

indicate a certain relative position in a series.

E.g. *first, second, third*

Articles:

There are two articles, the word *a* (or *an*) called the indefinite article, and the word *the* called the definite article. They are limiting adjectives.

The **indefinite** article *a* is used before words beginning with a consonant sound:

E.g. *a year, a man, a unit, a history*

An is used before words beginning with a vowel sound:

E.g. *an umbrella, an hour, an unusual day*

The **definite** article *the* is separated from the noun by the modifying adjective or adjectives if there are any:

E.g. *the sky, the beautiful sky, the beautiful blue sky*

Adverbs

- Adverbs may modify verbs:

E.g. *She smiled contentedly* (*contentedly* = adverb)
The news travelled fast (*fast* = adverb)

- Adverbs may modify adjectives:

E.g. *The applicant was under twenty-one* (*under* = adverb)

- Adverbs may modify other adverbs:

E.g. *I know that all too well* (*too* = adverb)

- Adverbs that modify a whole clause or sentence are called **sentence adverbs**:

E.g. *Unfortunately, no one saw him do it.* (*unfortunately* = adverb)

- Adverbs are used to tell where, how, when, or to what extent something is done:

E.g. *today, simply, here, now, more*

Some meanings of adverbs:

Time:	<i>now, when, then, finally, never, lately</i>
Place:	<i>where, there, here, below, far, downstairs</i>
Manner:	<i>well, ill, how, otherwise</i>
Degree:	<i>more, less, too, completely, much, equally</i>
Cause or Purpose:	<i>why, therefore, wherefore, consequently</i>
Number:	<i>firstly, secondly, thirdly</i>

Affirmative adverb:

yes

Negative adverb:

no, not

Interrogative adverb:

used in asking a direct or indirect question:

E.g. *How are you feeling?* (*how* = adverb)
They cannot tell why he did it. (*why* = adverb)

Relative adverb:

Relates the dependent clause to the independent clause.

E.g. *They cannot tell why he did it.* (*why* = adverb)

Conjunctive adverb:

Uses an adverb as a conjunction as well.

E.g. *I want to see him before the play begins.* (*before* = adverb)

Comparison:

Adverbs are compared similarly to adjectives.

Using *er* or *est*:

Positive	Comparative	Superlative
<i>tight</i>	<i>tighter</i>	<i>tightest</i>
<i>hard</i>	<i>harder</i>	<i>hardest</i>
<i>early</i>	<i>earlier</i>	<i>earliest</i>
<i>close</i>	<i>closer</i>	<i>closest</i>

Using *more* and *most* or *less* and *least*:

Positive	Comparative	Superlative
<i>slowly</i>	<i>more slowly</i>	<i>most slowly</i>
<i>often</i>	<i>less often</i>	<i>least often</i>
<i>eagerly</i>	<i>more eagerly</i>	<i>most eagerly</i>

Irregular forms:

Positive	Comparative	Superlative
<i>badly</i>	<i>worse</i>	<i>worst</i>
<i>far</i>	<i>farther (further)</i>	<i>farthest (furthest)</i>
<i>late</i>	<i>later</i>	<i>latest</i>
<i>little</i>	<i>less</i>	<i>least</i>
<i>much</i>	<i>more</i>	<i>most</i>
<i>near</i>	<i>nearer</i>	<i>nearest (next)</i>
<i>well</i>	<i>better</i>	<i>best</i>

Preposition

A preposition is a connective which combines with a noun or pronoun to form a prepositional phrase. The noun or pronoun is called the object of the preposition. The preposition shows the relation between this object and other words in the sentence.

E.g. *The children scampered down the street, but they were stopped at the corner by their father who was returning from his office.* (*down, at, by, from* = preposition)

In the phrase *at the corner*, *at* shows where they were stopped. The phrase is thus equivalent to an adverb.

Principal English prepositions:

<i>aboard</i>	<i>concerning</i>	<i>regarding</i>
<i>about</i>	<i>considering</i>	<i>respecting</i>
<i>above</i>	<i>down</i>	<i>round</i>
<i>across</i>	<i>during</i>	<i>since</i>
<i>after</i>	<i>for</i>	<i>through</i>
<i>against</i>	<i>from</i>	<i>throughout</i>
<i>along</i>	<i>in</i>	<i>till</i>
<i>amid</i>	<i>inside</i>	<i>to</i>
<i>among</i>	<i>into</i>	<i>toward(s)</i>
<i>around</i>	<i>like</i>	<i>under</i>
<i>at</i>	<i>near</i>	<i>underneath</i>
<i>before</i>	<i>of</i>	<i>until</i>
<i>behind</i>	<i>off</i>	<i>unto</i>
<i>below</i>	<i>on</i>	<i>up</i>
<i>beneath</i>	<i>onto</i>	<i>upon</i>
<i>beside(s)</i>	<i>outside</i>	<i>via</i>
<i>between</i>	<i>over</i>	<i>with</i>
<i>beyond</i>	<i>past</i>	<i>within</i>
<i>by</i>	<i>per</i>	<i>without</i>

Words like *concerning*, *considering* and *regarding* are often used with prepositional force:

E.g. *Considering his extreme youth, he has done well indeed.*
I spoke to him regarding his future plans.

Sometimes a phrase has the force of a preposition: E.g.

<i>according to</i>	<i>because of</i>	<i>in view of</i>
<i>ahead of</i>	<i>contrary to</i>	<i>on account of</i>
<i>apart from</i>	<i>due to</i>	<i>owing to</i>
<i>as far as</i>	<i>in place of</i>	
<i>back of</i>	<i>in spite of</i>	

Conjunction

Conjunctions are used either to connect words, phrases, clauses or sentences, or to show how one sentence is related to another.

There are three principal groups: **coordinating**, **subordinating** and **correlative**:

coordinating:

Joins two elements of equal grammatical value.

E.g. *They had bacon and eggs for breakfast.* (two nouns)

E.g. *The ball sailed over the wall and into the field.* (two prepositional phrases)

E.g. *Our players were ready and the game began.* (two independent clauses)

E.g. *When I went to Esther's wedding, I naturally expected to see the Andersons. And, sure enough, there they were.* (two sentences)

Important coordinating conjunctions:

and, but, or, nor, yet, for (when used as *the reason is that*)

subordinating:

Indicate one element is subordinate to another in a sentence.

E.g. *After the door closed, Ruth heaved a sigh of relief.*
(*after* = conj.)

If I reach Flinders Street before one o'clock, I shall be glad to get it for you. (*if* = conj.)

Albert wants to get his application in before it is too late.
(*before* = conj.)

Subordinating conjunctions may be used to denote such relations as:
reason, time, purpose, condition, result, place, comparison

Important subordinating conjunctions:

<i>after</i>	<i>in order that</i>	<i>what</i>
<i>although</i>	<i>in spite of</i>	<i>whatever</i>
<i>as</i>	<i>in that</i>	<i>when</i>
<i>as if</i>	<i>lest</i>	<i>whence</i>
<i>as long as</i>	<i>notwithstanding</i>	<i>whenever</i>
<i>as often as</i>	<i>now that</i>	<i>where</i>
<i>as soon as</i>	<i>provided that</i>	<i>wherever</i>
<i>as though</i>	<i>since</i>	<i>which</i>
<i>because</i>	<i>so that</i>	<i>whichever</i>
<i>before</i>	<i>so ... as (that)</i>	<i>while</i>
<i>but that</i>	<i>such ... as (that)</i>	<i>whither</i>
<i>even if</i>	<i>than</i>	<i>who</i>
<i>for the purpose of</i>	<i>that</i>	<i>whoever</i>
<i>how</i>	<i>though</i>	<i>why</i>
<i>if</i>	<i>till</i>	<i>with a view to</i>
<i>in case</i>	<i>unless</i>	
<i>inasmuch as</i>	<i>until</i>	

correlative:

Used to connect parallel sentence elements. They are used in pairs or in a series.

E.g. *They replied that they felt neither unusual cold nor unusual warmth.*
(*neither ... nor* = conj.)

E.g. *Either we must make up our minds at once or we must resign ourselves to doing without it.* (*either ... or* = conj.)

Both Frederick and his cousin became ill during the Christmas vacation.
(*both ... and* = conj.)

Important correlative conjunctions:

<i>not ... or</i>	<i>neither ... nor</i>
<i>not ... nor</i>	<i>both ... and</i>
<i>not only ... but also</i>	<i>so ... as</i>
<i>though .. yet</i>	<i>if ... then</i>
<i>whether ... or</i>	<i>as ... as</i>
<i>either ... or</i>	

Conjunctive adverbs:

These words should be regarded as adverbs except when they are used to connect two independent clauses.

<i>accordingly</i>	<i>nevertheless</i>
<i>also</i>	<i>notwithstanding</i>
<i>besides</i>	<i>otherwise</i>
<i>consequently</i>	<i>so</i>
<i>furthermore</i>	<i>still</i>
<i>hence</i>	<i>then</i>
<i>however</i>	<i>therefore</i>
<i>likewise</i>	<i>thus</i>
<i>moreover</i>	

E.g. *We must leave promptly at eight-thirty; otherwise we shall miss the first part of the ceremony.*

The semicolon shows that the word *for* has been omitted:

We must leave promptly at eight-thirty, for otherwise we shall miss the first part of the ceremony.

Interjection

A word or a group of words used to voice an exclamation. It is usually independent of the rest of the sentence; often it serves as an introduction:

E.g. *Oh! He's going to fall!*
Pshaw! I knew I couldn't do it.
What! Is it possible?

Most parts of speech can be used as interjections:

E.g. *Ridiculous! I don't believe it!*
My! What a hot day it is!
Helen! This can't be Helen!

Appendix B: LALR Skeleton for a Parallel Parser

Parser Skeleton:

PAS

Unit @2s;

```
{----- }  
{ A Turbo Pascal 6.0 LR parser generated by LALR 3.0 }  
{----- }
```

```
{ @0s.@3s + @1s.@4s -> @2s.@5s }
```

```
{----- }  
{  
{ The skeleton which generated this code, @1s.@4s, includes the  
{ following:  
{  
{ * Parsing code.  
{ * Parallel parsing of words with multiple parts of speech.  
{ * All possible parses are produced.  
{ * Call to input processing routines with none or one string argument.  
{ * Call to output processing routines with none or one string argument.  
{  
{ Two input processing routines are provided:  
{  
{ * Dictionary search for multiple word types.  
{ * Punctuator check.  
{  
{ If no input processing is specified in the grammar, no code will be  
{ generated for it.  
{  
{ Various output processing routines are provided.  
{  
{ If no output processing is specified in the grammar, no code will be  
{ generated for it.  
{  
{ If no multiple reductions are present, no code will be generated for them.  
{  
{----- }
```

Interface

Uses

GlobDefs, Stacks, Strings, MyScan, Dictnary, Dos;

Procedure StampOutput;

Procedure ParserInit(Var ContextP: ContextPtr);

Function Parse: Integer;

Procedure ProduceOutput;

@118!...

Function WordTypeCheck(ContextP: ContextPtr; TermNo: Integer): Integer;

Function PunctuatorCheck(ContextP: ContextPtr; TermNo: Integer): Integer;

@@ @118?...

Function WordTypeCheck(ContextP: ContextPtr; TermNo: Integer; Arg: String): Integer;
Function PunctuatorCheck(ContextP: ContextPtr; TermNo: Integer; Arg: String): Integer;

@@

@158!...

Function Emit(ContextP: ContextPtr): String;
Function EmitToken(ContextP: ContextPtr): String;
Function EmitTokenNI(ContextP: ContextPtr): String;
Function EmitTokenAndArg(ContextP: ContextPtr): String;
Function EmitArgAndToken(ContextP: ContextPtr): String;
Function EmitArgAndTokenNI(ContextP: ContextPtr): String;

@@

@158?...

Function Emit(ContextP: ContextPtr; Arg: String): String;
Function EmitToken(ContextP: ContextPtr; Arg: String): String;
Function EmitTokenNI(ContextP: ContextPtr; Arg: String): String;
Function EmitTokenAndArg(ContextP: ContextPtr; Arg: String): String;
Function EmitArgAndToken(ContextP: ContextPtr; Arg: String): String;
Function EmitArgAndTokenNI(ContextP: ContextPtr; Arg: String): String;

@@

{ _____ }

Implementation

{ _____ }
{ Generated parser tables }
{ _____ }

Const

N_TOKENS = @17d; { Number of tokens coming from scanner }
N_TERMS = @10d; { Number of terminal symbols in grammar }
N_HEADS = @20d; { Number of heads (nonterminal symbols }
N_PRODS = @54d; { Number of productions (rules) }
N_STATES = @50d; { Number of states in parser }

Type

IntegerPtr = ^Integer;

Const

{ Terminal symbols }

TermSymbol: Array[0..@10-1d] of SymString = (
 @10.1@'%s'@,@,\n @
);

@111?...

{ Descriptors associated with the terminal symbols }

TermDesc: Array[0..@11-1d] of SymString = {
 @11.1@'%s'@,@,\n @
};

@@

{ Non Terminal symbols }

NonTermSymbol: Array[0..**@20-1d**] of SymString = (

@20.1@'%s'@,@,\n @

);

@121?...

{ Descriptors associated with nonterminal symbols }

NonTermDesc: Array[0..**@21-1d**] of SymString = (

@21.1@'%s'@,@,\n @

);

@@

{ Pointers to terminal transition lists, one pointer }
{ for each state. This works with TermTrans[]. }

TermTransStart: Array[0..**@30-1d**] of Integer = (

@30.10@%5d@,@,\n @

);

{ Terminal transitions for each state. This works with TermTransStart[]. }
{ The terminal transition numbers are next state numbers, necessitating }
{ the use of Accessor[], the accessing symbol numbers for each state. }

TermTrans: Array[0..**@31-1d**] of Integer = (

@31.10@%5d@,@,\n @

);

{ Accessing symbols for each state. If the accessing symbol }
{ for a particular state is a terminal symbol it is positive. }
{ If it is a non terminal symbol, it is negative. }

Accessor: Array[0..**@32-1d**] of Integer = (

@32.10@%5d@,@,\n @

);

{ Pointers to non terminal transition lists, one pointer }
{ for each state. This works with NonTermTrans[]. }

NonTermTransStart: Array[0..**@40-1d**] of Integer = (

@40.10@%5d@,@,\n @

);

{ Non terminal transitions for each state. This works with }
{ NonTermTransStart[]. The non terminal transition numbers are }
{ next state numbers, necessitating the use of Accessor[], the }
{ accessing symbol numbers for each state. }

NonTermTrans: Array[0..**@41-1d**] of Integer = (

@41.10@%5d@,@,\n @

);

```

{ Default reduction (production) numbers, one for }
{ each state. A -1 indicates no default reduction. }
{ A zero indicates the goal symbol production. }

```

```

DefaultRedn: Array[0..@50-1d] of Integer = (
  @50.10@%5d@,@,\n @
);

```

@52?...

```

{ Pointers to the multiple reduction lists, one pointer for each }
{ state. This works with RednTermSymb[] and RednNumForTerm[] }

```

```

RednListStart: Array[0..@51-1d] of Integer = (
  @51.10@%5d@,@,\n @
);

```

```

{ Terminal symbol numbers causing a reduction. This list }
{ is indexed by RednListStart[]. The reduction that should }
{ take place is given by RednNumForTerm[] }

```

```

RednTermSymb: Array[0..@52-1d] of Integer = (
  @52.10@%5d@,@,\n @
);

```

```

{ Reduction (production) numbers associated with }
{ the terminal symbol numbers in RednTermSymb[] }

```

```

RednNumForTerm: Array[0..@53-1d] of Integer = (
  @53.10@%5d@,@,\n @
);

```

@@

```

{ Production lengths, right hand side lengths of the productions. }

```

```

ProdLength: Array[0..@54-1d] of Byte = (
  @54.10@%5d@,@,\n @
);

```

```

{ Head symbol numbers, one for each production. }

```

```

HeadSymbNum: Array[0..@55-1d] of Integer = (
  @55.10@%5d@,@,\n @
);

```

```

{-----}
{ Input processing functions }
{-----}

```

@116?Type

@116!...

@118?Type

@@

@116? InFunc = Function(ContextP: ContextPtr; TermNo: Integer): Integer;

@118? InFunc = Function(ContextP: ContextPtr; TermNo: Integer; Arg: String): Integer;

@16?...

Const

```
{ Input processing routine names as specified in the grammar. The }
{ numbers in list InputProcIndex[] are used as pointers into this list. }
```

InputProc: Array[0..@16-1d] of InFunc = (

@16.1@%s@,@,\n @

);

```
{ Input processing routine numbers associated with tokens (coming from }
{ the scanner). A number zero would indicate the first input processing }
{ routine as specified in the InputProc[] list. A number of -1 would }
{ indicate no input processor for this token symbol. }
```

InputProcIndex: Array[0..@17-1d] of Integer = (

@17.10@%5d@,@,\n @

);

@@

@118?...

```
{ Input processing routine arguments for the tokens. These strings may }
{ be either alphabetic or numeric, depending on how they were specified }
{ in the grammar, but they are always stored internally (in LALR 3.0) }
{ as string type. A -1 would indicate no argument for this token symbol. }
```

InputProcArg: Array[0..@18-1d] of String = (

@18.1@'%s'@,@,\n @

);

@@

```
{-----}
{ Output processing functions }
{-----}
```

@156?Type

@156!...

@158?Type

@@

@156? OutFunc = Function(ContextP: ContextPtr): String;

@158? OutFunc = Function(ContextP: ContextPtr; Arg: String): String;

@56?...

Const

```
{ Output processing routine names. The numbers in list }
{ OutputProcIndex[] are used as pointers to these names. }
```

OutputProc: Array[0..@56-1d] of OutFunc = (

@56.1@%s@,@,\n @

);

```
{ Pointers to output processing routines, one for each production. }
{ A number zero would indicate the first output processing routine }
{ as specified in the OutputProc list[]. A number of -1 would }
{ indicate no output processor for this production. }
```

OutputProcIndex: Array[0..@57-1d] of Integer = (

@57.10@%5d@,@,\n @

);

```

@@
@158?...
  { Output processing routine arguments for each production.      }
  { These strings may be either alphabetic or numeric, depending }
  { on how they were specified in the grammar, but they are      }
  { always stored internally (in LALR 3.0) as string type. A -1  }
  { would indicate no argument for this production.              }

```

```

OutputProcArg: Array[0..@58-1d] of String = (
  @58.1@'%s'@,@,\n @
);

```

```

@@
{-----}
{ Parser constants }
{-----}

```

```

Const
  GOAL = 0; { Production number of final goal }

{-----}

```

```

{-----}
{ Parser Utilities }
{-----}

```

```

{ StampOutput - put headings, time and date at top of output file }
Procedure StampOutput;
Const
  Days: Array[0..6] of String[9] =
    ('Sunday','Monday','Tuesday', 'Wednesday','Thursday','Friday', 'Saturday');

```

```

Var
  Hours, Mins, Secs, Hundredths: Word;
  Year, Month, Day, DayOfWeek: Word;

```

```

Function LeadingZero(w: Word): String;
Var
  s : String;
Begin { LeadingZero }
  Str(w:0, s);
  If Length(s) = 1 then
    s := '0' + s;
  LeadingZero := s
End; { LeadingZero }

```

```

Begin { StampOutput }
  GrammarName := '@0s.@3s';
  SkeletonName := '@1s.@4s';
  ParserName := '@2s.@5s';

  GetTime(Hours, Mins, Secs, Hundredths);
  GetDate(Year, Month, Day, DayOfWeek);

  Write(OutFile, 'Parser : ', ParserName:12);
  Writeln(OutFile, ' ');
  Write(OutFile, 'Grammar : ', GrammarName:12);
  Writeln(OutFile, ' } Created by LALR 3.0');
  Write(OutFile, 'Skeleton: ', SkeletonName:12);
  Writeln(OutFile, ' ');

  Writeln(OutFile, 'Time : ', LeadingZero(Hours),
    ':', LeadingZero(Mins), ':', LeadingZero(Secs));
  Writeln(OutFile, 'Date : ', Days[DayOfWeek], ' ',
    Day:0, '/', Month:0, '/', Year:0);
  Writeln(OutFile);
  Writeln(OutFile, '-----',
    + '-----');
  Writeln(OutFile)
End; { StampOutput }

{-----}

{ ParserInit -- initialise the parser, return pointer to the first context }
Procedure ParserInit(Var ContextP: ContextPtr);
Begin { ParserInit }
  InitContext(ContextP) { Initialise the context stack }
End; { ParserInit }

{-----}

{ DoInputProc -- call an input processing routine }
Procedure DoInputProc(ContextP: ContextPtr);
Begin { DoInputProc }
  With ContextP^ do
    If InputProcIndex[Terminal] >= 0 then
      Terminal := InputProc[InputProcIndex[Terminal]]    { Call processor }
@116?      (ContextP, Terminal)
@118?      (ContextP, Terminal, InputProcArg[Terminal])
End; { DoInputProc }

{-----}

{ PrintLine -- print source line }
Procedure PrintLine(ContextP: ContextPtr);
Var
  s: String;
  p: CharPtr;

```

```

Begin { PrintLine }
  With ContextP^ do
    Begin
      If LineStart^ <> EOF then           { If not at end of file }
        Begin
          s := '';
          p := LineStart;

          While (p^ <> CR) and (p^ <> EOF) do { Collect string up to end of line }
            Begin
              s := s + p^;
              Longint (p) := Longint (p) + 1
            End;

            Writeln(OutFile, LineNum:6, ' ', s)      { Print the line with its line number }
          End
        End { With ContextP^ do }
      End; { PrintLine }

```

```
{ ----- }
```

```
{ PrintMsg -- print error message }
```

```
Procedure PrintMsg(ContextP: ContextPtr; Msg, Symbol: String);
```

```
Var
```

```
  i, nl, Col, LastLineNum: Integer;
```

```
  p: CharPtr;
```

```
  c: Char;
```

```
Begin { PrintMsg }
```

```
  With ContextP^ do
```

```
    Begin
```

```
      nl := 0;
```

```
      Col := 0;
```

```
      LastLineNum := -1;
```

```
      Writeln(OutFile);           { Print the line }
```

```
      Writeln(OutFile);
```

```
      PrintLine(ContextP);
```

```
      LastLineNum := LineNum;     { Save line number }
```

```
      { Find column number of token causing error message }
```

```
      For Longint (p) := Longint (LineStart) to Longint (Token) - 1 do
```

```
        If p^ = TAB then
```

```
          Col := Col + 8 - (Col div 8)
```

```
        else
```

```
          Col := Col + 1;
```

```
      Write(OutFile, ' ');
```

```
      For i := 1 to Col do
```

```
        { Write out correct number of dashes }
```

```
        Write(OutFile, '-');
```

```
      Write(OutFile, '^ ' + Msg + Symbol); { Print pointer to erroneous symbol }
```

```
      { and the error message. }
```

```
      Writeln(OutFile)
```

```
    End { With ContextP^ do }
```

```
  End; { PrintMsg }
```

```
{ ----- }
```

```

Procedure Crash(ContextP: ContextPtr; Msg, Symbol: String);
Begin { Crash }
    PrintMsg(ContextP, Msg, Symbol);
    Close(OutFile);
    Halt(1)
End; { Crash }

```

```

{-----}

```

```

{ GetToken -- get token from scanner }

```

```

Procedure GetToken(ContextP: ContextPtr);

```

```

Begin { GetToken }

```

```

    With ContextP^ do

```

```

        Begin

```

```

            DictData.NumTypes := 0;           { Initialise dictionary return values }

```

```

            PartsOfSpeech := [];

```

```

        While true do

```

```

            Begin

```

```

                Scan(ContextP);           { Call the lexical scanner }

```

```

                If Terminal <= 0 then

```

```

                    Begin

```

```

                        If Terminal = 0 then           { If terminal unrecognised? }

```

```

                            Begin

```

```

                                If OutputOn then { If output turned on? }

```

```

                                    Begin

```

```

                                        NumErrors := NumErrors + 1;

```

```

                                        PrintMsg(ContextP, 'Ignoring token ',

```

```

                                            StrPtrsToStr(Token, Input))

```

```

                                    End

```

```

                                End { If Terminal = 0 then }

```

```

                            else

```

```

                                Begin

```

```

                                    If OutputOn then           { If output turned on? }

```

```

                                        Begin

```

```

                                            NumErrors := NumErrors + 1;

```

```

                                            PrintMsg(ContextP, 'Invalid token ',

```

```

                                                StrPtrsToStr(Token, Input))

```

```

                                        End;

```

```

                                    Terminal := -Terminal; { Make terminal positive and continue }

```

```

                                    DoInputProc(ContextP); { Any input processing for this terminal }

```

```

                                End

```

```

                            End { If Terminal <= 0 then }

```

```

                    else

```

```

                        Begin

```

```

                            DoInputProc(ContextP);           { Any input processing for this terminal }

```

```

                        Exit

```

```

                    End

```

```

                End { While true do }

```

```

            End { With ContextP^ do }

```

```

End; { GetToken }

```

```

{-----}

```

```

{ GetTokens - get token, check dictionary, spawn new contexts if needed }
Procedure GetTokens(ContextP: ContextPtr);
Var
  NewContextP: ContextPtr;
  TokenNum: Integer;
Begin { GetTokens }
  With ContextP^ do
    Begin
      GetToken(ContextP);
      With DictData do
        Begin
          If NumTypes > 0 then
            Begin
              For TokenNum := 1 to NumTypes do
                If TokenNum > 1 then
                  Begin
                    CreateContext(ContextP, NewContextP);
                    NewContextP^.Terminal :=
                      Ord(TypeData[TokenNum].WordType)
                  End
                else
                  Terminal := Ord(TypeData[TokenNum].WordType)
                End { If NumTypes > 0 then }
              End { With DictData do }
            End { With ContextP^ do }
          End; { GetTokens }

```

{ _____ }

{ _____ }
 { LR Parsing Engine }
 { _____ }

```

{ GetNextShiftStates - return possible next states for a state/terminal pair }
Procedure GetNextShiftStates(ContextP: ContextPtr);
Var
  TermTransNo: Integer;
Begin { GetNextShiftStates }
  With ContextP^ do
    Begin
      NumNextShiftStates := 0;
      { For all terminal transitions in this state }
      For TermTransNo := TermTransStart[State] to TermTransStart[State+1] - 1 do
        { If accessor of goto state matches incoming terminal symbol? }
        If Accessor[TermTrans[TermTransNo]] = Terminal then
          Begin
            NumNextShiftStates := NumNextShiftStates + 1;
            NextShiftStates[NumNextShiftStates] := TermTrans[TermTransNo]
          End
        End { With ContextP^ do }
      End; { GetNextShiftStates }

```

{ _____ }

```

@56?...
Procedure DoOutputProc(ContextP: ContextPtr);
Var
  OPProcMsg: String;
Begin { DoOutputProc }
  With ContextP^ do
    Begin
      { If any action specified for this production? }
      If OutputProcIndex[Prodn] >= 0 then
        Begin
@158!      { Put procedure and token on output stack }
@158?      { Put procedure, argument and token on output stack }
            OutputStackP := OutputStackP + 1;
            OutputStack[OutputStackP].ProcNo := OutputProcIndex[Prodn];
@158?      OutputStack[OutputStackP].ArgNo := Prodn;
            OutputStack[OutputStackP].TokenBeg := TokenBeg;
            OutputStack[OutputStackP].TokenEnd := TokenEnd;

            { Call the action procedure }
            OPProcMsg := OutputProc[OutputProcIndex[Prodn]]
@156?      (ContextP);
@158?      (ContextP, OutputProcArg[Prodn]);

            { If OP error message then print it }
            If OPProcMsg <> " then
              Begin
                NumErrors := NumErrors + 1;
                PrintMsg(ContextP, OPProcMsg, StrPtrsToStr(TokenBeg, TokenEnd))
              End
            End
          End { With ContextP^ do }
        End; { DoOutputProc }

```

```

@@
{ ----- }

```

```

{ DoShift - main code for the shift routine }
Procedure DoShift(ContextP: ContextPtr; NextState: Integer);
Var
  RednP: Integer;
  OPProcMsg: String;
Begin { DoShift }
  With ContextP^ do
    Begin
@56?...      If OutputOn then                                { If output turned on? }
              Begin
                { For all reductions on reduction stack }
                RednP := 0;
                While RednP < RednStackP do
                  Begin
                    RednP := RednP + 1;
                    Prodn := RednStack[RednP].Prodn;      { Get the production }
@156?      DoOutputProc(ContextP); { Call the action routine }
@158?      DoOutputProc(ContextP); { Call the action routine }
                  End { While RednP < RednStackP do }
              End; { If OutputOn then }
    End;

```

```

TokenBeg := Token;           { Point at beg & end of }
TokenEnd := Input;          { terminal sym accepted. }

@@
RednStackP := 0;             { Reset redn stack ptr }
If ParseStackP = MaxStack - 1 then { Parse stack overflow? Crash }
    Crash(ContextP, 'Tried to increment past the end of the parse stack.', '');
ParseStackP := ParseStackP + 1; { Put cur state on parse stack }
ParseStack[ParseStackP] := State;
State := NextState           { Define next state }
End { With ContextP^ do }
End; { DoShift }

```

```
{ ----- }
```

```

{ Shift – perform a shift action }
Function Shift(ContextP: ContextPtr): Boolean;
Var
    NextState, NextStateNum: Integer;
Begin { Shift }
    With ContextP^ do
        Begin
            GetNextShiftStates(ContextP);
            { Find a possible next state for this state/terminal pair }
            For NextStateNum := 1 to NumNextShiftStates do
                Begin
                    NextState := NextShiftStates[NextStateNum];
                    If Accessor[NextState] = Terminal then
                        Begin
                            DoShift(ContextP, NextState);
                            Shift := true;           { Return true for shift action }
                            Exit
                        End
                    End;
                Shift := false;           { Return failure for shift action }
            End { With ContextP^ do }
        End; { Shift }

```

```
{ ----- }
```

```

{ GetNextRednState - return possible next redn state for state/nonterminal pair }
Procedure GetNextRednState(ContextP: ContextPtr);
Var
    NonTermTransNo: Integer;
Begin { GetNextRednState }
    With ContextP^ do
        Begin
            NumNextRednStates := 0;
            { For all nonterminal transitions at the origin of this production ... }
            { A match for the nonterminal transition is always present }
            NonTermTransNo := NonTermTransStart[State];
            While true do
                Begin
                    { If the accessor of the goto state matches the head symbol? }

```

```

    If Accessor[NonTermTrans[NonTermTransNo]] = Head then
      Begin
        NumNextRednStates := NumNextRednStates + 1;
        NextRednStates[NumNextRednStates] :=
          NonTermTrans[NonTermTransNo];
      Exit
    End;
    NonTermTransNo := NonTermTransNo + 1
  End { While true do }
End { With ContextP^ do }
End; { GetNextRednState }

{ ----- }

{ DoReduce -- main code for Reduce routine }
Function DoReduce(ContextP: ContextPtr): Boolean;
Var
  NumSyms, NextState, NextStateNum: Integer;
Begin { DoReduce }
  With ContextP^ do
    Begin
      ParseStackP := ParseStackP + 1;
      If ParseStackP = MaxStack then           { If parse stack overflow, crash }
        Crash(ContextP, 'Tried to increment past end of parse stack.', '');

      { Save old state on parse stack before replacing with current one }
      RednStackP := RednStackP + 1;
      RednStack[RednStackP].State := ParseStack[ParseStackP];

      RednStack[RednStackP].Prodn := Prodn; { Save this production }
      ParseStack[ParseStackP] := State; { Put current state on parse stack }

      { Get head symbol for this production. Note that the }
      { accessors for nonterminal transitions are negative }
      Head := -HeadSymbNum[Prodn];

      { Reduce parse stack by length of production and get origin state }
      ParseStackP := ParseStackP - ProdLength[Prodn];
      State := ParseStack[ParseStackP];

      { Find a possible next state for this state/nonterminal pair }
      GetNextRednState(ContextP);
      If NumNextRednStates > 0 then
        Begin
          State := NextRednStates[1]; { Use the first state found }
          DoReduce := true; { Return true for shift action }
        Exit
        End;
        DoReduce := false { Return failure for shift action }
      End { With ContextP^ do }
    End; { DoReduce }

  { ----- }

```

```

@52?...
Procedure GctMultipleRedns(ContextP: ContextPtr);
Var
  TermSymNo: Integer;
Begin { GetMultipleRedns }
  With ContextP^ do
    Begin
      NumMultipleRedns := 0;
      { For all multiple reductions in this state }
      For TermSymNo := RednListStart[State] to RednListStart[State+1] - 1 do
        Begin
          { If there is a reduction specified for the incoming terminal? }
          If RednTermSymb[TermSymNo] = Terminal then
            Begin
              NumMultipleRedns := NumMultipleRedns + 1;
              { Define the production }
              MultipleRedns[NumMultipleRedns] := RednNumForTerm[TermSymNo]
            End
          End { For TermSymNo := RednListStart[State] to RednListStart[State+1] - 1 do }
        End { With ContextP^ do }
      End; { GetMultipleRedns }

```

{ ----- }

@@

```

{ Reduce -- perform a reduce action }
Function Reduce(ContextP: ContextPtr): Boolean;
Begin { Reduce }
  With ContextP^ do
    Begin

```

@52?...

```

  { Get all multiple reductions for this state/terminal pair }
  GetMultipleRedns(ContextP);
  If NumMultipleRedns > 0 then { If multiple redns, use the first }
    Begin
      Prodn := MultipleRedns[1]; { Define the production }
      If DoReduce(ContextP) then { And go do the reduction }
        Begin
          Reduce := true;
          Exit
        End
      End;

```

@@

```

  { If no default reduction for this state return failure to reduce }
  Prodn := DefaultRedn[State];
  If Prodn <= 0 then
    Begin
      Reduce := false;
      Exit
    End;
  Reduce := DoReduce(ContextP)
End { With ContextP^ do }
End; { Reduce }

```

{ ----- }

```

Procedure DoParse(ContextP: ContextPtr);
Var
    NextAction: ActionType;
    WordTypeNum: Integer;
Begin { DoParse }
    With ContextP^ do
        Begin
            Finished := false;
            Parsing := true;
            NextAction := ShiftAction;
            While not Finished do
                Begin
                    Case NextAction of
                        NoAction:
                            Begin
                                Finished := true;
                                Parsing := false
                            End;
                        ShiftAction:
                            Begin
                                If Shift(ContextP) then
                                    NextAction := GetNextTokens
                                else
                                    NextAction := ReduceAction
                                End;
                        ReduceAction:
                            Begin
                                If Reduce(ContextP) then
                                    NextAction := ShiftAction
                                else
                                    NextAction := NoAction
                                End;
                        GetNextTokens:
                            Begin
                                GetTokens(ContextP);
                                NextAction := ShiftAction
                            End
                    End; { Case NextAction of }
                End; { While not Finished do }

                If Prodn = GOAL then { Is this the goal production }
                    Begin
                        With ContextHead do
                            NumSuccessfulParses := NumSuccessfulParses + 1;
                        Exit
                    End
                else
                    Begin
                        DeleteContext(ContextP);
                    Exit
                End;
            End { With ContextP^ do }
        End; { DoParse }

{-----}

```

```

{ Parse -- find all possible LR parses }
Function Parse: Integer;
Var
  ContextP: ContextPtr;
  AllDone: Boolean;
Begin { Parse }
  With ContextHead do
    Begin
      ParserInit(ContextP); { Initialise the parser }
      GetTokens(ContextP); { Call scanner, get first terminal }
      Repeat
        AllDone := true; { Tell parser it is finished }

        ContextP := ContextStackTop; { Step through the contexts }
        While ContextP <> nil do
          Begin
            DoParse(ContextP); { Perform the parse }
            If not ContextP^.Finished then { If any parse is not finished }
              AllDone := false; { tell the parser. }
            ContextP := ContextP^.NextContext
          End

        Until AllDone; { Parsing is finished }

      Parse := NumSuccessfulParses
    End { With ContextHead do }
  End; { Parse }

{-----}

```

```
{-----}  
{ Input processing functions }  
{-----}
```

```
{ WordTypeCheck -- check types of a word in dictionary }
```

```
@118!...
```

```
Function WordTypeCheck(ContextP: ContextPtr; TermNo: Integer): Integer;
```

```
@@
```

```
@118?...
```

```
Function WordTypeCheck(ContextP: ContextPtr; TermNo: Integer; Arg: String): Integer;
```

```
@@
```

```
Var
```

```
  i: Integer;
```

```
  s: SymString;
```

```
Begin { WordTypeCheck }
```

```
  With ContextP^ do
```

```
    Begin
```

```
      s := StrPtrsToStr(Token, Input); { Put token into string }
```

```
      If RetrieveWord(s, PartsOfSpeech, DictData) then
```

```
        WordTypeCheck := TermNo { Return the original terminal number }
```

```
      else
```

```
        WordTypeCheck := -TermNo { Else return error code for this terminal }
```

```
      End { With ContextP^ do }
```

```
End; { WordTypeCheck }
```

```
{-----}
```

```
{ PunctuatorCheck -- check a punctuation symbol }
```

```
@118!...
```

```
Function PunctuatorCheck(ContextP: ContextPtr; TermNo: Integer): Integer;
```

```
@@
```

```
@118?...
```

```
Function PunctuatorCheck(ContextP: ContextPtr; TermNo: Integer; Arg: String): Integer;
```

```
@@
```

```
Begin { PunctuatorCheck }
```

```
  With ContextP^ do
```

```
    Begin
```

```
      { This routine is a dummy at present }
```

```
      PunctuatorCheck := TermNo { Return original number passed as argument }
```

```
    End { With ContextP^ do }
```

```
End; { PunctuatorCheck }
```

```
{-----}
```

```

{----- }
{ Output processing functions }
{----- }

```

{ Emit -- write out a string argument }

@158!...

Function Emit(ContextP: ContextPtr): String;

@@

@158?...

Function Emit(ContextP: ContextPtr; Arg: String): String;

@@

Var

EscPos: Byte;

s: String;

Begin { Emit }

With ContextP^ do

Begin

If Parsing then

Begin

Emit := " { No error }

End

else

Begin

@158?...

If Arg <> '-1' then { If there is an argument? }

Begin

Repeat

EscPos := Pos('\n', Arg); { Search for newline escape sequence }

If EscPos > 0 then

Begin

s := Copy(Arg, 1, EscPos - 1);

Delete(Arg, 1, EscPos + 1);

Writeln(OutFile, s); { Writeln parts of string up to newline }

End

Until EscPos = 0;

Write(OutFile, Arg) { Write remainder of string }

End;

@@

Emit := " { No error }

End { If not Parsing }

End { With ContextP^ do }

End; { Emit }

```

{----- }

```

```

{ EmitToken -- write out the current token }
@158!...
Function EmitToken(ContextP: ContextPtr): String;
@@
@158?...
Function EmitToken(ContextP: ContextPtr; Arg: String): String;
@@
Begin { EmitToken }
  With ContextP^ do
    Begin
      If Parsing then
        Begin
          EmitToken := " { No error }
        End
      else
        Begin
          Write(OutFile, StrPtrsToStr(TokenBeg, TokenEnd)); { Print token }
          EmitToken := " { No error }
        End { If not Parsing then }
      End { With ContextP^ do }
    End; { EmitToken }

```

{ ----- }

```

{ EmitTokenNI -- write out the current token & newline }
@158!...
Function EmitTokenNI(ContextP: ContextPtr): String;
@@
@158?...
Function EmitTokenNI(ContextP: ContextPtr; Arg: String): String;
@@
Begin { EmitTokenNI }
  With ContextP^ do
    Begin
      If Parsing then
        Begin
          EmitTokenNI := " { No error }
        End
      else
        Begin
          Writeln(OutFile, StrPtrsToStr(TokenBeg, TokenEnd)); { Print token }
          EmitTokenNI := " { No error }
        End { If not Parsing then }
      End { With ContextP^ do }
    End; { EmitTokenNI }

```

{ ----- }

```

{ EmitTokenAndArg -- write out a token followed by the string argument }
@158!...
Function EmitTokenAndArg(ContextP: ContextPtr): String;
@@
@158?...
Function EmitTokenAndArg(ContextP: ContextPtr; Arg: String): String;
@@
Begin { EmitTokenAndArg }
  With ContextP^ do
    Begin
      If Parsing then
        Begin
@158!...
          EmitTokenAndArg := Emit(ContextP)           { No error if "" }
@@
@158?...
          EmitTokenAndArg := Emit(ContextP, Arg)       { No error if "" }
@@
        End
      else
        Begin
@158!...
          Write(OutFile, StrPtrsToStr(TokenBeg, TokenEnd)); { Print token }
@@
@158?...
          EmitTokenAndArg := Emit(ContextP)           { No error if "" }
@@
@158?...
          EmitTokenAndArg := Emit(ContextP, Arg)       { No error if "" }
@@
        End { If not Parsing then }
      End { With ContextP^ do }
    End; { EmitTokenAndArg }

{ ----- }

```

```

{ EmitArgAndToken -- write out the string argument followed by current token }
@158!...
Function EmitArgAndToken(ContextP: ContextPtr): String;
@@
@158?...
Function EmitArgAndToken(ContextP: ContextPtr; Arg: String): String;
@@
Begin { EmitArgAndToken }
  With ContextP^ do
    Begin
      If Parsing then
        Begin
@158!...
          EmitArgAndToken := Emit(ContextP);           { No error if "" }
@@
@158?...
          EmitArgAndToken := Emit(ContextP, Arg);       { No error if "" }
@@
        End
      else
        Begin
@158!...
          EmitArgAndToken := Emit(ContextP);           { No error if "" }
@@

```

```

@158?...      EmitArgAndToken := Emit(ContextP, Arg);           { No error if " }
@@
      Write(OutFile, StrPtrsToStr(TokenBeg, TokenEnd))      { Print token }
      End { If not Parsing then }
      End { With ContextP^ do }
End; { EmitArgAndToken }

{ ----- }

{ EmitArgAndTokenNI -- write out the string argument, current token & newline }
@158!...
Function EmitArgAndTokenNI(ContextP: ContextPtr): String;
@@
@158?...
Function EmitArgAndTokenNI(ContextP: ContextPtr; Arg: String): String;
@@
Begin { EmitArgAndTokenNI }
  With ContextP^ do
    Begin
      If Parsing then
        Begin
@158!...
          EmitArgAndTokenNI := Emit(ContextP); { No error if " }
@@
@158?...
          EmitArgAndTokenNI := Emit(ContextP, Arg); { No error if " }
@@
        End
      else
        Begin
@158!...
          EmitArgAndTokenNI := Emit(ContextP); { No error if " }
@@
@158?...
          EmitArgAndTokenNI := Emit(ContextP, Arg); { No error if " }
@@
          Writeln(OutFile, StrPtrsToStr(TokenBeg, TokenEnd)) { Print token }
        End { If not Parsing then }
      End { With ContextP^ do }
End; { EmitArgAndTokenNI }

{ ----- }

```

```

{----- }
{ Output Production Routines }
{----- }

```

```

{ DoContextOutput - generate output from a context's output stack }

```

```

Procedure DoContextOutput(ContextP: ContextPtr);

```

```

Var

```

```

    p: Integer;

```

```

    OPProcMsg: String;

```

```

Begin { DoContextOutput }

```

```

    With ContextP^ do

```

```

        Begin

```

```

            p := 0;

```

```

            While p < OutputStackP do

```

```

                Begin

```

```

                    p := p + 1;

```

```

                    TokenBeg := OutputStack[p].TokenBeg;

```

```

                    TokenEnd := OutputStack[p].TokenEnd;

```

```

                    OPProcMsg := OutputProc[OutputStack[p].ProcNo] { Call action routine }

```

```

@156?                    (ContextP);

```

```

@158?                    (ContextP, OutputProcArg[OutputStack[p].ArgNo]);

```

```

                If OPProcMsg <> '' then { If OP error msg then print it }

```

```

                    Begin

```

```

                        NumErrors := NumErrors + 1;

```

```

                        PrintMsg(ContextP, OPProcMsg, StrPtrsToStr(TokenBeg, TokenEnd))

```

```

                    End

```

```

                End

```

```

            End { With ContextP^ do }

```

```

End; { DoContextOutput }

```

```

{----- }

```

```

{ ProduceOutput - generate output from all successful parses }

```

```

Procedure ProduceOutput;

```

```

Var

```

```

    ContextP: ContextPtr;

```

```

Begin { ProduceOutput }

```

```

    With ContextHead do

```

```

        Begin

```

```

            ContextP := ContextStackTop;

```

```

            While ContextP <> nil do

```

```

                Begin

```

```

                    With ContextP^ do

```

```

                        Begin

```

```

                            Writeln(OutFile, 'Context: ', ContextId:4);

```

```

                            Writeln(OutFile, '-----');

```

```

                            DoContextOutput(ContextP);

```

```

                            Writeln(OutFile)

```

```

                        End; { With ContextP^ do }

```

```

                    ContextP := ContextP^.NextContext

```

```

                End { While ContextP <> nil }

```

```

            End { With ContextHead do }

```

```

End; { ProduceOutput }

```

```

{----- }

```

```

End. { Unit @2s }

```

Appendix C: LALR Generated Parallel Parser for an English Language Subset

The parser described in this appendix was produced by LALR using as input the following very simple English subset grammar, designed to test the operation of the parallel parser:

A Very Simple English Subset grammar

/* Terminals */

/* The scanner returns only tokens of type <Error>, <Wordstr>, <Number>, */
/* <Punctuator>, <EndOfSentence> and <EndOfFile>. Most tokens will be of */
/* type <WordStr>. These are checked by the input routine WordTypeCheck */
/* to see if they are in the dictionary, and if so, whether they are of */
/* the terminal type specified in the current production. If so, the */
/* token for that type will be substituted for <WordStr>, otherwise */
/* <Error> will be returned. <Punctuators> are checked to see whether */
/* they represent an <SApostrophe> or an <ApostropheS>. */

<Error>
<WordStr> => WordTypeCheck
<Number>
<Punctuator> => PunctuatorCheck
<EndOfSentence>
<EndOfFile>

<Adjective>
<Adverb>
<ApostropheS>
<Binder>
<Complementizer>
<CompTo>
<Conjunction>
<Determiner>
<Interjection>
<Noun>
<Particle>
<Prefix>
<PrepFor>
<Preposition>
<Pronoun>
<Proper>
<Relative>
<SApostrophe>
<Send>
<Verb>
<WordElement>

/* Text production */

Text

-> Sentences <EndOfFile>

Sentences

-> S

-> S Sentences

/* Sentence Network */

S

-> NP VP <EndOfSentence> => Emit "Found a sentence\n\n"

NP

-> Determiner NP2 => Emit "Found a NP\n"

-> NP2 => Emit "Found a NP\n"

NP2

-> Noun => Emit "Found a NP2\n"

-> Pronoun => Emit "Found a NP2\n"

-> ProperNoun => Emit "Found a NP2\n"

-> AdjectiveList NP2 => Emit "Found a NP2\n"

-> NP2 PP => Emit "Found a NP2\n"

PP

-> Preposition NP => Emit "Found a PP\n"

VP

-> Verb Adverb NP => Emit "Found a VP\n"

-> Verb => Emit "Found a VP\n"

-> Verb Adverb => Emit "Found a VP\n"

-> Verb NP => Emit "Found a VP\n"

-> VP NP => Emit "Found a VP\n"

AdjectiveList

-> Adjective => Emit "Found an adjective list\n"

-> Adjective AdjectiveList => Emit "Found an adjective list\n"

Noun

-> <Noun> => EmitTokenAndArg " <Noun>\n"

Verb

-> <Verb> => EmitTokenAndArg " <Verb>\n"

Preposition

-> <Preposition> => EmitTokenAndArg " <Preposition>\n"

Adjective

-> <Adjective> => EmitTokenAndArg " <Adjective>\n"

Pronoun

-> <Pronoun> => EmitTokenAndArg " <Pronoun>\n"

ProperNoun

-> <Proper> => EmitTokenAndArg " <Proper>\n"

Determiner

-> <Determiner> => EmitTokenAndArg " <Determiner>\n"

Source Code of an LALR Parallel Parser for an English Language Subset

Unit MYPARSER;

```
{----- }
{ A Turbo Pascal 6.0 LR parser generated by LALR 3.0 }
{----- }

{ SIMPSENT.GRM + MYPARSER.SKL -> MYPARSER.PAS }

{----- }
{ }
{ The skeleton which generated this code, MYPARSER.SKL, includes the }
{ following: }
{ }
{ * Parsing code. }
{ * Parallel parsing of words with multiple parts of speech. }
{ * All possible parses are produced. }
{ * Call to input processing routines with none or one string argument. }
{ * Call to output processing routines with none or one string argument. }
{ }
{ Two input processing routines are provided: }
{ }
{ * Dictionary search for multiple word types. }
{ * Punctuator check. }
{ }
{ If no input processing is specified in the grammar, no code will be }
{ generated for it. }
{ }
{ Various output processing routines are provided. }
{ }
{ If no output processing is specified in the grammar, no code will be }
{ generated for it. }
{ }
{ If no multiple reductions are present, no code will be generated for them. }
{ }
{----- }
```

Interface

Uses

GlobDefs, Stacks, Strings, MyScan, Dictnary, Dos;

Procedure StampOutput;

Procedure ParserInit(Var ContextP: ContextPtr);

Function Parse: Integer;

Procedure ProduceOutput;

Function WordTypeCheck(ContextP: ContextPtr; TermNo: Integer): Integer;

Function PunctuatorCheck(ContextP: ContextPtr; TermNo: Integer): Integer;

```

Function Emit(ContextP: ContextPtr; Arg: String): String;
Function EmitToken(ContextP: ContextPtr; Arg: String): String;
Function EmitTokenNI(ContextP: ContextPtr; Arg: String): String;
Function EmitTokenAndArg(ContextP: ContextPtr; Arg: String): String;
Function EmitArgAndToken(ContextP: ContextPtr; Arg: String): String;
Function EmitArgAndTokenNI(ContextP: ContextPtr; Arg: String): String;

```

```
{-----}
```

Implementation

```

{----- }
{ Generated parser tables }
{----- }

```

Const

```

N_TOKENS = 27;   { Number of tokens coming from scanner }
N_TERMS = 28;   { Number of terminal symbols in grammar }
N_HEADS = 15;   { Number of heads (nonterminal symbols) }
N_PRODS = 26;   { Number of productions (rules) }
N_STATES = 33;  { Number of states in parser }

```

Type

```
IntegerPtr = ^Integer;
```

Const

```
{ Terminal symbols }
```

```
TermSymbol: Array[0..27] of SymString = (
```

```

'<Error>',
'<WordStr>',
'<Number>',
'<Punctuator>',
'<EndOfSentence>',
'<EndOfFile>',
'<Adjective>',
'<Adverb>',
'<ApostropheS>',
'<Binder>',
'<Complementizer>',
'<CompTo>',
'<Conjunction>',
'<Determiner>',
'<Interjection>',
'<Noun>',
'<Particle>',
'<Prefix>',
'<PrepFor>',
'<Preposition>',
'<Pronoun>',
'<Proper>',
'<Relative>',
'<SApostrophe>',
'<Send>',
'<Verb>',
'<WordElement>',
'Adverb'

```

```
);
```

{ Non Terminal symbols }

NonTermSymbol: Array[0..14] of SymString = (

'Text',
'Sentences',
'S',
'NP',
'NP2',
'PP',
'VP',
'AdjectiveList',
'Noun',
'Verb',
'Preposition',
'Adjective',
'Pronoun',
'ProperNoun',
'Determiner'

);

{ Pointers to terminal transition lists, one pointer }
{ for each state. This works with TermTrans[] } }

TermTransStart: Array[0..33] of Integer = (

0, 5, 5, 5, 5, 5, 5, 6, 11, 12,
16, 17, 17, 17, 17, 21, 22, 22, 22, 22,
28, 34, 35, 35, 35, 40, 41, 41, 41, 41,
46, 46, 46, 46

);

{ Terminal transitions for each state. This works with TermTransStart[] }
{ The terminal transition numbers are next state numbers, necessitating }
{ the use of Accessor[], the accessing symbol numbers for each state. } }

TermTrans: Array[0..45] of Integer = (

1, 2, 3, 4, 5, 16, 1, 2, 3, 4,
5, 18, 2, 3, 4, 5, 22, 2, 3, 4,
5, 5, 27, 1, 2, 3, 4, 5, 29, 1,
2, 3, 4, 5, 22, 1, 2, 3, 4, 5,
22, 1, 2, 3, 4, 5

);

{ Accessing symbols for each state. If the accessing symbol }
{ for a particular state is a terminal symbol it is positive. }
{ If it is a non terminal symbol, it is negative. } }

Accessor: Array[0..32] of Integer = (

-1, 13, 15, 20, 21, 6, -1, -2, -3, -14,
-4, -8, -12, -13, -7, -11, 5, -1, 25, -6,
-9, -4, 19, -5, -10, -4, -7, 4, -3, 27,
-3, -3, -3

);

```

{ Pointers to non terminal transition lists, one pointer    }
{ for each state. This works with NonTermTrans[].      }

```

```

NonTermTransStart: Array[0..33] of Integer = (
  0,  10,  10,  10,  10,  10,  10,  10,  20,  22,
  28,  30,  30,  30,  30,  36,  38,  38,  38,  38,
  46,  54,  56,  56,  56,  64,  66,  66,  66,  66,
  74,  74,  74,  74
);

```

```

{ Non terminal transitions for each state. This works with    }
{ NonTermTransStart[]. The non terminal transition numbers are }
{ next state numbers, necessitating the use of Accessor[], the }
{ accessing symbol numbers for each state.                    }

```

```

NonTermTrans: Array[0..73] of Integer = (
  6,  7,  8,  9,  10,  11,  12,  13,  14,  15,
  17,  7,  8,  9,  10,  11,  12,  13,  14,  15,
  19,  20,  21,  11,  12,  13,  14,  15,  23,  24,
  25,  11,  12,  13,  14,  15,  26,  15,  28,  9,
  10,  11,  12,  13,  14,  15,  30,  9,  10,  11,
  12,  13,  14,  15,  23,  24,  31,  9,  10,  11,
  12,  13,  14,  15,  23,  24,  32,  9,  10,  11,
  12,  13,  14,  15
);

```

```

{ Default reduction (production) numbers, one for    }
{ each state. A -1 indicates no default reduction.    }
{ A zero indicates the goal symbol production.        }

```

```

DefaultRedn: Array[0..32] of Integer = (
  -1,  25,  19,  23,  24,  22,  -1,  1,  -1,  -1,
  5,  6,  7,  8,  -1,  17,  0,  2,  20,  -1,
  13,  4,  21,  10,  -1,  9,  18,  3,  16,  14,
  15,  11,  12
);

```

```

{ Production lengths, right hand side lengths of the productions. }

```

```

ProdLength: Array[0..25] of Byte = (
  2,  1,  2,  3,  2,  1,  1,  1,  1,  2,
  2,  2,  3,  1,  2,  2,  2,  1,  2,  1,
  1,  1,  1,  1,  1,  1
);

```

```

{ Head symbol numbers, one for each production. }

```

```

HeadSymbNum: Array[0..25] of Integer = (
  0,  1,  1,  2,  3,  3,  4,  4,  4,  4,
  4,  5,  6,  6,  6,  6,  6,  7,  7,  8,
  9,  10,  11,  12,  13,  14
);

```

```

{----- }
{ Input processing functions }
{----- }

```

Type

```
InFunc = Function(ContextP: ContextPtr; TermNo: Integer): Integer;
```

Const

```
{ Input processing routine names as specified in the grammar. The }
{ numbers in list InputProcIndex[] are used as pointers into this list. }
```

```
InputProc: Array[0..1] of InFunc = (
    WordTypeCheck,
    PunctuatorCheck
);
```

```
{ Input processing routine numbers associated with tokens (coming from }
{ the scanner). A number zero would indicate the first input processing }
{ routine as specified in the InputProc[] list. A number of -1 would }
{ indicate no input processor for this token symbol. }
```

```
InputProcIndex: Array[0..26] of Integer = (
    -1,  0,  -1,  1,  -1,  -1,  -1,  -1,  -1,  -1,
    -1,  -1,  -1,  -1,  -1,  -1,  -1,  -1,  1,  -1,
    -1,  -1,  -1,  -1,  -1,  -1,  -1
);
```

```

{----- }
{ Output processing functions }
{----- }

```

Type

```
OutFunc = Function(ContextP: ContextPtr; Arg: String): String;
```

Const

```
{ Output processing routine names. The numbers in list }
{ OutputProcIndex[] are used as pointers to these names. }
```

```
OutputProc: Array[0..1] of OutFunc = (
    Emit,
    EmitTokenAndArg
);
```

```
{ Pointers to output processing routines, one for each production. }
{ A number zero would indicate the first output processing routine }
{ as specified in the OutputProc list[]]. A number of -1 would }
{ indicate no output processor for this production. }
```

```
OutputProcIndex: Array[0..25] of Integer = (
    -1,  -1,  -1,  0,  0,  0,  0,  0,  0,  0,
    0,  0,  0,  0,  0,  0,  0,  0,  0,  1,
    1,  1,  1,  1,  1,  1
);
```

```

{ Output processing routine arguments for each production.      }
{ These strings may be either alphabetic or numeric, depending }
{ on how they were specified in the grammar, but they are     }
{ always stored internally (in LALR 3.0) as string type. A -1 }
{ would indicate no argument for this production.             }

```

```

OutputProcArg: Array[0..25] of String = (

```

```

    '-1',
    '-1',
    '-1',
    'Found a sentence\n\n',
    'Found a NP\n',
    'Found a NP\n',
    'Found a NP2\n',
    'Found a PP\n',
    'Found a VP\n',
    'Found an adjective list\n',
    'Found an adjective list\n',
    '<Noun>\n',
    '<Verb>\n',
    '<Preposition>\n',
    '<Adjective>\n',
    '<Pronoun>\n',
    '<Proper>\n',
    '<Determiner>\n'

```

```
);
```

```

{----- }
{ Parser constants }
{----- }

```

```
Const
```

```
    GOAL = 0;          { Production number of final goal }
```

```
{----- }
```

```

{----- }
{ Parser Utilities }
{----- }

```

{ StampOutput - put headings, time and date at top of output file }

Procedure StampOutput;

Const

```

  Days: Array[0..6] of String[9] =
    ('Sunday','Monday','Tuesday', 'Wednesday','Thursday','Friday', 'Saturday');

```

Var

```

  Hours, Mins, Secs, Hundredths: Word;
  Year, Month, Day, DayOfWeek: Word;

```

Function LeadingZero(w: Word): String;

Var

```

  s : String;

```

Begin { LeadingZero }

```

  Str(w:0, s);
  If Length(s) = 1 then
    s := '0' + s;

```

```

  LeadingZero := s

```

End; { LeadingZero }

Begin { StampOutput }

```

  GrammarName := 'SIMPSENT.GRM';
  SkeletonName := 'MYPARSER.SKL';
  ParserName := 'MYPARSER.PAS';

```

```

  GetTime(Hours, Mins, Secs, Hundredths);

```

```

  GetDate(Year, Month, Day, DayOfWeek);

```

```

  Write(OutFile, 'Parser : ', ParserName:12);

```

```

  Writeln(OutFile, ' ');

```

```

  Write(OutFile, 'Grammar : ', GrammarName:12);

```

```

  Writeln(OutFile, ' } Created by LALR 3.0');

```

```

  Write(OutFile, 'Skeleton: ', SkeletonName:12);

```

```

  Writeln(OutFile, ' ');

```

```

  WriteLn(OutFile, 'Time : ', LeadingZero(Hours),
    ':', LeadingZero(Mins), ':', LeadingZero(Secs));

```

```

  WriteLn(OutFile, 'Date : ', Days[DayOfWeek], ' ',
    Day:0, '/', Month:0, '/', Year:0);

```

```

  Writeln(OutFile);

```

```

  Writeln(OutFile, '-----',
    + '-----');

```

```

  Writeln(OutFile)

```

End; { StampOutput }

```

{----- }

```

{ ParserInit – initialise the parser, return pointer to the first context }

Procedure ParserInit(Var ContextP: ContextPtr);

Begin { ParserInit }

```

  InitContext(ContextP) { Initialise the context stack }

```

End; { ParserInit }

```

{----- }

```

```

{ DoInputProc -- call an input processing routine }
Procedure DoInputProc(ContextP: ContextPtr);
Begin { DoInputProc }
  With ContextP^ do
    If InputProcIndex[Terminal] >= 0 then
      Terminal := InputProc[InputProcIndex[Terminal]]      { Call processor }
                  (ContextP, Terminal)
End; { DoInputProc }

{ ----- }

{ PrintLine -- print source line }
Procedure PrintLine(ContextP: ContextPtr);
Var
  s: String;
  p: CharPtr;
Begin { PrintLine }
  With ContextP^ do
    Begin
      If LineStart^ <> EOF then                                { If not at end of file }
        Begin
          s := '';
          p := LineStart;

          While (p^ <> CR) and (p^ <> EOF) do                  { Collect string up to end of line }
            Begin
              s := s + p^;
              Longint (p) := Longint (p) + 1
            End;

            Writeln(OutFile, LineNum:6, ' ', s)                { Print the line with its line number }
          End
        End { With ContextP^ do }
End; { PrintLine }

{ ----- }

{ PrintMsg -- print error message }
Procedure PrintMsg(ContextP: ContextPtr; Msg, Symbol: String);
Var
  i, nl, Col, LastLineNum: Integer;
  p: CharPtr;
  c: Char;
Begin { PrintMsg }
  With ContextP^ do
    Begin
      nl := 0;
      Col := 0;
      LastLineNum := -1;

      Writeln(OutFile);                                       { Print the line }
      Writeln(OutFile);
      PrintLine(ContextP);
      LastLineNum := LineNum;                                 { Save line number }
    End
  End
End;

```

```

    { Find column number of token causing error message }
    For Longint (p) := Longint (LineStart) to Longint (Token) - 1 do
        If p^ = TAB then
            Col := Col + 8 - (Col div 8)
        else
            Col := Col + 1;

    Write(OutFile, ' ');
    For i := 1 to Col do { Write out correct number of dashes }
        Write(OutFile, '-');

    Write(OutFile, '^ ' + Msg + Symbol);           { Print pointer to erroneous symbol }
                                                    { and the error message. }

    Writeln(OutFile)
    End { With ContextP^ do }
End; { PrintMsg }

{ ----- }

Procedure Crash(ContextP: ContextPtr; Msg, Symbol: String);
Begin { Crash }
    PrintMsg(ContextP, Msg, Symbol);
    Close(OutFile);
    Halt(1)
End; { Crash }

{ ----- }

{ GetToken -- get token from scanner }
Procedure GetToken(ContextP: ContextPtr);
Begin { GetToken }
    With ContextP^ do
        Begin
            DictData.NumTypes := 0;           { Initialise dictionary return values }
            PartsOfSpeech := [];

            While true do
                Begin
                    Scan(ContextP);           { Call the lexical scanner }
                    If Terminal <= 0 then
                        Begin
                            If Terminal = 0 then           { If terminal unrecognised? }
                                Begin
                                    If OutputOn then       { If output turned on? }
                                        Begin
                                            NumErrors := NumErrors + 1;
                                            PrintMsg(ContextP, 'Ignoring token ',
                                                StrPtrsToStr(Token, Input))
                                        End
                                    End { If Terminal = 0 then }
                                End
                            End
                        End
                    End
                End
            End
        End
    End
End;

```

```

else
  Begin
    If OutputOn then          { If output turned on? }
      Begin
        NumErrors := NumErrors + 1;
        PrintMsg(ContextP, 'Invalid token ',
          StrPtrsToStr(Token, Input))
      End;
      Terminal := -Terminal;    { Make terminal positive and continue
      }
      DoInputProc(ContextP); { Any input processing for this terminal }
    End
  End { If Terminal <= 0 then }
else
  Begin
    DoInputProc(ContextP);    { Any input processing for this terminal }
  Exit
  End
End { While true do }
End { With ContextP^ do }
End; { GetToken }

```

```
{ ----- }
```

```
{ GetTokens - get token, check dictionary, spawn new contexts if needed }
```

```
Procedure GetTokens(ContextP: ContextPtr);
```

```
Var
```

```
  NewContextP: ContextPtr;
```

```
  TokenNum: Integer;
```

```
Begin { GetTokens }
```

```
  With ContextP^ do
```

```
    Begin
```

```
      GetToken(ContextP);
```

```
      With DictData do
```

```
        Begin
```

```
          If NumTypes > 0 then
```

```
            Begin
```

```
              For TokenNum := 1 to NumTypes do
```

```
                If TokenNum > 1 then
```

```
                  Begin
```

```
                    CreateContext(ContextP, NewContextP);
```

```
                    NewContextP^.Terminal :=
```

```
                    Ord(TypeData[TokenNum].WordType)
```

```
                  End
```

```
                else
```

```
                  Terminal := Ord(TypeData[TokenNum].WordType)
```

```
            End { If NumTypes > 0 then }
```

```
          End { With DictData do }
```

```
        End { With ContextP^ do }
```

```
End; { GetTokens }
```

```
{ ----- }
```

```

{----- }
{ LR Parsing Engine }
{----- }

```

```

{ GetNextShiftStates - return possible next states for a state/terminal pair }
Procedure GetNextShiftStates(ContextP: ContextPtr);
Var
    TermTransNo: Integer;
Begin { GetNextShiftStates }
    With ContextP^ do
        Begin
            NumNextShiftStates := 0;
            { For all terminal transitions in this state }
            For TermTransNo := TermTransStart[State] to TermTransStart[State+1] - 1 do
                { If accessor of goto state matches incoming terminal symbol? }
                If Accessor[TermTrans[TermTransNo]] = Terminal then
                    Begin
                        NumNextShiftStates := NumNextShiftStates + 1;
                        NextShiftStates[NumNextShiftStates] := TermTrans[TermTransNo]
                    End
                End { With ContextP^ do }
        End; { GetNextShiftStates }

```

```

{----- }

```

```

Procedure DoOutputProc(ContextP: ContextPtr);
Var
    OPProcMsg: String;
Begin { DoOutputProc }
    With ContextP^ do
        Begin
            { If any action specified for this production? }
            If OutputProcIndex[Prodn] >= 0 then
                Begin
                    { Put procedure, argument and token on output stack }
                    OutputStackP := OutputStackP + 1;
                    OutputStack[OutputStackP].ProcNo := OutputProcIndex[Prodn];
                    OutputStack[OutputStackP].ArgNo := Prodn;
                    OutputStack[OutputStackP].TokenBeg := TokenBeg;
                    OutputStack[OutputStackP].TokenEnd := TokenEnd;

                    { Call the action procedure }
                    OPProcMsg := OutputProc[OutputProcIndex[Prodn]]
                        (ContextP, OutputProcArg[Prodn]);

                    { If OP error message then print it }
                    If OPProcMsg <> "" then
                        Begin
                            NumErrors := NumErrors + 1;
                            PrintMsg(ContextP, OPProcMsg, StrPtrsToStr(TokenBeg, TokenEnd))
                        End
                    End
                End
            End { With ContextP^ do }
        End; { DoOutputProc }

```

```

{----- }

```

```

{ DoShift - main code for the shift routine }
Procedure DoShift(ContextP: ContextPtr; NextState: Integer);
Var
  RednP: Integer;
  OPProcMsg: String;
Begin { DoShift }
  With ContextP^ do
    Begin
      If OutputOn then                                     { If output turned on? }
        Begin
          { For all reductions on reduction stack }
          RednP := 0;
          While RednP < RednStackP do
            Begin
              RednP := RednP + 1;
              Prodn := RednStack[RednP].Prodn;           { Get the production }
              DoOutputProc(ContextP);                     { Call the action routine }
            End { While RednNo < RednStackP do }
          End; { If OutputOn then }

          TokenBeg := Token; { Point at beg & end of }
          TokenEnd := Input; { terminal sym accepted. }
          RednStackP := 0; { Reset redn stack ptr }
          If ParseStackP = MaxStack - 1 then              { Parse stack overflow? Crash }
            Crash(ContextP, 'Tried to increment past the end of the parse stack.', '');
          ParseStackP := ParseStackP + 1;                 { Put cur state on parse stack }
          ParseStack[ParseStackP] := State;
          State := NextState                               { Define next state }
        End { With ContextP^ do }
      End; { DoShift }

      { ----- }
    End
  End

```

```

{ Shift -- perform a shift action }
Function Shift(ContextP: ContextPtr): Boolean;
Var
  NextState, NextStateNum: Integer;
Begin { Shift }
  With ContextP^ do
    Begin
      GetNextShiftStates(ContextP);
      { Find a possible next state for this state/terminal pair }
      For NextStateNum := 1 to NumNextShiftStates do
        Begin
          NextState := NextShiftStates[NextStateNum];
          If Accessor[NextState] = Terminal then
            Begin
              DoShift(ContextP, NextState);
              Shift := true; { Return true for shift action }
            End
          End
        End;
      Shift := false; { Return failure for shift action }
    End { With ContextP^ do }
  End; { Shift }

  { ----- }

```

{ GetNextRednState - return possible next redn state for state/nonterminal pair }

Procedure GetNextRednState(ContextP: ContextPtr);

Var

NonTermTransNo: Integer;

Begin { GetNextRednState }

With ContextP^ do

Begin

NumNextRednStates := 0;

{ For all nonterminal transitions at the origin of this production ... }

{ A match for the nonterminal transition is always present }

NonTermTransNo := NonTermTransStart[State];

While true do

Begin

{ If the accessor of the goto state matches the head symbol? }

If Accessor[NonTermTrans[NonTermTransNo]] = Head then

Begin

NumNextRednStates := NumNextRednStates + 1;

NextRednStates[NumNextRednStates] :=

NonTermTrans[NonTermTransNo];

Exit

End;

NonTermTransNo := NonTermTransNo + 1

End { While true do }

End { With ContextP^ do }

End; { GetNextRednState }

{ ----- }

{ DoReduce -- main code for Reduce routine }

Function DoReduce(ContextP: ContextPtr): Boolean;

Var

NumSyms, NextState, NextStateNum: Integer;

Begin { DoReduce }

With ContextP^ do

Begin

ParseStackP := ParseStackP + 1;

If ParseStackP = MaxStack then { If parse stack overflow, crash }

Crash(ContextP, 'Tried to increment past end of parse stack.', '');

{ Save old state on parse stack before replacing with current one }

RednStackP := RednStackP + 1;

RednStack[RednStackP].State := ParseStack[ParseStackP];

RednStack[RednStackP].Prodn := Prodn; { Save this production }

ParseStack[ParseStackP] := State; { Put current state on parse stack }

{ Get head symbol for this production. Note that the }

{ accessors for nonterminal transitions are negative }

Head := -HeadSymbNum[Prodn];

{ Reduce parse stack by length of production and get origin state }

ParseStackP := ParseStackP - ProdLength[Prodn];

State := ParseStack[ParseStackP];

```

    { Find a possible next state for this state/nonterminal pair }
    GetNextRednState(ContextP);
    If NumNextRednStates > 0 then
        Begin
            State := NextRednStates[1];           { Use the first state found }
            DoReduce := true;                     { Return true for shift action }
            Exit
        End;
        DoReduce := false                         { Return failure for shift action }
    End { With ContextP^ do }
End; { DoReduce }

```

```

{ ----- }

```

```

{ Reduce – perform a reduce action }
Function Reduce(ContextP: ContextPtr): Boolean;
Begin { Reduce }
    With ContextP^ do
        Begin
            { If no default reduction for this state return failure to reduce }
            Prodn := DefaultRedn[State];
            If Prodn <= 0 then
                Begin
                    Reduce := false;
                    Exit
                End;
            Reduce := DoReduce(ContextP)
        End { With ContextP^ do }
    End; { Reduce }

```

```

{ ----- }

```

```

Procedure DoParse(ContextP: ContextPtr);
Var
    NextAction: ActionType;
    WordTypeNum: Integer;
Begin { DoParse }
    With ContextP^ do
        Begin
            Finished := false;
            Parsing := true;
            NextAction := ShiftAction;
            While not Finished do
                Begin
                    Case NextAction of
                        NoAction:
                            Begin
                                Finished := true;
                                Parsing := false
                            End;
                        ShiftAction:
                            Begin
                                If Shift(ContextP) then
                                    NextAction := GetNextTokens
                                else
                                    NextAction := ReduceAction
                            End;

```

```

    ReduceAction:
      Begin
        If Reduce(ContextP) then
          NextAction := ShiftAction
        else
          NextAction := NoAction
        End;
      GetNextTokens:
      Begin
        GetTokens(ContextP);
        NextAction := ShiftAction
      End
    End; { Case NextAction of }
  End; { While not Finished do }

  If Prodn = GOAL then { Is this the goal production }
  Begin
    With ContextHead do
      NumSuccessfulParses := NumSuccessfulParses + 1;
      Exit
    End
  else
    Begin
      DeleteContext(ContextP);
      Exit
    End;
  End { With ContextP^ do }
End; { DoParse }

```

{ ----- }

{ Parse – find all possible LR parses }

Function Parse: Integer;

Var

ContextP: ContextPtr;

AllDone: Boolean;

Begin { Parse }

With ContextHead do

Begin

ParserInit(ContextP);

{ Initialise the parser }

GetTokens(ContextP);

{ Call scanner, get first terminal }

Repeat

AllDone := true;

{ Tell parser it is finished }

ContextP := ContextStackTop;

{ Step through the contexts }

While ContextP \neq nil do

Begin

DoParse(ContextP);

{ Perform the parse }

If not ContextP^.Finished then

{ If any parse is not finished }

AllDone := false;

{ tell the parser. }

ContextP := ContextP^.NextContext

End

Until AllDone;

{ Parsing is finished }

Parse := NumSuccessfulParses

End { With ContextHead do }

End; { Parse }

```
{-----}
```

```
{----- }  
{ Input processing functions }  
{----- }
```

```
{ WordTypeCheck -- check types of a word in dictionary }
```

```
Function WordTypeCheck(ContextP: ContextPtr; TermNo: Integer): Integer;
```

```
Var
```

```
  i: Integer;
```

```
  s: SymString;
```

```
Begin { WordTypeCheck }
```

```
  With ContextP^ do
```

```
    Begin
```

```
      s := StrPtrsToStr(Token, Input); { Put token into string }
```

```
      If RetrieveWord(s, PartsOfSpeech, DictData) then
```

```
        WordTypeCheck := TermNo { Return the original terminal number }
```

```
      else
```

```
        WordTypeCheck := -TermNo { Else return error code for this terminal }
```

```
      End { With ContextP^ do }
```

```
End; { WordTypeCheck }
```

```
{-----}
```

```
{ PunctuatorCheck -- check a punctuation symbol }
```

```
Function PunctuatorCheck(ContextP: ContextPtr; TermNo: Integer): Integer;
```

```
Begin { PunctuatorCheck }
```

```
  With ContextP^ do
```

```
    Begin
```

```
      { This routine is a dummy at present }
```

```
      PunctuatorCheck := TermNo { Return original number passed as argument }
```

```
    End { With ContextP^ do }
```

```
End; { PunctuatorCheck }
```

```
{-----}
```

```

{----- }
{ Output processing functions }
{----- }

```

{ Emit -- write out a string argument }

Function Emit(ContextP: ContextPtr; Arg: String): String;

Var

EscPos: Byte;

s: String;

Begin { Emit }

With ContextP^ do

Begin

If Parsing then

Begin

Emit := " { No error }

End

else

Begin

If Arg <> '-1' then

{ If there is an argument? }

Begin

Repeat

EscPos := Pos('\n', Arg); { Search for newline escape sequence }

If EscPos > 0 then

Begin

s := Copy(Arg, 1, EscPos - 1);

Delete(Arg, 1, EscPos + 1);

Writeln(OutFile, s); { Writeln parts of string up to newline }

End

Until EscPos = 0;

Write(OutFile, Arg)

{ Write remainder of string }

End;

Emit := " { No error }

End { If not Parsing then }

End { With ContextP^ do }

End; { Emit }

```

{----- }

```

{ EmitToken -- write out the current token }

Function EmitToken(ContextP: ContextPtr; Arg: String): String;

Begin { EmitToken }

With ContextP^ do

Begin

If Parsing then

Begin

EmitToken := " { No error }

End

else

Begin

Write(OutFile, StrPtrsToStr(TokenBeg, TokenEnd)); { Print token }

EmitToken := " { No error }

End { If not Parsing then }

End { With ContextP^ do }

End; { EmitToken }

```

{----- }

```

```

{ EmitTokenNI -- write out the current token & newline }
Function EmitTokenNI(ContextP: ContextPtr; Arg: String): String;
Begin { EmitTokenNI }
  With ContextP^ do
    Begin
      If Parsing then
        Begin
          EmitTokenNI := " { No error }
        End
      else
        Begin
          Writeln(OutFile, StrPtrsToStr(TokenBeg, TokenEnd)); { Print token }
          EmitTokenNI := " { No error }
        End { If not Parsing then }
      End { With ContextP^ do }
    End; { EmitTokenNI }

```

```
{ ----- }
```

```

{ EmitTokenAndArg -- write out a token followed by the string argument }
Function EmitTokenAndArg(ContextP: ContextPtr; Arg: String): String;
Begin { EmitTokenAndArg }
  With ContextP^ do
    Begin
      If Parsing then
        Begin
          EmitTokenAndArg := Emit(ContextP, Arg) { No error if " }
        End
      else
        Begin
          Write(OutFile, StrPtrsToStr(TokenBeg, TokenEnd)); { Print token }
          EmitTokenAndArg := Emit(ContextP, Arg) { No error if " }
        End { If not Parsing then }
      End { With ContextP^ do }
    End; { EmitTokenAndArg }

```

```
{ ----- }
```

```

{ EmitArgAndToken -- write out the string argument followed by current token }
Function EmitArgAndToken(ContextP: ContextPtr; Arg: String): String;
Begin { EmitArgAndToken }
  With ContextP^ do
    Begin
      If Parsing then
        Begin
          EmitArgAndToken := Emit(ContextP, Arg); { No error if " }
        End
      else
        Begin
          EmitArgAndToken := Emit(ContextP, Arg); { No error if " }
          Write(OutFile, StrPtrsToStr(TokenBeg, TokenEnd)) { Print token }
        End { If not Parsing then }
      End { With ContextP^ do }
    End; { EmitArgAndToken }

```

```
{ ----- }
```

```

{ EmitArgAndTokenNI -- write out the string argument, current token & newline }
Function EmitArgAndTokenNI(ContextP: ContextPtr; Arg: String): String;
Begin { EmitArgAndTokenNI }
  With ContextP^ do
    Begin
      If Parsing then
        Begin
          EmitArgAndTokenNI := Emit(ContextP, Arg);           { No error if " }
        End
      else
        Begin
          EmitArgAndTokenNI := Emit(ContextP, Arg);           { No error if " }
          Writeln(OutFile, StrPtrsToStr(TokenBeg, TokenEnd)) { Print token }
        End { If not Parsing then }
      End { With ContextP^ do }
    End; { EmitArgAndTokenNI }

{-----}

{----- }
{ Output Production Routines }
{----- }

{ DoContextOutput - generate output from a context's output stack }
Procedure DoContextOutput(ContextP: ContextPtr);
Var
  p: Integer;
  OPProcMsg: String;
Begin { DoContextOutput }
  With ContextP^ do
    Begin
      p := 0;
      While p < OutputStackP do
        Begin
          p := p + 1;
          TokenBeg := OutputStack[p].TokenBeg;
          TokenEnd := OutputStack[p].TokenEnd;
          OPProcMsg := OutputProc[OutputStack[p].ProcNo] { Call action routine }
            (ContextP, OutputProcArg[OutputStack[p].ArgNo]);

          If OPProcMsg <> " then { If OP error msg then print it }
            Begin
              NumErrors := NumErrors + 1;
              PrintMsg(ContextP, OPProcMsg, StrPtrsToStr(TokenBeg, TokenEnd))
            End
          End
        End { With ContextP^ do }
      End; { DoContextOutput }

{-----}

```

```

{ ProduceOutput - generate output from all successful parses }
Procedure ProduceOutput;
Var
  ContextP: ContextPtr;
Begin { ProduceOutput }
  With ContextHead do
    Begin
      ContextP := ContextStackTop;
      While ContextP <> nil do
        Begin
          With ContextP^ do
            Begin
              Writeln(OutFile, 'Context: ', ContextId:4);
              Writeln(OutFile, '-----');
              DoContextOutput(ContextP);
              Writeln(OutFile)
            End; { With ContextP^ do }
            ContextP := ContextP^.NextContext
          End { While ContextP <> nil }
        End { With ContextHead do }
      End; { ProduceOutput }

{-----}

End. { Unit MYPARSER }

```

Appendix D: - An ATN Grammar

Reference: Winograd, Terry, "Language as a cognitive process, volume 1: syntax", Addison-Wesley 1983.

The ATN description makes use of a number of abbreviations, as follows:

Referencing:	
C.R	The R of C
R.last	The last member of R
COPY	A copy based on ^
dummy X(xxx)	A dummy X with word = xxx
Initialisations:	
R1 ← ^.R2	Initialise R1 to the R2 of ^
Actions:	
R1 ← R2	Set R1 to R2
R1 ⇐ R2	Append R2 to R1
Conditions:	
C.R = X	The R of C is X
C.R != X	The R of C is not X
R = 0	R is empty
R != 0	R is not empty
R = xxx	The word in R is xxx
(R1 + R2) @ Dict	R1 and R2 share a dictionary entry
<p>In the above, R stands for a register (possibly *, the ATN node most recently parsed); C for a constituent; X for a feature or constituent; and ^ for the node from which a recursive call was made.</p>	

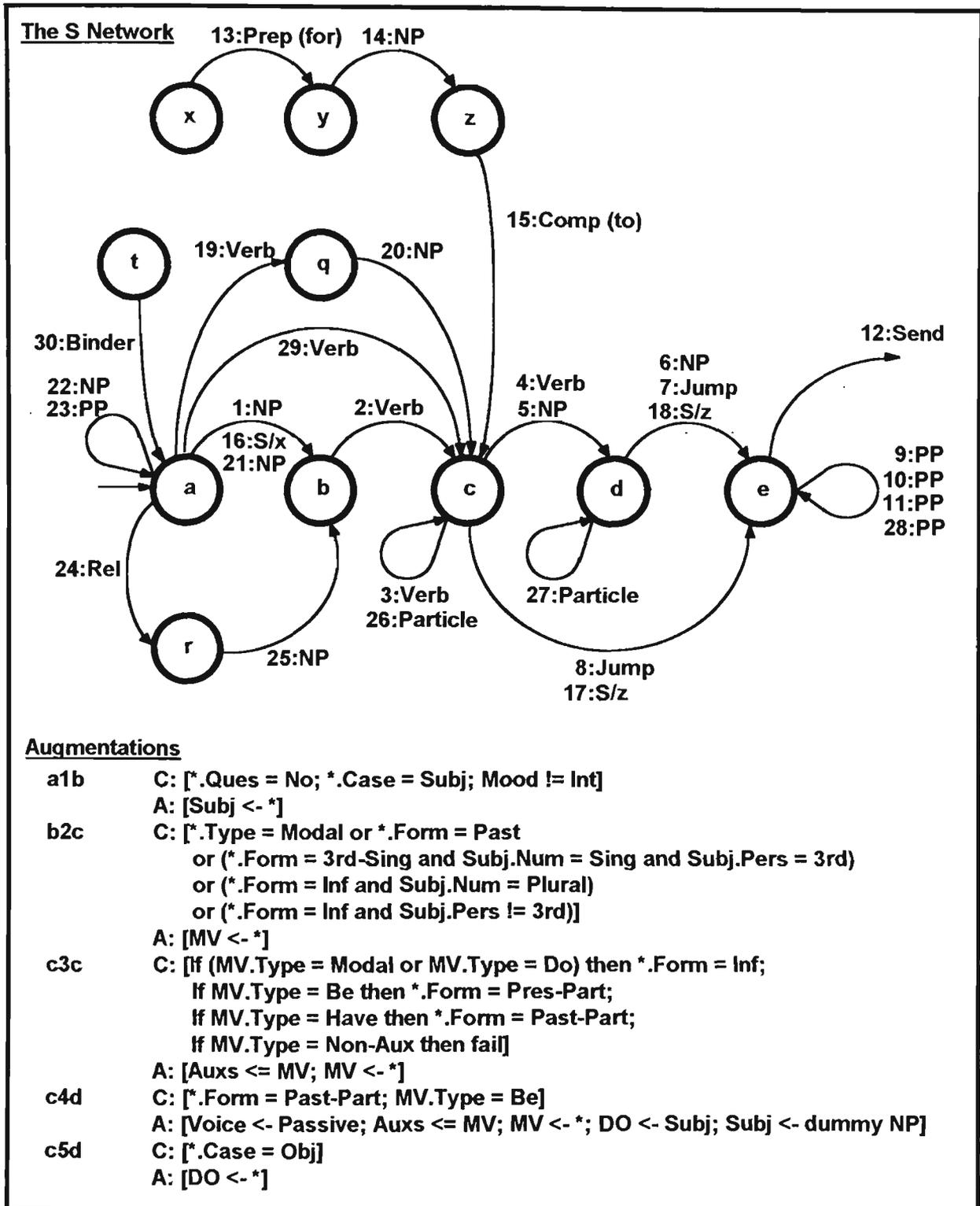
The S Network:

Roles:

- Subject
- Direct-Object
- Indirect-Object
- Main-Verb
- Binder
- Auxiliaries
- Modifiers
- Question-Element.

Feature Dimensions:

- **Voice:** Active, Passive; *default Active.*
- **Mood:** Declarative, Interrogative, Imperative, Bound, Relative, WhRel; *default Decl.*



Augmentations (continued)

d6e	C: [* .Case = Obj] A: [IO <- DO; DO <- *]
d7e	<i>No initialisations, conditions or actions</i>
d8e	<i>No initialisations, conditions or actions</i>
e9e	A: [Mods <= *]
e10e	C: [* .Prep = by; Voice = Pasive; Subj = dummy NP] A: [Subj <- * .PrepObj]
e11e	C: [* .Prep = to or * .Prep = for; IO = 0] A: [IO <- * .PrepObj]
e12	C: [If (Mood = Int or Mood = WhRel) then Hold = 0; If IO != 0 then MV .Transitivity = Bitransitive; If (IO = 0 and DO != 0) then MV .Transitivity = Transitive; If (DO = 0 and IO = 0) then MV .Transitivity = Intransitive]
x13y	C: [* = for]
y14z	C: [* .Case = Obj] A: [Subj <- *]
z15c	C: [* = to] A: [MV <- dummy Verb; MV .Type = Modal]
a16b	A: [Subj <- *]
c17e	I: [Subj <- ^ .Subj] A: [DO <- *]
d18e	I: [Subj <- ^ .DO] A: [IO <- DO; DO <- *]
a19q	C: [* .Type != Non-Aux; Mood = Decl or Mood = Int] A: [MV <- *; Mood <- Int]
q20c	C: [* .Ques = No; * .Case = Subj; MV .Type = Modal or MV .Form = Past or (MV .Form = 3rd-Sing and * .Num = Sing and * .Pers = 3rd) or (MV .Form = Inf and * .Num = Plural) or (MV .Form = Inf and * .Pers != 3rd)] A: [Subj <- *]
a21b	C: [* .Ques = Yes; * .Case = Subj; Mood = Decl] A: [Subj <- *; QE <- *]
a22a	C: [* .Ques = Yes; Mood = Decl] A: [QE <- *; Hold <- *; Mood <- Int]
r24a	<i>No initialisations, conditions or actions</i>
r25b	A: [Subj <- *]
c26c	C: [(MV + *) @Dict] A: [MV <- Dict]
d27d	C: [(MV + *) @Dict] A: [MV <- Dict]
e28e	C: [(MV + * .Prep) @Dict; DO = 0] A: [MV <- Dict; DO <- * .PrepObj]
a29c	C: [* .Form = Inf; Mood = Decl] A: [Subj <- dummy NP; Subj .Head <- you; MV <- *; Mood <- Imper]
t30a	A: [Binder <- *; Mood <- Bound]

The NP Network:

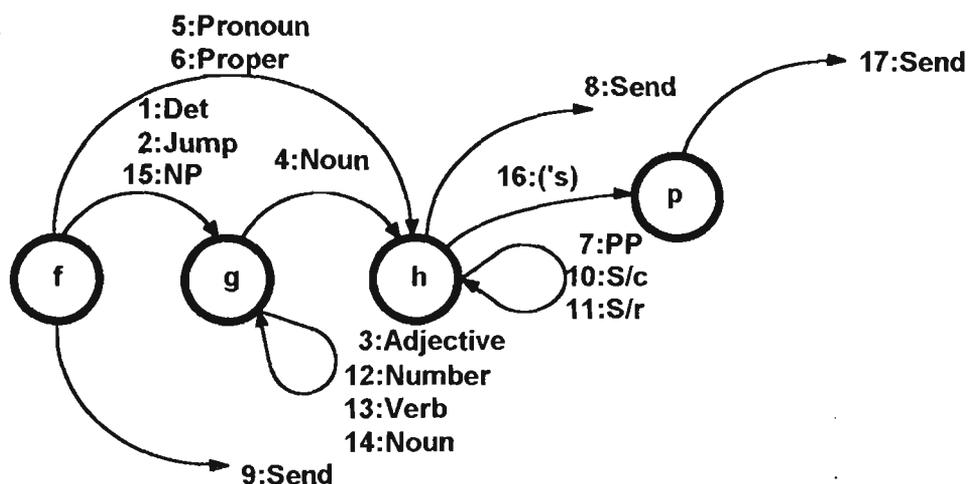
Roles:

- Determiner
- Head
- Describers
- Qualifiers

Feature Dimensions:

- **Number:** Singular, Plural; *default* Empty.
- **Person:** First, Second, Third; *default* Third.
- **Question:** Yes, No; *default* No.
- **Case:** Subjective, Objective, Possessive; *default* Empty.

The NP Network



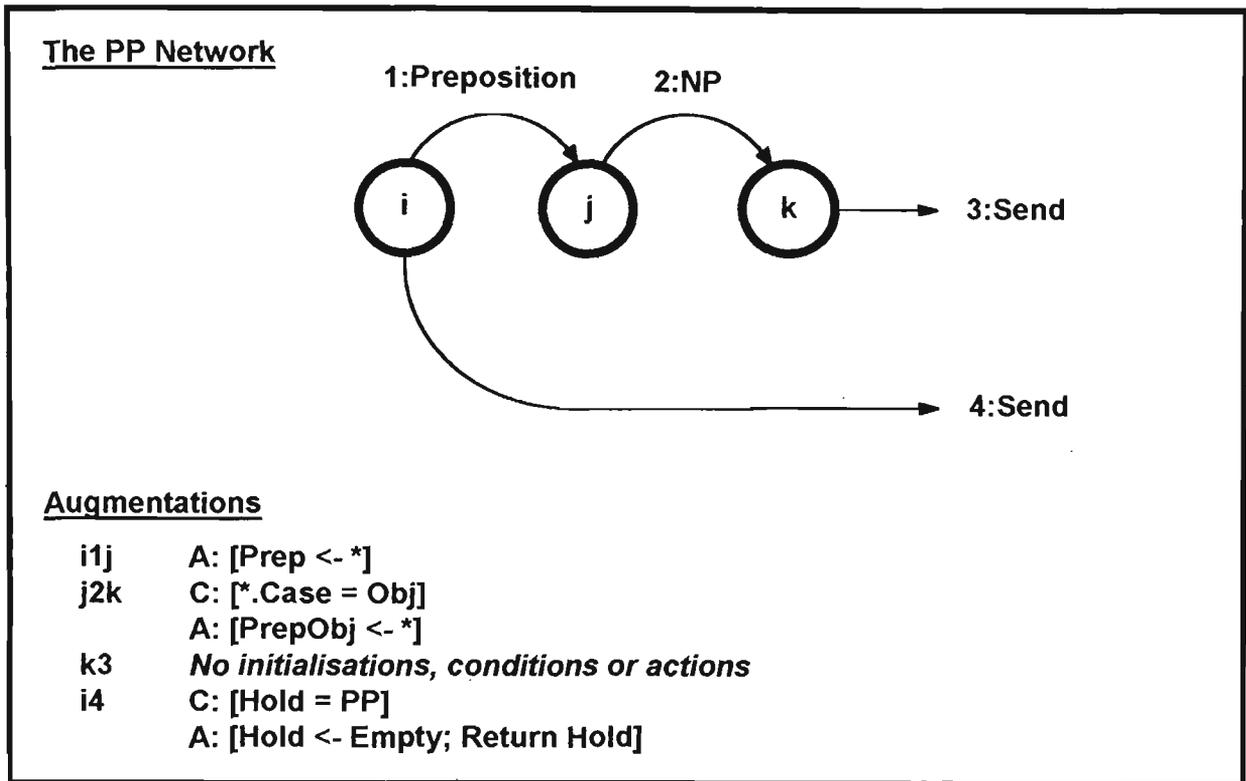
Augmentations

f1g	A: [Num <- *.Num; Ques <- *.Ques; Det <- *]
f2g	<i>No initialisations, conditions or actions</i>
g3g	A: [Desc <= *]
g4h	C: [*.Num = Num or Num = 0] A: [Num <- *.Num; Head <- *]
f5h	A: [Num <- *.Num; Per <- *.Per; Ques <- *.Ques; Head <- *]
f6h	A: [Num <- *.Num; Head <- *]
h7h	A: [Qual <= *]
h8	A: [Case <- Head.Case]
f9	C: [Hold = NP] A: [Hold <- Empty; Return Hold]
h10h	I: [Subj <- *COPY*; Mood <- Rel; MV <- dummy Verb; MV.Type <- Be] A: [Qual <= *]
h11h	I: [Hold <- *COPY*; Mood <- WhRel] A: [Qual <= *]
g12g	A: [Desc <= *]
g13g	C: [*.Form = Pres-Part or *.Form = Past-Part] A: [Desc <= *]
g14g	C: [*.Num = Sing] A: [Desc <= *]
f15g	C: [*.Case = Poss] A: [Det <- *]
h16p	C: [Head.Cat != Pronoun]
p17	A: [Case <- Poss]

The PP Network:

Roles:

- Preposition
- PrepObj.



Appendix E: Augmented Grammar for an English Language Subset

/* Augmented Context-free Grammar */

/* Goal */

Sentence -> S <eof>

/* Sentence Network Alternative Entry Points */

S-c -> S1

S-r -> Sra S
-> Srb Sbc S1

S-t -> Sta S

S-x -> Sxy Syz S1

S-z -> Szc S1

/* Sentence Network */

S -> Sac S1
-> Saa Sab Sbc S1
-> Saq Sqc S1

S1 -> Scc Scd Sdd Sde See Se
-> Sce See Se

Saa ->
-> Sa22a
-> Sa23a

Sab -> Sa1b
-> Sa16b
-> Sa21b

Sac -> Sa29c

Saq -> Sa19q

Sbc -> Sb2c

Scc ->
-> Scc Sc3c
-> Scc Sc26c

Scd -> Sc4d
-> Sc5d

Sce -> Sc8e
-> Sc17e

Sdd ->
-> Sdd Sd27d

Sde -> Sd6e
-> Sd7e
-> Sd18e

Se -> Se12

See ->
-> See Se9e
-> See Se10e
-> See Se11e
-> See Se28e

Sqc -> Sq20c

Sra -> Sr24a

Srb -> Sr25b

Sta -> St30a

Sxy -> Sx13y

Syz -> Sy14z

Szc -> Sz15c

Salb

"C:[*.Ques=No; *.Case=Subj; Mood!=Int] A:[Subj<-*]"
-> NP

Sb2c

"C:[*.Type=Modal or *.Form=Past
or (*.Form=3rd-Sing and Subj.Num=Sing and Subj.Pers=3rd)
or (*.Form=Inf and Subj.Num=Plural) or (*.Form=Inf and Subj.Pers!=3rd)]
A:[MV<-*]"
-> Verb

Sc3c

"C:[If (MV.Type=Modal or MV.Type=Do) then *.Form=Inf;
If MV.Type=Be then *.Form=Pres-Part;
If MV.Type=Have then *.Form=Past=Part;
If MV.Type=Non-Aux then fail] A:[Auxs<=MV; MV<-*]"
-> Verb

Sc4d

"C:[*.Form=Past-Part; MV.Type=Be]
A:[Voice<-Passive; Auxs<=MV; MV<-*; DO<-Subj; Subj<-dummy NP]"
-> Verb

Sc5d

"C:[*.Form=Past-Part; MV.Type=Be]
A:[Voice<-Passive; Auxs<=MV; MV<-*; DO<-Subj; Subj<-dummy NP]"
-> NP

Sd6e

"C:[*.Case=Obj] A:[IO<-DO; DO<-*]"
-> NP

Sd7e

-> Jump

Sc8e

-> Jump

Se9e

"A:[Mods<=*]"
-> PP

Se10e

"C:[*.Prep=by; Voice=Passive; Subj=dummy NP] A:[Subj<-*.PrepObj]"
-> PP

Se11e

"C:[*.Prep=to or *.Prep=for; IO=0] A:[IO<-*.PrepObj]"
-> PP

Se12

"C:[If (Mood=Int or Mood=WhRel) then Hold=0;
If IO!=0 then MV.Transitivity=Bitransitive;
If (IO=0 and DO!=0) then MV.Transitivity=Transitive;
If (DO=0 and IO=0) then MV.Transitivity=Intransitive]"
-> Send

Sx13y

"C:[*=for]"
-> Prep-for

Sy14z

"C:[*.Case=Obj] A:[Subk<-*]"
-> NP

Sz15c

"C:[*=to] A:[MV<-dummy Verb; MV.Type=Modal]"
-> Comp-to

Sa16b

"A:[Subj<-*]"
-> S-x

Sc17e

"I:[Subj<-^Subj] A:[DO<-*]"
-> S-z

Sd18e

"I:[Subj<-^DO] A:[IO<-DO; DO<-*]"
-> S-z

Sa19q

"C:[*.Type!=Non-Aux; Mood=Decl or Mood=Int] A:[MV<-*; Mood<-Int]"
-> Verb

Sq20c

"C:[*.Ques=No; *.Case=Subj; MV.Type=Modal or MV.Form=Past
or (MV.Form=3rd-Sing and *.Num=Sing and *.Pers=3rd)
or (MV.Form=Inf and *.Num=Plural) or (MV.Form=Inf and *.Pers!=3rd)]
A:[Subj<-*]"
-> NP

Sa21b

"C:[*.Ques=Yes; *.Case=Subj; Mood=Decl] A:[Subj<-*; QE<-*]"
-> NP

Sa22a

"C:[*.Ques=Yes; Mood=Decl] A:[QE<-*; Hold<-*; Mood<-Int]"
-> NP

Sa23a

"C:[*.PrepObj.Ques=Yes; Mood=Decl]
A:[QE<-*.PrepObj; Hold<-*; Mood<-Int]"
-> PP

Sr24a

-> Rel

Sr25b

"A:[Subj<-*]"
-> NP

Sc26c

"C:[(MV+*)@Dict] A:[MV<-Dict]"
-> Particle

Sd27d

"C:[(MV+*)@Dict] A:[MV<-Dict]"
-> Particle

Se28e

"C:[(MV+*.Prep)@Dict; DO=0] A:[MV<-Dict; DO<-*.PrepObj]"
-> PP

Sa29c

"C:[*.Form=Inf; Mood=Decl] A:[Subj<-dummy NP; Subj.Head<-you; MV<-*;
Mood<-Imper]"
-> Verb

St30a

"A:[Binder<-*; Mood<-Bound]"
-> Binder

/* Noun Phrase Network */

NP -> NPf
-> NPfg NPgg NPgh NPhh NPh
-> NPfg NPgg NPgh NPhh NPhp NPp
-> NPfh NPhh NPh
-> NPfh NPhh NPhp NPp

NPf -> NPf9

NPfg -> NPf1g
-> NPf2g
-> NPf15g

NPfh -> NPf5h
-> NPf6h

NPgg ->
-> NPgg NPg3g
-> NPgg NPg12g
-> NPgg NPg13g
-> NPgg NPg14g

NPgh -> NPg4h

NPh -> NPh8

NPhh ->

-> NPhh NPh7h

-> NPhh NPh10h

-> NPhh NPh11h

NPhp -> NPh16p

NPp -> NPp17

NPf1g

"A:[Num<-* .Num; Ques<-* .Ques; Det<-*]"

-> Det

NPf2g

-> Jump

NPg3g

"A:[Desc<=*]"

-> Adjective

NPg4h

"C:[* .Num=Num or Num=0] A:[Num<-* .Num; Head<-*]"

-> Noun

NPf5h

"A:[Num<-* .Num; Pers<-* .Pers; Ques<-* .Ques; Head<-*]"

-> Pronoun

NPf6h

"A:[Num<-* .Num; Head<-*]"

-> Proper

NPh7h

"A:[Qual<=*]"

-> PP

NPh8

"A:[Case<-Head .Case]"

-> Send

NPf9

"C:[Hold=NP] A:[Hold<-Empty; Return Hold]"

-> Send

NPh10h

"I:[Subj<=*COPY*; Mood<-Rel; MV<-dummy Verb; MV.Type<-Be]
A:[Qual<=*]"
-> S-c

NPh11h

"I:[Hold<-*COPY*; Mood<-WhRel] A:[Qual<=*]"
-> S-r

NPg12g

"A:[Desc<=*]"
-> Number

NPg13g

"C:[*.Form=Pres-Part or *.Form=Past-Part] A:[Desc<=*]"
-> Verb

NPg14g

"C:[*.Num=Sing] A:[Desc<=*]"
-> Noun

NPf15g

"C:[*.Case=Poss] A:[Det<-*]"
-> NP

NPh16p

"C:[Head.Cat!=Pron]"
-> Apostrophe-s

NPp17

"A:[Case<-Poss]"
-> Send

/* Preposition Phrase Network */

PP -> PPi
-> PPij PPjk PPk

PPi -> PPi4

PPij -> PPi1j

PPjk -> PPj2k

PPk -> PPk3

PPi1j

"A:[Prep<-*]"
> Preposition

PPj2k

"C:[*.Case=Obj] A:[PrepObj<-*]"
-> NP

PPk3

-> Send

PPi4

"C:[Hold=PP] A:[Hold<-Empty; Return Hold]"
-> Send

Appendix F: Grammar for Augmentation Language

/* Terminals */

<error>

!=

(

)

*

+

.

'0'

;

<-

<=

=

\"

]

^

A:[

AdjClass

AdjDegree

AdjKind

AdjType

Adjective

AdvDegree

AdvKind

AdvMeaning

AdvType

Adverb

C:[

COPY

ConjGroup

Conjunction

Determiner

Dict

Dictionary

Hold

I:[

If

Interjection

NP

NPCase

NPNumber

NPPerson

NPQuestion

Noun

NounCase

NounClass
NounGender
NounGroup
NounNumber
NounPerson
NPDescribers
NPDeterminer
NPHead
NPQualifiers
PP
Preposition
PPPprep
PPPprepObj
ProductionNumber
PronCase
PronClass
PronGender
PronNumber
PronPerson
Pronoun
Return
S
SMood
SVoice
SAuxiliaries
SBinder
SDirectObject
SIndirectObject
SMainVerb
SModifiers
SSubject
Verb
VerbForm
VerbKind
VerbMood
VerbNumber
VerbPerson
VerbTense
VerbTrans
VerbVoice
abstract
abs
active
affirmative
affirm
and
article
auxiliary
aux

be
bitransitive
bitrans
bound
by
can
cardinal
card
cause
collective
collctv
common
comparative
comprtv
comparison
comprsn
comptv
concrete
concr
conjunctive
conjtv
coordinating
coord
correlative
correl
declarative
decl
definite
def
degree
demonstrative
demonstv
descriptive
descptv
do
dummy
fail
feminine
fem
first
for
future-perfect
future-perf
fut-perfect
fut-perf
have
imperative
imper
in

indefinite
indef
indicative
indctv
infinitive
inf
interrogative
interr
intransitive
intrans
irregular
irreg
last
limiting
lim
manner
masculine
masc
may
modal
must
negative
neg
neuter
neut
no
nominative
nomntve
non-comparable
non-comp
number
numeral
objective
objctv
or
ordinal
ord
passive
past
past-participle
past-part
past-perfect
past-perf
perfect
perf
personal
pers
place
plural

plur
positive
pos
possessive
poss
present
pres
principal
prncpl
proper
purpose
regular
reg
relative
rel
second
shall
singular
sing
subjunctive
subjctv
subordinating
subord
superlative
super
then
third
time
to
transitive
trans
wh-relative
wh-rel
will
yes
you
<number>
<eof>

/ Augmentation */*

AugmentationGrammar

-> AugmentationFile <eof>

AugmentationFile

-> AugmentationEntries

AugmentationEntries

-> AugmentationEntry

-> AugmentationEntries AugmentationEntry

AugmentationEntry

-> ProductionNumber AugmentationString

ProductionNumber

-> <number>

=> StoreProdnNumber

AugmentationString

-> \" Entries \"

Entries

-> Entry

-> Entries Entry

Entry

-> Conditions

-> Actions

-> Initialisations

/ Words */*

Word

-> by

-> for

-> to

-> you

/* Features */

Feature

- > SentenceFeature
- > NounPhraseFeature
- > NounFeature
- > PronounFeature
- > VerbFeature
- > AdjectiveFeature
- > AdverbFeature
- > ConjunctionFeature

SentenceFeature

- > SMood
- > SVoice

NounPhraseFeature

- > NPNumber
- > NPPerson
- > NPQuestion
- > NPCase

NounFeature

- > NounClass
- > NounGroup
- > NounPerson
- > NounNumber
- > NounGender
- > NounCase

PronounFeature

- > PronClass
- > PronPerson
- > PronNumber
- > PronGender
- > PronCase

VerbFeature

- > VerbKind
- > VerbTrans
- > VerbVoice
- > VerbMood
- > VerbTense
- > VerbPerson
- > VerbNumber
- > VerbForm

AdjectiveFeature

- > AdjClass
- > AdjKind
- > AdjDegree
- > AdjType

AdverbFeature

- > AdvMeaning
- > AdvType
- > AdvKind
- > AdvDegree

ConjunctionFeature

- > ConjGroup

/* Feature Dimensions */

FeatureDim

- > SentenceFeatureDim
- > NounPhraseFeatureDim
- > NounFeatureDim
- > PronounFeatureDim
- > VerbFeatureDim
- > AdjectiveFeatureDim
- > AdverbFeatureDim
- > ConjunctionFeatureDim

SentenceFeatureDim

- > SentenceMoodDim
- > SentenceVoiceDim

NounPhraseFeatureDim

- > NounPhraseNumberDim
- > NounPhrasePersonDim
- > NounPhraseQuestionDim
- > NounPhraseCaseDim

NounFeatureDim

- > NounClassDim
- > NounGroupDim
- > NounPersonDim
- > NounNumberDim
- > NounGenderDim
- > NounCaseDim

PronounFeatureDim

- > PronounClassDim
- > PronounPersonDim
- > PronounNumberDim
- > PronounGenderDim
- > PronounCaseDim

VerbFeatureDim

- > VerbKindDim
- > VerbTransitivityDim
- > VerbVoiceDim
- > VerbMoodDim
- > VerbTenseDim
- > VerbPersonDim
- > VerbNumberDim
- > VerbFormDim

AdjectiveFeatureDim

- > AdjectiveClassDim
- > AdjectiveKindDim
- > AdjectiveDegreeDim
- > AdjectiveTypeDim

AdverbFeatureDim

- > AdverbMeaningDim
- > AdverbTypeDim
- > AdverbKindDim
- > AdverbDegreeDim

ConjunctionFeatureDim

- > ConjunctionGroupDim

/* Sentence Feature Dimensions */

SentenceMoodDim

- > declarative
- > decl
- > interrogative
- > interr
- > imperative
- > imper
- > bound
- > relative
- > rel
- > wh-relative
- > wh-rel

SentenceVoiceDim

- > active
- > passive

/* Noun Phrase Feature Dimensions */

NounPhraseNumberDim

- > singular
- > sing
- > plural
- > plur

NounPhrasePersonDim

- > first
- > second
- > third

NounPhraseQuestionDim

- > yes
- > no

NounPhraseCaseDim

- > nominative
- > nomntv
- > possessive
- > possv
- > objective
- > objctv

/* Noun Feature Dimensions */

NounClassDim

- > common
- > proper

NounGroupDim

- > abstract
- > abs
- > concrete
- > cncrt
- > collective
- > collctv

NounPersonDim

- > first
- > second
- > third

NounNumberDim

- > singular
- > sing
- > plural
- > plur

NounGenderDim

- > masculine
- > masc
- > feminine
- > fem
- > neuter
- > neut
- > common

NounCaseDim

- > nominative
- > nomntv
- > possessive
- > possv
- > objective
- > objctv

/* Pronoun Feature Dimensions */

PronounClassDim

- > personal
- > pers
- > relative
- > rel
- > interrogative
- > interr
- > demonstrative
- > demonstv
- > indefinite
- > indef

PronounPersonDim

- > first
- > second
- > third

PronounNumberDim

- > singular
- > sing
- > plural
- > plur

PronounGenderDim

- > masculine
- > masc
- > feminine
- > fem
- > neuter
- > neut
- > common

PronounCaseDim

- > nominative
- > nomntv
- > possessive
- > possv
- > objective
- > objctv

/* Verb Feature Dimensions */

VerbKindDim

- > principal
- > princpl
- > auxiliary
- > aux
- > regular
- > reg
- > irregular
- > irreg
- > modal
- > infinitive
- > inf

VerbTransitivityDim

- > transitive
- > trans
- > intransitive
- > intrans
- > bitransitive
- > bitrans

VerbVoiceDim

- > active
- > passive

VerbMoodDim

- > indicative
- > indctv
- > subjunctive
- > subjnctv
- > imperative
- > impertv

VerbTenseDim

- > present
- > pres
- > past
- > past-participle
- > past-part
- > perfect
- > perf
- > past-perfect
- > past-perf
- > future-perfect
- > future-perf
- > fut-perfect
- > fut-perf

VerbPersonDim

- > first
- > second
- > third

VerbNumberDim

- > singular
- > sing
- > plural
- > plur

VerbFormDim

- > be
- > can
- > do
- > have
- > may
- > must
- > shall
- > will

/* Adjective Feature Dimensions */

AdjectiveClassDim

- > descriptive
- > descptv
- > limiting
- > lim
- > proper

AdjectiveKindDim

- > regular
- > reg
- > irregular
- > irreg
- > numeral
- > article

AdjectiveDegreeDim

- > positive
- > pos
- > comparative
- > comprtv
- > superlative
- > super
- > non-comparable
- > non-comp

AdjectiveTypeDim

- > cardinal
- > card
- > ordinal
- > ord
- > definite
- > def
- > indefinite
- > indef

/* Adverb Feature Dimensions */

AdverbMeaningDim

- > time
- > place
- > manner
- > degree
- > cause
- > purpose
- > number

AdverbTypeDim

- > affirmative
- > affirm
- > negative
- > neg
- > interrogative
- > interr
- > relative
- > rel
- > conjunctive
- > conjtv
- > comparison
- > comprsn

AdverbKindDim

- > regular
- > reg
- > irregular
- > irreg

AdverbDegreeDim

- > positive
- > postv
- > comparative
- > comptv
- > superlative
- > super
- > non-comparable
- > non-comp

/ Conjunction Feature Dimensions */*

ConjunctionGroupDim

- > coordinating
- > coord
- > subordinating
- > subord
- > correlative
- > correl

/ Roles */*

Role

- > SentenceRole
- > NounPhraseRole
- > PrepositionPhraseRole

SentenceRole

- > SSubject
- > SAuxiliaries
- > SMainVerb
- > SIndirectObject
- > SDirectObject
- > SBinder
- > SModifiers

NounPhraseRole

- > NPDeterminer
- > NPHead
- > NPDescribers
- > NPQualifiers

PrepositionPhraseRole

- > PPPrep
- > PPPrepObj

/* Registers */

RegisterRef

- > Register
- > Register . RegisterRef
- > * . RegisterRef
- > ^ . RegisterRef
- > ^
- > *
- > RegisterRef . last
- > RegisterRef . first

RoleRef

- > Role
- > Role . FeatureRef

RegisterOrRoleRef

- > RegisterRef
- > RoleRef

FeatureRef

- > Feature
- > RegisterRef . Feature
- > RoleRef . Feature

Register

- > S
- > NP
- > PP
- > Verb
- > Determiner
- > Noun
- > Pronoun
- > Adjective
- > Adverb
- > Preposition
- > Conjunction
- > Interjection
- > Hold
- > Dict

/* Conditions */

Conditions

-> ConditionStart ConditionList ConditionEnd => StartCondition

ConditionStart

-> C:[=> StartCondition

ConditionEnd

->] => EndCondition

ConditionList

-> ConditionExp
-> ConditionList ; ConditionExp

ConditionExp

-> RegisterOrRoleRef = Register
-> RegisterOrRoleRef != Register
-> RegisterOrRoleRef = '0'
-> RegisterOrRoleRef != '0'
-> RegisterOrRoleRef = Word
-> (RegisterOrRoleRef + RegisterOrRoleRef) in Dictionary
-> FeatureRef = Feature
-> FeatureRef != Feature
-> FeatureRef = '0'
-> FeatureRef != '0'
-> (ConditionExp)
-> ConditionExp or ConditionExp
-> ConditionExp and ConditionExp
-> If ConditionExp then ActionExp
-> If ConditionExp then fail

/* Actions */

Actions

-> A:[ActionList]

ActionStart

-> A:[=> StartAction

ActionEnd

->] => EndAction

ActionList

-> ActionExp

-> ActionList ; ActionExp

ActionExp

-> RegisterOrRoleRef <- RegisterOrRoleRef

-> FeatureRef <- FeatureRef

-> FeatureRef <- FeatureDim

-> RegisterOrRoleRef <- dummy Register

-> RoleRef <= RegisterOrRoleRef

-> RegisterOrRoleRef <- '0'

-> Return RegisterOrRoleRef

/* Initialisations */

Initialisations

-> I:[InitialisationList]

InitialisationList

-> InitialisationExp

-> InitialisationList ; InitialisationExp

InitialisationExp

-> RegisterOrRoleRef <- RegisterOrRoleRef

-> FeatureRef <- FeatureRef

-> FeatureRef <- FeatureDim

-> RegisterOrRoleRef <- dummy Register

-> RegisterOrRoleRef <- COPY

Appendix G: Prefixes

Prefixes are in alphabetical order.

<i>a</i>	<i>birth</i>	<i>dead</i>	<i>free</i>
<i>ab</i>	<i>black</i>	<i>dec</i>	<i>fug</i>
<i>abs</i>	<i>blood</i>	<i>deca</i>	<i>fuge</i>
<i>ac</i>	<i>blow</i>	<i>deci</i>	<i>gastero</i>
<i>acri</i>	<i>blue</i>	<i>dek</i>	<i>gastr</i>
<i>acro</i>	<i>brain</i>	<i>deka</i>	<i>gastro</i>
<i>ad</i>	<i>broad</i>	<i>dem</i>	<i>gen</i>
<i>aer</i>	<i>bronch</i>	<i>demo</i>	<i>geno</i>
<i>aero</i>	<i>broncho</i>	<i>dendr</i>	<i>geo</i>
<i>af</i>	<i>by</i>	<i>dendro</i>	<i>giga</i>
<i>after</i>	<i>bye</i>	<i>di</i>	<i>Graec</i>
<i>ag</i>	<i>caec</i>	<i>dia</i>	<i>Graeco</i>
<i>agri</i>	<i>caeco</i>	<i>dicho</i>	<i>grand</i>
<i>agro</i>	<i>caeno</i>	<i>dino</i>	<i>graph</i>
<i>air</i>	<i>caino</i>	<i>dipl</i>	<i>grapho</i>
<i>al</i>	<i>calli</i>	<i>diplo</i>	<i>great</i>
<i>all</i>	<i>cardi</i>	<i>dis</i>	<i>Grec</i>
<i>allo</i>	<i>cardio</i>	<i>dodec</i>	<i>Greco</i>
<i>alt</i>	<i>carn</i>	<i>dodeca</i>	<i>green</i>
<i>alti</i>	<i>carni</i>	<i>door</i>	<i>gyro</i>
<i>alto</i>	<i>cat</i>	<i>down</i>	<i>hair</i>
<i>ambi</i>	<i>cata</i>	<i>duo</i>	<i>half</i>
<i>amphi</i>	<i>cath</i>	<i>dyna</i>	<i>hand</i>
<i>an</i>	<i>cent</i>	<i>dynam</i>	<i>hard</i>
<i>ana</i>	<i>centen</i>	<i>dynamo</i>	<i>head</i>
<i>angelo</i>	<i>centi</i>	<i>dys</i>	<i>heart</i>
<i>Anglo</i>	<i>centr</i>	<i>e</i>	<i>hect</i>
<i>ant</i>	<i>centri</i>	<i>ec</i>	<i>hecto</i>
<i>ante</i>	<i>centro</i>	<i>eco</i>	<i>hekto</i>
<i>antho</i>	<i>chem</i>	<i>ef</i>	<i>heli</i>
<i>anthropo</i>	<i>chemo</i>	<i>electr</i>	<i>helio</i>
<i>anti</i>	<i>Chino</i>	<i>electro</i>	<i>hemi</i>
<i>ap</i>	<i>choreo</i>	<i>em</i>	<i>hept</i>
<i>aph</i>	<i>Christo</i>	<i>en</i>	<i>hepta</i>
<i>apo</i>	<i>chrom</i>	<i>ep</i>	<i>herb</i>
<i>aqua</i>	<i>chromo</i>	<i>eph</i>	<i>herbi</i>
<i>ar</i>	<i>chron</i>	<i>epi</i>	<i>here</i>
<i>arch</i>	<i>chrono</i>	<i>equi</i>	<i>heter</i>
<i>aristo</i>	<i>cine</i>	<i>ethno</i>	<i>hetero</i>
<i>as</i>	<i>circum</i>	<i>eu</i>	<i>hex</i>
<i>astro</i>	<i>co</i>	<i>ex</i>	<i>hexa</i>
<i>at</i>	<i>col</i>	<i>exa</i>	<i>hier</i>
<i>atto</i>	<i>com</i>	<i>exo</i>	<i>hiero</i>
<i>audio</i>	<i>con</i>	<i>extra</i>	<i>high</i>
<i>aut</i>	<i>contr</i>	<i>extro</i>	<i>hippo</i>
<i>auto</i>	<i>contra</i>	<i>femto</i>	<i>holo</i>
<i>back</i>	<i>cor</i>	<i>ferri</i>	<i>hom</i>
<i>baro</i>	<i>cosm</i>	<i>ferro</i>	<i>home</i>
<i>batho</i>	<i>cosmo</i>	<i>fire</i>	<i>homeo</i>
<i>be</i>	<i>counter</i>	<i>fluor</i>	<i>homo</i>
<i>bed</i>	<i>cred</i>	<i>fluoro</i>	<i>homoeo</i>
<i>bene</i>	<i>credo</i>	<i>fly</i>	<i>homoio</i>
<i>bi</i>	<i>cross</i>	<i>foot</i>	<i>horo</i>
<i>biblio</i>	<i>cruci</i>	<i>for</i>	<i>hot</i>
<i>bin</i>	<i>cut</i>	<i>fore</i>	<i>house</i>
<i>bio</i>	<i>de</i>	<i>forti</i>	<i>hydr</i>

hydro
hyper
ice
ideo
idio
il
ill
im
in
infra
inter
intra
intro
ir
iron
is
iso
kilo
king
klept
klepto
land
laryng
laryngo
lati
left
life
light
lith
litho
logo
long
love
low
lumin
macr
macro
magni
mal
mar
mare
matri
mechan
mechano
meg
mega
megalo
melan
melano
meta
metr
metri
metro
micr
micro
mid
milli
mini
mis
mole
mon

mono
morph
morpho
motor
multi
mytho
nano
neo
neur
neuro
news
night
non
nor
nudi
nulli
o
ob
oc
oct
octa
octo
of
off
ole
oleo
omni
one
onto
op
ortho
oste
osteo
out
over
palae
palaeo
pale
paleo
pan
panto
par
para
patho
patri
ped
pedati
pedi
pent
penta
per
peri
peta
phil
philo
phleb
phlebo
phon
phono
photo
phylo

physio
pico
plan
plani
plano
play
pleisto
pluto
poly
post
pre
preter
pro
prot
proto
pseud
pseudo
psych
psycho
quadr
quadri
quasi
quin
radio
ram
re
rect
recti
red
retro
road
rock
Russo
sacro
sand
saw
se
sea
self
semi
sept
septe
septem
septi
sesqui
sex
short
side
small
snow
socio
soli
south
space
steno
step
stere
stereo
steth
stetho
stock

stop
su
sub
subter
suc
suf
sug
sui
sum
sun
sup
super
supra
sur
sy
syl
sym
syn
sys
tauto
taxe
taxi
taxo
techn
techno
tel
tele
tera
tetra
the
theo
there
therm
thermo
time
top
topo
trans
tri
tubercul
tuberculo
turbo
two
ultra
un
under
uni
up
vice
water
wave
well
where
wind
with
work
xyl
xylo
yester

Appendix H: Suffixes

Suffixes are in reverse alphabetic order - last character to first character.

<i>+'</i>	<i>+ualised</i>	<i>+able</i>	<i>-e+ose</i>
<i>+s'</i>	<i>-e+ated</i>	<i>+C+able</i>	<i>-ke+cose</i>
<i>+ia</i>	<i>+ized</i>	<i>-e+able</i>	<i>+iose</i>
<i>-iac+ia</i>	<i>+ualized</i>	<i>-ate+able</i>	<i>-lysis+lyse</i>
<i>-ic+ia</i>	<i>+oid</i>	<i>-y+iabile</i>	<i>-a+ate</i>
<i>-e+ia</i>	<i>-e+oid</i>	<i>+isable</i>	<i>-e+ate</i>
<i>-ical+ia</i>	<i>+fold</i>	<i>+izable</i>	<i>-eal+ellate</i>
<i>-y+ia</i>	<i>+end</i>	<i>+ible</i>	<i>+uate</i>
<i>-ary+ia</i>	<i>-two+second</i>	<i>+C+ible</i>	<i>+ite</i>
<i>+mania</i>	<i>+hood</i>	<i>-e+ible</i>	<i>-e+ite</i>
<i>+omania</i>	<i>-y+ihood</i>	<i>-ge+sible</i>	<i>-re+site</i>
<i>+opia</i>	<i>+ard</i>	<i>-t+sible</i>	<i>+ette</i>
<i>-ia+iana</i>	<i>+C+ard</i>	<i>+acle</i>	<i>+lyte</i>
<i>-e+iana</i>	<i>+ward</i>	<i>-ate+acle</i>	<i>+agogue</i>
<i>-ian+iana</i>	<i>-ree+ird</i>	<i>-e+ile</i>	<i>+esque</i>
<i>+ac</i>	<i>-t+ce</i>	<i>+phile</i>	<i>-e+esque</i>
<i>-ia+iac</i>	<i>-ny+ce</i>	<i>-e+tule</i>	<i>+ve</i>
<i>-y+iac</i>	<i>+ice</i>	<i>+cule</i>	<i>-fe+ve</i>
<i>+ic</i>	<i>-e+ice</i>	<i>+uncule</i>	<i>-f+ve</i>
<i>-ia+ic</i>	<i>-y+ice</i>	<i>+drome</i>	<i>+ive</i>
<i>+C+ic</i>	<i>+ance</i>	<i>+chrome</i>	<i>-e+ive</i>
<i>-e+ic</i>	<i>+C+ance</i>	<i>+some</i>	<i>-de+sive</i>
<i>-o+ic</i>	<i>-ate+ance</i>	<i>-an+ane</i>	<i>+ative</i>
<i>-y+ic</i>	<i>-ant+ance</i>	<i>+cene</i>	<i>-ate+ative</i>
<i>-y+fic</i>	<i>-y+iance</i>	<i>+ine</i>	<i>-y+ative</i>
<i>-d+ific</i>	<i>+ulance</i>	<i>+C+ine</i>	<i>+itive</i>
<i>-ify+ific</i>	<i>-ty+ulance</i>	<i>-e+ine</i>	<i>-e+itive</i>
<i>+pathic</i>	<i>+ence</i>	<i>+phone</i>	<i>+ize</i>
<i>+tic</i>	<i>+C+ence</i>	<i>+scape</i>	<i>+C+ize</i>
<i>-sis+tic</i>	<i>-e+ence</i>	<i>+scope</i>	<i>-e+ize</i>
<i>-y+tic</i>	<i>-ent+ence</i>	<i>+trope</i>	<i>-ice+ize</i>
<i>+atic</i>	<i>+escence</i>	<i>+type</i>	<i>-y+ize</i>
<i>-ar+atic</i>	<i>-esce+escence</i>	<i>+re</i>	<i>-t+dize</i>
<i>-y+etic</i>	<i>-ept+ipience</i>	<i>+ware</i>	<i>+ualize</i>
<i>+istic</i>	<i>-us+ulence</i>	<i>+sphere</i>	<i>-lysis+lyze</i>
<i>-a+istic</i>	<i>+esce</i>	<i>+osphere</i>	<i>-ve+f</i>
<i>-ic+istic</i>	<i>+ade</i>	<i>+where</i>	<i>+proof</i>
<i>-e+istic</i>	<i>-e+ade</i>	<i>+metre</i>	<i>+ing</i>
<i>+ualistic</i>	<i>-h+cade</i>	<i>+litre</i>	<i>+C+ing</i>
<i>+lytic</i>	<i>+grade</i>	<i>+ure</i>	<i>-e+ing</i>
<i>-lysis+lytic</i>	<i>+cide</i>	<i>-e+ure</i>	<i>+ling</i>
<i>+d</i>	<i>+ode</i>	<i>-ed+dure</i>	<i>+ising</i>
<i>+d</i>	<i>+tude</i>	<i>-ce+se</i>	<i>+ualising</i>
<i>-t+d</i>	<i>+ee</i>	<i>-rt+se</i>	<i>-e+ating</i>
<i>+ad</i>	<i>+C+ee</i>	<i>+ese</i>	<i>+izing</i>
<i>+head</i>	<i>-e+ee</i>	<i>+ise</i>	<i>+ualizing</i>
<i>+ed</i>	<i>-eal+ellee</i>	<i>+C+ise</i>	<i>+arch</i>
<i>+C+ed</i>	<i>+age</i>	<i>-e+ise</i>	<i>+graph</i>
<i>-e+ed</i>	<i>-e+age</i>	<i>-ice+ise</i>	<i>+morph</i>
<i>+/-headed</i>	<i>+fuge</i>	<i>-y+ise</i>	<i>+ish</i>
<i>-y+tied</i>	<i>+ie</i>	<i>-t+dise</i>	<i>+C+ish</i>
<i>+red</i>	<i>+C+ie</i>	<i>+ualise</i>	<i>-e+ish</i>
<i>-e+red</i>	<i>-e+ie</i>	<i>+wise</i>	<i>+th</i>
<i>+ised</i>	<i>+like</i>	<i>+ose</i>	<i>-ve+th</i>

+path	-a+an	-ive+ption	+s
-e+opath	+ian	-ve+ution	+mas
+eth	-ia+ian	+hedron	+ics
+C+eth	-e+ian	+person	-e+ics
-e+eth	-y+ian	+ton	-o+ics
-y+ieth	+bian	+ern	-y+ics
-ve+fth	+ician	+ship	-y+tics
-ong+ength	-ic+ician	-th+ship	-a+atics
+lith	-y+ician	+manship	-ar+atics
+with	-e+arian	+smanship	+es
-ep+pth	-y+arian	+up	-e+es
+speak	-ary+arian	+/-up	-is+es
+ock	-ory+orian	+ar	-ex+ices
+al	+sman	+C+ar	-ix+ices
-a+al	+en	-e+ar	-y+ies
-e+al	+C+en	-al+lar	-fe+ves
-us+al	-e+en	-le+ular	-ft+ves
+ical	-er+en	-ule+ular	+polis
-e+ical	+teen	+er	+osis
-y+ical	-t+teen	+C+er	+biosis
-r+tical	+ren	-e+C+er	+gnosis
-cy+tical	-other+ethren	-e+er	+lysis
+atical	+/-in	-y+er	+itis
+istical	+kin	-ry+er	+ess
-a+istical	+gon	+der	+C+ess
-e+istical	+ion	+eer	-e+ess
+C+ial	-e+ion	-e+eer	-er+ess
-ary+C+ial	-ite+ion	-e+ifer	-or+ess
-is+ial	-ect+icion	+ier	-y+ess
-y+ial	-d+ision	-e+ier	+less
-e+ential	-ce+ision	-y+ier	-y+iless
+ional	-de+ision	-ry+ier	+ness
-ed+dural	-ge+ision	+erer	-e+ableness
+ual	-se+ision	-ry+erer	+ibleness
+eval	-t+ision	+ater	+iveness
-e+rel	-e+ision	+olater	-y+iness
+erel	-el+ulsion	-ol+olater	-ve+fullness
+phil	-ine+ension	+meter	-ire+ress
+ll	-it+ission	-p+C+meter	-eror+ress
+ful	+tion	-y+imeter	-ter+ress
-y+iful	-e+tion	+ometer	-tor+ress
-ve+tful	-l+tion	+ster	+stress
+gram	+ation	+aster	-ulate+ulus
+dom	-a+ation	+yer	-cule+culus
+form	-e+ation	+or	+unculus
-e+iasm	-y+ication	+C+or	+ous
+ism	-fy+fication	-e+or	-e+ous
-ic+ism	-e+ification	-ate+or	-ar+ous
-e+ism	-ify+ification	-y+or	-y+ous
-ise+ism	+isation	+C+ior	-ety+ous
-ive+ism	-e+isation	-ire+eror	-ity+ous
-ize+ism	-ise+isation	-e+ator	-arity+ous
-ist+ism	+uation	+ceptor	+eous
-y+ism	+ization	+saur	-y+eous
+ualism	-e+ization	+our	+C+ious
-e+atism	-ize+ization	-e+our	-ion+ious
+endum	-fy+faction	-e+iour	-io+ious
+an	-ear+arition	+s	-y+ious

<i>-iety+ious</i>	<i>-ept+ipent</i>	<i>-e+ify</i>	<i>+ery</i>
<i>-ity+ious</i>	<i>-ear+arent</i>	<i>-iful+ify</i>	<i>+C+ery</i>
<i>-iosity+ious</i>	<i>-ve+pt</i>	<i>-y+ify</i>	<i>-e+ery</i>
<i>-acy+acious</i>	<i>-ive+pt</i>	<i>+logy</i>	<i>-er+ery</i>
<i>-acity+acious</i>	<i>+C+art</i>	<i>+ology</i>	<i>-ler+ery</i>
<i>-ocity+ocious</i>	<i>+ast</i>	<i>+inology</i>	<i>+ory</i>
<i>-ite+itious</i>	<i>-ary+ast</i>	<i>+urgy</i>	<i>-e+ory</i>
<i>-ition+itious</i>	<i>-e+iast</i>	<i>+archy</i>	<i>-el+ulsory</i>
<i>+philous</i>	<i>+est</i>	<i>+graphy</i>	<i>-e+atory</i>
<i>+ulous</i>	<i>+C+est</i>	<i>+sophy</i>	<i>+atry</i>
<i>-ke+ulous</i>	<i>-e+est</i>	<i>+pathy</i>	<i>+iatry</i>
<i>+gamous</i>	<i>+fest</i>	<i>+ly</i>	<i>+olatory</i>
<i>-er+rous</i>	<i>-y+iest</i>	<i>-e+ly</i>	<i>-ol+olatory</i>
<i>+erous</i>	<i>+ist</i>	<i>-le+ly</i>	<i>+metry</i>
<i>+iferous</i>	<i>-a+ist</i>	<i>+ably</i>	<i>+ty</i>
<i>-ic+iferous</i>	<i>-e+ist</i>	<i>-e+ably</i>	<i>-te+ty</i>
<i>-e+iferous</i>	<i>-ise+ist</i>	<i>-able+ably</i>	<i>-al+ty</i>
<i>+pteros</i>	<i>-ize+ist</i>	<i>+ibly</i>	<i>-ive+ifty</i>
<i>-our+orous</i>	<i>-ism+ist</i>	<i>-ible+ibly</i>	<i>+ity</i>
<i>+vorous</i>	<i>-y+ist</i>	<i>+ively</i>	<i>-e+ity</i>
<i>+ways</i>	<i>+ualist</i>	<i>-e+ingly</i>	<i>-acious+acity</i>
<i>+n't</i>	<i>+nist</i>	<i>-y+ily</i>	<i>-ocious+ocity</i>
<i>-ll+n't</i>	<i>+ionist</i>	<i>-ial+ially</i>	<i>-e+ility</i>
<i>-n+n't</i>	<i>+tist</i>	<i>-y+ially</i>	<i>-ice+ility</i>
<i>-ill+on't</i>	<i>+most</i>	<i>+ually</i>	<i>-le+ility</i>
<i>-d+t</i>	<i>-one+first</i>	<i>-ual+ually</i>	<i>-ble+ility</i>
<i>-se+t</i>	<i>-e+trix</i>	<i>-ve+fully</i>	<i>-ile+ility</i>
<i>-ve+t</i>	<i>-or+trix</i>	<i>+/-ply</i>	<i>-il+ility</i>
<i>-cy+t</i>	<i>+y</i>	<i>+gamy</i>	<i>-ble+bility</i>
<i>+crat</i>	<i>+C+y</i>	<i>+nomy</i>	<i>+ability</i>
<i>+stat</i>	<i>-e+y</i>	<i>+tomy</i>	<i>-e+ability</i>
<i>+sect</i>	<i>+cy</i>	<i>-ix+ectomy</i>	<i>-able+ability</i>
<i>+et</i>	<i>-tic+cy</i>	<i>+thermy</i>	<i>-ate+ability</i>
<i>-e+et</i>	<i>-ce+cy</i>	<i>+phany</i>	<i>+ibility</i>
<i>+let</i>	<i>-te+cy</i>	<i>+geny</i>	<i>-ible+ibility</i>
<i>+tight</i>	<i>-t+cy</i>	<i>+gony</i>	<i>-emy+mity</i>
<i>+ant</i>	<i>-atic+acy</i>	<i>+phony</i>	<i>+osity</i>
<i>-e+ant</i>	<i>-e+acy</i>	<i>+mony</i>	<i>-ose+osity</i>
<i>-ance+ant</i>	<i>-ate+acy</i>	<i>+scopy</i>	<i>-us+osity</i>
<i>-ate+ant</i>	<i>-at+acy</i>	<i>+ry</i>	<i>-ous+osity</i>
<i>-eal+ellant</i>	<i>+cracy</i>	<i>-l+ry</i>	<i>+iosity</i>
<i>-ty+ulant</i>	<i>+ancy</i>	<i>-er+ry</i>	<i>-iose+iosity</i>
<i>+ent</i>	<i>-e+ancy</i>	<i>+ary</i>	<i>+evity</i>
<i>-e+ent</i>	<i>-ant+ancy</i>	<i>-e+ary</i>	<i>+ivity</i>
<i>-ence+ent</i>	<i>-y+ancy</i>	<i>-ate+ary</i>	<i>-e+ivity</i>
<i>-ate+ent</i>	<i>-ty+ulancy</i>	<i>-nial+ary</i>	<i>-ive+ivity</i>
<i>-ency+ent</i>	<i>-e+ency</i>	<i>-an+ary</i>	<i>-wo+wenty</i>
<i>+escent</i>	<i>-us+ulency</i>	<i>-eer+ary</i>	<i>-ree+irty</i>
<i>-esce+escent</i>	<i>+ey</i>	<i>-ant+ary</i>	<i>-our+orty</i>
<i>+ulent</i>	<i>-id+efy</i>	<i>-ample+emplary</i>	
<i>-us+ulent</i>	<i>+ify</i>	<i>-ain+anary</i>	
<i>+ment</i>	<i>-ific+ify</i>	<i>-our+orary</i>	

Appendix I: Suffix Transforms and their Effects:

Adj_Adj	adj -> adj
Adj_Adj_Causing	adj -> adj (causing)
Adj_Adj_FullOf	adj -> adj (full of)
Adj_Adj_Number	adj -> adj (number)
Adj_Adj_Producing	adj -> adj (producing)
Adj_Adj_Superlative	adj -> adj (superlative)
Adj_Adj_Superlative__Adv_Adv_Superlative	adj -> adj (superlative)/ adv -> adv (superlative)
Adj_AdjAdv_Number	adj -> adj,adv (number)
Adj_Noun	adj -> noun
Adj_Noun_Ability	adj -> noun (ability)
Adj_Noun_ActionOrCondition	adj -> noun (action or condition)
Adj_Noun_Actor	adj -> noun (actor)
Adj_Noun_Collection	adj -> noun (collection)
Adj_Noun_ConditionOrCharacteristics	adj -> noun (condition or characteristics)
Adj_Noun_ImperfectResemblance	adj -> noun (imperfect resemblance)
Adj_Noun_Language	adj -> noun (language)
Adj_Noun_Person	adj -> noun (person)
Adj_Noun_Ply	adj -> noun (ply)
Adj_Noun_QualityOrStateOfBeing	adj -> noun (quality or state of being)
Adj_Noun_State	adj -> noun (state)
Adj_Noun_StateOrQuality	adj -> noun (state or quality)
Adj_Noun_Ware	adj -> noun (ware)
Adj_Noun_Where	adj -> noun (where)
Adj_Verb_Past	adj -> verb (past,pastpart)
AdjAdv_Noun_State	adj,adv -> noun (state)
AdjAdvNounVerb_Noun_Proficiency	adj,adv,noun,verb -> noun (proficiency)
AdjAdvVerb_Noun_State	adj,adv,verb -> noun (state)
AdjNoun_Adj	adj,noun -> adj
AdjNoun_Adj_Direction	adj,noun -> adj (direction)
AdjNoun_Noun	adj,noun -> noun
AdjNoun_Noun_ActionOrCondition	adj,noun -> noun (action or condition)
AdjNoun_Noun_ConditionOrRank	adj,noun -> noun (condition or rank)
AdjNoun_Noun_Hypocoristic	adj,noun -> noun (hypocoristic)
AdjNoun_Noun_StateOrCondition	adj,noun -> noun (state or condition)
AdjNoun_Verb	adj,noun -> verb
AdjNoun_Verb_Action	adj,noun -> verb (action)
AdjNoun_Verb_CauseToBe	adj,noun -> verb (cause to be)
AdjNounPfxVerb_Adj_Tendency	adj,noun,pfx,verb -> adj (tendency)
AdjNounPfxVerb_AdjNoun_FunctionLocationRelation	adj,noun,pfx,verb -> adj,noun (function, location or relation)
AdjNounPfxVerb_Noun_State	adj,noun,pfx,verb -> noun (state)
AdjNounPrep_Adj_Like	adj,noun,prep -> adj (like)
AdjNounVerb_Adj_CharacterisedBy	adj,noun,verb -> adj (characterised by)
AdjNounVerb_Adj_Pertaining	adj,noun,verb -> adj (pertaining)
AdjNounVerb_Adv	adj,noun,verb -> adv
AdjNounVerb_Noun	adj,noun,verb -> noun
AdjNounVerb_Noun_ActionOrCondition	adj,noun,verb -> noun (action or condition)

AdjNounVerb_Noun_ConditionOrCharacteristicsadj,noun,verb -> noun
(condition or characteristics)**AdjNounVerb_Noun_ConditionOrPractice**

adj,noun,verb -> noun (condition or practice)

AdjNounVerb_Noun_Identification

adj,noun,verb -> noun (identification)

AdjNounVerb_Noun_Object

adj,noun,verb -> noun (object)

AdjNounVerb_Noun_Practitioner

adj,noun,verb -> noun (practitioner)

AdjNounVerb_Noun_StateOrQuality

adj,noun,verb -> noun (state or quality)

AdjNounVerb_Verb

adj,noun,verb -> verb

AdjPfx_Adj_Moving

adj,px -> adj (moving)

AdjPrep_Adj_Superlative

adj,prep -> adj (superlative)

AdjPrep_AdjAdv_Direction

adj,prep -> adj,adv (direction)

Adv_Adv_Conjunction

adv -> adv (conjunction)

Noun_Adj

noun -> adj

Noun_Adj_AboundingIn

noun -> adj (abounding in)

Noun_Adj_FullOf

noun -> adj (full of)

Noun_Adj_ImperviousTo

noun -> adj (impervious to)

Noun_Adj_Like

noun -> adj (like)

Noun_Adj_Manner

noun -> adj (manner)

Noun_Adj_Pertaining

noun -> adj (pertaining)

Noun_Adj_Pertaining_Verb_Noun_Agent

noun -> adj (pertaining) / verb -> adj (agent)

Noun_Adj_Place

noun -> adj (place)

Noun_Adj_Producing

noun -> adj (producing)

Noun_Adv_Direction

noun -> adv (direction)

Noun_Adv_Manner

noun -> adv (manner)

Noun_Noun

noun -> noun

Noun_Noun_ActionOrCondition

noun -> noun (action or condition)

Noun_Noun_Agent

noun -> noun (agent)

Noun_Noun_Collection

noun -> noun (collection)

Noun_Noun_Condition

noun -> noun (condition)

Noun_Noun_Diminutive

noun -> noun (diminutive)

Noun_Noun_DomainOrCondition

noun -> noun (domain or condition)

Noun_Noun_Expert

noun -> noun (expert)

Noun_Noun_FeastOrHoliday

noun -> noun (feast or holiday)

Noun_Noun_FeminineAgent

noun -> noun (feminine agent)

Noun_Noun_FemOrDim

noun -> noun (feminine or diminutive)

Noun_Noun_Festivity

noun -> noun (festivity)

Noun_Noun_Identification

noun -> noun (identification)

Noun_Noun_Passion

noun -> noun (passion)

Noun_Noun_Person

noun -> noun (person)

Noun_Noun_Plural

noun -> noun (plural)

Noun_Noun_Plur_Verb_Verb_3PersActivenoun -> noun (plural) / verb -> verb
(3rd pers sing indictv active)**Noun_Noun_Possessive**

noun -> noun (possessive)

Noun_Noun_Producing

noun -> noun (producing)

Noun_Noun_ScienceOrArt

noun -> noun (science or art)

Noun_Noun_State

noun -> noun (state)

Noun_Noun_StateOrQuality

noun -> noun (state or quality)

Noun_Noun_Suffering

noun -> noun (suffering)

Noun_Noun_View

noun -> noun (view)

Noun_Noun_WorshiperOf

noun -> noun (worshiper of)

Noun_Verb

noun -> verb

Noun_Verb_Past

noun -> verb (past,pastpart)

NounPfx_Adj	noun,pfx -> adj
NounPfx_Adj_BeingOrBecoming	noun,pfx -> adj (being or becoming)
NounPfx_AdjNoun_Action	noun,pfx -> adj,noun (action)
NounPfx_AdjNoun_Belonging	noun,pfx -> adj,noun (belonging)
NounPfx_AdjNoun_Expert	noun,pfx -> adj,noun (expert)
NounPfx_Noun_Abstract	noun,pfx -> noun (abstract)
NounPfx_Noun_ActionOrCondition	noun,pfx -> noun (action or condition)
NounPfx_Noun_Actor	noun,pfx -> noun (actor)
NounPfx_Noun_Chief	noun,pfx -> noun (chief)
NounPfx_Noun_City	noun,pfx -> noun (city)
NounPfx_Noun_Collection	noun,pfx -> noun (collection)
NounPfx_Noun_ConditionOrResult	noun,pfx -> noun (condition or result)
NounPfx_Noun_Cutting	noun,pfx -> noun (cutting)
NounPfx_Noun_Decomposition	noun,pfx -> noun (decomposition)
NounPfx_Noun_Diminutive	noun,pfx -> noun (diminutive)
NounPfx_Noun_Distribution	noun,pfx -> noun (distribution)
NounPfx_Noun_Drawn	noun,pfx -> noun (drawn)
NounPfx_Noun_Drawing	noun,pfx -> noun (drawing)
NounPfx_Noun_Flight	noun,pfx -> noun (flight)
NounPfx_Noun_Form	noun,pfx -> noun (form)
NounPfx_Noun_GeometricalShape	noun,pfx -> noun (geometrical shape)
NounPfx_Noun_GeometricalSolid	noun,pfx -> noun (geometrical solid)
NounPfx_Noun_Government	noun,pfx -> noun (government)
NounPfx_Noun_Graph	noun,pfx -> noun (graph)
NounPfx_Noun_Heat	noun,pfx -> noun (heat)
NounPfx_Noun_ImperfectResemblance	noun,pfx -> noun (imperfect resemblance)
NounPfx_Noun_Inflammation	noun,pfx -> noun (inflammation)
NounPfx_Noun_Knowledge	noun,pfx -> noun (knowledge)
NounPfx_Noun_Leading	noun,pfx -> noun (leading)
NounPfx_Noun_Like	noun,pfx -> noun (like)
NounPfx_Noun_Litre	noun,pfx -> noun (litre)
NounPfx_Noun_Lizard	noun,pfx -> noun (lizard)
NounPfx_Noun_Lover	noun,pfx -> noun (lover)
NounPfx_Noun_Manifestation	noun,pfx -> noun (manifestation)
NounPfx_Noun_Marriage	noun,pfx -> noun (marriage)
NounPfx_Noun_Measuring	noun,pfx -> noun (measuring)
NounPfx_Noun_MeasuringInstrument	noun,pfx -> noun (measuring instrument)
NounPfx_Noun_MedicalCare	noun,pfx -> noun (medical care)
NounPfx_Noun_Metre	noun,pfx -> noun (metre)
NounPfx_Noun_New	noun,pfx -> noun (new)
NounPfx_Noun_Origin	noun,pfx -> noun (origin)
NounPfx_Noun_Origination	noun,pfx -> noun (origination)
NounPfx_Noun_Passion	noun,pfx -> noun (passion)
NounPfx_Noun_Process	noun,pfx -> noun (process)
NounPfx_Noun_Recognition	noun,pfx -> noun (recognition)
NounPfx_Noun_Ruler	noun,pfx -> noun (ruler)
NounPfx_Noun_Running	noun,pfx -> noun (running)
NounPfx_Noun_See	noun,pfx -> noun (see)
NounPfx_Noun_Seeing	noun,pfx -> noun (seeing)
NounPfx_Noun_Sight	noun,pfx -> noun (sight)
NounPfx_Noun_Sound	noun,pfx -> noun (sound)
NounPfx_Noun_Sphere	noun,pfx -> noun (sphere)
NounPfx_Noun_State	noun,pfx -> noun (state)
NounPfx_Noun_Stationary	noun,pfx -> noun (stationary)
NounPfx_Noun_Stone	noun,pfx -> noun (stone)
NounPfx_Noun_Suffering	noun,pfx -> noun (suffering)
NounPfx_Noun_Technology	noun,pfx -> noun (technology)
NounPfx_Noun_ThoughtSystem	noun,pfx -> noun (thought system)

NounPfx_Noun_ToKill	noun,pfx -> noun (to kill)
NounPfx_Noun_Turning	noun,pfx -> noun (turning)
NounPfx_Noun_Type	noun,pfx -> noun (type)
NounPfx_Noun_WayOfLife	noun,pfx -> noun (way of life)
NounPfx_Verb_Become	noun,pfx -> verb (become)
NounPfxVerb_Adj_Form	noun,pfx,verb -> adj (form)
NounPfxVerb_Adj_Pertaining	noun,pfx,verb -> adj (pertaining)
NounPfxVerb_Noun_Action	noun,pfx,verb -> noun (action)
NounPlur_Noun_PlurPossessive	noun (plural) -> noun (plural possessive)
NounPrepVerb_Noun_DiminOrPerjor	noun,prep,verb -> noun (diminutive or perjorative)
NounSing_Noun_PlurPossessive	noun (singular) -> noun (plural possessive)
NounVerb_Adj	noun,verb -> adj
NounVerb_Adj_Ability	noun,verb -> adj (ability)
NounVerb_Adj_FullOf	noun,verb -> adj (full of)
NounVerb_Adj_Pertaining	noun,verb -> adj (pertaining)
NounVerb_Adj_Tendency	noun,verb -> adj (tendency)
NounVerb_Adj_TendingTo	noun,verb -> adj (tending to)
NounVerb_Adj_Without	noun,verb -> adj (without)
NounVerb_AdjNoun_ActivityIntensity	noun,verb -> adj,noun (activity intensity)
NounVerb_AdjNoun_FullOf	noun,verb -> adj,noun (full of)
NounVerb_AdjVerb_PresPart	noun,verb -> adj,verb (prespart)
NounVerb_Noun	noun,verb -> noun
NounVerb_Noun_ActionOrCondition	noun,verb -> noun (action or condition)
NounVerb_Noun_Actor	noun,verb -> noun (actor)
NounVerb_Noun_ConditionOrCharacteristics	noun,verb -> noun (condition or characteristics)
NounVerb_Noun_Feminine	noun,verb -> noun (feminine)
NounVerb_Noun_Identification	noun,verb -> noun (identification)
NounVerb_Noun_Identity	noun,verb -> noun (identity)
Pfx_Adj	pfx -> adj
Pfx_Adj_Colour	pfx -> adj (colour)
Pfx_Adj_Eating	pfx -> adj (eating)
Pfx_Adj_Loving	pfx -> adj (loving)
Pfx_Adj_Marriage	pfx -> adj (marriage)
Pfx_Adj_Pertaining	pfx -> adj (pertaining)
Pfx_Adj_Suffering	pfx -> adj (suffering)
Pfx_Adj_TendingTo	pfx -> adj (tending to)
Pfx_Adj_Winged	pfx -> adj (winged)
Pfx_Verb_Cut	pfx -> verb (cut)
Prep_Noun_End	prep -> noun (end)
PersPron_PersPron_Are	pron (personal) -> pron (personal) + 'are' eg. we're, they're
PersPron_PersPron_Did	pron (personal) -> pron (personal) + 'did' eg. I'd, he'd
PersPron_PersPron_Have	pron (personal) -> pron (personal) + 'have' eg. we've, they've
PersPron_PersPron_Will	pron (personal) -> pron (personal) + 'will' eg. I'll, they'll

Verb_Adj	verb -> adj
Verb_Adj_BeingOrBecoming	verb -> adj (being or becoming)
Verb_Adj_Capability	verb -> adj (capability)
Verb_Adj_Causing	verb -> adj (causing)
Verb_Adj_FullOf	verb -> adj (full of)
Verb_AdjNoun_Agency	verb -> adj,noun (agency)
Verb_AdjNoun_FunctionEffectOrPurpose	verb -> adj,noun (function, effect or place)
Verb_Noun	verb -> noun
Verb_Noun_ActionOrCondition	verb -> noun (action or condition)
Verb_Noun_ActionOrState	verb -> noun (action or state)
Verb_Noun_Actor	verb -> noun (actor)
Verb_Noun_Agent	verb -> noun (agent)
Verb_Noun_CommunalActivity	verb -> noun (communal activity)
Verb_Noun_Diminutive	verb -> noun (diminutive)
Verb_Noun_FeminineAgent	verb -> noun (feminine agent)
Verb_Noun_ImperfectResemblance	verb -> noun (imperfect resemblance)
Verb_Noun_State	verb -> noun (state)
Verb_Noun_StateOrCondition	verb -> noun (state or condition)
Verb_Noun_StateOrQuality	verb -> noun (state or quality)
Verb_Verb_Archaic	verb -> verb (archaic)
Verb_Verb_Not	verb -> verb (verb + 'not')
Verb_Verb_Past	verb -> verb (past, pastpart)
Verb_Verb_Past_Noun_Adj	verb -> verb (past, pastpart) / noun -> adj
Verb_Verb_PastPart	verb -> verb (pastpart)
Verb_Verb_Plural	verb -> verb (plural)

Appendix J: Suffix Rules and Associated Transforms:

Suffix rules are in reverse alphabetic order - last character to first character

+'	NounPlur_Noun_PlurPossessive
+s'	NounSing_Noun_PlurPossessive
+ia	Noun_Noun
-iac+ia	Noun_Noun
-ic+ia	Adj_Noun
-e+ia	Noun_Noun
-ical+ia	Adj_Noun
-y+ia	Noun_Noun
-ary+ia	AdjNoun->Noun
+mania	NounPfx_Noun_Passion
+omania	Noun_Noun_Passion
+opia	NounPfx_Noun_Sight
-ia+iana	Noun_Noun_Collection
-e+iana	Noun_Noun_Collection
-ian+iana	Adj_Noun_Collection
+ac	Pfx_Adj_Pertaining
-ia+iac	Noun_Adj_Pertaining
-y+iac	Noun_Adj_Pertaining
+ic	Noun_Adj_Pertaining
-ia+ic	Noun_Adj_Pertaining
+C+ic	Noun_Adj_Pertaining
-e+ic	Noun_Adj_Pertaining
-o+ic	Noun_Adj_Pertaining
-y+ic	NounVerb_Adj_Pertaining
-y+fic	Verb_Adj_Causing
-d+ific	Adj_Adj_Causing
-ify+ific	Verb_Adj_Causing
+pathic	Pfx_Adj_Suffering
+tic	Pfx_Adj_Pertaining
-sis+tic	Noun_Adj_Pertaining
-y+tic	Noun_Adj_Pertaining
+atic	Noun_Adj_Pertaining
-ar+atic	Noun_Adj_Pertaining
-y+etic	Noun_Adj
+istic	AdjNoun_Adj
-a+istic	Noun_Adj
-ic+istic	Noun_Adj
-e+istic	AdjNoun_Adj
+ualistic	Noun_Adj
+lytic	Pfx_Adj
-lysis+lytic	Noun_Adj
+’d	PersPron_PersPron_Did
+d	Verb_Verb_Past
-t+d	Noun_Verb
+ad	NounPfx_Noun_Collection
+head	AdjNoun_Noun
+ed	Verb_Verb_Past&Noun_Adj
+C+ed	Verb_Verb_Past
-e+ed	Verb_Verb_Past
+/-headed	Adj_Adj
-y+ied	Verb_Verb_Past
+red	Noun_Noun_Condition

-e+red	Noun_Noun_Condition
+ised	Adj_Verb_Past
+ualised	Noun_Verb_Past
-e+ated	Adj_Verb_Past
+ized	Adj_Verb_Past
+ualized	Noun_Verb_Past
+oid	Noun_Adj_Like
-e+oid	Noun_Adj_Like
+fold	Adj_AdjAdv_Number
+end	Prep_Noun_End
-two+second	Adj_Adj
+hood	Noun_Noun_State
-y+ihood	AdjAdv_Noun_State
+ard	AdjNounVerb_Noun
+C+ard	AdjNounVerb_Noun
+ward	AdjPrep_AdjAdv_Direction
-ree+ird	Adj_Adj_Number
-t+ce	Adj_Noun
-ny+ce	AdjNoun_Noun
+ice	AdjNounPfxVerb_Noun_State
-e+ice	AdjNounPfxVerb_Noun_State
-y+ice	AdjNounPfxVerb_Noun_State
+ance	AdjAdvVerb_Noun_State
+C+ance	AdjAdvVerb_Noun_State
-ate+ance	Adj_Noun_State
-ant+ance	Adj_Noun_State
-y+iance	Verb_Noun_State
+ulance	Noun_Noun_State
-ty+ulance	Adj_Noun_State
+ence	Verb_Noun_State
+C+ence	Verb_Noun_State
-e+ence	Verb_Noun_State
-ent+ence	Adj_Noun_State
+escence	NounPfx_Noun_State
-esce+escence	Verb_Noun_State
-ept+ipience	Noun_Noun_State
-us+ulence	Noun_Noun_State
+esce	NounPfx_Verb_Become
+ade	NounPfxVerb_Noun_Action
-e+ade	NounPfxVerb_Noun_Action
-h+cade	NounPfxVerb_Noun_Action
+grade	AdjPfx_Adj_Moving
+cide	NounPfx_Noun_ToKill
+ode	NounPfx_Noun_Like
+tude	NounPfx_Noun_Abstract
+ee	AdjNounVerb_Noun_Object
+C+ee	AdjNounVerb_Noun_Object
-e+ee	AdjNounVerb_Noun_Object
-eal+tellee	AdjNounVerb_Noun_Object
+age	NounVerb_Noun
-e+age	NounVerb_Noun
+fuge	NounPfx_Noun_Flight
+ie	AdjNoun_Noun_Hypocoristic
+C+ie	AdjNoun_Noun_Hypocoristic
-e+ie	AdjNoun_Noun_Hypocoristic
+like	Noun_Adj_Like
+able	NounVerb_Adj_Ability
+C+able	NounVerb_Adj_Ability

-e+able	NounVerb_Adj_Ability
-ate+able	NounVerb_Adj_Ability
-y+iable	NounVerb_Adj_Ability
+isable	Adj_Noun_Ability
+izable	Adj_Noun_Ability
+ible	NounVerb_Adj_Ability
+C+ible	NounVerb_Adj_Ability
-e+ible	NounVerb_Adj_Ability
-ge+sible	NounVerb_Adj_Ability
-t+sible	NounVerb_Adj_Ability
+acle	Noun_Noun
-ate+acle	Verb_Noun
-e+ile	Verb_Adj_Capability
+phile	NounPfx_Noun_Lover
-e+tule	NounPfx_Noun_Diminutive
+cule	NounPfx_Noun_Diminutive
+uncule	NounPfx_Noun_Diminutive
+drome	NounPfx_Noun_Running
+chrome	Pfx_Adj_Colour
+some	AdjNounPfxVerb_Adj_Tendency
-an+ane	Adj_Adj
+cene	NounPfx_Noun_New
+tine	NounPfx_AdjNoun_Action
+C+tine	NounPfx_AdjNoun_Action
-e+tine	NounPfx_AdjNoun_Action
+phone	NounPfx_Noun_Sound
+scape	Noun_Noun_View
+scope	NounPfx_Noun_See
+trope	NounPfx_Noun_Turning
+tpe	NounPfx_Noun_Type
+re	PersPron_PersPron_Are
+ware	Adj_Noun_Ware
+sphere	NounPfx_Noun_Sphere
+osphere	NounPfx_Noun_Sphere
+where	Adj_Noun_Where
+metre	NounPfx_Noun_Metre
+litre	NounPfx_Noun_Litre
+ture	NounVerb_Noun_Action
-e+ture	NounVerb_Noun_Action
-ed+dure	NounVerb_Noun_Action
-ce+se	Noun_Verb
-rt+se	Verb_Noun
+ese	Noun_Adj_Place
+ise	AdjNoun_Verb_Action
+C+ise	AdjNoun_Verb_Action
-e+ise	AdjNoun_Verb_Action
-ice+ise	AdjNoun_Verb_Action
-y+ise	AdjNoun_Verb_Action
-t+dise	AdjNoun_Verb_Action
+ualise	AdjNoun_Verb_Action
+wise	Noun_Adv_Direction
+ose	Noun_Adj_FullOf
-e+ose	Noun_Adj_FullOf
-ke+cose	Noun_Adj_FullOf
+iose	Noun_Adj_FullOf
-lysis+lyse	AdjNoun_Verb_Action
-a+ate	AdjNoun_Verb_Action
-e+ate	AdjNoun_Verb_Action

-eal+ellate	NounVerb_Adj_Pertaining
+uate	AdjNoun_Verb_Action
+ite	NounVerb_Noun_Identity
-e+ite	NounVerb_Noun_Identity
-re+site	NounVerb_Noun_Identity
+ette	Noun_Noun_FeminineOrDiminutive
+lyte	NounPfx_Noun_Process
+agogue	NounPfx_Noun_Leading
+esque	Noun_Adj_Manner
-e+esque	Noun_Adj_Manner
+’ve	PersPron_PersPron_Have
-fe+ve	Noun_Verb
-f+ve	Noun_Verb
+ive	NounVerb_Adj_Tendency
-e+ive	NounVerb_Adj_Tendency
-de+sive	NounVerb_Adj_Tendency
+ative	NounVerb_Adj_Tendency
-ate+ative	NounVerb_Adj_Tendency
-y+ative	NounVerb_Adj_Tendency
+itive	NounVerb_Adj_Tendency
-e+itive	NounVerb_Adj_Tendency
+ize	AdjNoun_Verb_Action
+C+ize	AdjNoun_Verb_Action
-e+ize	AdjNoun_Verb_Action
-ice+ize	AdjNoun_Verb_Action
-y+ize	AdjNoun_Verb_Action
-t+dize	AdjNoun_Verb_Action
+ualize	AdjNoun_Verb_Action
-lysis+lyze	AdjNoun_Verb_Action
-ve+f	Verb_Noun
+proof	Noun_Adj_ImperviousTo
+ing	NounVerb_AdjVerb_PresPart
+C+ing	NounVerb_AdjVerb_PresPart
-e+ing	NounVerb_AdjVerb_PresPart
+ling	NounPrepVerb_Noun_DiminOrPerjor
+ising	Adj_AdjVerb_PresPart
+ualising	Noun_AdjVerb_PresPart
-e+ating	Adj_AdjVerb_PresPart
+izing	Adj_AdjVerb_PresPart
+ualizing	Noun_AdjVerb_PresPart
+arch	NounPfx_Noun_Chief
+graph	NounPfx_Noun_Graph
+morph	NounPfx_Noun_Form
+ish	AdjNounPrep_Adj_Like
+C+ish	AdjNounPrep_Adj_Like
-e+ish	AdjNounPrep_Adj_Like
+th	Adj_Adj_Number
-ve+th	Adj_Adj_Number
+path	NounPfx_Noun_Suffering
-e+opath	Noun_Noun_Suffering
+eth	Verb_Verb_Archaic
+C+eth	Verb_Verb_Archaic
-e+eth	Verb_Verb_Archaic
-y+ieth	Adj_Adj_Number
-ve+fth	Adj_Adj_Number
-ong+ength	Adj_Noun
+lith	NounPfx_Noun_Stone
+with	Adv_Adv_Conjunction

-ep+pth	Adj_Noun
+speak	Adj_Noun_Language
+ock	Noun_Noun_Diminutive
+al	AdjNounVerb_Adj_Pertaining
-a+al	AdjNounVerb_Adj_Pertaining
-e+al	AdjNounVerb_Adj_Pertaining
-us+al	AdjNounVerb_Adj_Pertaining
+ical	NounPfx_Adj
-e+ical	NounPfx_Adj
-y+ical	NounPfx_Adj
-r+tical	NounPfx_Adj
-cy+tical	NounPfx_Adj
+atical	NounPfx_Adj
+istical	NounPfx_Adj
-a+istical	NounPfx_Adj
-e+istical	NounPfx_Adj
+C+ial	NounPfxVerb_Adj_Pertaining
-ary+C+ial	NounPfxVerb_Adj_Pertaining
-is+ial	NounPfxVerb_Adj_Pertaining
-y+ial	NounPfxVerb_Adj_Pertaining
-e+ential	NounPfxVerb_Adj_Pertaining
+ional	NounPfxVerb_Adj_Pertaining
-ed+dural	NounPfxVerb_Adj_Pertaining
+ual	NounPfxVerb_Adj_Pertaining
+eval	AdjNounVerb_Adj_Pertaining
-e+rel	NounPrepVerb_Noun_DiminOrPerjor
+erel	NounPrepVerb_Noun_DiminOrPerjor
+phil	NounPfx_Noun_Lover
+’ll	PersPron_PersPron_Will
+ful	NounVerb_AdjNoun_FullOf
-y+iful	NounVerb_AdjNoun_FullOf
-ve+iful	NounVerb_AdjNoun_FullOf
+gram	NounPfx_Noun_Drawn
+dom	Noun_Noun_DomainOrCondition
+form	NounPfxVerb_Adj_Form
-e+iasm	Verb_Noun
+ism	AdjNoun_Noun_ActionOrCondition
-ic+ism	Adj_Noun_ActionOrCondition
-e+ism	AdjNoun_Noun_ActionOrCondition
-ise+ism	Verb_Noun_ActionOrCondition
-ive+ism	Adj_Noun_ActionOrCondition
-ize+ism	Verb_Noun_ActionOrCondition
-ist+ism	Noun_Noun_ActionOrCondition
-y+ism	AdjNoun_Noun_ActionOrCondition
+ualism	Noun_Noun_ActionOrCondition
-e+atism	Verb_Noun_ActionOrCondition
+endum	Verb_Noun
+an	NounPfx_AdjNoun_Belonging
-a+an	NounPfx_AdjNoun_Belonging
+ian	NounPfx_AdjNoun_Belonging
-ia+ian	NounPfx_AdjNoun_Belonging
-e+ian	NounPfx_AdjNoun_Belonging
-y+ian	NounPfx_AdjNoun_Belonging
+bian	NounPfx_AdjNoun_Belonging
+ician	NounPfx_AdjNoun_Expert
-ic+ician	NounPfx_AdjNoun_Expert
-y+ician	NounPfx_AdjNoun_Expert
-e+arian	NounPfx_AdjNoun_Belonging

-y+arian	NounPfx_AdjNoun_Belonging
-ary+arian	NounPfx_AdjNoun_Belonging
-ory+orian	NounPfx_AdjNoun_Expert
+sman	Noun_Noun_Expert
+en	AdjNoun_Verb
+C+en	AdjNoun_Verb
-e+en	Verb_Verb_PastPart
-er+en	AdjNoun_Verb
+teen	Adj_Adj_Number
-t+teen	Adj_Adj_Number
+ren	Verb_Verb_Plural
-other+ethren	Verb_Verb_Plural
+/-in	Verb_Noun_CommunalActivity
+kin	Noun_Noun_Diminutive
+gon	NounPfx_Noun_GeometricalShape
+ion	NounVerb_Noun_ActionOrCondition
-e+ion	AdjNounVerb_Noun_ActionOrCondition
-ite+ion	Verb_Noun_ActionOrCondition
-ect+icion	Verb_Noun_ActionOrCondition
-d+sion	Verb_Noun_ActionOrCondition
-ce+sion	NounVerb_Noun_ActionOrCondition
-de+sion	Verb_Noun_ActionOrCondition
-ge+sion	Verb_Noun_ActionOrCondition
-se+sion	AdjNounVerb_Noun_ActionOrCondition
-t+sion	Verb_Noun_ActionOrCondition
-e+ision	AdjNounVerb_Noun_ActionOrCondition
-el+ulsion	Verb_Noun_ActionOrCondition
-ine+ension	Verb_Noun_ActionOrCondition
-it+ission	Verb_Noun_ActionOrCondition
+tion	Verb_Noun_ActionOrCondition
-e+tion	Verb_Noun_ActionOrCondition
-l+tion	Adj_Noun_ActionOrCondition
+ation	AdjNounVerb_Noun_ActionOrCondition
-a+ation	NounVerb_Noun_ActionOrCondition
-e+ation	NounVerb_Noun_ActionOrCondition
-y+ication	Verb_Noun_ActionOrCondition
-fy+fication	Verb_Noun_ActionOrCondition
-e+ification	Adj_Noun_ActionOrCondition
-ify+ification	Verb_Noun_ActionOrCondition
+isation	Adj_Noun_ActionOrCondition
-e+isation	Adj_Noun_ActionOrCondition
-ise+isation	Verb_Noun_ActionOrCondition
+uation	NounVerb_Noun_ActionOrCondition
+ization	Adj_Noun_ActionOrCondition
-e+ization	Adj_Noun_ActionOrCondition
-ize+ization	Verb_Noun_ActionOrCondition
-fy+faction	Verb_Noun_ActionOrCondition
-ear+arition	Verb_Noun_ActionOrCondition
-ive+ption	Verb_Noun_ActionOrCondition
-ve+ution	Verb_Noun_ActionOrCondition
+hedron	NounPfx_Noun_GeometricalSolid
+person	Noun_Noun_Person
+ton	Adj_Noun_Person
+ern	AdjNoun_Adj_Direction
+ship	AdjNoun_Noun_StateOrCondition
-th+ship	AdjNoun_Noun_StateOrCondition
+manship	AdjAdvNounVerb_Noun_Proficiency
+smanship	AdjAdvNounVerb_Noun_Proficiency

+up	NounVerb_AdjNoun_ActivityIntensity
+/-up	NounVerb_AdjNoun_ActivityIntensity
+ar	Noun_Adj_Pertaining&Verb_Noun_Agent
+C+ar	Noun_Adj_Pertaining&Verb_Noun_Agent
-e+ar	Noun_Adj_Pertaining&Verb_Noun_Agent
-al+lar	Noun_Adj_Pertaining&Verb_Noun_Agent
-le+ular	Noun_Adj_Pertaining&Verb_Noun_Agent
-ule+ular	Noun_Adj_Pertaining&Verb_Noun_Agent
+er	AdjNounVerb_Noun_Identification
+C+er	AdjNounVerb_Noun_Identification
-e+C+er	AdjNounVerb_Noun_Identification
-e+er	AdjNounVerb_Noun_Identification
-y+er	AdjNounVerb_Noun_Identification
-ry+er	AdjNounVerb_Noun_Identification
+der	AdjNounVerb_Noun_Identification
+eer	Noun_Noun_Identification
-e+eer	Noun_Noun_Identification
-e+ifer	Noun_Noun_Producing
+ier	NounVerb_Noun_Identification
-e+ier	NounVerb_Noun_Identification
-y+ier	NounVerb_Noun_Identification
-ry+ier	NounVerb_Noun_Identification
+erer	AdjNounVerb_Noun_Identification
-ry+erer	AdjNounVerb_Noun_Identification
+ater	Noun_Noun_Identification
+olater	Noun_Noun_WorshiperOf
-ol+olater	Noun_Noun_WorshiperOf
+meter	NounPfx_Noun_MeasuringInstrument
-p+C+meter	NounPfx_Noun_MeasuringInstrument
-y+imeter	NounPfx_Noun_MeasuringInstrument
+ometer	NounPfx_Noun_MeasuringInstrument
+ster	AdjNounVerb_Noun_Identification
+aster	NounPfx_Noun_ImperfectResemblance
+yer	NounVerb_Noun_Identification
+or	NounVerb_Noun_Actor
+C+or	NounVerb_Noun_Actor
-e+or	NounVerb_Noun_Actor
-ate+or	Verb_Noun_Actor
-y+or	NounVerb_Noun_Actor
+C+ior	NounVerb_Noun_Actor
-ire+eror	NounVerb_Noun_Actor
-e+ator	Adj_Noun_Actor
+ceptor	NounPfx_Noun_Actor
+saur	NounPfx_Noun_Lizard
+our	Verb_Noun_StateOrCondition
-e+our	Verb_Noun_StateOrCondition
-e+iour	Verb_Noun_StateOrCondition
+’s	Noun_Noun_Possessive
+s	Noun_Noun_Plural_Verb_Verb_3PersActive
+mas	Noun_Noun_FeastOrHoliday
+ics	Noun_Noun_ScienceOrArt
-e+ics	Noun_Noun_ScienceOrArt
-o+ics	Noun_Noun_ScienceOrArt
-y+ics	Noun_Noun_ScienceOrArt
-y+tics	Noun_Noun_ScienceOrArt
-a+atics	Noun_Noun_ScienceOrArt
-ar+atics	Noun_Noun_ScienceOrArt
+es	Noun_Noun_Plural_Verb_Verb_3PersActive

-e+es	Noun_Noun_Plural__Verb_Verb_3PersActive
-istes	Noun_Noun_Plural
-ex+ices	Noun_Noun_Plural
-ix+ices	Noun_Noun_Plural
-y+ies	Noun_Noun_Plural__Verb_Verb_3PersActive
-fe+ves	Noun_Noun_Plural__Verb_Verb_3PersActive
-f+ves	Noun_Noun_Plural__Verb_Verb_3PersActive
+polis	NounPfx_Noun_City
+osis	NounPfx_Noun_ActionOrCondition
+biosis	NounPfx_Noun_WayOfLife
+gnosis	NounPfx_Noun_Recognition
+lysis	NounPfx_Noun_Decomposition
+itis	NounPfx_Noun_Inflammation
+ess	NounVerb_Noun_Feminine
+C+ess	NounVerb_Noun_Feminine
-e+ess	NounVerb_Noun_Feminine
-er+ess	NounVerb_Noun_Feminine
-or+ess	NounVerb_Noun_Feminine
-y+ess	NounVerb_Noun_Feminine
+less	NounVerb_Adj_Without
-y+iless	NounVerb_Adj_Without
+ness	Adj_Noun_StateOrQuality
-e+ableness	Verb_Noun_StateOrQuality
+ibleness	Noun_Noun_StateOrQuality
+iveness	Noun_Noun_StateOrQuality
-y+iness	Adj_Noun_StateOrQuality
-ve+fullness	Verb_Noun_StateOrQuality
-ire+ress	NounVerb_Noun_Feminine
-eror+ress	NounVerb_Noun_Feminine
-ter+tress	NounVerb_Noun_Feminine
-tor+tress	NounVerb_Noun_Feminine
+stress	NounVerb_Noun_Feminine
-ulate+ulus	Verb_Noun_Diminutive
-cule+culus	Noun_Noun_Diminutive
+unculus	NounPfx_Noun_Diminutive
+ous	Noun_Adj_FullOf
-e+ous	Noun_Adj_FullOf
-ar+ous	Adj_Adj_FullOf
-y+ous	Noun_Adj_FullOf
-ety+ous	Noun_Adj_FullOf
-ity+ous	Noun_Adj_FullOf
-arity+ous	Noun_Adj_FullOf
+eous	Noun_Adj_FullOf
-y+eous	Noun_Adj_FullOf
+C+ious	Noun_Adj_FullOf
-ion+ious	Noun_Adj_FullOf
-io+ious	Noun_Adj_FullOf
-y+ious	NounVerb_Adj_FullOf
-iety+ious	Noun_Adj_FullOf
-ity+ious	Noun_Adj_FullOf
-iosity+ious	Noun_Adj_FullOf
-acy+acious	Noun_Adj_FullOf
-acity+acious	Noun_Adj_FullOf
-ocity+ocious	Noun_Adj_FullOf
-ite+itious	Verb_Adj_FullOf
-ition+itious	Noun_Adj_FullOf
+philous	Pfx_Adj_Loving
+ulous	Pfx_Adj_TendingTo

-ke+ulous	NounVerb_Adj_TendingTo
+gamous	Pfx_Adj_Marriage
-er+rous	Noun_Adj_FullOf
+erous	Noun_Adj_FullOf
+iferous	Noun_Adj_Producing
-ic+iferous	Adj_Adj_Producing
-e+iferous	Noun_Adj_Producing
+pteros	Pfx_Adj_Winged
-our+orous	Noun_Adj_FullOf
+vorous	Pfx_Adj_Eating
+ways	Noun_Adv_Manner
+n't	Verb_Verb_Not
-ll+n't	Verb_Verb_Not
-n+n't	Verb_Verb_Not
-ill+on't	Verb_Verb_Not
-d+t	Verb_Verb_Past
-se+t	AdjNounVerb_Verb
-ve+t	Verb_Noun
-cy+t	Noun_Adj
+crat	NounPfx_Noun_Ruler
+stat	NounPfx_Noun_Stationary
+sect	Pfx_Verb_Cut
+et	Noun_Noun_Diminutive
-e+et	Noun_Noun_Diminutive
+let	Noun_Noun_Diminutive
+tight	Noun_Adj_ImperviousTo
+ant	Verb_AdjNoun_Agency
-e+ant	Verb_AdjNoun_Agency
-ance+ant	Noun_Adj
-ate+ant	Verb_AdjNoun_Agency
-eal+ellant	Verb_AdjNoun_Agency
-ty+ulant	Adj_Adj
+ent	Verb_AdjNoun_Agency
-e+ent	Verb_AdjNoun_Agency
-ence+ent	Verb_AdjNoun_Agency
-ate+ent	Verb_AdjNoun_Agency
-ency+ent	Verb_AdjNoun_Agency
+escent	NounPfx_Adj_BeingOrBecoming
-esce+escent	Verb_Adj_BeingOrBecoming
+ulent	Noun_Adj_AboundingIn
-us+ulent	Noun_Adj_AboundingIn
+ment	Verb_Noun_ActionOrState
-ept+ipent	Noun_Noun_Agent
-ear+arent	Verb_Adj_BeingOrBecoming
-ve+pt	Verb_Noun
-ive+pt	Verb_Noun
+C+art	Verb_Noun
+ast	NounPfx_Noun_ImperfectResemblance
-ary+ast	Adj_Noun_ImperfectResemblance
-e+iast	Verb_Noun_ImperfectResemblance
+est	Adj_Adj_Superlative
+C+est	Adj_Adj_Superlative
-e+est	Adj_Adj_Superlative
+fest	Noun_Noun_Festivity
-y+iest	Adj_Adj_Superlative&Adv_Adv_Superlative
+ist	AdjNounVerb_Noun_Practitioner
-a+ist	AdjNounVerb_Noun_Practitioner
-e+ist	AdjNounVerb_Noun_Practitioner

-ise+ist	AdjNounVerb_Noun_Practitioner
-ize+ist	AdjNounVerb_Noun_Practitioner
-ism+ist	AdjNounVerb_Noun_Practitioner
-y+ist	AdjNounVerb_Noun_Practitioner
+ualist	AdjNounVerb_Noun_Practitioner
+nist	AdjNounVerb_Noun_Practitioner
+ionist	AdjNounVerb_Noun_Practitioner
+tist	AdjNounVerb_Noun_Practitioner
+most	AdjPrep_Adj_Superlative
-one+first	Adj_Adj_Number
-e+trix	Verb_Noun_FeminineAgent
-or+trix	Noun_Noun_FeminineAgent
+y	AdjNounVerb_Adj_CharacterisedBy
+C+y	AdjNounVerb_Adj_CharacterisedBy
-e+y	AdjNounVerb_Adj_CharacterisedBy
+cy	AdjNoun_Noun_ConditionOrRank
-tic+cy	AdjNoun_Noun_ConditionOrRank
-ce+cy	AdjNoun_Noun_ConditionOrRank
-te+cy	AdjNoun_Noun_ConditionOrRank
-t+cy	AdjNoun_Noun_ConditionOrRank
-atic+acy	AdjNounVerb_Noun_StateOrQuality
-e+acy	AdjNounVerb_Noun_StateOrQuality
-ate+acy	AdjNounVerb_Noun_StateOrQuality
-at+acy	AdjNounVerb_Noun_StateOrQuality
+cracy	NounPfx_Noun_Government
+ancy	AdjNounVerb_Noun_StateOrQuality
-e+ancy	Verb_Noun_StateOrQuality
-ant+ancy	Adj_Noun_StateOrQuality
-y+ancy	Verb_Noun_StateOrQuality
-ty+ulancy	Adj_Noun_StateOrQuality
-e+ency	Verb_Noun_StateOrQuality
-us+ulency	Noun_Noun_StateOrQuality
+ey	AdjNounVerb_Adj_CharacterisedBy
-id+efy	AdjNoun_Verb_CauseToBe
+ify	AdjNoun_Verb_CauseToBe
-ific+ify	AdjNoun_Verb_CauseToBe
-e+ify	AdjNoun_Verb_CauseToBe
-iful+ify	AdjNoun_Verb_CauseToBe
-y+ify	AdjNoun_Verb_CauseToBe
+logy	NounPfx_Noun_Knowledge
+ology	NounPfx_Noun_Knowledge
+inology	NounPfx_Noun_Knowledge
+urgy	NounPfx_Noun_Technology
+archy	NounPfx_Noun_Government
+graphy	NounPfx_Noun_Drawing
+sophy	NounPfx_Noun_ThoughtSystem
+pathy	NounPfx_Noun_Suffering
+ly	AdjNounVerb_Adv
-e+ly	AdjNounVerb_Adv
-le+ly	AdjNounVerb_Adv
+ably	AdjNounVerb_Adv
-e+ably	AdjNounVerb_Adv
-able+ably	AdjNounVerb_Adv
+ibly	AdjNounVerb_Adv
-ible+ibly	AdjNounVerb_Adv
+ively	AdjNounVerb_Adv
-e+ingly	AdjNounVerb_Adv
-y+ily	AdjNounVerb_Adv

-ial+ially	AdjNounVerb_Adv
-y+ially	AdjNounVerb_Adv
+ually	AdjNounVerb_Adv
-ual+ually	AdjNounVerb_Adv
-ve+fully	AdjNounVerb_Adv
+/-ply	Adj_AdjNoun_Ply
+gamy	NounPfx_Noun_Marriage
+nomy	NounPfx_Noun_Distribution
+tomy	NounPfx_Noun_Cutting
-ix+ectomy	NounPfx_Noun_Cutting
+thermy	NounPfx_Noun_Heat
+phany	NounPfx_Noun_Manifestation
+geny	NounPfx_Noun_Origin
+gony	NounPfx_Noun_Origination
+phony	NounPfx_Noun_Sound
+mony	NounPfx_Noun_ConditionOrResult
+scopy	NounPfx_Noun_Seeing
+ry	AdjNounVerb_Noun_ConditionOrPractice
-ltry	AdjNounVerb_Noun_ConditionOrPractice
-ertry	AdjNounVerb_Noun_ConditionOrPractice
+ary	AdjNounPfxVerb_AdjNoun_FunctionLocationOrRelation
-etary	AdjNounPfxVerb_AdjNoun_FunctionLocationOrRelation
-ate+ary	AdjNounPfxVerb_AdjNoun_FunctionLocationOrRelation
-nial+ary	AdjNounPfxVerb_AdjNoun_FunctionLocationOrRelation
-an+ary	AdjNounPfxVerb_AdjNoun_FunctionLocationOrRelation
-eer+ary	AdjNounPfxVerb_AdjNoun_FunctionLocationOrRelation
-ant+ary	AdjNounPfxVerb_AdjNoun_FunctionLocationOrRelation
-ample+emplary	AdjNounPfxVerb_AdjNoun_FunctionLocationOrRelation
-ain+anary	AdjNounPfxVerb_AdjNoun_FunctionLocationOrRelation
-our+orary	AdjNounPfxVerb_AdjNoun_FunctionLocationOrRelation
+ery	AdjNounVerb_Noun
+C+ery	AdjNounVerb_Noun
-etary	AdjNounVerb_Noun
-er+ery	AdjNounVerb_Noun
-ler+ery	AdjNounVerb_Noun
+ory	Verb_AdjNoun_FunctionEffectOrPlace
-etary	Verb_AdjNoun_FunctionEffectOrPlace
-el+ulsory	Verb_AdjNoun_FunctionEffectOrPlace
-etary	Verb_AdjNoun_FunctionEffectOrPlace
+atry	Noun_Noun_WorshipOf
+iatry	NounPfx_Noun_MedicalCare
+olatry	Noun_Noun_WorshipOf
-ol+olatry	Noun_Noun_WorshipOf
+metry	NounPfx_Noun_Measuring
+ty	Adj_Adj_Number
-te+ty	Verb_Noun_StateOrQuality
-al+ty	Adj_Noun_StateOrQuality
-ive+ifty	Adj_Adj_Number
+ity	Adj_Noun_ConditionOrCharacteristics
-e+ity	Adj_Noun_ConditionOrCharacteristics
-acious+acity	Adj_Noun_QualityOrStateOfBeing
-ocious+ocity	Adj_Noun_QualityOrStateOfBeing
-e+ility	NounVerb_Noun_ConditionOrCharacteristics
-ice+ility	NounVerb_Noun_ConditionOrCharacteristics
-le+ility	Adj_Noun_ConditionOrCharacteristics
-ble+ility	Adj_Noun_ConditionOrCharacteristics
-ile+ility	Adj_Noun_ConditionOrCharacteristics
-il+ility	Adj_Noun_ConditionOrCharacteristics

-ble+bility	Adj_Noun_ConditionOrCharacteristics
+ability	NounVerb_Noun_ConditionOrCharacteristics
-e+ability	NounVerb_Noun_ConditionOrCharacteristics
-able+ability	Adj_Noun_ConditionOrCharacteristics
-ate+ability	NounVerb_Noun_ConditionOrCharacteristics
+ibility	NounVerb_Noun_ConditionOrCharacteristics
-ible+ibility	Adj_Noun_ConditionOrCharacteristics
-emy+mity	Noun_Noun_StateOrQuality
+osity	NounVerb_Noun_ConditionOrCharacteristics
-ose+osity	Adj_Noun_ConditionOrCharacteristics
-us+osity	Adj_Noun_ConditionOrCharacteristics
-ous+osity	Adj_Noun_ConditionOrCharacteristics
+iosity	Adj_Noun_ConditionOrCharacteristics
-iose+iosity	Adj_Noun_ConditionOrCharacteristics
+evity	Adj_Noun_ConditionOrCharacteristics
+ivity	AdjNounVerb_Noun_ConditionOrCharacteristics
-e+ivity	AdjNounVerb_Noun_ConditionOrCharacteristics
-ive+ivity	Adj_Noun_ConditionOrCharacteristics
-wo+wenty	Adj_Adj_Number
-ree+irty	Adj_Adj_Number
-our+orty	Adj_Adj_Number

APPENDIX K: SARLIB Library Routines and Error Codes

The SARLIB library consists of a Turbo Pascal unit called SAR10, containing routines to carry out the functions shown in Table K.1:

<p>Train a speech recognition pattern Update a speech recognition pattern Digitise or record an audio response Output an audio response Perform speech recognition Change recognition cluster Change reject threshold Delete speech recognition or audio response patterns Upload speech recognition or audio response patterns Download speech recognition or audio response patterns Download a speech recognition vocabulary Set or inquire about parameters or flags Inquire about speech recognition or audio response status Inquire about error status Various control and memory housekeeping functions</p>
--

Table K.1 Facilities Provided by SARLIB

These routines may be grouped into three categories as follows:

- Control and test routines, listed in Table K.2,
- Speech recognition routines, listed in Table K.3,
- Audio response routines, listed in Table K.4.

<p>SarErrorStatus - obtain error code from SAR-10 SarSetRespFormatFlag - set SAR-10 response format flag SarGetRespFormatFlag - return SAR-10 response format flag SarChangeMemory - change contents of SAR-10 memory SarDumpMemory - dump contents of SAR-10 memory SarTestWorkMem - test SAR-10 work memory SarTestSRRefPatMem - test SAR-10 SR reference pattern memory SarTestARSpeechPatMem - test SAR-10 AR speech pattern memory SarInitialise - initialise the SAR-10 SarCancel - quit SAR-10 command execution SarPause - stop transmitting pattern data from SAR-10 to PC SarResum - resume transmitting pattern data from SAR-10 to PC SarBeep - cause SAR-10 speaker to beep</p>

Table K.2 Control and Test Routines

SarTrain - train SAR-10 for speech recognition
SarUpdate - update specified speech recognition reference pattern
SarRecogOne - recognise first reference pattern candidate
SarRecogTwo - recognise first & second reference pattern candidates
SarRecogOneAllClust - recognise first reference pattern candidate (all clusters)
SarRecogTwoAllClust - recognise first & second reference pattern candidates (all clust.)
SarRecogOneZeroClust - recognise first reference pattern candidate (cluster 0)
SarRecogTwoZeroClust - recognise first & second reference pattern candidates (clust 0)
SarStartRecog - put SAR-10 into recognition mode
SarStartRecogAllClust - put SAR-10 into recognition mode (all clusters)
SarStartRecogZeroClust - put SAR-10 into recognition mode (cluster 0)
SarSetRecogClusters - set SAR-10 recognition clusters
SarSetAllRecogClusters - set all SAR-10 recognition clusters
SarSetWordRejectThresh - set recognition reject threshold for a word
SarSetAllRejectThresh - set recognition reject threshold for all words
SarGetAllRejectThresh - return recognition reject threshold for all words
SarDeleteAllSRPatterns - delete all SR reference patterns
SarDeleteSRWordPatterns - delete SR reference patterns for a word
SarDeleteSRWordLastPattern - delete last SR reference pattern for a word
SarDeleteSRWordNthPattern - delete Nth SR reference pattern for a word
SarUploadSRPattern - upload one SR word vocab entry & reference pattern
SarUploadAllSRPatterns - upload all SR vocab entries & reference patterns
SarDownloadSRPattern - download one SR word vocab entry & reference pattern
SarDownloadAllSRPatterns - download all SR vocab entries & reference patterns
SarDownloadAllSRVocab - download all SR vocab entries to SAR-10
SarSetSRParameters - set SAR-10 recognition parameters
SarGetSRParameters - return SAR-10 recognition parameters
SarSetRecogFlags - set SAR-10 recognition flags
SarGetRecogFlags - return SAR-10 recognition flags
SarGetSRStatus - return SAR-10 SR status

Table K.3 Speech Recognition Routines

SarDigitise - digitise speech for SAR-10 audio output
SarRecord - record speech for SAR-10 audio output
SarAROutputOneWord - produce one word of SAR-10 audio output
SarAROutputOneMacro - produce one macro of SAR-10 audio output
SarAROutputOnePause - produce one pause in mSec in SAR-10 audio output
SarAROutputWordArray - produce a word array of SAR-10 audio output
SarAROutputMacroArray - produce a macro array of SAR-10 audio output
SarARDefineMacro - define a SAR-10 audio output macro
SarGetARMacro - return a SAR-10 audio output macro
SarDeleteARMacros - delete all SAR-10 audio output macros
SarDeleteOneARPattern - delete one SAR-10 AR speech pattern
SarDeleteAllARPatterns - delete all SAR-10 AR speech patterns
SarUploadARPatterns - upload one AR speech pattern
SarUploadAllARPatterns - upload all AR speech patterns
SarDownloadARPattern - download one AR speech pattern
SarDownloadAllARPatterns - download all AR speech patterns
SarSetARParameters - set SAR-10 AR parameters
SarGetARParameters - return SAR-10 AR parameters
SarGetARStatus - return SAR-10 AR status

Table K.4 Audio Response Routines

Error conditions cause the error codes shown in Table K.5 to be returned.

SAR-10 Errors:

- 000 -- No error
- 001 -- Command error - illegal command format error
- 002 -- Command parameter error - parameter out of range
- 010 -- Training error - SR reference pattern/AR speech pattern not trained
- 020 -- Memory full error - SR reference pattern/AR speech pattern memory full
- 022 -- Memory full error - SR output code memory full
- 030 -- Load data error - SR reference pattern/AR speech pattern load data error
- 040 -- Memory write error - memory write inhibited
- 050 -- Memory test error - bad memory
- 060 -- Response macro error - macro not defined

System Errors:

- 100 -- Unknown error
- 101 -- Wrong response from upload

Table K.5 SARLIB Error Codes