Aston University

Some pages of this thesis may have been removed for copyright restrictions.

If you have discovered material in AURA which is unlawful e.g. breaches copyright, (either yours or that of a third party) or any other law, including but not limited to those relating to patent, trademark, confidentiality, data protection, obscenity, defamation, libel, then please read our Takedown Policy and contact the service immediately

The Use of Deterministic Parsers on Sublanguage for Machine Translation

Jacqueline McEwan Archibald

Doctor of Philosophy

The University of Aston in Birmingham

August 1992

c This copy of the Thesis has been supplied on condition that anyone who consults it is understood to recognize that its copyright rests with its author and that no quotation from the thesis and no information derived from it may be published without the author's prior, written consent.

The University of Aston in Birmingham

The Use of Deterministic Parsers on Sublanguage for Machine Translation

Jacqueline McEwan Archibald

Doctor of Philosophy

August 1992

Summary

For more than forty years, research as been on going in the use of the computer in the processing of natural language. During this period methods have evolved, with various parsing techniques and grammars coming to prominence. Problems still exist, not least in the field of Machine Translation. However, one of the successes in this field is the translation of sublanguage.

The present work reports Deterministic Parsing, a relatively new parsing technique, and its application to the sublanguage of an aircraft maintenance manual for Machine Translation. The aim has been to investigate the practability of using Deterministic Parsers in the analysis stage of a Machine Translation system.

Machine Translation, Sublanguage and parsing are described in general terms. with a review of Deterministic Parsing systems, pertinent to this research, being presented in detail. The interaction between Machine Translation, Sublanguage and Parsing, including Deterministic Parsing, is also highlighted.

Two types of Deterministic Parser have been investigated, a Marcus-type parser, based on the basic design of the original Deterministic Parser (Marcus, 1980) and an LR-type Deterministic Parser for natural language, based on the LR parsing algorithm. In total, four Deterministic Parsers have been built and are described in the thesis. Two of the Deterministic Parsers are prototypes from which the remaining two parsers to be used on sublanguage have been developed.

This thesis reports the results of parsing by the prototypes, a Marcus-type parser and an LR-type parser which have a similar grammatical and linguistic range to the original Marcus parser. The Marcus-type parser uses a grammar of production rules, whereas the LR-type parser employs a Definite Clause Grammar(DCG).

Also, reported are the results of parsing using the Deterministic Parsers for sublanguage on the noun-phrases from an aircraft maintenance manual. This includes a discussion of how to deal with complex nouns, which appear in the aircraft maintenance manual.

Keywords: Natural Language Processing, Parsing Strategies
Deterministic Parsers. Sublanguage, Machine Translation, Artificial Intelligence.

To my Parents

and

in memory of my Gran
Sarah (Cis) Walker Snaddon

ACKNOWLEDGEMENTS

I would like to express my gratitude to the following people:

Dr Peter Hancox, my associate supervisor, for his support and guidance and for having faith in me and my work, at those times, when I did not have it myself,

Mum, Dad, Iain and other members of my family for their endless support and encouragement in this undertaking and everything that I have done,

Dr Abdellah Salhi for his constant support and encouragement,

Dr Brian Gay for his help and guidance

all members of the Academic and Technical Support Staff of Department of Computer Science & Applied Mathematics at Aston University,

all colleagues and former colleagues of MB268, Aston University for their advice, suggestions and friendship especially Mr Richard Gatward and Dr Neil Simpkins.

Finally, I acknowledge the financial support provided by an Aston University Studentship.

LIST OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1

INTRODUCTION

1.1 Aims of the Research

Since the immediate post-war period, the use of the computer in processing natural language has occupied many researchers. The first major systems built for dealing with natural language were Machine Translation systems. Machine Translation systems were built to translate one natural language into another natural language, e.g., Russian into English. Expectations of the results that these systems would produce proved to be too optimistic, as it became apparent that fully automatic high quality translation was virtually impossible. However, although research in Machine Translation floundered in the 1960's, a by-product of this research began to gain credence; this was the more general area of Computational Linguistics. The field of Computational Linguistics deals, in part, with theories of grammar, grammar formalisms and parsing techniques that can be computerised to perform natural language processing. Since the beginning of research in Computational Linguistics several notable grammar formalisms and parsing techniques have been developed.

Today's researchers in Machine Translation, which has quietly re-emerged as an area for research, and Computational Linguistics are building both theoretical and practical natural language systems. The majority of practical systems built are Machine Translation systems and Natural Language Front Ends to Databases.

At present, parsing methods used by the majority of Machine Translation systems either have been of a non-deterministic nature, e.g., ATNs that employ back-tracking or the parsing method has used pseudo-parallelism, such as, Chart Parsing systems. This research involves the investigation of the use of a relatively untried method of parsing in a practical situation, such as Machine Translation, namely Deterministic Parsing.

The Deterministic Parsing of natural language was put forward by Marcus (1980) in opposition to backtracking non-deterministic systems and systems that worked in parallel. He developed the first Deterministic Parser, PARSIFAL, dedicated to parsing Natural Language. Marcus believed that the Deterministic Parser reflected how humans parse - humans did not back-track or process in parallel. Other researchers, namely Shieber (1983) and Pereira(1985) have also investigated Deterministic Parsing using an LR type Deterministic Parser for the processing of natural language. Both types of Deterministic Parser, i.e., the Marcus and LR type, in the main have been used as components of theoretical systems.

As Deterministic Parsers have been used significantly as part of theoretical systems, it was felt that there was a need to investigate their use in a practical situation; the logical step of putting theory into practise. Specifically, the aim of this research is to evaluate the practability of using these Deterministic Parsers in the analysis stage of a Machine Translation system with an example sublanguage of an Aircraft Maintenance Manual serving as the source language text. The use of a technical sublanguage adds to the practability aspect of the research, as it is an example of the type of text that has already been used successfully in Machine Translation. (Machine Translation and Sublanguage are discussed in more detail below).

To fulfil the aims of the research, four parsers have been constructed. Two of these four parsers are prototypes, one of which is of the Marcus type, MParser, the other of the LR type, LParser designed to process natural language. The Marcus type Deterministic Parser follows the basic design of Marcus' parser with amendments made to enhance the functioning. The grammar of production rules used is similar to that used by Berwick (1985), itself based on Marcus (1980). The LR type Deterministic Parser, LParser, is based on the LALR(1) algorithm. Modifications have been made to allow it to cope with natural language. LParser is a substantial extension of the Shieber (1983) and Pereira (1985) parser. The grammar used written in a Declarative Clause Grammar (DCG) formalism. The construction, functioning and linguistic range of both these parsers are discussed in the thesis.

From these prototypes, two Deterministic Parsers have been used on the sublanguage of a Rolls-Royce aircraft maintenance manual, as part of the analysis stage of a Machine Translation system, have been developed, i.e., a Marcus type parser, MParserSub, for sublanguage and an LR type parser, LParserSub, for sublanguage. Both parsers use similar type grammars to their prototypes. The Deterministic Parsers applied to the sublanguage only process the noun-phrases of that sublanguage. However, these noun-phrases encapsulate many differing patterns of phrases, thus providing the Deterministic Parsers with a wide grammatical range to process. The thesis also reports the construction, functioning and linguistic range of the two parsers used on the sublanguage.

Having defined the aims of the thesis, the rest of the chapter concentrates on defining and discussing the major areas that are pertinent to the research and put it into context, namely Machine Translation, Parsing and Sublanguage. The final section of the chapter highlights the contents of the remaining chapters of the thesis.

1.2 Machine Translation

Machine Translation (MT) involves the use of a computer to translate one language into another, i.e., a source language into a target language. In the early days of MT the strategy used was direct translation. This involved translating sentences in stages with the output of one stage being the input to the next stage. There was no analysis of the source text, no linguistic theory or parsing method was used. The system worked by processing large bi-lingual dictionaries, analysing word categories and their morphological endings and text processing.

The method of Machine Translation used by the vast majority MT systems since the 1960's is the indirect method. The term indirect relates to the fact that there is not a direct link between source and target language. Two strategies of indirect MT exist - interlingual and transfer. An interlingua can be described as a language-independent representation of semantico-syntactic structures that produces a relevant meaning of both source language and target language text. The interlingual approach to translation has two stages - the analysis and synthesis stages. The analysis stage of the system processes the source language into an interlingua and the synthesis stage generates the target language text from the interlingua. Both the analysis and synthesis stages use their own dictionaries and grammars.

It is the second indirect method, the transfer strategy of MT which the research parsers to be reported in this thesis would be part. MT systems that use the transfer strategy function in three stages - analysis, transfer and synthesis. Each stage processes independently with its own dictionary. An important concept of transfer strategy systems is the flow of information from source language to target language; the flow of information passes from the analysis of the source language through transfer into target language structures, to the final stage of the generation of the translation. The analysis stage can comprise modules for morphological analysis, syntactic analysis and semantic analysis. However many systems mostly perform syntactic analysis or morphological and syntactic analysis with a little semantic analysis.

The analysis stage of MT is of most interest to this research, as this is the stage where the parsing of the source text into an abstract internal representation takes place. This analysis information is recorded on a results data structure which is passed to the transfer component. The analysis stage involves syntactic analysis, the most important procedure of the analysis stage, perhaps the whole system, as any errors occurring during syntactic processing would be passed through each stage of the whole MT system.

As a result of the importance placed on Syntactic Analysis in the majority of MT systems, the process of analysing can prove costly, i.e. financially, and time consuming. This has lead to the need when developing MT systems to concentrate on Syntactic Analysis with the aim of decreasing time spent on analysis.

1.2.1 Coverage of Machine Translation Systems

Since Machine Translation systems were first developed, they have varied in size and the type of text of they processed. Many of the MT systems developed were large general purpose systems that tended to specialise in a subject area. An example of direct MT system that fell into this category was Systran. Systran began as a Russian - English system used by the Wright-Patterson Air Force Base (Hutchins, 1986). As it was a direct MT system, it was not based on linguistic theory, rather a series of programs that followed the tasks of input, main dictionary look-up, analysis, transfer and synthesis. Each of these tasks had no distinct result structure; the output of one task was the immediate input of another task. Later adaptations to Systran allowed this general purpose system translate texts from English to French, French to English and English to Italian. Systran has been used by the Commission of European Communities to translate internal memos relating to Agriculture in these language pairs. Output from Systran usually needed much post-editing.

A large general purpose MT system that used the interlingual approach was CETA (Hutchins, 1986). Although general purpose, the main subject areas translated by the system were Maths and Physics; the language pair used was Russian - French. The system was based on Chomsky's theory of Transformational Grammar.

Examples of general purpose systems that used the Transfer Strategy are Eurotra and GETA (Hutchins, 1986). Both systems were designed and developed using linguistic theories that incorporated parsing strategies for use in the Analysis Stage of MT. The subject areas covered by each system were various. All of the above general purpose systems were large systems in that they comprised large dictionaries and grammars; thus much time was spent on syntactic analysis, in spite of the fact that most of the systems concentrated on translating certain subject areas.

A few MT systems have been designed and developed for specific tasks; the most well known are Meteo and Aviation developed by the TAUM group. Both systems were developed as sublanguage systems in that they could only translate text specific to certain sublanguage. The Meteo system is dedicated to translating Weather Forecast from English into French. This is a fully working system that has proven to be one of the most successful translation systems in the world. Aviation was dedicated to translating aircraft maintenance manuals. As both systems were based on sublanguages, their dictionaries and grammars were relatively small compared to general purpose systems. Thus, in theory, processing time during the Analysis Stage should be shorter. It is a sublanguage system of which the parsers, discussed above, would be part.

1.3 Parsing Systems

For the purpose of this research, parsing can be split into two types - Deterministic and Non-Deterministic Parsing. The emphasis of the research is to investigate the use of Deterministic Parsers on a sublanguage, however, it would be appropriate in this introduction to define parsing, in general, and examine the characteristics of parsing for MT.

Fundamental to a parsing system is a grammar and a parsing algorithm. The input for a parsing system, usually in the case of MT, is a sentence. The output is a representation usually in the form of a Phrase Structure Tree, but this is not always the case. There are a number of grammar formalisms that can be used in parsing systems and these are discussed in detail in the next chapter. Parsing comprise the functions of searching and matching which is common to both types of parsing - Deterministic and Non-deterministic. Within these types there are various parsing methods that can be adopted, which are also discussed in the next chapter.

The function of parsing, in general, is to produce as output a syntactic or syntactic/semantic representation from a given input. In terms of Machine Translation, parsing, which takes place in the analysis phase of MT, can be considered to be transducing an output from an input. In his 1983 paper, Rod Johnson discusses parsing from an MT perspective, which is the basis of the following discussion. The parser's function in MT is to convert source language text into an abstract structure, usually referred to as an Intermediate Representation. This definition could reasonably refer to parsers, in general. However, most parsers in MT systems produce syntactic representations. For Johnson, the criteria for deciding on the form of a MT parser should be engineering criteria. These criteria state that the properties of the input and output must be known, as an applications parser is only successful if it gets the mappings between input and output that are correct. As Johnson (1983) further states,

> " It seems somewhat gratuitous to enter into debate on the plausibility/efficiency of a parsing strategy without reference to the credibility/usefulness of the idealised artefact which the parser is intended to build."

Parsers, in general, contain various types of knowledge about language and linguistic processes. For parsers used in MT systems, Johnson (1983) considers that four types of knowledge that should be embedded within the parsers - well-formedness conditions, structure-building rules, knowledge about control and appropriateness criteria. In other words MT parsers have to be able to deal with input that is not recognised and be flexible enough to deal with modifications to the input. The parsers must be able to differentiate between decisions that increase efficiency and effect the nature of the result. Finally parsers should only pass on one output, so that the other stages of the translation process are free of the ambiguities found within the source language.
If necessary the parsers should choose the 'best' of the outputs by use of a heuristic, if more than one has been processed.

Although Johnson's (1983) comments are very pertinent to MT, some of the comments can be considered as being more relevant to multi-purpose MT systems than those systems that are dedicated to processing a sublanguage. Parsers for sublanguage would not need to be as flexible as those used in multi-purpose MT systems to deal with new classes of input, as there would not be as many new classes of input within a sublanguage system. Johnson (1983) does not differentiate in types of parsers that should be used, although his comments may be more pertinent to non-deterministic parsers. However, MT systems, especially sublanguage systems could use, it is suggested, deterministic parsers. Although the parser used in the Meteo system was not a deterministic parser, it did only produce one syntactic structure from each parse, which is what a deterministic parse would do. The individual structure produced from the parse was sufficient.

Parsing for MT is, in reality, not that different to parsing for any situation. As MT systems, particularly transfer systems, are modular in design with the parsing process being distinct from the other stages of the MT system.

1.4 Parsing Sublanguage

Sublanguage has many definitions which are discussed in Chapter 4, but the most common are:

A sublanguage of a natural language L is part of L.

A sublanguage is identified with a particular semantic domain.

(Lehrberger, 1986)

The reason that sublanguage is considered to be part of natural language is that various restrictions are found within sublanguages. The restrictions are of a lexical, syntactic and semantic nature. Lexical restrictions in a sublanguage mean that the size of the vocabulary is reduced especially the number of nouns and verbs. Syntactic restrictions found within a sublanguage are the reductions in type and number of grammar rules. The semantic restrictions within a sublanguage are more important than lexical restrictions in that they restrict words being attached to one category and thus leads to a reduction in polysemy.

These restrictions can lead to a reduction in lexical and syntactic ambiguity. The reductions in lexical and syntactic ambiguity can be an aid to the parsing process, which can be caused problems by ambiguous structures. However, it does not mean that the structures that parsers have to parse are simple, some sublanguages have quite complex syntactic structures. Parsers, in general, used in the processing of sublanguages have been of a non-deterministic type. However, the aims of the thesis described the design of deterministic parsers to be used to process sublanguage. The fulfilment of the aims is discussed within the following chapters of the thesis.

1.5 Outline of Following Chapters

The remaining chapters of the thesis can be considered as falling into three sections. The first section consists of chapters 2 - 4, which consider the literature related to the research. The second section contains chapters 5 - 7, which discuss and describe the four parsers built and their linguistic range. The final section contains only Chapter 8, which draws conclusions on the parsers for sublanguage, suggestions for future work and concluding remarks.

Chapter 2 discusses grammars, parsing strategies and parsing systems. The parsing systems considered are those found in Machine Translation systems. Chapter 3 reviews Deterministic Parsers, beginning with Marcus' parser PARSIFAL (Marcus, 1980) and continuing with several of the Deterministic Parsers discussed in the literature during the past decade. This includes an examination of parsers of the LR type. Chapter 4 examines sublanguage, describing what it is, how it is parsed and why it should be used in the practical situation of Machine Translation.

Chapter 5 discusses in detail MParser, its grammar, its parsing rules and its grammatical and linguistic range. The amendments made to the basic design of Marcus' parser are also discussed. Chapter 6 examines LParser, its construction, parsing rules and grammatical and linguistic range. The modifications made to the parser to allow it to parse natural language are discussed. Chapter 7 describes the workings of MParserSub and LParserSub and how they are applied to the noun-phrases of the sublanguage. The processing of complex nouns is also considered.

CHAPTER 2

GRAMMARS, PARSING STRATEGIES AND PRACTICAL PARSERS

2.1 Introduction

The preceding chapter introduced the use of Deterministic Parsers in the practical situation of Machine Translation as the area to be investigated. This chapter concentrates the discussion on the fundamentals of parsing systems - grammar and parsing strategies - looking at the formal properties of both. The chapter ends with an examination of some current parsers and their use in practical Machine Translation systems to emphasise what parsers need to produce for a practical Machine Translation system.

In linguistic theory, grammars are commonly defined as generators of language, i.e., grammars describe the structures of a language. In section 2.2 grammars are defined in more detail, with examples being given of the different types of grammar commonly used in conjunction with practical parsing systems.

There are several parsing strategies that can be considered for use in NLP systems. These strategies have been adopted both in theoretical and practical systems. As the aim of this current work is to investigate an issue of practability in Machine Translation, the discussion will concentrate on the latter type of system in this field. In section 2.3, there is an examination of current parsing strategies, with the formal properties of parsing being discussed. In section 2.4, the chapter concludes with an examination of the parsers used in practical Machine Translation systems.

2.2 Grammar

It is generally regarded that grammar can be defined in the manner explained by Winograd (1983)

" A formal linguist describes the structures of a language by devising collection of rules, called a grammar, that can be used in a systematic way to generate the sentences of a language."

The generating of sentences can be defined as the specification of how to form, interpret and pronounce sentences (cf Radford, 1982,p12). The generating of sentences is the reason for including this discussion on the theoretical properties of grammar as interesting parsers can be engineered from the grammatical theories which generate sentences.

However, Briscoe (1987) suggests that consideration should also be given in a definition to the 'function of grammatical information in language, i.e.,'

" to encode explicitly by means of word order and inflection limited aspects of meaning and thereby reduce ambiguity." (Briscoe, 1987, p69)

The function of grammatical information is very important in Machine Translation systems. The majority of Machine Translation systems have tended to rely on the syntactic (grammatical) information as the main (often the only) way of discovering 'semantic relations' during processing. Both of the above definitions have equal importance when describing grammar.

In the following sub-sections some of the most common grammar formalisms used in the field of computational linguistics are examined. The formalisms discussed are classified in a hierarchy of power, Type 0 - Type 3, and were widely expounded by Noam Chomsky in his book *Syntactic Structures* (1957). The classes of grammar discussed by Chomsky and also unification grammar are examined in sub-section 2.2.1.

2.2.1 The Chomsky Hierarchy

Noam Chomsky classified grammars incorporating rewrite rules into a four member hierarchy based on power: type 0 grammars are the most powerful and are referred to as unrestricted rewriting systems, type 1 grammars are the second most powerful and also can be referred to as context sensitive grammars, type 2 grammar are the third most powerful also being referred to as context free grammars and finally type 3 grammars are the least powerful and can also be referred to as regular languages or finite state grammars.

As stated, each of the grammar classes of the Chomsky hierarchy represents a different level of power. According to Chomsky's classification, the more powerful the grammar type, the more classes of languages it can generate. In other words, a language defined by type 3 grammar can be defined by types 2, 1 and 0, but a language defined by type 0 cannot be defined by type 1. The power of each of grammar class is now examined in terms of what it covers and its inherent computability or non-computability of the grammar. The type 0 grammar can generate natural languages which can be generated by any deterministic computational machine, thus can be considered inherently computable. The type 1 grammar can also generate most natural language structures. These natural language structures can be recognised by a deterministic computational machine called a linear bounded automaton. Once more, this type of grammar can be considered inherently computable. The type 2 grammar which is less powerful than type 0 and type 1 grammars contains the restrictions of type 1 grammars as well as its own restrictions. As Winograd (1983) notes, type 2 grammars can not generate languages that contain $X^nY^nZ^n$ structures or WW structures where W represent a string of terminal symbols and the two W's are identical. An example would be any sentence that contained the English 'respectively' construction, e.g., Jane, Martin and Keith go to Kent and Sussex respectively. This type of grammar because of its restrictions is not inherently computable. Type 3 grammars are the least powerful type of grammar. As well as having the restrictions of type 1 and type 2 grammars, type 3 grammars cannot generate languages that have any embedded structures, i.e., $X^nY^kZ^n$. As with the type 2 grammars, type 3 grammars are not inherently computable.

Above, it is noted that Chomsky classified grammars that incorporated rewrite rules. Rewrite rules are now described. The classes of grammar discussed by Chomsky (1957) are examples of generative devices. Generative grammars have the capacity to define the set of grammatical sentences in a language. Grammar rules in generative grammars tend are rewrite rules which take the form X -> Y, with the symbol to the left of arrow representing a *single structural element* and the symbol to the right representing a string of one or more elements. There are generally two sets of symbols used in rewrite rules, the set of non-terminal symbols and the set of terminal symbols. Non-terminal symbols usually represent nodes in phrase markers, such as NP (noun-phrase), VP (verb-phrase) or PP (prepositional phrase) etc. Terminal symbols refer to the units, such as *a, boy, -ed*, used in the syntactic representation of a phrase or sentence, when rules have been applied. Below examples of the classes of grammar are examined, beginning with the Type 3 grammars, i.e., simple finite state grammars, also known as regular grammars. The description continues through Type 2 and Type 1 grammars, i.e., phrase structure grammars, which describe both context free and context sensitive grammars, to Type 0 grammars which include an example of an unrestricted rewrite systems, transformational grammar and unification grammars.

2.2.1.1 Type 3 - Finite State Grammars

Finite state grammars are regarded as simple Generative Devices. Grammars of this type generate by working through a sentence or phrase from left to right in the following manner: an element is selected as first element and the possible occurrences of all other elements are dependent on the type of elements by which they have been preceded. This type of grammar can only cope with very simple sentence constructions. An example grammar is given below:

```
((S0 (a S1))
 (S1 (b S2)
     (b S3))
 (S2 (c S4))
 (S3 (d S2))
 (S4 (DONE)))
```

Phrase structure grammars and unrestricted rewrite system are considered to improve on the Finite State Grammar in several ways. The enhancements are discussed in conjunction with each of the classes of grammar.

2.2.1.2 Type 2 and Type 1 - Phrase Structure Grammars

Phrase structure grammars are another type of generative device which contain rules that can generate *strings of linguistic elements* and also provide a *constituent analysis of the string*. By providing the constituent analysis, the phrase structure grammar provides more information than that provided by the Finite State Grammar. Phrase Structure Grammars are regarded as the basis for generative linguistics (Winograd, 1983, p72) and computers systems that deal with both programming languages and natural language. Phrase structure grammars are generally divided into two types which are context-free grammars and context-sensitive grammars. Each of these grammar formalisms is now discussed.

A context-free grammar consists of a set of rewrite rules which contain non-terminal and terminal symbols as shown in the example below.

S -> NP VP	Det -> The
NP -> Pnoun	Noun -> man
NP -> Det Noun	Pnoun -> Mary
VP -> V NP	Pnoun -> John
V -> loves	

The context-free grammar formalism is widely used in computational linguistics because it provides a simple method of providing the information needed about the language. As Winograd (1983) states,

> " A context-free grammar provides an especially simple way of describing t h e structures of a language and of setting up a correspondence between the knowledge structures, the structures generated in producing or recognizing a sentence and the processes of recognition and production."

Context-free grammars also can be referred to as *immediate constituent grammars, Backus normal form* and *recursive patterns.*

Context-sensitive grammars are similar in appearance to context free grammars, in that they are composed of non-terminal and terminal symbols, but are different in that the left-hand side of a rule in a context sensitive grammar can have more than one element which can be used to indicate context dependencies. Also the right-hand side of a context sensitive rule consists of the left-hand side with a single symbol expanded. An example of a context-sensitive rule is the following:

Plural NP -> Plural Det Noun

2.2.1.3 Type 0 - Unrestricted Rewrite Systems and Unification Grammars

The unrestricted rewrite system is another example of a generative device which is described as having 'Turing Machine' power. This classification of grammar has no restriction on rule form. As a result, the majority of natural language structures can be generated by unrestricted rewrite systems. An example of the type 0, unrestricted rewrite system is Transformational grammar.

Transformational grammar, as described by Crystal (1985), utilises the linguistic operation of transformation which allows *two levels of structural representation to be placed in correspondence.* The rules of transformational grammar comprise of sequences of non-terminal and terminal symbols which can be rewritten as different sequences of the same non-terminal and terminal symbols by application of a specific operation. Transformational rules are described as having *inputs* which are structural descriptions. The *structural descriptions* define the type of *phrase-markers* to which the transformational rules can apply. Transformational rules can perform a structural change on the *input* by employing at least one of several operations. Some of these operations are *movement, adjunction, insertion* and *deletion.*

Since the first discussion of Transformational Grammar in Chomsky (1957) several models of this type of grammar have been suggested. The Standard Model described by Chomsky (1965) has three components: a syntactic component, a phonological component and a semantic component. Crystal (1985) defines the components thus: the syntactic component is made up of the base component of phrase structure rules, which along with *lexical information* give *deep structure* information and a set of transformational rules which generate the *surface structure.* The phonological component functions as a converter of syntactic elements into *pronounceable utterances.*

The semantic component provides a representation of the meaning of *lexical items.*

In the early 1970's a model of generative grammar developed out of 'standard theory' known as 'extended standard theory'. The extension to the standard theory was the range of semantic rules which could according to the new theory operate with surface structure as input, which meant that deep structure was not the only determinant of the semantic representation (Crystal, 1985).

In later work on Transformational Grammar, Chomsky discussed a different method for representing the structure of sentences in natural language called X-bar syntax (Radford, 1981). X-bar theory is considered, by its proponents, to be an answer to the problems of phrase structure. The problems of phrase structure are considered to be the following:

a) too restricted in the categories it permits, i.e., phrase structure restricts types of categories.

b) too unconstrained in the sets of possible Phrase Structure Rules it permits, i.e., phrase structure has no constraint on the type of phrase structure rules that can be generated. (Radford, 1981).

X-bar theory allows more than one phrasal expansion because of its range of category types :

'X with no bars' or lexical category X

'X-bar' or X^1

'X-double-bar' or X^2

'X-treble-bar' or X^3

etc...

By having a range of category types, constituents can be better described as it allows intermediate categories that do not appear in phrase structure syntax.

The reason that phrase structure syntax is considered unconstrained with regard to sets of phrase structure rules is that it does not disallow theoretically incorrect structures such as NP -> V VP. In X-bar theory the following rule applies:

X^n -> ...X^m... (where m = n or n - 1).

In other words, a noun-phrase must have as its head a noun, a verb-phrase must have as its head a verb, etc...

As described by Crystal (1985), Transformational grammar allows the economic derivation of many sentence types by adding to the constituent analysis rules of phrase structure grammars, transformation rules which can change one sentence into another. Thus, more sentence types can be derived by Transformational Grammar than can be derived by finite state grammars or phrase structure grammars.

The name unification grammar is given to a variety of different grammars which, however, have not been developed specifically as unification grammars: the grammars have been described as *unification-based* or *complex-feature-based* because of the approach taken by developers to encoding syntactic and semantic linguistic information (Shieber, 1985). Many of the developers of the various grammars that come under the heading unification grammar took as their starting point the fact that generative grammar formalisms such as Transformational Grammar were becoming increasingly complex and abstract. As Winograd (1983) states,

> " Generative grammar has arrived at a point where the concepts of 'rule' and 'generation' are extremely distant from naive intuitions of how we produce and understand linguistic patterns."

The developers wished to incorporate the advantages of the simpler generative formalisms, but not disregard the power of the more complex generative formalisms. Examples of unification grammars are: *Lexical Functional Grammar (LFG), Categorial Grammar, Generalised Phrase Structure Grammar (GPSG), Head Phrase Structure Grammar, Functional Unification Grammar (FUG)* and *Definite Clause Grammar (DCG).* LFG, FUG and DCG are examined in more detail below.

Each of the above grammar formalisms was developed independently by researchers in various fields such as computational linguistics, linguistics and natural language processing (Shieber, 1985). However, there are some common assumptions made regarding the nature of the grammar formalism, i.e.:

surface-based:	providing a direct characterization of the actual surface order of string elements in a sentence.
informational:	associating with the strings information from some informational domain.
inductive:	defining the association of strings and informational elements recursively, new pairings being derived by merging substrings according to some string combining operations, and merging the associated information. The commonly used method is unification.
declarative:	defining the association between strings and informational elements in terms of what associations are permissible, not how the associations are computed.
complex-feature-based:	as associations between features and values taken from some well defined, possibly structured, set.

(Shieber, 1985)

The informational elements mentioned above have several names dependent on the grammar formalism, but as Shieber (1985) states can be commonly referred to as *feature structures*. It is with the manipulation of these feature structures that the notion of unification arises. The combination of two sets of feature structures will involve the union of feature/value pairs and perhaps recursively combining the values. The process can be referred to as graph-combining, which encapsulates the notion of unification, as an essential operation of FUG, LFG, GPSG. There are other operations such as disjunction, generalization and overwriting relevant to the grammar formalisms, but unification is central to all the grammar formalisms mentioned. The previously mentioned unification-based grammar formalisms are now examined individually.

Lexical Functional Grammar (LFG) was designed as a theory of language, with the mental representation of grammatical constructs and the universal constraints on natural languages being particularly stressed (Shieber, 1985). It is called Lexical-Functional Grammar because it stresses the role of the lexicon. LFG was developed as a result of work that had been done by its developers, Bresnan and Kaplan, on a complex Chomsky grammar such as Transformational Grammar. LFG tries to solve problems found in Transformational grammar by incorporating *additive description*. For example, in place of transforming trees, LFG has a multi-layered description with each layer augmenting the contents of the other layers. The Lexical-Functional description of a sentence has two components, which are *constituent structure* and *functional structure*. The constituent structure is a context-free surface parse of the sentence and the functional structure is generated by *equations* which are linked to the context-free rules. LFG uses *syntactic metavariables*, represented by arrows, in place of names for grammatical constituents of rules and paired feature structures (f-structures in LFG).

Functional Unification Grammar (FUG) developed by Martin Kay out of *unification grammar* and *functional grammar* is described as a *general linguistic tool* which employs unification as its only operation (Shieber, 1985). FUG comprises a set of constituent descriptions which are arranged into sets of alternatives. The set of alternatives can be regarded as new feature *patterns* including syntactic patterns and word-defining patterns. FUG is used for both generation or parsing, with its goal being to produce a functional description of a sentence.

Declarative Clause Grammars (DCGs) were developed from research done on the logic programming language Prolog by Pereira and Warren (1980). DCGs employ a form of unification based on *term structures* not feature structures. Terms are informational elements which are notated in form similar to that used in logic and maths - a predicate symbol followed by a parenthesized series of smaller terms.

DCG rules are an extended form of context-free grammar rules which link *string-combining* and *information-combining* operations. DCGs are more powerful than context-free grammars in that they can allow context-dependency within a grammar and also allow structure building during parsing. DCGs are discussed in more detail in Chapter 6. An example DCG grammar rule is given below.

```
sent(s(NP,VP) --> np(NP),vp(VP))
```

2.3 Parsing Algorithms

In Chapter 1, the function of parsing has been discussed in relation to its use within Machine Translation systems. This section continues the discussion on parsing with an examination of the types of parsing algorithms used not only in Machine Translation, but in Natural Language Processing in general, the methods of parsing employed by these parsing algorithms, and the formal properties of such algorithms.

2.3.1 Types of Parsing Algorithms

In the previous chapter, the discussion on parsing briefly mentioned the types of parsing algorithms, i.e., non-deterministic parsing algorithms and deterministic parsing algorithms. Non-deterministic parsing algorithms, in the Natural Language Processing context, include depth-first and breadth-first parsers. Depth-first parsers use backtracking algorithms which allow the re-parsing of words or phrases when the original parse proves to be incorrect. Conceptually backtracking algorithms use a stack to store alternatives and backtracking is very simply a matter of holding the current state of the computation with each entry on the stack, so (on backtracking) the state of the computation can be reinstated. Breadth-first parsers use a queue and can be said to parsing one or more hypothesis at a time. The current state of the computation has to be held in the queue assuming, of course, that the machine is sequential and not a true parallel machine. Deterministic parsing algorithms on the other hand, do not allow either re-parsing by use of backtracking or processing in parallel, but have the ability to use lookahead to aid parsing. In the next sub-section, the discussion focuses on the parsing methods used within the aforementioned algorithms.

2.3.2 Parsing Methods

In the literature (e.g. De Roeck, 1983; Winograd, 1983), parsing methods are usually described in connection with non-deterministic parsing algorithms and Context-Free Grammars (CFGs), although most parsers can cope with grammars of a greater formal power, as described above. The parsers that cope with grammars of a greater power than that of CFGs have to have extra code to be able to cope with the augmented features.

Parsing, in this discussion, is taken to comprise searching and matching. Searching in the parsing process uses methods of strategy and control to search the search space of a grammar during the building of the parse tree structure. When describing strategy the following definition can be used:

> The first part of the method used for building a structure, that is starting either with the starting symbol (S in the grammar below) or the input string, in other words Top-down or Bottom-up Analysis.

When describing control the following definition can be used:

> The method used for examining the derivation tree of a grammar either vertically or horizontally; that is Depth-first or Breadth-first Analysis.

Matching in the parsing process occurs when the current input, i.e., the grammatical constituent being processed, matches with a grammatical construct from the grammar. When matching is successful a structure is built (usually a tree structure but perhaps a LFG f-structure). The discussion in this section primarily considers the methods used by non-deterministic parsing algorithms for Natural Language Processing and will use the example CFG given below.

```
s -> np vp                    adj -> flying
vp -> verb_group np           aux -> are
vp -> verb np                 noun -> planes
verb_group -> aux verb        pronoun -> they
np -> pronoun                 verb -> are
np -> noun                    verb -> flying
np -> adj noun
```

There are two possible phrase structure trees for an input of 'They are flying planes'. It is easy to work out 'intuitively' how the phrase structure trees may be derived using top-down analysis, but more difficult to work out an easily stateable algorithm for bottom-up parsing.

2.3.2.1 Top-down Analysis

Parsers that uses a top-down strategy begin by searching the grammar rules for the rule containing the starting symbol, in the case of the example grammar, s.

```
s -> np vp
```

The parser checks whether the first of the constituents that form the rule, i.e., nps is a terminal, if not this constituent is expanded.

```
np -> pronoun
np -> noun
np -> adj noun
```

There are three choices that can be made. If, for example, the first of the three alternatives is chosen, the constituents are checked for terminals and in this case the non-terminal is expanded:

```
pronoun -> they
```

The constituent of the rule on this occasion is a terminal and matches with the first word of the input string. The parser continues working in the same fashion, searching the search space, until all terminals have been found.

The parser meets no problems when parsing the above, i.e., all branches of search tree chosen are correct but if either of the other nps had been chosen matching would have failed. For example the following rule could have been chosen. `np -> noun` On expanding the rules for finding the first terminal, the match between the terminal and the first element of the input string would fail.

2.3.2.2 Bottom-up Analysis

When a parser uses bottom-up strategy it begins by looking at the input string and reduces until the rule for the start symbol S is found. There are many bottom-up algorithms, of which one is presented here. The algorithm is based on the Bottom-up Parser (BUP) developed by Matsumoto *et al.* (1983).

For example, beginning the parse with the leftmost element of the input string 'They', which has the ultimate 'goal' of being covered by constituent 'np', the initial goal to be matched is

```
    pronoun -> they
```

Pronoun is then reduced to NP as a result of the rule

```
    np -> pronoun
```

The parser continues in this manner until the whole input string is covered by the distinguished symbol S. An example of the search tree for bottom-up analysis is given below.

According to Winograd (1983) the difference between top-down analysis and bottom-up analysis is that top-down analysis can be defined as goal directed processing guided by the goals it is trying to achieve and bottom-up analysis as data directed processing which is guided by the availability of specific data.

2.3.2.3 Depth-first Analysis

As stated above control comprises depth-first and breadth-first analysis. Using depth-first analysis, a single branch of the search tree is followed to its very bottom. In fact, both the top-down and bottom-up procedures described above, work in this manner. Non-deterministic parsers use a stack to store alternative branches in the search tree, but also have the ability to reinstate the state of the parse at the choice point. Programming languages such as Prolog, LISP and Pascal (with facilities for recursion) can be used to implement depth-first search easily.

2.3.2.4 Breadth-first Analysis

If a parser using Depth-first analysis works in serial, it can be said that a parser using Breadth-first analysis is concerned with working in parallel, i.e., it looks all possible branches of the tree are searched in a horizontal manner although, of course, this parallelism has to be simulated in current sequential machines. During the searching each state of the computation is recorded in a queue, i.e., when successful matching occurs during top-down or bottom-up strategy new states are added FIFO. Using this strategy means that there is no need for a backtracking facility as all possibilities are covered.

2.3.2.5 Advantages of Depth-first and Breadth-first Analysis

Both control mechanisms, depth-first and breadth-first, search the same search space and give the same solutions, but not necessarily in the same order. Depth-first will find the 'first' path (by convention the left-hand branch in the search tree): breadth-first always finds the shortest solution ("most compact") first - i.e., the one with fewer nodes in the search tree.

The order of solutions becomes important when:

a) interest is in the first solution found, or

b) use of some heuristic is important.

2.3.2.6 Combining the Methods

It is now possible to combine the methods, as a result obtaining four different parsing methods. The result of the combination set out in De Roeck (1983), is depicted in Table 2.0. These are considered the basic parsing methods, but not many parsers strictly adhere to the rules governing methods. For example, parsers exist that combine bottom-up analysis with top-down filtering such as BUP (Matsumoto *et al.,* 1983); a linking device which works top-down.

	Depth-first Analysis	Breadth-first Analysis
Top-down Analysis	Top-down, Depth-first	Top-down, Breadth-first
Bottom-up Analysis	Bottom-up, Depth-first	Bottom-up, Breadth-first

Table 2.0 Parsing Strategies

2.3.2.7 Parsing Methods for Deterministic Parsing Algorithms

The parsing methods used by deterministic parsing algorithms for Natural Language Processing use variations on the methods described above. As deterministic parsing algorithms do not allow backtracking or processing in parallel, lookahead is used to solve any parsing problems. The deterministic parsing algorithms commonly used in Natural Language Processing are the Marcus deterministic parsing algorithm (Marcus, 1980) and the LR(k) type deterministic parsing algorithm (Aho and Ullman, 1972), which will be discussed in detail in the next chapter. The Marcus parsing algorithm uses a combination of parsing strategy of bottom-up analysis with top-down filtering. The LR(k) type algorithm uses a bottom-up strategy. Both algorithms follow a depth-first search path, but of course do not need to store alternatives on the stack because there is no backtracking. Matching takes place on current input and up to three buffer cells of lookahead.

2.3.3 The Formal Properties of Parsing Algorithms

In the sub-section above parsing strategies used by parsing algorithms have been described. In this sub-section the methods of measuring the efficiency of parsing algorithms are described. This is an important feature of work on parsing algorithms as by examining the formal properties of parsing algorithms, the concern is with evaluating the performance of the parsing algorithms. This is done by means of measuring time and space consumption. The measuring of time and space consumption involves issues of complexity theory in that the criteria used in evaluation are

1. The number of elementary mechanical operations executed as a function of size of the input (time complexity).

2. How large an auxiliary memory is required to hold intermediate results that arise during the execution, again as a function of the size of the input (space complexity). (Aho and Ullman, 1972, pp27-28)

Another factor to be taken into consideration when evaluating parsing algorithms is the type of grammar it is used on. As context-free grammars are most often used as examples of grammar, the discussion on the formal properties of parsing algorithms concentrates on the use of parsing algorithms on context-free grammars; the results of use on other types of Chomsky grammars are reflected in the results of use on context-free grammars (Briscoe, 1987; Berwick and Weinberg, 1984).

The use of non-deterministic parsing algorithms on context-free grammars have been found to require polynomial and in the worst case exponential time and linear space as a function of the input length of the string to be parsed (Aho and Ullman, 1972). Deterministic parsing algorithms as used on context-free grammars have been found to require linear time and linear space. In other words by using deterministic algorithms, the operations needed to parse sentences of ten words are not polynomially or exponentially greater than those needed for sentences consisting of five words (Briscoe, 1987). Whereas, by using non-deterministic parsing algorithms the number of operations needed to parse five words would increase non-linearly.

Cautious consideration has to be given to these claims about the algorithms and complexity theory as other factors such as the size of the grammar may affect results (Briscoe, 1987). However, deterministic parsing algorithms have working memory limitations, thus they use less resources whether measured in time or space.

2.4 Parsing Strategies used in Machine Translation Systems

In Chapter 1 the characteristics of parsing for Machine Translation have been examined. In this section three actual parsers used in practical Machine Translation systems are discussed; two of the systems are sublanguage systems. The discussion will concentrate on the function of the parsers and their output in the context of Machine Translation to highlight what a practical parser should produce for a Machine Translation system. In the final sub-section each of the three parsers will be compared and contrasted.

2.4.1 Taum-Meteo

The Taum-Meteo Machine Translation system has been one of the most successful practical Machine Translation systems developed and is still in use today. The Meteo system translates Canadian weather forecasts from English into French. The main reason for its success seems to stem from the fact that the system only translates the restricted language of weather forecasts.

The Meteo system is not a Machine Translation system of traditional three phase design. The three phases found in a more traditional system are, in Meteo, dealt with almost completely by a multi-pass syntactic and semantic parser. The designers found that a conventional syntactic parser was not suitable due to the telegraphic style of the text, therefore there was a need to utilise semantic information.

The aim of the parser is to produce for each input string a single description giving categories and translations realized for the given string (Chandioux, 1976). Following idiom and main dictionary look-up, the parsing process begins. In the first pass of the parser, substrings containing numerals are identified as dates, hours or temperatures. In the second pass, time and location expressions are identified. The third pass of the parser deals with the remaining substrings with the parsing rules dependent on the semantic sub-categorization introduced in the dictionary to choose the correct translation. In the fourth pass, sequences of conditions, time references and locations are tested for ambiguity and well-formedness. In the fifth and final pass incomplete parses are rejected and stylistic adjustments made.

After parsing, the resulting structure is passed to the generator where it is decomposed, French articles inserted where necessary, French word order considered and agreement checked.

Meteo's parser performs the majority of the functions of machine translation by side-effect, i.e., having found a certain string in the input buffer, it executes arbitrary code in order to produce some TL output. This process makes Meteo untypical amongst current Machine Translation systems. It can be regarded as a direct system which can deal only with English to French. The structure produced by the end of the parse is almost a complete translation apart from cosmetic changes required to accommodate French. Meteo could only be used to translate meteorological texts because it is based on the syntax and semantics of a restricted language. However, the system could be adapted for another restricted domain, but perhaps not another language pair, as the system had been specifically designed to the English - French language pair.

2.4.2 Taum-Aviation

The Taum-Aviation Machine Translation system was designed by the same team who developed the Meteo system. However, this system has not been practically successful, although successful as a research system, due to the fact that funding was withdrawn before the system was fully operational. The Aviation system was designed to translate from English to French an aircraft maintenance manual which can be considered as yet another sublanguage.

Taum-Aviation used a usual three phase approach: analysis, transfer and synthesis. The parser developed for the analysis phase was REZO, a version of an Augmented Transition Network (ATN) (Woods, 1970). An ATN is usually classed as a non-deterministic, backtracking parser. REZO functioned in a similar manner by parsing in a top-down, depth-first, left-to-right serial manner.

Prior to parsing, the input string was pre-processed and put through dictionary look-up. Pre-processing involved writing the input string onto a chart structure. REZO took the chart structure with encoded lexical ambiguities and applied the grammar in to all paths of the chart producing another chart structure. The parsing process eliminated many of the lexical ambiguities, but also introduced structural ambiguities which had to be dealt with by semantic processing. Semantic processing attached to each tree built within the chart structure a set of semantic features to be used in the transfer phase.

The chart structure produced at the end of the parsing process is an intermediate representation of the source language, described as a type of semantically annotated deep structure (Isabelle and Bourbeau, 1985). The chart structure is still in the source language. The target language is not considered during the parsing process or the analysis phase in general. When the chart structure is passed to the transfer module, it undergoes both lexical and structural transfer which transforms the structure into an intermediate representation of the target language.

2.4.3 Metal

The Machine Translation system Metal has been developed by the Linguistic Research Center (LRC) group for Siemens Ag, Munich and has had to continually prove itself to be cost-effective, so as to avoid the problems that beset the Taum Aviation Project (Slocum *et al.*, 1985). Metal is another three phase analysis, transfer and synthesis system, with the parsing process, as in other systems of this type taking place in the analysis phase. German was the source language used.

The LRC research team decided that the parser to be used in the analysis had to be computationally efficient with regard to time and space consumption as set out by complexity theory discussed above. The team compared eight parsing systems based on two parsers, Cocke-Kasmi-Young and Left corner, with variations of top-down filtering and early constituent analysis. Data was gathered for 35 variables measuring various aspects of behaviour - general information, search space, processing time and memory requirements. Storage management was measured by evaluating the 'conses' executed by Lisp, the language in which the parsers were programmed. Search space was considered in the manner of grammar versus chart structure with the grammar considered as being processed bottom-up and top-down. Time and space also were separately measured for words, phrases, idioms and operating rule body procedures. In measuring the performance everything considered relevant was taken into consideration i.e., paging, storage management, building "interpretations" as well as parse time (Slocum, 1981).

The parser chosen by the LRC team was one that proved best in overall performance according to the above tests. This parser is a some-paths-parallel bottom-up parser. The parser processes in conjunction with rule body procedures which accept or reject grammar rule application. (Rule bodies are arbitrary code that perform tests or building structures in the same way that ATNs are augmented with arbitrary Lisp code.) The rule body procedures are called when the parser finds a phrase which matches the rule's constituent phrase structure. The procedures also aid in building an interpretation of the phrase. Metal's parser differs from traditional syntactic parsers which build parse trees and may add semantic information to these trees. Rather the parser does not construct a syntactic structure, but the linguistic procedures build an interpretation and compute its weight or plausibility measure. The weight of the phrase is used when comparing it with other phrases that could be built from the same sequence of words in order to identify the most likely reading. The phrase interpretations, in most instances, are *topologically equivalent* to the tree structure that could be produced by the parser.

The interpretations produced by the parser and rule body procedure are considered to be shallow analyses of the input string. The interpretations are passed to the Transfer module with *suspended* rule body procedures attached. These procedures are re-invoked by the Transfer module and aid in the Transfer process. The interpretations remain in the source language, with the Target language firstly being considered in the Transfer module. As in the other systems discussed both syntax and semantics have to be considered in the analysis phase.

It is interesting that Slocum *et al* (1985) in their evaluation make the following statement about evaluating parsers

> " ...the researcher studying NLP can be justified in concerning himself more with issues of practical performance in parsing sentences encountered in language as humans actually use it using a grammar expressed in a form convenient to the human linguist who is writing it. Moreover, very occasional poor performance may be quite acceptable, particularly if real-time considerations are not involved... provided the overall average performance is superior."

2.4.4 Concluding Remarks on Parsers for Machine Translation

Although the discussion on parsers for Machine Translation has focused on only three parsers, it can be concluded that there are areas of similarity between the three which would suggest the importance of some aspects for all parsers used in Machine Translation, not least the parsers developed for this research. The discussion has also primarily dealt with parsers and parsing during the analysis phase. It is also important to point out that dictionaries and lexicons are also an important feature of the analysis phase of all three systems. As stated above comparisons between the parsers designed for this research and those discussed in this section are considered in Chapter 7.

The major similarity between the three parsers is the importance of processing both syntax and semantics of the source language in the analysis phase. In the case of Taum-Aviation and LRC Metal this means that structure passed to the transfer phase carries as much information as possible about the source language text. The Taum-Meteo parser, as a result of the restricted language it is dealing with, can perform the translation process during the parsing procedure but relies heavily on the dictionaries for semantic information.

Another similarity between the analysis phases of Taum-Aviation and LRC Metal is the fact that the Target Language is not considered during this phase at all. The parsers of these systems do not have to deal with Target language, they have solely to deal with the source language.

In short, the parsers of all these systems have to deal both with syntax and semantics and those used in the usual three phase system consider solely the Source Language.

CHAPTER 3

DETERMINISTIC PARSERS

3.1 Introduction

In Chapter 1, the research area has been described as being the use of Deterministic Parsers on Sublanguage for Machine Translation. The aim of this chapter is to introduce and discuss Deterministic Parsers, of the Marcus and LR type, on which the parsers developed for this research have been based. The discussion considers the parsers from a practical viewpoint, as Machine Translation systems (of which the research parsers would be part) can be regarded as practical or engineering systems. Brief consideration is given to the comparison of the research parsers and the parsers discussed in this chapter. This will be covered more fully in Chapters 5 and 6. (All references to the term "practical system" stands for Machine Translation system.)

The majority of Deterministic Parsers for natural language processing are based on the original, PARSIFAL, developed by Mitchell Marcus and reported in his 1977 PhD thesis, which was published in book form in 1980 (Marcus, 1980). However, not all the Deterministic Parsers that are discussed here can be described as Marcus type parsers: one parser in particular has been developed from the deterministic LR(k) algorithm (Aho and Ullman, 1972), primarily used as a model for parsers of programming languages. However, since it has been suggested (Berwick, 1985) that Marcus' parser was a variant of the LR(k) model, the parser of the LR(k) type is pertinent to the discussion. Also it is valid to state that the majority of the parsers discussed here follow Marcus' Determinism Hypothesis and rules guiding the building of constituent structures as described below. Other Marcus type parsers are discussed briefly to highlight the work done on Deterministic Parsing.

3.2 General Rules of Deterministic Parsers

As stated above, the majority of Marcus type and LR type Deterministic Parsers are based on Marcus' rules and hypothesis. Prior to the Deterministic Parser, most parsers simulated Non-deterministic machines by using backtracking or pseudo-parallelism. Marcus (1980) was motivated to design the Deterministic Parser on watching an ATN backtracking, concluding that there must be a better way to parse English computationally, preferably using a method similar to the way humans do it.

The design for PARSIFAL, Marcus' parser, came from his psychological model of how humans parse and was based on his Determinism Hypothesis:

> " The syntax of any natural language can be parsed by a mechanism, which operates strictly deterministically in that it does not simulate a Non-deterministic machine."(Marcus, 1980, p2)

The limitations Marcus placed on his parser, which he considered fundamental to parsing deterministically, were as follows:

a) All syntactic substructures created by the grammar interpreter are permanent, eliminating the possibility of simulating determinism by "backtracking".

b) All syntactic substructures created by the grammar interpreter for a given input must be output as part of a syntactic structure assigned to that input.

c) The internal state of the mechanism must be constrained in such a way that no temporary syntactic structures are encoded within the internal state of the machine.

3.3 Marcus Parsing

In the following sub-sections the Deterministic Parsers of the Marcus type used as the basis for the design of the parsers used in this research are examined. The discussion begins with describing PARSIFAL (Marcus, 1980), followed by ROBIE (Milne, 1983) and L.PARSIFAL (Berwick, 1985). The parsers are discussed in the context of searching and matching as highlighted in Chapter 2.

As the first Deterministic Parser for Natural Language Processing, PARSIFAL set the design for the developers of later Deterministic Parsers to follow and improve on. Both ROBIE and L.PARSIFAL are in some way different to the Marcus original, but, in the main, keep to Marcus' original proposals.

3.3.1 The Marcus Parser - PARSIFAL

The Deterministic Parsing system PARSIFAL comprised three major components - two data structures, the 'push-down' stack and the Lookahead Buffer, and a grammar of pattern/action rules; the parsing process being an interaction between all three components. The stack was called the Active Node Stack and the Lookahead Buffer could hold three constituents. A constituent could range from a single word to a clause. These constituents, on being processed, were given the name "parse nodes", which could form tree-like structures of "NPs", "VPs", "Ss" etc. The function of parsing has been defined as one of searching and matching. PARSIFAL is now considered in this context.

In terms of parsing, searching comprises strategy and control. The strategy used by PARSIFAL was mixed; the parser processed bottom-up but incorporated top-down prediction. For example, PARSIFAL would process 'with Mary' in its lookahead buffer by first creating a prepositional phrase node, triggered by the leading edge of the phrase, which would be pushed onto the Active Node Stack. This would be the new top node of the Active Node Stack. PARSIFAL would subsequently parse in a bottom-up fashion 'with' as a preposition. Top-down prediction would force creation of a noun-phrase on finding the leading edge 'Mary', which would be parsed bottom-up as a proper-noun. In a strict bottom-up parser the phrase 'with Mary' would be processed respectively as preposition and proper-noun, the proper-noun would be reduced to a noun-phrase and finally prep and noun-phrase reduced to a prepositional phrase.

Control within the searching action of parsing can be depth-first, breadth-first or a mixture of both. PARSIFAL employed a depth-first approach. Thus, the parser follows one branch of the search tree to the bottom. As a result of determinism, only one branch of the search tree is followed unlike non-deterministic depth-first parses where alternative branches can be searched and stored in a stack. PARSIFAL is aided by the three cell Lookahead Buffer to guide it through the single branch of the search tree. PARSIFAL's Active Node Stack contained only constituents that feature in the final parse, it did not contain any which were superfluous or unnecessary due to following depth-first a single branch of the search tree.

Matching in PARSIFAL was on syntactic categories. The actual production of parsing output was based on matching. Words, having gone through dictionary look-up, entered the Lookahead Buffer with category attached. PARSIFAL's parsing strategy allowed the prediction of phrase type by matching on, at most, the three constituents in the Lookahead Buffer. PARSIFAL processed the constituent in first cell of the

Lookahead Buffer by matching the contents of all the cells of the Lookahead Buffer and the Current Active Node, the bottom node in the Active Node Stack, with one of the pattern/action grammar rules. Once the constituent in the first cell of the Lookahead Buffer was processed, it was pushed onto the Current Active Node; the pushing of the constituent onto the Current Active Node was activated by the action part of the pattern/action rules. The parse tree structure was built in the Current Active Node. Depending on the type of constituent being processed, the constituent would be pushed into the Current Active Node either to combine with constituent or constituents already contained in the Current Active Node adding to the building of a branch of the parse tree or be the only constituent in Current Active Node initiating a new branch of the parse tree. If there were constituents in the Current Active Node that could not combine with the constituent being processed they would be pushed further into the Active Node Stack to await further processing. The constituent which previously had been in the second buffer cell entered, the first buffer cell. This constituent was processed in a similar fashion, by the matching of contents of Lookahead Buffer and the Current Active Node with a pattern/action grammar rule.

If the contents of the Lookahead Buffer and the Active Node Stack did not match with a pattern/action grammar rule, the contents of the Current Active Node popped into the Lookahead Buffer. Processing would be reinitiated by the matching of the contents of the Current Active Node and Lookahead Buffer with a pattern/action grammar rule. PARSIFAL also used special rules known as Attention Shift rules that allowed the parser to shift attention from the constituent in the first cell of the buffer to another constituent, if there was evidence (as Marcus called it) that this other constituent initiated a higher level constituent of a given type. These rules were used mainly for "NPs", but could initiate other constituents, e.g., clauses beginning with "as".

When the higher-level constituent had been built, the parser's attention was returned to the initial constituent with the newly formed constituent, in its place in the buffer, being able to signify the context of the constituent in the first cell. Examples of sentences that would have needed the facility of attention shift are as follows:

Have the students, who missed the exam, taken the exam today?

Have the students, who missed the exam, take the exam today.

In Figure 3.0 are snapshots of the Active Node Stack and Lookahead Buffer (Marcus, 1980, p18) part way through parsing.

```
S1 (S DECL MAJOR S)/ (PARSE-AUX CPOOL)
       NP : (John)
C AUX1(MODAL PAST VSPL AUX)/(BUILD-AUX)
       MODAL: (should)
```

Figure 3.0 (a) PARSIFAL's Active Node Stack

```
WORD3(*HAVE VERB TNSLESS AUX VERB PRES V-3S) :
       (have)
 WORD4(*SCHEDULE COMP-OBJ VERB INF-OBJ V-3S
       ED = EN PART PAST ED) : (scheduled)
 Yet unseen words:   the meeting.
```

Figure 3.0 (b) PARSIFAL's Lookahead Buffer

PARSIFAL's grammar, involved in the matching process, was based on Chomsky's theory of Transformational Grammar, i.e., Extended Standard Theory, discussed in Chapter 2. The grammar was especially based on trace theory expounded by Chomsky in the seventies; this theory places constraints on transformations. A trace is a dummy NP that represents a NP that once occupied that position in the sentence, but was subsequently deleted during movement. Transformations were implemented and traces added within parsing output. Once again the contents of the Current Active Node and the Lookahead Buffer would match with a pattern/action grammar rule, with the action part of the rule initiating the transformation or adding the trace. As a result of using this type of grammar, the parse tree built by the parser is an annotated surface structure.

The grammar comprised a series of the pattern/action rules mentioned above. The rules were written in PIDGIN and translated into LISP by a grammar translator. Figure 3.1 depicts the structure of Marcus' grammar (Marcus, 1980, p56). The rules were ordered according to numerical priority, with "0" signifying the highest priority. PARSIFAL always began processing with the rules whose patterns matched with the highest priority. If there was more than one rule with this priority that matched, then the parser could arbitrarily choose one of them.

```
-Packets of pattern/action rules
-Matched Against The Buffer, The Current Active Node
-Ordered By Priority

PRIORITY        PATTERN                    ACTION

        1st         2nd     3rd    C

                PACKET1

5:if    [...]   [...]                      then ACTION1
10:if   [...]               [...]          then ACTION2
10:if   [...]               [...]  [...]   then ACTION3

                PACKET2

10:if           [...]                      then ACTION4
15:if   [...]               [...]          then ACTION5

                PACKET3

5:if    [...]       [...]                  then ACTION6
15:if   [...]                      [...]   then ACTION7
```

Figure 3.1 Marcus' Grammar

PARSIFAL's range of coverage was a set of sample sentences rather than a text. The coverage was atypical not reflecting the grammar, for example, of an aircraft manual. Marcus listed the sample sentences in his book (Marcus, 1980).

The components of PARSIFAL now have been described. As the research is considering practical issues rather than the theoretical issues of Deterministic Parsers due to the practical nature of Machine Translation systems, it is relevant to discuss PARSIFAL in this light. PARSIFAL was designed to test theoretical issues rather than practical issues, but could it be considered as a parser that could be used in a practical situation?

PARSIFAL's grammar, as previously mentioned, is based on Chomsky's Transformational Grammar. The grammar contained a wide variety of grammar rules that covered a number of grammatical constructions. The actual structure of the packets of pattern/action rules, which represented the grammar, was rather clumsy. Each pattern/action rule could comprise many pattern and actions which made the rules idiosyncratic and overburdened with information. Berwick (1985) recognised that the rules were clumsily structured, as a result the pattern/action rules of his system (see below) were much more precise. Berwick, also, got rid of the packeting of rules, which is discussed below.

Although PARSIFAL's grammar contained several grammatical constructions, the number was few compared to those of other systems. In some practical Machine Translation systems, especially sublanguage systems, grammars are restricted and small. This suggests suitability for use of Deterministic Parsing within a Machine Translation system.

PARSIFAL's data structures, the Active Node Stack and the Lookahead Buffer, from a practical viewpoint seem sufficiently rigorous to cope with processing. However, the three buffer lookahead is regarded as being superfluous as it is considered that two buffer lookahead is sufficient; this was the conclusion reached by Milne (1983) and is discussed in the next sub-section. The Attention Shift also appears to have been a rather clumsy and superfluous mechanism. Both Milne (1983) and Berwick (1985), as discussed below, regarded Attention Shift as an unnecessary mechanism.

PARSIFAL could process test sentences containing both transformational and non-transformational type constructions. This, of course, is not proof that it could be used in a practical situation, but, in simple terms, it does prove that the parser can parse. However, the test sentences used by Marcus (1980) did not contain any lexical ambiguities, i.e., part of speech ambiguities. PARSIFAL, for example, could not cope with 'block' as a noun and 'block' as a verb. Obviously, parsers used in the analysis phase of a Machine Translation system would have to deal with part of speech ambiguity. This is a major failure in PARSIFAL which (if left unresolved) would limit its use as part of a Machine Translation system. However part of speech ambiguity was considered by Milne (1983) and is discussed in more detail in the next sub-section.

3.3.2 The Milne Parser - ROBIE

The second Deterministic Parser on which the research parsers of the Marcus type are based is ROBIE (Milne, 1982, 1983, 1986). Robert Milne, the developer of ROBIE, researched further the topic of determinism, concentrating on lexical ambiguity, i.e., part of speech ambiguity, which Marcus had not considered. Milne's parser was implemented in Prolog. ROBIE had an Active Node Stack that was identical in design to PARSIFAL's Active Node Stack. However Milne designed different structures for ROBIE's buffer mechanism - two static buffers positioned below the bottom of the Active Node Stack (Milne, 1982). The reasons Milne gave for using only two buffers were as follows: if we consider that an ATN parser had only one buffer lookahead and with this it backtracks, a two buffer lookahead must be the minimum number needed to avoid backtracking. Consequently, Milne considered three buffer lookahead to be excessive (personal communication). Milne also stated that the two buffer lookahead was sufficient to get rid of several types of lexical ambiguity; this aspect is discussed below. Firstly, as with PARSIFAL, ROBIE is now considered in the context of searching and matching.

The method of searching used by ROBIE was the same as that used by PARSIFAL. Although lookahead comprised only two buffers, similar to buffer cells in PARSIFAL, the strategy remained bottom-up with top-down filtering and control remained depth-first.

Matching within ROBIE was performed in a similar manner to that within PARSIFAL, i.e., matching on syntactic categories held in the Active Node Stack and Lookahead Buffer and resulting in the production of a parse tree. In addition, the matching process within ROBIE helped resolve some of the problems of lexical ambiguity.

To handle the problem of lexical ambiguity ROBIE's dictionary contained compound lexical entries that had attached to them all the features for all the words' possible parts of speech and thus, when words were looked up all the possible parts of speech were returned. After the look-up stage, words were morphologically checked. This process was the initial aid to disambiguation. For example, when the word 'talked' was morphologically analyzed "ed past" was added as a feature to the list of features of the word 'talk'. If 'talk' was defined as both a noun and a verb, 'talked' could not be a noun so disambiguation took place. Features corresponding to the verb, e.g., 'tenseless', were removed as were other parts of speech and corresponding features.

Morphological analysis could identify adverbs, adjectives and verbs in the same fashion.

The matching process dealt with lexical ambiguity and (as a side-effect) the productions of the parse tree, simultaneously. The matching of the contents of the Active Node Stack and the Lookahead Buffer with a pattern/action grammar rule in ROBIE also meant the matching of the features of either the one or the two buffers and once the features had matched the word had always that feature. How did the disambiguation process work? In the example, 'The falling block needs painting', the word 'falling' could be an adjective or a verb and 'block' could be a noun or a verb (Milne, 1986). Having parsed 'the', which would have started an "NP" node and been attached to it as a determiner, the rule to parse an adjective was activated and, thus the matching of the rule ADJECTIVE with the contents of the Active Node Stack and the Lookahead Buffer. The word 'falling' was then attached, disambiguated as an adjective and therefore was not considered to be a verb. There were no more adjectives, so the rule to parse the head-noun would be activated due to the matching of contents of buffer and stack. 'block' matched with noun and so would be attached as a noun. Yet again the word would not be considered as a verb. Milne stated that the following noun phrase ambiguities could be treated likewise: 'singular head-nouns', 'verb/adjective ambiguity' and other 'pre-nominal ambiguities' because of the word-order of the "NP". Main verbs could also be disambiguated in this way.

Matching aided the process of disambiguation in other examples of part of speech ambiguity. However, certain other rules had to be called upon to aid disambiguation. For example, ROBIE could disambiguate between 'to' as an auxiliary verb and a preposition using the method previously described, but, ungrammatical sentences, such as the example below, would be parsed as being grammatical by ROBIE.

I want to the school with you. (Milne, 1986)

Milne suggested that 'verb subcategorization' would solve the problem of ungrammatical sentences. In certain cases rules could be introduced that could handle "VPs", i.e., 'to' could only be an auxiliary verb starting a "VP" when the verb took the infinitive complement. Thus a rule could be called only when an infinitive was permitted. If verbs for "PPs" with 'to' as a preposition were classified this would help disambiguation. ROBIE could also check verb and subject agreement to aid disambiguation of phrases which included 'that', 'which', 'what' and 'have'.

Milne found that adjective/noun ambiguity and noun/noun ambiguity were difficult to handle and only proposed a simple method that would not handle all cases, e.g.,

The old can get in for half price. (Milne, 1986)

could not be disambiguated correctly by ROBIE (Milne, 1986). Other cases of ambiguity Milne suggested could be resolved by checking agreement; that is by looking at the structure of verb groups and person/number codes.

ROBIE did not use Attention Shift rules since Milne considered that two buffers could handle most of the cases covered by PARSIFAL's Attention Shift rules. In those cases where three buffers seem to be needed Milne suggests that a non-syntactic processor would be called. Again this would be part of the matching process. Each grammar rule had as its final action a 'recursive call' to a rule matcher that searched the patterns of each rule in each active packet comparing the pattern with the current state of ROBIE. The rules also called semantic routines as constituents were built. This non-syntactic processor used the name of the rule that was executing, the Current Active Node and the two buffers to obtain information it required. These semantic routines produced predicate calculus assertions of the semantic information gleaned (Milne, 1983).

Similar to PARSIFAL, ROBIE's grammar was based on Chomsky's Extended Standard Theory. ROBIE's grammar, as stated, was also made up of packets of pattern/action rules, eighty percent of which were identical to PARSIFAL's. The rules worked in exactly the same fashion as in PARSIFAL.

ROBIE's range of coverage was that of PARSIFAL with the addition of an extension to cover sentences from the MECHO world, related to Mechanical engineering. MECHO was not a text, but a set of sentences relating to mechanical engineering problems.

The main features of Milne's parser have been described. As with PARSIFAL, the discussion now considers ROBIE from a practical viewpoint. ROBIE's Active Node Stack, as stated, was similar to PARSIFAL's in that it has the same structure and performs the same function.

ROBIE's grammar was similarly structured into packets of pattern/action rules, and guided by a numbering priority. The rules were similarly large and cumbersome containing seven functions that rules could perform. As stated in relation to PARSIFAL, this has proved to be unnecessary as rules do not have to contain so much information, see discussion on Berwick below.

ROBIE's Lookahead Buffer was different from the Lookahead Buffer used by PARSIFAL. ROBIE's buffer contained only two cells whereas PARSIFAL's held three cells. Milne's reasons for two buffer cells being sufficient to cope with the processing of the test sentences are valid. However, the method for dealing with those sentences which could not be processed within two buffer cells, by using a semantic processor, could be regarded as a simple way of solving the problem, but probably the best way. The two cell Lookahead Buffer could still be considered a rigorous data structure that could be part of a parser used in a practical situation.

Another difference between the two parsers was that ROBIE did not use attention shift. The reason given by Milne for this relates to the discussion above whereby sentences that could not be processed with a two cell buffer would call up the semantic processor. Therefore, sentences processed by PARSIFAL with attention shift were processed by ROBIE by using a call to the semantic processor. This does not pose a problem from a practical viewpoint as one of the criticisms of PARSIFAL was its failure to deal with semantics. The facility for dealing with semantics enhances the parser for use in a practical situation.

The major difference between PARSIFAL and ROBIE is the latter's capability to deal with various part of speech ambiguities. It cannot cope with all ambiguities of this type, but this does not detract from having the capability of dealing with ambiguity. It enhances further the facility of semantic processing, which itself enhances ROBIE as a Deterministic Parser for use in a practical situation.

As with PARSIFAL, the type of practical Machine Translation system for which ROBIE would seem suitable to use in the analysis phase is a sublanguage/restricted language system. ROBIE's grammar is restricted and small, which as highlighted above can be a feature of sublanguage systems. The semantic processor and the dealing with part of speech ambiguity are restricted but these are also aspects of restricted Machine Translation systems.

3.3.3 The Berwick Parser - L.PARSIFAL

Berwick's parser L.PARSIFAL was a version of PARSIFAL used as part of an implementation to acquire parsing rules to parse English syntax (Berwick, 1982, 1985). The changes Berwick made were to simplify the parser, so that it would be easier for his acquisition procedure to learn, and have a more standard design (Berwick, 1982). L.PARSIFAL had an Active Node Stack and a three cell Lookahead Buffer similar to PARSIFAL. It was implemented in LISP without the use of PIDGIN. However the grammar rules of the parser were somewhat different. The rules, which were of a pattern/action type, were acquired whilst processing the input sentences; this provided the learning theory. As mentioned, they were of a pattern/action type but there were also base phrase structure rules that corresponded to PARSIFAL's packets. These rules were used for activation and deactivation. The standard design that Berwick used was that of a bottom-up parser.

As with the previous parsers discussed, the L.PARSIFAL parsing system will be considered from the viewpoint searching and matching. The searching aspect of L.PARSIFAL was exactly similar to that of PARSIFAL and ROBIE, i.e., the strategy was bottom-up with top-down filtering and control was depth-first.

The matching process in L.PARSIFAL, as with the previous parsers discussed, resulted in producing the parsing output. The matching was on syntactic categories; the contents of the Active Node Stack and Lookahead Buffer matched with a pattern/action grammar rule. The action part of the rule, as with ROBIE and PARSIFAL, manipulated the contents of the stack and buffer to produce the eventual output. However, if the parsing system was in acquisition mode, the matching on the stack and buffer contributed to building a grammar rule.

In L.PARSIFAL, matching produced the output in a manner similar to most bottom-up parsers, whereas PARSIFAL and ROBIE were considered to deviate from standard bottom-up parsers, as noted below. All three parsers deviated from bottom-up parsers in that they predicted the parent node of the subconstituents. However PARSIFAL and ROBIE dealt with constituents individually, i.e., pushed constituents onto the stack without waiting for the analysis of the last constituent. In L.PARSIFAL it was not necessary to push the individual constituents whose parents were known onto the Active Node Stack until all constituents were processed, due to the use of dotted rules, discussed below.

Another difference between PARSIFAL and ROBIE and L.PARSIFAL is the type of output produced. L.PARSIFAL did not produce a tree structure output: rather complex non-terminal symbols, dotted rules, represented the state of the parse. A dotted rule is a context free rewrite rule with a marker dot placed at some point in the right-hand-side of the expansion showing how much has been recognized of the rule. If the grammar rule is S -> NP VP and the sentence, Joan liked the boy, which has had it first word processed, in this case the proper noun Joan which is the "NP", this would be represented in dotted rule notation by:

S -> NP * VP

and when the VP constituent is processed, the dot will move past it, signalling that the S had been completely processed.

PARSIFAL's packet names turned out to be in one-to-one correspondence with dotted rules. For example, the call of the packets in PARSIFAL to parse a simple declarative sentence would be the following:

1. Sentence-Start 2. Parse-Subject 3. Parse-Aux 4. Parse-VP

which in one-to-one correspondence would gives us the associated re-write rule S -> NP AUX VP, with dotted rules:

S -> * NP AUX VP

S -> NP * AUX VP

S -> NP AUX * VP

S -> NP AUX VP *

These dotted rules were easily translated into the X-bar notation of the grammar, discussed in the previous chapter and below. Unlike in PARSIFAL where more than one packet of rules could be active at one time, L.PARSIFAL had only a single template packet active which related to one of the three elements of the template in X-bar notation, e.g., Parse-specifier, Parse-head, or Parse-complement.

L.PARSIFAL similar to ROBIE had no attention shift mechanism. In place of the mechanism, L.PARSIFAL was designed to reduce the second leftmost complete constituent before the first leftmost complete constituent when processing the following type of sentences.

Have the students take the exam today!

Have the students taken the exam today? (Berwick, 1985)

Matching on the contents of the stack and buffer with a pattern/action grammar rule brought about the above procedure.

In the L.PARSIFAL parsing system, the grammar was based on Chomsky's X-bar theory. As stated in the previous chapter, X-bar theory shows that the base phrase structure system of a particular language is fixed by template filling decisions (Berwick 1985). As well as not having a full packeting system, L.PARSIFAL's grammar rules themselves were modified as compared with PARSIFAL and ROBIE. In L.PARSIFAL the grammar rules did not have a numbered priority where higher priority rules were able to execute before those of lower priority. Instead, specific rules executed before general rules. A specific rule was a rule with more pattern matches called for in the first buffer cell. If this failed the second buffer was checked. If there were no specific rules, the general rules would then fire. In Figure 3.2 are examples of similar grammar rules used by PARSIFAL and ROBIE and L.PARSIFAL (Berwick, 1985).

```
Main-verb in packet  parse-vp
 priority: 10
 IF: The first element in the buffer is a verb
 THEN  DEACTIVATE packet parse-vp
        if  the active mode is a major sentence
        then  ACTIVATE packet ss-final;
        else if   the active node is a secondary clause
        then  ACTIVATE packet emb-s-final.
 CREATE a VP node.
 ATTACH the VP node to the S.
 ATTACH the first element in the buffer to
 the active node as a verb.
 ACTIVATE the clause level packet cpool
    if  the verb is labeled passive
    then  ACTIVATE the packet passive
    and RUN the grammar rule passive  next.
    if  the verb takes an infinitive without to
    then  ACTIVATE the packet to-less-inf-obj
```

Figure 3.2 (a) Example of a PARSIFAL grammar rule

```
Main-verb
IF
    current active node is S
    current cyclic node is nil
    1st buffer cell is V
THEN
    ATTACH
```

Figure 3.2 (b) Example of a L.PARSIFAL grammar rule

The range of coverage of L.PARSIFAL was similar to that of PARSIFAL. The sentences were not from a definite text, rather a set of sentences that represented that grammatical constructs of the grammar.

Finally, L.PARSIFAL is consider in terms of practability. L.PARSIFAL has been described as a simplified version of PARSIFAL. Its data structures are similar, i.e., it has an Active Node Stack and a Lookahead Buffer with three cells of lookahead which are considered rigorous enough for use in a practical application.

The simplified grammar rules of L.PARSIFAL, compared with the packets of pattern/action rules used by both PARSIFAL and ROBIE, contribute in making the parser more beneficial for use in a practical system because of their precise structure and compact nature. The compact nature of L.PARSIFAL's grammar i.e., the reduced size of the grammar rules would mean less processing time which would make a system using these type of rules more efficient than that using rules of the PARSIFAL or ROBIE type.

L.PARSIFAL could not cope with lexical ambiguity which, as stated in relation to PARSIFAL, is a major lacking for a parsing system to be used as part of a practical system. However, as in the case of ROBIE it has been proved that a deterministic parser can cope with aspects of lexical ambiguity; the same method could be applied to L.PARSIFAL. Similarly to PARSIFAL and ROBIE, L.PARSIFAL can be considered as suitable for use in a practical system, such as the analysis stage of sublanguage/restricted language Machine Translation system.

3.3.4 Other Deterministic Parsing Systems

As stated in the introduction to this chapter, other Deterministic Parsers of the Marcus type have been developed. In this sub-section some of these other deterministic parsers are discussed. The majority of the deterministic parsing systems that have been developed since PARSIFAL have similar data structures and use grammars of pattern/action rules although the purpose of the systems has been different. Similarly to PARSIFAL, ROBIE and L.PARSIFAL, most of the parsers discussed below were

ASTON UNIVERSITY
LIBRARY AND

built as theoretical systems. At the end of this sub-section, the prospect of using these parsers in a practical system is discussed.

A modified form of PARSIFAL designed to be a Finite State Deterministic Parser was YAP (Church, 1980). YAP had two buffers, the Upper Buffer which was similar to PARSIFAL's Active Node Stack and the Lower Buffer which was identical Lookahead Buffer. The grammar also comprised pattern/action rules and was referred to as a Deterministic Finite State Control Device. However, as Church states,

> " The problem with writing a grammar in production rules (pattern/action rules) is that the performance and the competence components tend to become tangled; it is very difficult to write a good structured program (grammar) with elementary building blocks." (Church, 1980, p49)

Church, thus, decided to use 'phrase structure rules', so that the next action could be selected in orderly fashion. The phrase structure component could cover most normal 'unmarked' cases, pattern/action rules were used for 'marked' examples.

Another Marcus type parsing system where the grammar rules were simplified by use of phrase structure rules was PARAGRAM (Charniak, 1983). PARAGRAM was developed to test whether a syntactic parser could handle ungrammatical sentences. The data structures have not been described by Charniak (1983), but he does state that there were many differences between PARAGRAM and PARSIFAL, although there was only a small modification to the original design to allow for the processing of the ungrammatical sentences.

Three systems which followed the Marcus design dealt with semantic aspects of processing, in addition to ROBIE (Milne, 1983). Firstly, Lesmo, Magnini and Torasso (1981) built an analyzer to be used as a natural language front end to a medical database of liver complaints. The natural language used was Italian. The analyzer incorporated a lookahead buffer and a grammar of pattern/action rules. However, instead of building or storing a structure on an Active Node Stack, the interpretation process translated the input command into a set of frame instantiations, linked together. A frame was a collection of slots. The semantic analysis was incorporated into the action part of the grammar rules which built representations of the constituents in a manner which would allow semantic checking by consulting the semantic information held in the lexicon.

A second system which incorporated semantics was designed by Sabah and Rady (1983). In this instance, the parser could cope with both English and French syntax. The parser incorporated a lookahead buffer and a grammar of pattern/ action rules. As with the analyzer described above, the Sabah and Rady parser did not have an Active Node Stack. A final syntactic structure was produced, but the more important representation produced, which was required, was of a semantic nature. The pattern/action rules again dealt with both the syntax and semantics.

Finally, the third system, dealing with semantics, was developed by Carter and Freiling (1984). The aim when developing the parser was to produce a parser that would make simple the writing and understanding of deterministic grammars. As with the two previous parsers discussed, the Carter and Freiling parser incorporated a lookahead buffer and grammar of pattern/action rules, but not an Active Node Stack. The pattern/action rules dealt with the syntax and semantics.

The above parsers are now considered in the searching and matching context of parsing. With regard to searching, the strategy used by all parsers was bottom-up with top-down prediction and the control, once more, depth-first. The matching process of the syntactic categories of the Lookahead Buffer and Active Node Stack, or its equivalent, with pattern/action grammar rules was similar to that of PARSIFAL. ROBIE and L.PARSIFAL. The action part of the rules also produced the parsing output. All the purely syntactic parsers produced tree structures and those that dealt with semantics produced a semantic representation, in most instances, in addition to a syntactic structure.

The majority of the grammars of these systems, where mentioned, were based on Chomsky's Transformational Grammar. The coverage of all but one of these parsing systems was a set of sentences that represented respective grammatical constructs of the grammars. The coverage of the Lesmo, Magnini and Torasso analyzer was a text related to Liver Complaints, which classifies it as one of a very few practical deterministic parsing systems.

Most of the parsers that have been discussed in this section, as stated, were designed as theoretical systems. The question of whether any of these parsers could be incorporated as part of practical systems is now discussed, accepting that data structures and simplified grammar of systems previously discussed are suitable.

The parsers YAP and PARAGRAM both had grammars with simplified rules as compared with the cumbersome and idiosyncratic pattern/action rules used by PARSIFAL. In the discussion on L.PARSIFAL, the use of the simplified rules was highlighted as being beneficial to a practical parser, which suggests that both YAP and PARAGRAM could be considered for practical applications. However, both parsing systems had relatively small theoretical type grammars which would be unsuitable in practical systems. The use of practical grammars in each of the parsing systems is not considered to be an impossibility: once more a sublanguage/restricted language grammar would seem easier to test because of the restriction on size and type of grammar rules.

The practical use of the three systems that incorporated semantics is varied. The deterministic analyzer (Lesmo, Magnini and Torasso, 1981) was already part of a practical application by providing the natural language interface to a medical database. The developers did not discuss successes or failures of their system, but by adopting the deterministic concept they must have been convinced of its potential. The remaining two parsers were theoretical parsers and were very similar in design and structure; neither incorporated an Active Node Stack into their system. Both parsing systems had small grammars of pattern/action rules but the concern was the semantic as well as the syntactic output of the parsing process. Yet again, the use of practical grammars by the systems would provide good test ground for use in practical systems.

3.4 Deriving the Marcus Type Research Parsers

The parsers PARSIFAL, ROBIE and L.PARSIFAL, discussed above, form the basis of the Marcus type research parsers that have been built for this research - MParser and MParserSub. Both MParser and MParserSub comprise the data structures of the Active Node Stack, two cell Lookahead Buffer and grammar of pattern/action rules of the type used by L.PARSIFAL: these components were considered to be the most suitable for use in a practical parser. However, MParser has not been built for use in a practical system. Rather, it was built as a prototype from which MParserSub, the proposed Machine Translation system parser, was developed. MParser, therefore, had to be built from a practical viewpoint. MParser and MParserSub are discussed in more detail in Chapters 5 and 7.

The Active Node Stack was used in all three parsers. It had proved to be capable of handling and storing grammatical structures. No problems were, therefore, foreseen in using the Active Node stack as part of MParser or MParserSub.

A similar argument is given for employing the Lookahead Buffer, i.e., it could handle grammatical structures when used as a theoretical parser, so the assumption was made that it could be part of a practical parser. However, consideration was taken of the fact that neither the two cell nor three cell lookahead buffer had been fully tested on all types of sentences or phrases. It is possible that certain sentences or phrases could turn up in a practical situation which MParser or MParserSub, with a two cell lookahead buffer, could not handle.

The grammar of pattern/action rules used by MParser adopts the same format and type as that used in the L.PARSIFAL system. This was expected to prove to be suitable because similar grammars were being used. But, the grammar used by MParserSub was to be derived from an aircraft maintenance manual and would, therefore, contain untested grammatical structures that had not appeared in the grammars of PARSIFAL, ROBIE or L.PARSIFAL. However, the sublanguage/restricted language domain of aircraft maintenance manuals would mean the grammar would be of a restricted nature, i.e., a smaller grammar with fewer types of grammatical structures.

3.5 LR Parsers For Natural Language Processing.

In this section an LR parser, which has been used in natural language processing and on which the LR type parser developed for this research is based, is examined. The LR parser is a variant of the deterministic LR parsing technique described by Aho and Ullman (1972) for use in parsing programming languages. The parser that will be discussed is the parser developed by Shieber (1983) and Pereira (1985).

3.5.1 The Shieber and Pereira Parser

The parser developed by Shieber (1983) and Pereira (1985) was a variant of the

LALR(k) parser (Aho and Johnson, 1974) ,which itself was a specialization of LR(k) parser. The LALR(k) parser is a shift-reduce parser that works in a bottom-up manner.

As with the Marcus type parsers, the Shieber and Pereira parser will be considered in the context of searching and matching, the procedures which constitute parsing. Within the searching procedure, the parsing strategy used by the Shieber and Pereira LALR(1) parser is purely bottom-up with no top-down prediction of phrase types. Using a bottom-up parser of the LALR type, the example phrase given above 'with Mary' would be processed in the same manner as above, i.e., respectively as preposition and proper-noun; the proper-noun would be reduced to a noun-phrase and, finally, prep and noun-phrase reduced to a prepositional phrase.

The method of control used by the LALR(1) parser is depth-first. Similar to the Marcus type parsers only a single branch of the search tree would be examined because of the limitations of determinism. The Shieber and Pereira LALR(1) parser had one cell of lookahead which aided the search through the single branch of the search tree.

Matching in the LALR(1) parser is on syntactic categories, the final result being the production of the parsing output, a parse tree. The components which made-up the LALR(1) Shieber and Pereira parser were an input buffer, a stack for storing constituents constructed during the parse and a parse table that guided the parse. The parse table is produced by manipulation of the grammar. This process is discussed in more detail in Chapter 6. At every stage in the parse, the parser consults the parse table or shift reduce table, as it is sometimes known. Matching took place at this stage of processing. The contents of the parse table match with either the syntactic category in the first cell of the input buffer or the top node in the stack. Matching resulted in the parser making either a shift move, when matching was on the first cell of the input buffer, or a reduce move when matching was on the top node of the stack. The shift comprises of removing the next word from the input sentence on to the top of the stack and reduce takes constituents from the top of the stack and reduces them into a new constituent that is placed back on the top of the stack, e.g., if the top of the stack held a Det and a Noun the result of the reduce operation would be to replace them at the top of the stack with a NP. After matching and the shift or reduce operation have taken place, the whole process is repeated until processing is complete.

Within the matching process of a true LALR(1) parser, problems would have occurred in dealing with a natural language grammar. Shieber (1983) and Pereira (1985) built their parser so that it would work in the same manner as Marcus' parser, i.e., that there would be no simulation of non-determinism. However there was a difference. The Marcus parser could only use an unambiguous grammar, whereas with the LALR(k) parser, Shieber and Pereira used an ambiguous grammar.

The use of an ambiguous grammar made the LALR(1) Shieber and Pereira parser a variant of the true LALR(k) algorithm, but in order to be able to parse natural language the parser had to be amended to allow it to cope, in this case, with the syntax of English. Since the grammar was ambiguous this meant that the parse table held conflicting actions at certain points. Shieber (1983) dealt with this by introducing rules that would, when there was a conflict between shift and reduce (i.e., matching with the parse table could be on both the first buffer cell and the stack) choose the shift option and when the conflict was between two reduce options (i.e., matching of the top of the stack with the parse table could result in two different reductions) choose the reduce option that would perform the longer reduction.

Another variation added to the parser was that of preterminal delaying introduced by Shieber (1983). In certain situations, a word among preterminals may be ambiguous: where this is the case the assignment of the preterminal symbol can be delayed until the ambiguity is solved. For example, the preterminal 'that' could be a determiner or a complementizer and further information would be needed before disambiguating between the two; the following two sentences are examples of this.

That problem is important.

That problems are difficult to solve is important. (Shieber, 1983)

In both of these examples the assigning of the preterminal determiner or complementizer is delayed until the point where the first reduction takes place involving the word. In the first case the reduction will be to NP because of the rule NP --> Determiner, Noun and the preterminal can be assigned Determiner. In the second case, the rule NP does not apply because of number agreement and the first possible reduction would be Complementizer S to S.

Shieber and Pereira used the above changes to prove that the parser could account for theories about preference, i.e., Right Association and Minimal Attachment.

> Right Association, as put forward by Kimball, cited by Pereira, (1985), is the principle that in the absence of other information phrases are attached to a partial analysis as far right as possible.
>
> Minimal Attachment, as put forward by Frazier and Fodor, cited by (Pereira, 1985), has the principle that in the absence of other information phrases are attached so as to minimize the complexity of the analysis.

Pereira (1985) found that the problem of Right Association involved a shift-reduce conflict and thus could be solved by shifting in preference to reducing, whereas Minimal

Attachment involved a reduce-reduce conflict and could be solved by choosing the longer reduction. Shieber (1983) also looked at the theory that attachment preferences depended on lexical choice. He amended the rule concerning reduce-reduce conflicts so that longer reductions would be performed with the strongest leftmost stack element, preterminals word pairs having been assigned with information as to whether they were weak or strong.

As with the Marcus type parsers, it is pertinent to the discussion to consider whether the Pereira and Shieber parser would be of use in a practical system. However, similarly to Marcus type parsers, the Pereira and Shieber parser was built as a theoretical parser. Although the Pereira and Shieber parser had been already amended to cope with those aspects of natural language as mentioned above, the grammar used by the parser was very small. As it stood, the parsing system could not be recommended for use in a practical system.

However, the parser, itself, - the stack, the buffer and the parse table - can be regarded as being suitable for use in a practical system, as it has been incorporated in many compilers and proved to be suitable. For the whole parsing system to be suitable, a larger grammar, containing a much more varied group of grammatical constructs than that found in the Pereira and Shieber Parser, would need to be incorporated into the system. However, the variety of the grammatical constructions may not have to be that varied if the practical system of which the parser was to be a part were a sublanguage/restricted language system where there is typically a restriction on the range of grammatical constructs.

3.6 Deriving the LR Research Parsers

The Shieber and Pereira Parser was the basis from which the LR research parsers were developed. As with the Marcus type parsers, two parsers, one a prototype, would be developed for use in a practical situation. LParser would be the prototype from which LParserSub would be developed.

The components incorporated into LParser and LParserSub are similar to the components used in the Shieber and Pereira parser, i.e., a stack, an input buffer and parse table derived from a grammar. The major difference between the Shieber and Pereira parser and the research parsers would be the type and size of grammar. A Definite Clause Grammar (DCG) would be the grammar used by both LParser and LParserSub: LParser's DCG would be based on the grammar used by L.PARSIFAL and LParserSub's DCG derived from the aircraft maintenance manual. LParser, LParserSub and Definite Clause Grammar are discussed in more detail in Chapters 6 and 7.

CHAPTER 4

SUBLANGUAGE

4.1 Introduction

In this chapter the discussion is concerned with sublanguage. The term sublanguage came to prominence with the publication of the book "Mathematical Structures of Language" by Zellig Harris. This work, cited by Kittredge and Lehrberger (1982), reported research in the theory of language structure in the area of scientific texts. Harris' definition of sublanguage was as follows:

> " Certain proper subsets of the sentences of a language may be closed under some or all of the operations defined in the language, and thus constitute a sublanguage of it."
>
> (Kittredge and Lehrberger, 1982)

Implicit in this definition is the assumption that sublanguage is similar to a subsystem in mathematics. Harris, as cited by Kittredge and Lehrberger (1982), considered that sentences that made-up a sublanguage were a subset of the whole language, but the sublanguage grammar did not have to be part of the grammar of the whole language; both grammars could, in fact, intersect.

In section 4.2, more recent definitions of sublanguage are examined. This is followed, in section 4.3, by an examination of examples of the restrictions found in sublanguages, i.e, the restrictions on semantics, grammar and syntax. In section 4.4, a specific definition about the formation of a sublanguage grammar is described. This is based on work done by Naomi Sager (Sager, 1986). In section 4.5, the discussion focuses on the significance of using a sublanguage within Machine Translation. Finally, in section 4.6 the discussion focuses on whether deterministic parsing is a suitable parsing method for the processing of sublanguage within Machine Translation. The section concentrates on sublanguage in general and not specifically on the research area sublanguage.

4.2 The Definition Controversy

In recent years, the definition of sublanguage has come under much scrutiny. It has been argued, justifiably, that a sublanguage of natural language cannot be compared with a mathematical subsystem, since the boundaries of both sublanguage and natural language are not rigid. However can it be said that a sublanguage is an independent system, unrelated to the whole language? This problem will now be discussed as highlighted in Lehrberger (1986) and Fitzpatrick, Bachenko and Hindle (1986).

Lehrberger put forward five points that he thought should be considered when defining a sublanguage:

A sublanguage of a natural language L is part of L.

A sublanguage is identified with a particular semantic domain.

A sublanguage of a natural language L grows in a natural way through the use of L, albeit in special circumstances.

What we refer to as sublanguage texts usually contain some material that does not belong to the sublanguage proper.

Whatever can be said in a sublanguage of a natural language L can be paraphrased in Lstd (Standard Language). (Lehrberger, 1986)

Lehrberger took each of the above points in turn and discussed their significance. This discussion is now considered.

Lehrberger began his discussion of the first point by suggesting the following scenario. If a person ignorant of the field of nuclear physics were to read some article on the subject, he or she would not necessarily understand the topic but he or she would recognize the language that it was written in. According to the above, in this situation it could be said that the sublanguage of a natural language L is part of L, but as Lehrberger suggested it also could be said that the grammar of the sublanguage of the language is a subgrammar of the language. This would not be true since constructions could exist within the subgrammar that would not belong to a grammar of natural language, such as the telegraphic construction, predominant in telegraphic sublanguages.

Regarding the second point, according to Lehrberger, the majority of investigations into sublanguage have been concerned with writings in scientific and technical areas with restricted semantic domains, but Harris' definition of sublanguage did not mention any limitations of semantics. In a later paper cited by Lehrberger, (1986), Harris called such sublanguages subject-matter sublanguages and suggested that they did not contain sentences of their own metalanguage; natural language, in general, had its own metalanguage, the grammar of the language.

The third point is connected with the first two points. Lehrberger considered that a sublanguage, like natural language, is not constructed but develops through use by groups of specialists communicating with each other about their specific area. However, Lehrberger noted specially constructed languages had not been studied in great detail, so it was difficult to make any claim about their growth patterns.

The fourth point is concerned with information that may be in the form of sentences or phrases that appear throughout the sublanguage text but which are not part of the sublanguage proper. There exists in certain texts two levels, one level concerns the actual subject matter, the other, perhaps a discussion about the subject matter. The result of this, as Lehrberger stated, was a discourse within a particular domain and a metadiscourse about it. It is possible to say that any text that appeared frequently in the writings of a specific area was part of the sublanguage of that area. Thus, a grammar of a sublanguage may have in it more than constitutes the sublanguage grammar.

A suggestion to clear up the point, as put forward by Lehrberger, was to state that there is a difference between sublanguage and discourse. If it was considered that it was the discourse that would be processed, then the sublanguage grammar would be larger to accommodate it. However by allowing the discourse analysis, based on the grammar, to distinguish between discourse and metadiscourse this grammar would have a component to match with the grammar of the actual sublanguage and rules for dealing with the metadiscourse and how it corresponded to the rest of the discourse.

For the fifth point the concept of Lstd was introduced by Lehrberger which was considered to be a grammar of the standard language, a more comprehensible grammar than that of the whole language, L. It was also assumed that the sublanguage consisted of deviant grammatical constructions that could be replaced by constructions of Lstd. This was not simple, since certain examples of deletion are known only to the specialists in the sublanguage and need to be explained by discourse rules and pragmatic principles.

The stance taken by Fitzpatrick, Bachenko and Hindle (1986) was that telegraphic sublanguages, e.g., the language of military messages, patient records, newspaper headlines and weather reports, should be regarded as independent systems. They took the view that Lehrberger's first point, highlighted above, was incorrect in that they believed telegraphic sublanguages were not part of standard language but independent systems. They argued that the view taken by Kwasny and Sondheimer (1981) that sublanguages were ill-formed in relation to standard language (implying that they were to be regarded as ungrammatical) was wrong. (Kwasny and Sondheimer refer to telegraphic sublanguages as being elliptic, with a need to be treated as special with regard to the grammar.) Fitzpatrick, Bachenko and Hindle also did not agree with the theory that sublanguages were reduced forms of language, with their sentences regarded as suitable in specific areas. They believed that the error in the above assumptions was that a telegraphic sublanguage was derived from deletions in the standard language and thus could only be understood by reinserting the deletions. In essence, they did not agree with the assumption that a telegraphic sentence is dependent on a full standard language sentence.

In their research on telegraphic sublanguages, looking specifically at Navy messages, Fitzpatrick, Bachenko and Hindle regarded the telegraphic sentence as a full form, not a sentence with deletions. Therefore, their definition of the telegraphic sublanguage was that it was not based on standard language, but had its own internal consistency. In order to substantiate the above claim they had the restriction (see below) of using only the transitive or intransitive meaning of a verb. They state that it was not possible to prove that a restriction on transitivity forms part of the navy-message sublanguage. However Harris' statements on semantic domains of sublanguages being subsets of those of the standard language gave credence to their theory, i.e., lexical options appearing in the standard language may not appear in the sublanguage.

Fitzpatrick, Bachenko and Hindle (1986) discussed areas in navy-message sublanguage, which, it was stressed, needed a restriction on verb transitivity to produce a correct analysis of text. Other areas considered were the gapping of noun phrase objects, passivization and apparent syntactic ambiguity. In the discussion on the gapping of noun phrase objects the following were given as examples:

Five man hours expended to correct _

Attempt to procure _ locally to deliver _ on 05 April

Ship's technician will repair _

System currently unable to process _

(Fitzpatrick, Bachenko and Hindle,1986)

In the navy-message sublanguage the verbs correct, procure, repair and process were all transitive and could not be intransitive because of the transitivity restriction. The assumption was made that the gap, found during syntactic processing, would be filled with a noun phrase provided by the semantic processor. This is not the view taken by those who follow the deletion theory since they would regard the gaps as syntactic deletions of the full noun phrase object. Having the gapped object in the sublanguage is dependent on the above verbs never being intransitive. As Fitzpatrick, Bachenko and Hindle observed, this assumption was not true for Standard English and hence could not be true for the syntactic deletion approach. The three remaining areas were discussed thoroughly, with the conclusion being drawn that the telegraphic sublanguage has a grammar made-up of a subset of rules of the standard language and other interdependent rules that were peculiar to the sublanguage. This rule interdependence along with being sublanguage-specific, showed the sublanguage grammar to be independent in relation to the standard language (Fitzpatrick, Bachenko and Hindle, 1986).

Lehrberger (1986) believed that independent systems could be relevant in practical applications of certain domains, since it was possible that a greater economy could be gained in their description. However, he also believed that there were reasons for describing some sublanguages with reference to the structures and operations of the standard language because particular applications use hierarchies of sublanguages with overlap occurring between sublanguages in different hierarchies. Sublanguages with non-similar semantic areas could be grammatically related; how they were related may be shown with respect to standard language. Variant sublanguage texts may contain metadiscourse expressions of similar type, so that even grammars of practical applications could be connected with the grammar of the standard language.

In subject-matter sublanguages, grammaticality depends on the norms of usage of the field's specialists. It could occur that what is grammatical in the sublanguage is not grammatical in standard language; the structures of the sublanguage conform to an internal consistency and could be viewed as independent systems. However, Lehrberger stated that considering

> " ...sublanguage as part of a larger entity (the whole natural language) then sublanguage structures that do not conform to the standard grammar can legitimately be described as deviant." (Lehrberger, 1986)

Structures also could be regarded as marked, the subject-matter restriction functioning as part of extra linguistic context. All this, Lehrberger (1986) stressed, gave strength to the part-whole relation between sublanguage and standard language with the standard language being a point of reference.

4.3 Restrictions

The variety of restrictions found in sublanguages are now considered. These restrictions can be of the following types: lexical, syntactic, semantic or may represent constraints in discourse or individual sentences. Lexical restrictions are examined first.

The most well known sublanguages as discussed to date in the literature have been the language of weather broadcasts, maintenance manuals, stock market reports and medical reports. In the sublanguage of aircraft maintenance manuals it would be expected to find words such as aileron, motor compressor, jack or filter but not cabbage, belief, hope. However words are not exclusive to one particular sublanguage; filter can appear in both the language of pharmacology and aircraft maintenance manuals (Lehrberger, 1982).

The restrictions on vocabulary vary between categories. Nouns, verbs, adjectives and adverbs are the most restricted categories (Lehrberger, 1982), whereas the majority of members of other categories, e.g., coordinate conjunctions and articles, are found in most sublanguages. This emphasizes that the semantic burden is carried by nouns and verbs.

Syntactic restrictions depend on the type of sublanguage, e.g., descriptive. In a descriptive sublanguage it would not be usual to find direct questions or exclamatory sentences. In the particular example of aircraft maintenance manuals, the simple past tense is not used. But this is not to say that syntactic restriction means that the structure of a sentence will be simple in sublanguages. It is still possible to have complex sentences with, e.g., passives, nominalization, conjunctions and relative clauses; these sentences can be long, even if they belong to a telegraphic sublanguage. Syntactic restrictions do not ease completely the linguistic problems met by the parser.

The semantic restrictions of a sublanguage carry a greater importance than any restriction on vocabulary size. The results of such restrictions can mean that there is reduction in polysemy and words being attached to only one category. Examples from an aircraft maintenance sublanguage are the following, with * signifying the category not accepted by the sublanguage:

Case (N)	*Case the joint.
Lug (N)	*They lugged the equipment from the plane.
Cake (V)	*The pilot likes banana cake.
Jerky (Adj)	*Carry a pound of jerky on long flights.
Cable (N)	*Cable the forward department.

(Lehrberger, 1982)

There are other words which can be restricted within their categories, e.g.,

Eccentric (Adj)	Cannot apply to animate objects.
Ball (N)	Can only be a spherical object.
Check (N)	Abstract only.
Bore (V)	Cannot take human object.

(Lehrberger, 1982)

Other semantic restrictions that can occur depend on the number and kinds of semantic features needed for parsing (Lehrberger, 1982). Nouns which can take either concrete or abstract objects in the standard language may take only the concrete in a sublanguage. Also, the amount of semantic restriction in a sublanguage has an effect on the way semantic features are represented. Two methods, suggested by Lehrberger (1982) for implementing this restriction were using the system of assignment of unary or binary features. By these methods, if binary features were used, nouns such as air, oil, water, etc., would be assigned +fluid the remaining nouns in the dictionary would then be assigned -fluid and if the unary features were used the above nouns would be assigned +fluid and the remaining nouns would not have any assignment attached.

As Kittredge (1982) observed various sublanguages employ in different ways a language's linguistic means of textual cohesion. Many sublanguages drop the use of the definite article and copula. As well as this restriction, adjectives can be constrained to appearing in only the predicative or attributive position. In the language of aircraft maintenance manuals many adjectives have the attrib feature assigned to them and thus cannot appear in the predicate position. The above are just some of the restrictions found in sublanguages and give an outline of the depth of the restrictions.

4.4 Defining A Sublanguage Grammar

Naomi Sager addressed this subject in her article Sublanguage: Linguistic Phenomenon, Computational Tool from where much of the information is drawn (Sager, 1986). To form a sublanguage grammar it is necessary to establish the domain-specific noun classes and the verb and other linguistic operators that co-occur with them in simple structures (Sager, 1986). For example certain sublanguages within the science field have such well-classified noun classes that they can be used as they stand. Word classification can also be defined by sorting the words of sublanguage texts into classes by referring to their appearance in similar environments. Sager described use of a clustering program which worked on sentences of a sublanguage that have been transformationally analyzed to produce domain-specific classifications of words.

Having made an analysis of the texts by hand, the results, syntactic operator-argument sentence trees, were put into the computer to produce the domain-specific co-occurrence patterns. The nouns of the sublanguage were always placed on the bottom nodes of the tree; the verbs were operators on the nouns. When the tree structures were in the computer, a list of the main noun classes of the sublanguage was typed in. The program replaced class names with class-member occurrences in the tree structures. Tables of co-occurrence patterns were produced by the computer providing material for the most elementary portion of the sublanguage grammar (Sager, 1986). These co-occurrence patterns of word classes were used as the main method of defining a sublanguage, usually being members of larger sentence structures that were common amongst the sublanguage texts and forming a second level of sublanguage description.

4.5 Automatic Machine Translation and Sublanguage

The discussion on sublanguage continues in this section with the investigation of the use of sublanguages in the field of Machine Translation. As noted in Chapter 1, difficulties were encountered by the developers of Machine Translation systems in their efforts to produce efficient working systems. One of the more successful working systems was the TAUM-METEO system developed by researchers at the University of Montreal for the Canadian Government; the function of the system being to translate public weather broadcasts from English to French (Chandioux, 1976). The reason for the success of this system seems to stem from the fact that it was constructed to only translate the language of weather reports, i.e., a sublanguage.

Why use a sublanguage for Machine Translation? A valid point that Lehrberger made was that since grammarians have not yet succeeded in writing a definitive formal grammar for natural language could it be assumed that it could be done for Machine Translation systems (Lehrberger, 1982)? One argument suggested by Lehrberger is that translating from one language to another does not need a full grammar, only context sensitive transfer rules to gain the correct lexical item in the target language and rules for restructuring these items. However, the need for context sensitivity and restructuring imply that structures of lexical items of source and target languages have to be recognized, which is no easy task. A system as small as METEO had still to perform much grammatical processing of both source and target languages at each stage of translation, i.e., parsing, transfer and generation. Using a sublanguage grammar, however, with its built-in restrictions, would not produce as many problems as those that would occur while trying to translate a whole language.

As Kittredge (1987) stated, analysis of the source text is very important part of Machine Translation and if a sublanguage was used the analysis of its grammar would be more efficient. This is due to the reduction of parsing time, itself a result of the relatively small size of the grammar. There would also be a decrease in structural and lexical ambiguity because of the fact that many words and phrases appearing in the standard language would not be permitted in the sublanguage.

The benefits of using sublanguage were also found during transfer and generation. Kittredge (1987) stressed that there was evidence to prove that there existed a strong similarity in structure of text and sentences of scientific and technical works of different languages over and above any similarities in the full languages. As he observed, if taking this into consideration whilst developing a Machine Translation system, it would be necessary to have the transfer grammars on the level of sentence and text. Thus, the output produced from transfer could be computed with regard to a particular sublanguage and not the whole language. Taking the example of TAUM-METEO, it was found that the format of written text in English and French weather broadcasts was totally similar. Deletion patterns in sentences of both languages' broadcasts were also similar with semantic and syntactic categories almost in one-to-one correspondence. Results from the TAUM-AVIATION project showed that in the language of aircraft maintenance manuals there were close links between French and English. The stylistic patterns were so similar that translation could be done on the level of phrase structure (Kittredge, 1982). These findings prompted further investigation into the similarities of parallel sublanguages in English and French. The results published showed (Kittredge, 1982) that, generally, parallel sublanguages of English and French were much more similar structurally than were dissimilar sublanguages of the same language, especially if the domain of reference of the parallel sublanguages was technical.

The combination of less polysemy due to semantic restrictions, use of text norms, limitations of vocabulary and restriction in syntax enable Machine Translation to be practicable for sublanguages. Practicability for sublanguages is enhanced further with the indication that parallel sublanguages have a great similarity structurally.

4.6 Deterministic Parsing and Sublanguage

As the main interest of the research is the use of deterministic parsers on sublanguage for Machine Translation, the sections discusses the viability of this method of processing sublanguage. It is accepted that other types of parser could be used and are used in the processing of sublanguage for Machine Translation, but the object of the research, as stated, is the testing the possibility of applying the deterministic parser to this area. The discussion will concentrate on what are considered the best properties of sublanguage that make it suitable to be processed by a deterministic parser in Machine Translation.

Firstly, a recap on Deterministic Parsing and Machine Translation. In the discussion on deterministic parsers in the previous chapter, the rules for deterministic parsing have been set out. These rules demand that syntactic processing should not involve backtracking or pseudo-parallelism and that there should be a limited lookahead into the input sentence buffer: in essence, there can only be a single parse of a sentence, nothing within the sentence can be reparsed.

In Chapter 1, it was suggested that the potential advantages of using a deterministic parser in a Machine Translation system would be even better if a sublanguage was the text to be processed. All sublanguage restrictions would also enhance the tentative proposal regarding the results from computational complexity theory that deterministic parsers processed in linear time and space which make them more efficient than non-deterministic parsers. This is discussed in more detail below.

By adopting the stance taken by Fitzpatrick, Bachenko and Hindle (1986), discussed above, that a sublanguage can be considered an independent system, accepting the sublanguage grammar is correct, then it can be suggested that sublanguage, especially a technical, telegraphic sublanguage, is suitable for deterministic parsing. The reasons are linked to the characteristics of sublanguages which indicate differences from standard language. These characteristics are the restrictions, previously discussed, which are placed on the syntax, lexicon and semantics of sublanguage. Below these sublanguage restrictions are examined in relation to deterministic parsing.

Technical sublanguages, such as aircraft maintenance manuals, must not be ambiguous because of the precise nature of what they have to describe. As stated, the restrictions found in sublanguages lead to reductions in syntactic, semantic and lexical ambiguity. Reductions in ambiguity would favour deterministic parsing, as this does not allow the reparsing of incorrect parses. For example, if a sentence contains a word restricted lexically or semantically which leads to a reduction in syntactic ambiguity then this would ease processing for a deterministic parser as there would be only one correct parse of the sentence which the parser would produce. An actual example sentence, where this applies, is from the Navy message sublanguage (Fitzpatrick, Bachenko and Hindle, 1986) is 'Failed klystron. Unable to radiate continuous wave 1'. 'Failed' can be an intransitive left modifier of 'klystron' or a transitive verb and 'klystron' is its object. When the transitivity constraint is applied, so that 'failed' is no longer ambiguous, this means that the sentence would no longer be structurally ambiguous. It's obvious that 'failed' would be restricted to its intransitive meaning, which would mean, accepting the independent status of the sublanguage with transitivity constraint, that wherever 'failed' appeared in the sublanguage it would be in the intransitive sense.

As stated previously, other types of parsers can cope with processing sublanguage for Machine Translation. However, the restrictions found in sublanguage suggest that a deterministic parser would be able to cope with that type of language. The restrictions also mean that less ambiguity may occur within the sublanguage corpus, which could lead to processing being faster due to the tentative results concerning complexity theory and deterministic parsers. No problems are foreseen with the processing of the sublanguage being for Machine Translation, as the structure produced would also have been semantically processed.

CHAPTER 5

MPARSER - A MARCUS TYPE PARSER

5.1 Introduction

In this chapter, MParser is examined in detail. The aim of building MParser, programmed in Prolog, was to produce a prototype Deterministic Parser of the Marcus type that would be tested on a set of sentences that cover a wide-range of grammatical and linguistic aspects, taken from Marcus (1980). From this prototype another parser would be developed that could be used on a sublanguage. MParser is built according to the basic design of the first Deterministic Parser, PARSIFAL (Marcus, 1980) and the grammar of L.PARSIFAL (Berwick, 1985), and follows the rules set out for maintaining determinism within a system:

a) All syntactic substructures created by the machine are permanent, reflecting that the parser should be partially data driven and prevent backtracking

b) All syntactic substructures created by the machine for a given input must be output as part of a syntactic structure, showing that the pad reflect expectations and stop pseudo-parallelism.

c) The internal state of the mechanism must be constrained in such a way that no temporary syntactic structures are encoded within the internal state of the machine, meaning that there must be a limited lookahead facility.

(Marcus, 1980, p12-13)

MParser also incorporates structural modifications suggested by Berwick (1985) and Milne (1982, 1983, 1986), which are considered to enhance the functioning of the parser.

In section 5.2 the description begins by looking at one of the important components of the MParser parsing system, the grammar, followed by a description of the grammatical theory on which the grammar is based and the specific grammar rules used by the system. The discussion continues in section 5.3 with the workings of the parser being described in detail, beginning with an examination of MParser's two major data structures, the Pushdown Stack and Lookahead Buffer. Finally, in section 5.4, the grammatical and linguistic range of MParser is discussed, with examples of the type of sentences it can process being given.

5.2 The Grammar

The grammar used by MParser is similar to the target generative grammar acquired by Berwick's (1985) system, which incorporates Chomsky's X-bar theory of phrase structure. The grammar has two parts: base phrase structure grammar rules and transformational-type grammar rules. The base phrase structure system contains the following: noun-phrases, verb-phrases, auxiliaries, main sentences, prepositional phrases, relative clauses and embedded sentences. There are two types of transformational rule; one type deals with simple local transformations, such as subject-auxiliary inversion and the other deals with wh-movement. MParser's grammar is an adapted form of the grammar used by L.PARSIFAL (Berwick, 1985), which itself was an adapted form of the grammar used by PARSIFAL (Marcus, 1980). The grammar rules are actually a set of production rules, with each production rule being made up of a series of patterns and a single action. Below X-bar theory is explained in some detail, followed by an examination of the specific grammar rules that MParser utilises.

5.2.1 X-bar theory

Berwick (1985) put forward pertinent reasons for using the X-bar theory of phrase structure within L.PARSIFAL, his system being designed to mimic a child's acquisition of syntactic knowledge. The main reason being that it made learning easier for his system. The reason that it was chosen for MParser is that by using X-bar theory it is possible to make redundant the packets of rules as used in Marcus' system, the names of which were placed on the stack to indicate the state of the parse. By making the packeting system redundant, the parser is rid of the process of activating and deactivating the packets; thus enhancing the functioning of the parser.

X-bar theory of phrase structure is designed to be restrictive (Jackendoff, 1977). In X-bar theory, the structure of the phrase can have, at most, three branches: specifier, head and complement. Specifiers correspond to determiners, quantifiers and/or adjectives. Complements can be embedded sentences, or phrases such as prepositional phrases or relative clauses. Of the three branches, the head, since it is the core of the phrase, is the most important: it is a noun in a noun-phrase, a verb in a verb-phrase, a preposition in a prepositional phrase, etc. An important restriction in X-bar theory is that phrases assume the features of their heads by percolation. Specifiers and complements, which are optional within X-bar theory, function as modifiers and arguments. For example, the determiner and/or adjective can be the specifier of a noun-phrase, which also can have a sentential complement.

By being restrictive, X-bar theory allows the base phrase structure rules of the grammar used by MParser to function by means of what Berwick (1985) called a *small number of template filling decisions*. As stated above, by using X-bar theory Marcus' packeting system becomes redundant. In its place it is possible to use a *single template* *packet*. In MParser's system the template is placed on the Pushdown Stack, as it is in Berwick's system. The template contains three descriptors, which correspond to the three branches of a phrase. The template placed on MParser's Pushdown Stack is represented by the following:

```
    [spec_,head,comp]
```

The three descriptors themselves also represent types of rules used by MParser's grammar - specifiers, heads and complements, which have been described above.

The symbol '_' attached in the example above to *spec*, signifies what type of rule is being processed. Once a rule has been processed, '_' either will remain attached to the same descriptor, if the next rule to be processed is of the same rule type or it will attach itself to the next descriptor, if the next rule is of a different type. If, for example, a simple noun-phrase is to be parsed, e.g., *the girl*, the template above would signify that '_'attaches itself to *head* since there are no more specifiers to be processed. The template on the Pushdown Stack would be as follows:

```
    [spec,head_,comp]
```

The symbol '_' can move only in a left-to-right direction and cannot jump descriptors. The use of this notation is similar to using the dotted rule notation, which incorporates context-free rewrite rules with marker dots placed in the right-hand side of the expansion representing how much of a phrase has been parsed. The difference being that MParser's grammar does not incorporate context-free type rules, but rather groups of features which correspond to specifier, head or complement type rules.

In X-bar theory groups of features represent the following lexical categories: N(Noun), A(Adjective and Adverb), DET(Determiners and Quantifiers), V(Verb), P(Preposition), INFL(Inflectional Elements) and COMP(Complement). Lexical categories and corresponding feature representations are shown below:

```
N -> +N-V+A-P   V -> -N+V-A+P

ADJ -> +N-V+A+P      INFL -> -N+V+A+P

DET -> +N-V-A-P      COMP -> -N-V+A+P
```

The *N* and *V* of the feature value representations correspond to noun and verb respectively with *A* and *P* corresponding to argument and predicate: a typical argument is a noun-phrase and typical predicate is a verb-phrase. With regard to S(Sentence), this has INFL as its head. INFL is governor of the Subject NP, also functioning as a predicate. S can also be an argument; S has COMP as its head. Using the feature value representations eases conforming to X-bar theory, with any illegal structure being ruled out early in the processing.

5.2.2 The Grammar Rules

MParser's grammar is invoked by a call to *grammar_rule/5* by the parsing rules. The grammar is, in fact, a Prolog procedure called *grammar_rule/5*; each rule consists of a grammatical category and representation of the Pushdown Stack and Lookahead Buffer. When there is a call to *grammar_rule/5*, the present state of the stack and buffer tries to match with one of the rules in the database. If the matching succeeds, *grammar_rule/5* calls one of the following rules to perform an action:

attach - The function of this rule is to attach the contents of the first buffer to the Current Active Node.

switch - This rule is called when the contents of the stack and buffer signify that subject-auxiliary inversion should occur, i.e, the contents of the first and second buffer cell are switched.

insert - This rule is called when a lexical item or trace needs to be added to the parse tree, i.e, an empty np-node, is inserted into the first buffer cell.

Below two differing descriptions of that part of the grammar rule which includes representations of the stack and buffer are given.

```
[Gram-category,[[xmax,Structure,Features,Template]|Reststack]
    [[[First_cell],[Second_cell]]|Outbuffer])
```

Figure 5.0 Individual Components of a Grammar Rule

In Figure 5.0 the description is that of the individual components that make up the part of the grammar rule which matches with the state of the Pushdown Stack and Lookahead Buffer MParser when it is ready to fire *grammar-rule.*

Gram-category refers to the type of grammatical category that is processed if matching succeeds.

xmax, Structure, Features, Template are all in the Current Active Node of the Pushdown Stack, with *xmax* signifying that the constituent being processed is a *maximal projection.* A maximal projection is the highest level of projection and represents the phrasal level. *Structure* represents a structure that has been processed and which may have another constituent added to it. *Features* signifies the type of grammatical category being processed. *Template* holds the descriptors, which signify to which group the rule belongs. *Reststack* represents the remainder of the Pushdown Stack.

First_cell and *Second_cell* represent the Lookahead Buffer. *Outbuffer* is the remainder of the input string that has not yet entered the Lookahead Buffer.

In Figure 5.1, below, an actual grammar rule is depicted i.e., the representations of the stack and buffer which match with the state of the Pushdown Stack and Lookahead Buffer. This example rule processes *nouns*. In the Current Active Node there is already a determiner which has been previously processed. If the representation of the stack and buffer in the rule match with the state of Pushdown Stack and Lookahead Buffer, the noun in the first cell of the buffer is added to the determiner in the Current Active Node.

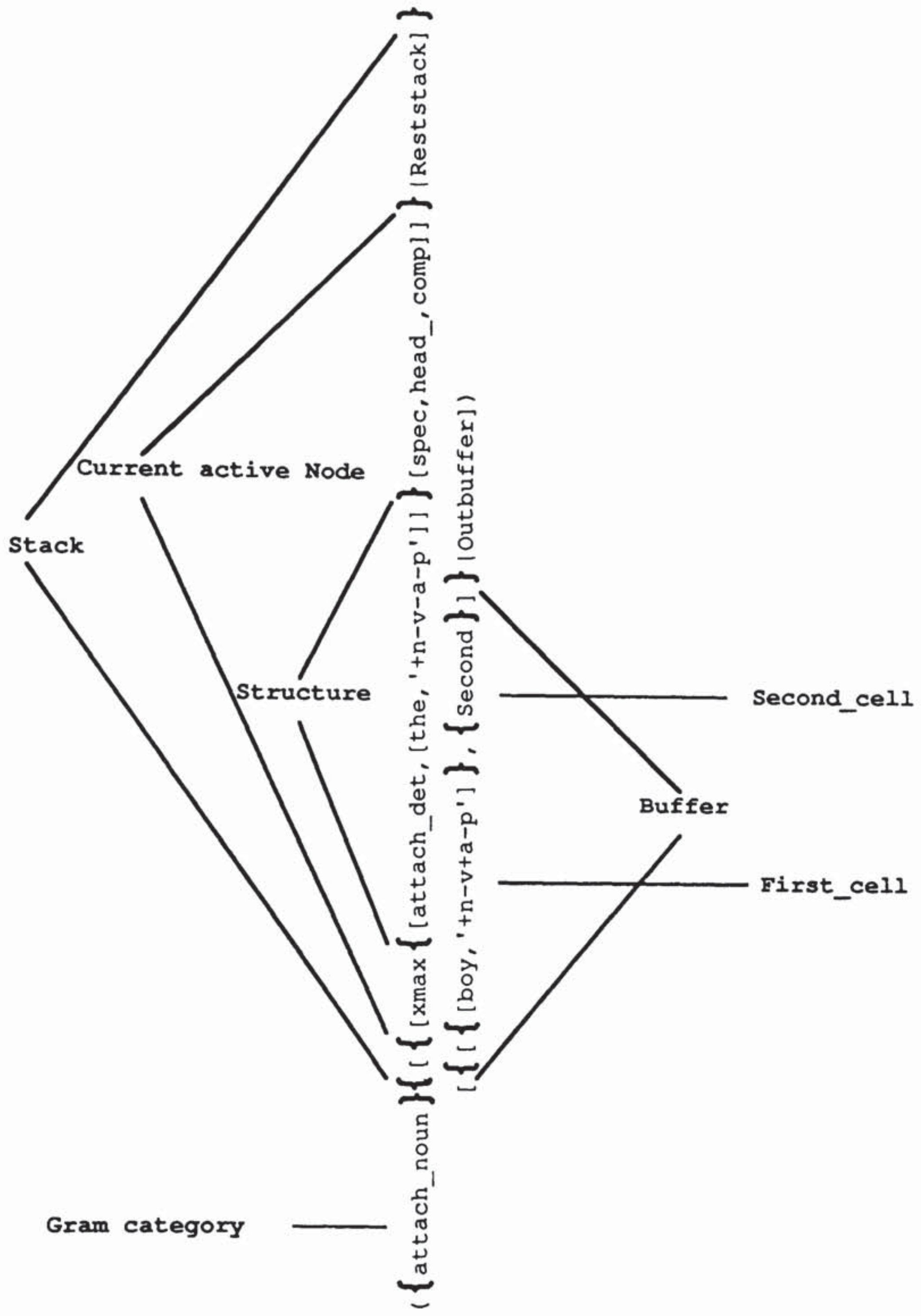

({attach_noun}{[{[xmax{[attach_det,[the,'+n-v-a-p']]}[spec,head_,comp]]}|Reststack]}
[{[{[boy,'+n-v+a-p']},{Second}]]}|Outbuffer])
Stack
Current active Node
Structure
Second_cell
Buffer
First_cell
Gram category

Figure 5.1 Actual Grammar Rule

The rules of MParser's grammar are grouped by type. There are three groups, which are classified by the descriptor in the template, i.e., specifier, head or complement. Each group varies in size, with the complement group being the largest. If considering grammatical categories, there are the thirty eight rules in total, but there are a number of rules with the same lexical category because of the different conditions that apply to attaching the contents of the first buffer cell to the Current Active Node. Below examples from each group are given; the whole grammar being described in Appendix A.

5.2.2.1 Examples of Specifier Group

attach_det - This is one of the rules for processing determiners.

```
grammar_rule(attach_det,[[xmax,'+n-v+a-p',[spec_,head,comp]
]|RSt],
[[FW,'+n-v-a-p'],[SW,'+n-v+a-p']]|OutB],NewStack,NewBuffer):-
attach([[xmax,'+n-v+a-p',[spec_,head,comp]]|RSt],
[[[FW,'+n-v-a-p'],[SW,'+n-v+a-p']]|OutB],NewStack,NewBuffer).
```

attach_adj - This is one of the rules for processing adjectives.

```
grammar_rule(attach_adj,[[xmax,S,'+n-v+a-p',[spec_,head,com
p]]|R],
[[[FW,'+n-v+a+p'],[SW,'+n-v+a-p']]|OutB],NewStack,NewBuffer
):-
attach([[xmax,S,'+n-v+a-p',[spec_,head,comp]]|R],
[[[FW,'+n-v+a+p'],[SW,'+n-v+a-p']]|OutB],NewStack,NewBuffer).
```

5.2.2.2 Examples of Head Group

perfective - This is one of the rules for processing perfectives.

```
grammar_rule(perfective,[[xmax,'-n+v',[spec,head_,comp]]|RS
t],
[[[FW,'-n+v'],[SW,'-n+v','+',en]]|OutB],NewStack,NewBuffer):-
attach([[xmax,'-n+v',[spec,head_,comp]]|RSt],
[[[FW,'-n+v'],[SW,'-n+v','+',en]]|OutB],NewStack,NewBuffer).
```

attach_rpron - This is the rule for processing relative pronouns.

```
grammar_rule(attach_rpron,[[xmax,'-n-v+a+p',[spec,head_,com
p]],
[xmax,S,'+n-v+a-p',Temp]]|RSt],
[[[FW,'-n-v+a+p'],Second]|OutB],NewStack,NewBuffer):-
attach([[xmax,'-n-v+a+p',[spec,head_,comp]]
[xmax,S,'+n-v+a-p',Temp]]|RSt],
[[[FW,'-n-v+a+p'],Second]|OutB],Newstack,NewBuffer).
```

5.2.2.3 Examples of Complement Group

attach-vp - This is one of the rules for processing verb-phrases.

```
grammar_rule(attach_vp,[[xmax,S,'-n+v+a+p',[spec,head,comp_
]]|RS],
[[[xmax,S1'-n+v-a+p',Ty],Second]|OutB],NewStack,NewBuffer):-
attach([[xmax,S,'-n+v+a+p',[spec,head,comp_]]|RS],
[[[xmax,S1'-n+v-a+p',Ty],Second]|OutB],NewStack,NewBuffer).
```

attach_pp - This is the rule for processing prepositional phrases.

```
grammar_rule(attach_pp,[[xmax,S,'+n-v+a-p',[spec,head,comp_
]],
[xmax,S1,'-n-v',Temp]]|Rst],
[[[xmax,S1,'n-v'],Second]|OutB],NewStack,NewBuffer):-
attach([[xmax,S,'+n-v+a-p',[spec,head,comp_]],
[xmax,S1,'-n-v',Temp]]|RSt],
[[[xmax,S1,'-n-v'],Second]|OutB],NewStack,NewBuffer).
```

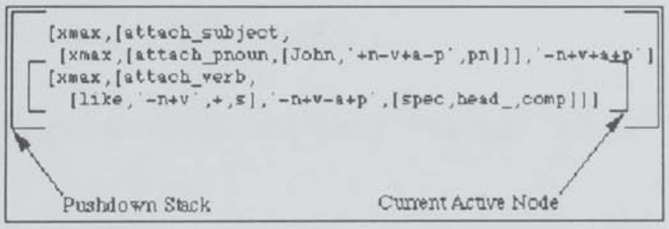
5.3 MParser's Data Structures

Two of the major components of MParser are the Pushdown Stack and the Lookahead Buffer, each of which is described in detail. The section ends with a description of MParser in action, which is based on the manipulation of Stack and Buffer.

5.3.1 The Pushdown Stack

The Pushdown Stack or Active Node Stack, by which name it is also referred in the literature, plays a similar role in most Deterministic Parsing systems - a store of the incomplete constituents. The Pushdown Stack used by MParser is no different. The bottom node of the Pushdown Stack, referred to as the Current Active Node, is where constituents are added to the node being built. Figure 5.2 shows a snapshot of the stack part way through a parse.

If there is a need for further processing after a new constituent is added to the Current Active Node, the still incomplete constituent in the Current Active Node is dropped into the first buffer cell of the Lookahead buffer. Otherwise, the incomplete constituent is pushed further onto the stack to allow for processing of the next node, but this incomplete constituent itself will be processed later in the parse when it will be added to another incomplete constituent to form a larger constituent. For example, in the situation depicted below in the snapshot of the stack, a *verb* is the Current Active Node, and since no other grammatical constituent can be added to it at this stage of the parse, it is pushed further into the stack to allow for processing of the next node. If the next node, after processing, is a *noun-phrase* this can be added to the verb, which will have become the Current Active Node again and a verb-phrase will have been formed.



```
[xmax,[attach_subject,
 [xmax,[attach_pnoun,[John,'+n-v+a-p',pn]]],'-n+v+a±p']
[xmax,[attach_verb,
  [like,'-n+v',+,s],'-n+v-a+p',[spec,head_,comp]]]
```

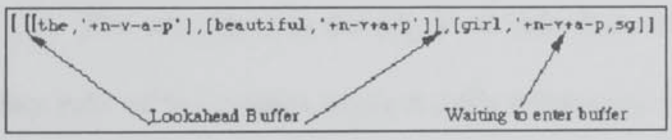
5.2 Snapshot of Pushdown Stack

5.3.2 The Lookahead Buffer

The Lookahead buffer used by MParser has only two cells of lookahead. This differs from the Lookahead used by PARSIFAL (Marcus, 1980) and L.PARSIFAL (Berwick, 1985), which both used three cell lookahead. There is a similarity to the Lookahead Buffer used by ROBIE (Milne, 1982, 1983, 1986) which also used a lookahead of two, but this data structure incorporated two single static buffers, whereas

MParser's buffer is one buffer containing two cells. The choice of employing a buffer with two cell lookahead is enough to prevent back-tracking and process phrases and sentences. However there are certain ambiguities that cannot be resolved no matter how many cells of lookahead are used. These ambiguities are mentioned below.

The Lookahead Buffer is a queue with the individual words entering from the left. MParser processes the constituent in the first buffer cell, having the ability to look at the second cell and "knowing" with what sort of clause it is dealing. Figure 5.3 shows a snapshot of the Lookahead buffer and the remainder of the input string. The remainder of the input string enters the buffer in the second cell when the contents of the first buffer cell are pushed onto the stack and the contents of the second cell enter the first buffer cell.



```
[ [[the,'+n-v-a-p'],[beautiful,'+n-v+a+p']],[girl,'+n-v+a-p,sg]]
```

Figure 5.3 Snapshot of Lookahead Buffer and Input String

After processing, the constituent in the first buffer cell is pushed onto the stack, where the parser constructs an enlarged grammatical constituent, as described above.

Prior to entering the Lookahead Buffer, the words that form the input string will have already gone through dictionary look-up and a morphological analyzer. As a result, each word in the input string will have attached its grammatical category and if the word is a noun, it will also have attached its grammatical number, i.e., either singular or plural.

5.3.3 Interaction of Stack and Buffer - MParser in Action

In this section the method by which MParser manipulates the stack and buffer to produce the parse of a sentence is described. This description involves referring to the series of Prolog rules that constitute MParser. The purpose of these rules is to get the stack and buffer into a state ready for the firing of a grammar rule. As stated above, prior to entering the Lookahead Buffer individual words of the input string pass through dictionary look-up and morphological analysis; these procedures are discussed in a subsequent section in connection with the parsing of certain types of sentences. Below each rule used in the parsing process is examined. The Prolog code for MParser appears also in Appendix A.

5.3.3.1 *input*

The rule *input* is the top-level rule that calls *enter, prep* and *parse*; *parse* is the key rule of the system, since it calls rules to process an input string. However, before any processing can take place, the input string has to be entered into the buffer and the stack prepared for the constituent in the first buffer cell; the rules *enter* and *prep* perform this pre-parsing preparation.

5.3.3.2 ***enter***

The rule *enter* places the first two words of the input sentence into the two cell Lookahead Buffer; the rest of the input string remains outside the buffer until the second buffer cell is empty. (At that point, the third word of the input string enters the second buffer cell, the contents of the cell now being in the first buffer cell, the contents of which have been pushed onto the Pushdown Stack. This procedure occurs after a call to *grammar_rule*.)

5.3.3.3 *prep*

This rule calls several rules, *create_max*, *perc_features*, *add_template*, *add_features* and *pre_test* to prepare the stack for the first constituent from the buffer. These rules are called also by *act_create_node* and *drop*, which are discussed later.

5.3.3.3.1 *create_max*

The rule *create_max* places a marker in the Current Active Node of the Pushdown Stack to signify that a maximal projection, i.e., head of a phrase, will be held there. This rule fires even if the first constituent of a phrase is not its head, e.g., the constituent may be a specifier.

5.3.3.3.2 *perc_features*

The purpose of the rule *perc_features* is to percolate the features of the constituent in the first cell of the buffer to the Current Active Node. If the constituent cannot be a maximal projection, i.e., it is a specifier, the features of the governing head are placed on the stack.

5.3.3.3.3 *add_template*

The rule adds a template of descriptors, *spec, head,* and *comp* to the Pushdown Stack that describes the function of the constituent to be placed in the Current Active Node; the symbol '_' is attached to the descriptor to signify which function the constituent holds, as described above. On occasions it is necessary to consider the contents of the second cell and/or the stack, as well as the first buffer cell.

5.3.3.3.4 *add_features*

The rule *add_features* adds features to or amends the features on the stack after percolation has occurred. Occasionally, the need for this arises when the features of a head of phrase are incomplete, e.g., the features *A* and *P* are missing and have to be added. On rarer occasions, an anomaly occurs in the theory and the features percolated have to be amended, e.g., '+' changes to '_' or vice versa.

5.3.3.3.5 *pre_test*

The purpose of *pre_test* is to check the contents of the buffer for the sequence of auxiliary verb followed by a determiner, adjective or noun. If there is no such sequence *pre_test* will return contents of stack and buffer as they were in the previous rule. However, if there is such a sequence *pre_test* calls *create_max_s, perc_feat_s, add_template_s, attach_s, attach_s1, attach_s2, attach_s3* and *drop_s*, which are rules to deal with building a maximal projection from the contents of the second buffer cell. The rules correspond to the rules with similar names which have already been described. The four *attach* rules attach the contents of the second buffer to the stack in order to build a maximal node. Four rules are needed to deal with determiners and adjectives. *drop_s* drops the contents of the stack back into the second buffer cell to allow for the processing of subject-auxiliary inversion.

5.3.3.4 *parse*

As stated above, *parse* is the key rule of the system. It calls *grammar_rule* which invokes the grammar. If *grammar_rule* fires there is a call to *process* which in turn calls a series of rules that makes adjustments to the stack and buffer after the firing of *grammar_rule* and also prepare them for the firing of the next grammar rule. However, if *grammar_rule* does not fire, *parse* then calls *drop*, another rule to manipulate the stack and buffer. But, if *drop* fails this is a signal that parsing has ended and a final parse tree is dropped into the buffer.

5.3.3.5 *process*

The rule *process* calls the rules *annotate_node, amend_stack, amend_template, act_create_node, add_feat_spec* and *drop*.

5.3.3.5.1 *annotate_node*

This rule attaches the name of the *grammar_rule* to the Current Active Node.

5.3.3.5.2 *amend_stack*

The purpose of *amend_stack* is to add a description of the type of verb or *wh*-word to the Current Active Node, if the rules to attach verb or *wh*-words have just fired.

5.3.3.5.3 *amend_template*

With *amend_template*, the pointer to the descriptor in the template can be moved to the right, remain the same or be moved from comp to spec, depending on the contents of the stack and/or buffer. The rule calls *member* or *equal* and *change_template* or *change_template1* or *change_template2* to aid the decision of whether the pointer should move or remain in the same position.

5.3.3.5.4 *act_create_node*

This rule, on examination of stack and buffer, either keeps the Current Active Node or builds another node at the bottom of the stack, which will become the Current

Active Node and pushes the contents of the present Current Active Node further into the stack. In order to create the new node, *act_create_node* calls upon *create_max, perc_features, add_template* and *add_features*, which we have already described.

5.3.3.5.5 *drop*

The rule *drop* fires if *grammar_rule* fails. The main purpose of *drop is to* remove the contents of the Current Active Node and place them back in the first buffer cell. Depending on the contents of the stack and buffer after this action has taken place, it may be necessary to build a new node on the bottom of the stack by calling the same rules as described above.

5.4 The Grammatical and Linguistic Range of MParser

In this section the grammatical and linguistic range of MParser is discussed. The discussion includes the examination of two types of ambiguity, structural and part of speech ambiguity, and how MParser copes with the problems that arise. The problem of part of speech ambiguity was the main theme of Milne (1983), whereas Marcus (1980) did not consider the problem at all. However, both Milne and Marcus looked at aspects of structural ambiguity. The discussion begins by looking at the dictionary and dictionary look-up, which includes a procedure for morphological analysis. The subsequent sub-section deals with the type of sentences MParser parses, which are not ambiguous and the final sub-section ends with an examination of ambiguous sentences.

5.4.1 The Dictionary

Any serious parsing system needs a good dictionary for it to work effectively. The dictionary used by MParser, a series of Prolog rules, see below, has approximately one hundred words covering the major grammatical constituents such as verbs, nouns, pronouns, proper nouns, determiners, adjectives, prepositions, relative pronouns and *wh*-pronouns. The dictionary may seem relatively small, but the vocabulary allows great scope for forming many grammatical constructions and handling ambiguity. The dictionary contains, at most, double entries for a word, for example:

```
1.   word(walk,Full,_,_,[Full,'+n-v+a-p']).

2.   word(walk,Full,PN,Tense,[Full,'-n+v',PN,Tense]).
```

Example 1 is the dictionary entry for the noun *walk* and example 2 is the entry for the verb *walk*.

5.4.1.1 Dictionary Look-up

The dictionary look-up procedure *check* is called immediately after the input string has been read in from the keyboard. *check* processes each word of the input string separately. The first procedure *check* calls is the procedure *end* which performs morphological analysis; this procedure is discussed below. After morphological analysis, which returns the root form of the word and its ending, there is a call to the dictionary procedure *word*. The procedure *word* matches the root form of a word and returns this word with ending attached and grammatical category. If the word is a noun, the grammatical number, i.e., singular or plural, also is returned.

If the word being processed by dictionary look-up has two entries in the dictionary, *check* returns both entries. Both entries remain in the processed input string until entering the first cell of MParser's Lookahead Buffer. MParser makes the distinction between entries depending on the state of the Pushdown Stack and the second cell of the Lookahead Buffer. For example, if, in the double entry of the example given above, *walk* is in the first cell of the Lookahead Buffer, MParser would examine the state of the Pushdown Stack and if there is a *determiner* in the Current Active Node, *walk* would be attached as a noun. As the example shows, a double entry of noun/verb following a determiner is disambiguated as noun. This is due to the rules of the grammar for noun phrases, which would also be true of a double entry of noun/verb following a determiner and adjective. If an adjective/verb double entry followed a determiner, the entry would be disambiguated as adjective as a result of the grammar rules for noun phrases. Similarly, if a noun/verb followed an adjective/verb which followed a determiner, the double entries would be disambiguated as noun and adjective respectively, once more as a result of the grammar rules for noun phrases. The restrictions imposed by the X-bar theory of phrase structure state clearly the components of the phrases of the grammar, thus the grammar is an aid to the method used for disambiguation of the particular examples discussed. Milne (1983) uses a similarly method for dealing with polysemy.

5.4.1.2 Morphological Analysis

As stated above, the procedure *end*, which performs morphological analysis, is the first procedure called by *check* and is based on the algorithm for dealing with morphology given in Winograd (1972). *end* examines words for typical endings, such as *s, es, ed, en, er, ly, ing,* or *est.* If *end* finds one of these endings, it removes the ending and then checks the last or last two letters of the remaining letters, since they might need to be replaced. For example, if the word being morphologically analyzed is *pennies*, the analyzer removes the *es* ending and on checking the last letter finds an *i* which is replaced with a *y*. The word *penny* is subsequently passed to the remaining procedures of dictionary look-up.

5.4.2 The Processing of Unambiguous Sentences

The grammatical and linguistic range of MParser is represented by the type of sentence it can process. In this section the unambiguous sentences that MParser can parse are examined. An unambiguous sentence is a sentence where there is no structural ambiguity or where there is no part of speech ambiguity. MParser can deal with a varied set of sentences covering the following grammatical constructions and sentence types:

1. Auxiliary Verbs - *to*-infinitive, perfective, progressive, modal, *do*, passive_*be*
2. Simple Declarative Sentences. 3.Prepositional Phrase.
4. Verb Complements. 5.*That* Complements.
6. Deleted *That* Complements. 7.Subject_Auxiliary Inversion.
8. Passive Constructions. 9.Missing Subject Sentences
10. Wh Questions 11.*There* Insertions.
12. Imperatives. 13.Relative Pronouns.
14. Reduced Relatives. 15.Possessive Determiners.

Of course, some of the above can produce ambiguity within a sentence, but not always. Below a few examples of unambiguous sentences employing the above constructs and the resulting parses are given. Further example parses appear in Appendix C.

Will the beautiful blushing bride kiss John.

Who broke the jar?

The large boy loves the small girl.

Is there a meeting scheduled for Wednesday?

```
?- readin(S).

|: will the beautiful blushing bride kiss John.

S= [[[xmax,[attach_subject,
     [xmax,[[the,'+n-v-a-p'],[beautiful,'+n-v+a+p'],
          [blushing,'+n-v+a+p'],[bride,'+n-v+a-p',sg]],
          '+n-v+a-p']],
          [attach_vp, [xmax,[[aux_sai,[will,'-n+v+a+p']],
     [attach_vp,
     [xmax,[[attach_verb,[kiss,'-n+v',-,tense]],
                    [attach_object,[xmax,
                    [attach_propnoun,['John','+n-v+a-p',pn]],
                    '+n-v+a-p']]],
               '-n+v-a+p',subj]]],
          '-n+v+a+p']]],
     '-n+v+a+p'],[]],[]]

?- readin(S).|: who broke the jar.

S=[[[xmax,[[attach_wh_comp,[who,'-n-v+a+p']],
          [attach_sent,[xmax,[[attach_embedded_subject,
          [np_empty,'+n-v+a-p'],
          [attach_vp,[xmax,
               [[attach_verb,[broke,'-n+v',-,tense]],
                    [attach_object,
                    [xmax,[[attach_det,[the,'+n-v-a-p']],
                         [attach_noun,[jar,'+n-v+a-p',sg]]],
                    '+n-v+a-p']],
               '-n+v-a+p',obj]]],
               '-n+v+a+p']]],
      '-n-v+a+p',wh],[]],[]]

| ?- readin(S).|: the large boy loves the small girl.

S=[[[xmax,[attach_subject,
          [xmax,[[attach_det,[the,'+n-v-a-p']],
          [attach_adj,[large,'+n-v+a+p']],
          [attach_noun,[boy,'+n-v+a-p',sg]]],'+n-v+a-p']],[
               [attach_vp,
          [xmax,[[attach_verb,[love,'-n+v',+,s]],
          [attach_object,
               [xmax,[[attach_det,[the,'+n-v-a-p']],
               [attach_adj,[small,'+n-v+a+p']]],
               [attach_noun,[girl,'+n-v+a-p',sg]]],
               '+n-v+a-p']],
          '-n+v-a+p',subj]]],
      '-n+v+a+p'],[]],[]]

|?- readin(S).|: is there a meeting scheduled for Wednesday.

S = [[[xmax,[attach_subject,[xmax,[there,'+n-v+a-p'],
          '+n-v+a-p']],
             [attach_vp,[xmax,[[copula,[is,'-n+v']],
          [attach_object,[xmax,[[attach_det,[a,'+n-v-a-p']],
               [attach_noun,[meeting,'+n-v+a-p',sg]]],
               [attach_zrpron_sent,[xmax,
                 [attach_reduce_rel,
         [schedule,'-n+v-a+p',+,ed]],
             [attach_pp,[xmax,[[attach_prep,[for,'-n-v']],
             [attach_pp_object,[xmax,
             [attach_noun,['Wednesday','+n-v+a-p',sg]],
                 '+n-v+a-p']],
             '-n-v']]],
         '-n+v+a+p']],
         '+n-v+a-p']],
    -n+v']]],
'-n+v+a+p'],[]],[]]
```

5.4.3 The Processing of Ambiguous Sentences

Above, it was stated that MParser could handle ambiguity, namely aspects of structural and part of speech ambiguity. Structural ambiguity occurs when a sequence of grammatical constituents can form differing grammatical constructs, e.g.,

Have the students take the exam.

Have the students taken the exam?

The two sentences given above are structurally ambiguous because it is impossible to decide whether Have is a main verb of an imperative sentence or an auxiliary of a yes/no question until the verbs take or taken are parsed.

Part of speech ambiguity occurs when a word in a sentence can be more than one type of grammatical constituent, e.g.,

1. The block is on the table.

2. I know that boy likes football.

3. I know that boys like football.

In example 1, it is obvious to the reader that "block" is a noun, but the reader also knows that "block" can be a verb. In example 2, the reader knows "that" is a determiner and in example 3, "that" is a complementiser. How does the reader know the difference? In example 1 "block" appears after a determiner, so it can only be a noun. In example 2 "that" appears before a singular noun and in example 3 "that" appears before a plural noun, in both cases the grammatical number of the noun determines how each "that" is processed.

MParser can handle the above ambiguities. As stated above, if a word occurs more than once in the dictionary, all entries of the word are passed, at dictionary look-up, to the parsing process. When this group of entries appears in the first cell of the Lookahead Buffer, only one of the entries is pushed onto the Current Active Node. The method MParser uses for making the choice is to look at the contents of the Pushdown Stack and/or the second buffer cell and compare with each of the multiple entries when a *grammar_rule* is about to fire. In example 1, above, MParser could discriminate between *noun* and *verb* at the crucial point because a *determiner* would be in the Current Active Node. In examples 2 and 3, MParser could discriminate between *complementiser* and *determiner* by examining the contents of the second buffer cell and finding either a singular or plural noun.

Milne (1983) followed a similar method for dealing with examples like example 1, above, and cites Winograd (1972) as the instigator of the method by using compound lexical entries and pattern matching to disambiguate between different grammatical categories. Other examples (Milne, 1983) that can be dealt with by looking at the type of grammatical constituents in the Current Active Node or second buffer cell are ambiguities surrounding the noun-phrase, e.g., singular head nouns, verb/adjective ambiguity and other pre-nominal ambiguities. This method also can deal with disambiguating between *to* as a preposition and *to* as the infinitive auxiliary by looking at the contents of the second buffer cell. However, the method in this case allows ungrammatical sentences. This problem can be solved by using verb sub-categorization on verbs preceding *to*, i.e., certain verbs can take infinitive complements others cannot. When verbs precede *to* which is followed by noun/verb ambiguity, the ambiguity can normally be disambiguated due to verb sub-categorization. For example,

I want to school the boys.

I went to school yesterday.

It may occur that a verb may not precede *to* followed by noun/verb ambiguity, in this instance it would be difficult to disambiguate between noun and verb. MParser can deal with the above ambiguities.

In examples 2 and 3, above, disambiguation takes place because examination of the second cell includes checking for number agreement. Milne (1983) gives other examples of part of speech ambiguity that can be solved by checking *person/number codes* and *verb agreement*. These examples include disambiguating between *for* as a preposition and *for* as a complementizer, noun/verb ambiguity in plural head nouns, dealing with *what* and *which* noun/modal ambiguity, dealing with *her*, dealing with *that* and dealing with *have*. MParser has not been programmed to deal with all of these aspects of ambiguity but it is considered that it would be an easy task as it involves checking for either verb agreement or person/number agreement.

MParser and all Marcus type parsers have difficulty in dealing with constructions that exhibit unbounded dependencies, e.g., constituent questions, as stated by Church, cited in (Briscoe, 1987). The reason for this is that it is impossible for a deterministic parser trying to solve local ambiguities by using lookahead to find the correct gap in unbounded dependencies; lookahead in a deterministic parser is restricted. In order to resolve the difficulty lookahead would have to be unrestricted. Whether this causes a problem for parsing sublanguage is discussed in Chapter 7.

Other ambiguities can occur when relative clauses and prepositional phrases are subject to rightward movement; an example trace is given in the first example below showing 'point of extraction'. Examples of rightward movement are:

The boy 'e' dropped in who lives down the lane.

A girl took the job who was attractive.

A girl took the job that was attractive. (Briscoe, 1987)

Deterministic parsers, although MParser does not deal with this construction, can cope with the ambiguities of rightward movement. Briscoe (1987) notes that rightward movement in the English language is treated:

> " as a case of canonical attachment of the postmodifier to the S node, where semantics remains a matter of more general inference rather than a fully determinate, grammatically defined binding."

This allows deterministic parsing of the rightwardly moved relative clause or prepositional phrase.

Below are examples of sentences with ambiguities and the resulting parses. These examples includes both types of ambiguity discussed and varying aspects of ambiguity that occur within part of speech ambiguity.

The tall skinny girl with red hair that I met at the party is coming.

Have the students taken the exam?

The block is on the table.

I know that boy likes football.

I know that boys like football.

```
?- readin(S).
```

```
|: the tall skinny girl with red hair that I met at the party
is coming.

S = [[[xmax,[[attach_subject,
        [xmax,[[attach_det,[the,'+n-v-a-p']],
        [attach_adj,[tall,'+n-v+a+p']],
        [attach_adj,[skinny,'+n-v+a+p']],
        [attach_noun,[girl,'+n-v+a-p',sg]],
            [attach_pp,[xmax,[[attach_prep,[with,'-n-v']],
                [attach_pp_object,[xmax,
                    [attach_adj,[red,'+n-v+a+p']],
                    [attach_noun,[hair,'+n-v+a-p',sg]],
                '+n-v+a-p']],
            '-n-v']]],
        '+n-v+a-p'],
        [attach_relative_clause,
      [xmax,[[attach_rpron,[that,'-n-v+a+p']],
            [attach_sent,[xmax,[[attach_embedded_subject,
                [xmax,[attach_propnoun,['I','+n-v+a-p',pn]],
                    '+n-v+a-p']],
                [attach_vp,
                [xmax,[[attach_verb,[met,'-n+v',-,tense]],
                [attach_pp,[xmax,[[attach_prep,
                        [at,'-n-v']],
                    [attach_pp_object,[xmax,
                    [attach_det,[the,'+n-v-a-p']],
                    [attach_noun,[party,'+n-v+a-p',sg]],
                    '+n-v+a-p']],
                '-n-v']]],
            '-n+v-a+p',subj]]],
        '-n+v+a+p']]],
        '-n-v+a+p']]],
        '+n-v+a-p'],
        [attach_vp,[xmax,[[progressive,[is,'-n+v']],
        [attach_vp,[xmax,[[attach_verb,[come,'-n+v',+,ing]],
        '-n+v-a+p',tnsl]]],
        '-n+v']]],
    '-n+v+a+p'],[]],[]]


| ?- start(S).

|: have the students taken the exam.

S=[[[xmax,[[attach_subject,[xmax,[the,'+n-v-a-p'],
        [students,'+n-v+a-p',pl]],'+n+v+a-p']],
    [attach_vp,[xmax,[[perfective,[have,'-n+v']],
    [attach_vp,[xmax,[[attach_verb,[take,'-n+v',+,en]],
        [attach_object,[xmax,[[attach_det,[the,'+n-v-a-p']],
        [attach-noun,[exam,'+n-v+a-p',sg]]],'+n-v-a-p']]],
'-n+v-a+p',obj]]],
    '-n+v']]],
'-n+v+a+p'],[]],[]]

|-?- start(S).
```

```
|: the block is on the table.

S=[[[xmax,[[attach_subject,
        [xmax,[[attach_det,[the,'+n-v-a-p']],
        [attach_noun,[block,'+n-v+a-p',sg]]],'+n-v+a-p']],
        [attach_vp,[xmax,[[copula,[is,'-n+v']],
        [attach_pp,[xmax,[[attach_prep,[on,'-n-v']],
            [attach_pp_object,[xmax,
                [[attach_det,[the,'+n-v-a-p']],
                [attach_noun,[table,'+n-v+a-p',sg]]],
            '+n-v+a-p']]],
        '-n-v']]],
        '-n+v']]],
       '-n+v+a+p'],[]],[]]


| ?- start(S).

|: I know that boys like football.

S= [[[xmax,[[attach_subject,[xmax,
        [attach_propnoun,['I','+n-v+a-p',pn]],'+n-v+a-p']],
        [attach_vp,
        [xmax,[[attach_verb,'[know,'-n+v',-,tense]],
            [attach_comp_phr,
            [xmax,[[attach_comp,[that,'-n-v+a+p'],
                [attach_sent,[xmax,
                [[attach_embedded_subject,
                    [xmax,[attach_noun,[boys,'+n-v+a-p',pl]],
                        '+n-v+a-p'],
                            [attach_vp,[xmax,
                    [[attach_verb,[like,'-n+v',-,tense]],
                    [attach_object,[xmax,
                    [attach_noun,[football,'+n-v+a-p',sg]],
                    '+n-v+a-p']],
                    '-n+v-a+p',subj]]],
            '-n+v+a+p']]],
         '-n-v+a+p']]],
          '-n+v-a+p',subj]]],
    '-n+v+a+p',[]],[]]

| ?- start(S).
|: I know that boy likes football.
S = [[[xmax,[[attach_subject,
            [xmax,[attach_propnoun,['I','+n-v+a-p',pn]]],
        '+n-v+a-p']],
    [attach_vp,
    [xmax,[[attach_verb,[know,'-n+v',-,tense]],
       [attach_zcomp_sent,
        [xmax,[[attach_embedded_subject,
        [xmax,[[attach_det,[that,'+n-v-a-p']],
            [attach_noun,[boy,'+n-v+a-p',sg]]],'+n-v+a-p']],
        [attach_vp,
        [xmax,[[attach_verb,[like,'-n+v',+,s]],
            [attach_object,[xmax,
        [attach_noun,[football,'+n-v+a-p',sg]],'+n-v+a-p']]
        '-n+v-a+p',subj]]],
       '-n+v+a+p']]],
'-n+v-a+p',subj]]],
 '-n+v+a+p'],[]],[]]
```

5.4.4 Parsing Output in Relation to Analysis in Machine Translation

The parsing output produced by MParser tries to take account of syntactic theory, i.e., reflect X-bar theory in the context of Transformational Grammar. In reflecting syntactic theory, the parsing output represents the left and right branching of structures. Thus, repetitions of prepositional phrases within a noun phrase would be represented as a right branching structure, e.g.,

The son of the servant of the mother of the daughter.

The left branching structure of a similar noun phrase would be

The daughter's mother's servant's son.

As the previous examples show, English contains both left branching and right branching structures. There are languages, such as Turkish and Japanese, where the recursive structures are predominantly left branching structures.

In terms of Machine Translation the adoption of a specific syntactic theory and analysis would be made in the analysis stage, i.e., during the processing of the source language. As stated previously, certain languages can be, for example, predominantly left branching or a mixture of both left and right branching. During the Machine Translation of English into Japanese and/or Turkish or vice versa, account would have to be taken of the differences in the branching structures. As the analysis stage of a Machine Translation system deals primarily with source language, the difference in branching would be accounted for in the transfer stage of the system which deals with aspects of both source and target languages. Any other differences in theory and analysis between source and target language would be, firstly, accounted for in the transfer stage.

5.5 Summary

MParser has been designed according to the principles for Deterministic Parsing set out by Marcus (1980) and includes structural modifications suggested by Berwick (1985) and Milne (1982, 1983, 1986), with some further structural modifications involving abandoning the packeting system used by Marcus (1980) and Milne (1983) and using a two cell Lookahead Buffer. MParser's grammar incorporates X-bar theory of phrase structure which makes easier representing the state of the parse in the Pushdown Stack. As MParser's grammar was based on that of L.PARSIFAL, it was relatively easy to produce. The notion of using X-bar theory came from Berwick (1985) which posed no problem in implementing. The grammatical and linguistic range of MParser is similar to that of PARSIFAL (Marcus, 1980), ROBIE (Milne, 1983) and L.PARSIFAL (Berwick, 1985), in that it can parse the majority of the sentences that these systems can parse.

The output produced by MParser is a syntactic structure of the input sentence. The purpose of producing the output is proof of the building of the syntactic structure by the parsing mechanism. As the main aim of building MParser has been to provide a prototype Deterministic Parser that can be used as a model for a parser to be built for processing a sublanguage; the output produced is not used for any other purpose as than that of proof of the working of the parsing process. With some changes to the grammar and parsing rules, MParser is easily modified to become MParserSub, which is discussed in Chapter 7. The output from MParserSub and its purpose is also discussed in Chapter 7.

No measurement of MParser performance has been made. The performance results of MParsersub are discussed in Chapter 7.

As MParser is based on PARSIFAL, ROBIE and L.PARSIFAL, it would be pertinent to compare MParser with them. MParser, like to PARSIFAL, ROBIE and L.PARSIFAL, comprises an Active Node Stack, Lookahead Buffer and a grammar of pattern/action rules. PARSIFAL was the first of the Marcus type parsers and stands as the basis on which parsers of this type are modelled. MParser differs from PARSIFAL in several ways. Firstly, MParser can cope with aspects of lexical ambiguity, specifically part of speech ambiguity, which PARSIFAL cannot. Secondly MParser has a lookahead of two cells whereas PARSIFAL has a lookahead of three which is deemed to be excessive by Milne (1986). This topic has been discussed in more detail in Chapter 3.

Finally with regard to grammar and grammar rules, MParser varies from PARSIFAL in that its grammar rules were not grouped into packets of rules. The packeting system was considered a redundant mechanism (Berwick, 1985), with the suggested replacement being the dotted rule. The dotted rule encoded a representation of constituents that have been parsed and replaced the tree representation built by use of the packeting mechanism. The use of the dotted rule simplified this representation in that the representation did not contain superfluous information such as both packet name and non-terminal name.

The grammar used by MParser is based on Chomsky's X-bar theory of syntax which is more restrictive than Phrase Structure syntax in terms of types rules that it allows. PARSIFAL's grammar, although based on Chomsky's work, does not incorporate X-bar theory.

The differences between MParser and ROBIE lie primarily in the grammar and grammar rules. The type of grammar and the packets of grammar rules used by ROBIE are similar to those used by PARSIFAL.

MParser differs from L.PARSIFAL in two aspects: L.PARSIFAL, like PARSIFAL could not cope with lexical ambiguity and also, similar to PARSIFAL, L.PARSIFAL had a lookahead of three buffer cells.

MParser, in essence, is an amalgam of what are considered, by the author, to be the best components for a parser of the Marcus type. The Active Node Stack, Lookahead Buffer and Grammar of Pattern/Action Rules have already been discussed as being suitable components for a parser. The use of lookahead of two buffer cells has been decided to be adequate for processing the language produced from the grammar. It has also proved suitable for processing of the sublanguage (see Chapter 7). The amendments to the grammar, by removing the packeting system, help reduce the amount of information held in the structure produced during processing. The restrictions of X-bar theory help in recognising the language processed by the parser and simplify the processing mechanism.

CHAPTER 6

LPARSER - AN LR TYPE PARSER

6.1 Introduction

In this chapter, LParser is discussed in detail. LParser, like MParser, has been built as a prototype to be tested on a set of sentences covering a wide range of grammatical and linguistic aspects taken from Marcus (1980). From this prototype another parser, LParserSub, has been developed to be used on a sublanguage. LParserSub is discussed in Chapter 7.

LParser is derived from an LR(k) parsing model. An LR(k) (left-to-right scan, rightmost derivation) parser is a deterministic parser of the shift-reduce, bottom-up variety that can work with up to k symbols of lookahead. This type of parser traditionally belongs in the world of compiler writing and context-free parsing theory, i.e., the pure side of Computer Science, and for which Aho and Ullman (1972) is the classic reference. Yet, as a result of the research done by Shieber (1983), Pereira (1985) and Tomita (1986), it has been proven that modified versions of the LR(k) parsing technique can be applied to natural language processing. LParser has been developed as a substantial extension of the Shieber and Pereira type parser, described in Chapter 3, with different modifications added to allow it to deal with ambiguity. The Shieber and Pereira parser used a small GPSG type grammar, which covered only a few set example sentences. By incorporating a much larger grammar, LParser can process a much wider range of grammatical and linguistic categories. LParser follows all the rules set out in Marcus (1980) with regard to deterministically parsing natural language.

LParser's grammar is discussed in the next section. In section 6.3, there is a detailed look at LParser, the construction of its parse table and the parsing rules. This is followed in section 6.4, by an examination of the linguistic and grammatical range of LParser, which includes a discussion of the modifications made to LParser to allow it to parse natural language.

6.2 The Grammar - A Definite Clause Grammar (DCG)

The grammar used by LParser is notated in the form of a Definite Clause Grammar (DCG) as described by Pereira and Warren (1980). LParser's grammar is similar in many ways to the grammar used by MParser, which is based on the target generative grammar acquired by Berwick's system, itself based on Marcus' grammar. The difference is that there is no transformational component in LParser's grammar. (This is not to imply that DCGs could not deal with transformation, as a transformation is a rewriting of one string into another or the writing of an arbitrary string when you have found another string - a DCG could be made to do this.) However, LParser does parse the type of sentence that would be transformed by a transformational grammar, but without a resulting transformation. All other types of sentence are parsed by LParser in a similar way to those by MParser. In the sub-section below DCGs are defined. This is followed by an examination of the grammar rules used by the system.

6.2.1 Defining DCGs

As stated above the grammar formalism used by LParser is a DCG. DCGs are more than descriptions of a language, they also can process language and be made to produce grammatical structures. DCGs, in fact, are executable Prolog programs. DCGs are explained in some detail by Pereira and Warren (1980) and are described as,

" ...a formalism originally described by Colmerauer (1975), in which grammars are expressed as clauses of first-order predicate logic, providing a natural generalisation of context-free grammars."

(Pereira and Warren, 1980, p231)

DCGs, in effect, are an extendable form of context-free grammars (CFGs). CFGs are not considered to be powerful enough for describing all aspects of natural language, whereas DCGs as extensions of CFGs are attributed with more grammatical power. The extensions to CFGs result in the following:

a) DCGs allow context-dependency within a grammar. This results in the form of a phrase, possibly, being dependent on the context in which that phrase occurs in the string.

b) DCGs permit the building of structures during parsing. The structures are not constrained by the recursive structure of the grammar

c) DCGs permit extra conditions to be part of grammar rules. These conditions can make parsing depend on auxiliary computations.

(Pereira and Warren, 1980, p233)

The extensions to the CFG that produce the DCG formalism are represented in the grammatical notation. Firstly non-terminals in DCGs can be compound terms as well as the simple atoms found in CFGs, which allows the expression of context dependency and the building structures. Secondly, the right-hand side of DCG rule can have, written within braces, procedure calls, which represent the extra conditions that can be applied to the grammar. These calls must execute for the rule to be valid. Below in Figure 6.0 are examples of a simple CFG and DCG with number agreement and structure building.

S -> NP VP	Pnoun -> John
NP -> Pnoun	Noun -> girl
NP -> Det Noun	Det -> the
VP ---> Verb NP	Verb ---> likes

Figure 6.0 (a) A Simple CFG

```
(sentence(s(NP,VP))-->noun_phrase(NP),verb_phrase (VP)).

(noun_phrase(np(singular, Pnoun))-->p_noun(singular, Pnoun)).

(noun_phrase(N, np(Det,Noun))-->det(N, Det),noun(N,Noun)).

(verb_phrase(N, vp(Verb NP))-->verb(N, Verb),
                              noun_phrase(N1,NP)).

(p_noun(singular,  p_n(Word))  -->  [Word],  {is_pnoun(Word,
                                                 singular)}).
(det(N,d(Word)) --> [Word], {is_det(Word,N)}).

(noun(N,n(Word)) --> [Word], {is_noun(Word,N)}).

(verb(N,v(Word) --> [Word], {is_verb(Word,N)}).
```

Figure 6.0 (b) A Simple DCG

How do the above mentioned extensions enhance the capabilities of the DCG compared with the CFG?

a) Context-Dependency: The use of context dependency within the formalism allows contextual information to be held within the non-terminals of the DCG rules. Non-terminals of CFGs are only singular atoms and cannot represent context within grammar rules. Contextual information can be tested, which can result in certain grammatical structures not being accepted by the parser, in this case LParser. An example of contextual information that can be held within non-terminals and subsequently tested is number agreement. Number agreement is usually held in an extra argument of the non-terminals. The example DCG above includes examples which handle number agreement.

b) Structure Building: The structure building capabilities of the DCG make it a powerful grammar formalism. CFGs do not have any explicit structure building capabilities, so they can only recognise the language not parse it. Structure building is dependent on the extra arguments of the non-terminals, which expand by means of matching with grammar rules. Grammatical structures are systematically constructed in this way until matching is complete.

c) Extra Conditions: The extra conditions found in DCGs are in the form of procedure calls added to the right-hand side of the grammar rules. These extra procedures act as a restriction to the type of constituent accepted by the DCG. The CFG has no such facility.

For the purposes of this research it is only the descriptive properties of the DCG that are necessary. The structure building properties of the DCG, i.e., the parsing capabilities have not been necessary for the research undertaken, as a separate parser, LParser has been used. For similar reasons, the extra conditions, i.e., the procedure calls, that DCGs can provide have not been made use of in this work. However, the context-dependency aspect of the DCG has been utilised, as this adds quite a powerful ingredient to the descriptive capabilities of the grammar formalism.

6.2.2 The Grammar Rules

In the previous section a small example DCG was given, in this section the rules in LParser's DCG are discussed, with examples of rules been given. In total, LParser's grammar holds seventy-seven rules, which represent a wide range of sentence types, phrases and constituents, including the following: declarative sentences, noun-phrases, verb-phrases, relative-clauses, prepositional phrases, embedded sentences, imperatives and auxiliaries. Other types of phrase and constituent are also represented in the grammar. All grammar rules appear in Appendix B.

6.2.2.1 Grammar Rules for Sentences

There are seventeen grammar rules that represent sentences. Examples of the sentences are the following:

```
(sentence(N,s(NP,VP))-->noun_phrase(N,NP),verb_phrase(N,VP)).

(sentence(N,s(NP,Aux,VP))-->noun_phrase(N,NP),aux(N,Aux),
                                         verb_phrase(N,VP)).

(sentence(N,s(Wh_phr,E_Sent))-->wh_phrase(N,Wh_phr),
                                     e_sentence(N,E_Sent)).

(sentence(N,s(Aux1,NP,VP))-->aux1(N,Aux1),noun_phrase(N,NP)),
                                         verb_phrase(N,VP)).

(sentence(N,s(IVP))-->imp_verb_phrase(N,IVP)).
```

6.2.2.2 Grammar Rules for Noun-phrases

There are seven rules that represent noun-phrases. Examples of the noun-phrases are the following:

```
(noun_phrase(plural,np(Noun))-->noun(plural,Noun)).

(noun_phrase(N1,np(Det,Adj,Noun))-->det(N1,Det),Adj(N1,Adj),
                                              noun(N1,Noun)).

(noun_phrase(N1,np(NP,PP))-->noun_phrase(N1,NP),
                                    prep_phrase(N1,PP)).

(noun_phrase(N1,np(NP,RC))-->noun_phrase(N1,NP),
                                    r_clause(N1,RC)).
```

6.2.2.3 Grammar Rules for Verb-phrases

There are eleven rules that represent verb-phrases. Examples of the verb-phrases are the following:

```
(verb_phrase(N,vp(V,NP))-->verb(N,V),noun_phrase(N1,NP)).
(verb_phrase(N,vp(V,PP))-->verb(N,V),prep_phrase(N1,PP)).
(verb_phrase(N,vp(V,VP2))-->verb(N,V),verb_phrase2(N,VP2)).
(verb_phrase2(N,vp2(Inf,Verb))-->inf(N,Inf),verb(N,V)).
```

6.2.2.4 Other Grammar Rules

The remaining rules of the grammar represent a variety of phrase and constituent types. Examples of these other types are the following:

```
(comp_phrase(N,c_ph(Comp,E_Sent))-->comp(N,Comp),
                                     e_sentence(N,E_Sent)).
(cop_phrase(N,cp(Aux,NP))-->aux(N,Aux),noun_phrase(N,NP)).
(aux(N,aux(Prog))-->prog(N,Prog)).
(r_clause(N,r_c(RPron,VP))-->rpron(N,RPron),
                              verb_phrase(N,VP)).
```

6.3 LParser, the Parse Table and Interaction with Stack and Buffer

As stated previously, LParser has been derived from an LR(k) parsing model and is actually a modified version of an LALR(1) parser, which has been defined as an,

" LR(1) parser in which all states that differ only in the lookahead component of configurations are merged."

(Fischer and Leblanc, 1988, p165)

LR parsers have to work with deterministic grammars. However natural language grammars can be full of ambiguities which result in irregularities appearing in the parse table, one of the components of an LR parser. These irregularities have to be dealt with to allow the parser to process ambiguous natural language grammars. Another problem occurs when a word can have two or more different grammatical categories. Both these problems lead to modifications being made to LParser, but do not allow it to break the rules of determinism. The modifications are discussed in the next section.

LParser comprises three components - a stack, a buffer and a parse table. LParser's stack, like MParser's stack, is where grammatical structures are built during a parse. Unlike MParser, once a constituent is shifted on to the stack, it remains on the stack. The buffer, as used by LParser, takes no part in the processing of constituents. Its only function is to hold the constituents of the input string until they have to be processed. The parse table could be considered the most important of the three components that make up LParser. It is, actually, the result of first preprocessing the grammar into a finite-state transition network and from the transition network constructing a parse table. In the following sub-section the construction of the parse table is examined in detail.

6.3.1 The Construction of LParser's Parse Table

The method for constructing LParser's quite substantial parse table follows a similar method to that used for constructing an LR(1) parser parse table, but the algorithm takes account of merging those states where only lookahead differs. The algorithm also has to take account of the fact that a DCG is a complex-valued feature system - a unification type grammar. As noted by Shieber (1985a)

> " Such formalisms can be thought of by analogy to context-free grammars, as generalizing the notion of non-terminal symbol from a finite domain of atomic elements to a possibly infinite domain of directed graph structures."

The DCG used to build LParser's parse table contains sets of features to represent categories, for example noun-phrase and verb-phrase. The sets of features also contain variable values, which represent number. By including a variable value for number, the sets of features representing the different categories can represent both singular and plural versions of categories. Shieber (1985a) commented on the difficulties that can occur in using a complex based feature system, i.e., moving to an infinite non-terminal domain. The problems relate to the fact that standard methods of parsing may no longer be applicable because of the gross inefficiency or nontermination of algorithms. How this problem can be resolved is discussed below.

When using the DCG formalism to build the parse table and the item sets from which the parse table is built, consideration has to be given to the fact that categories represented within the DCG have more than one representation, e.g., there is more than one rule for noun-phrase and verb-phrase. Therefore, when during the invoking of the 'Closure' function, items are generated from features preceded by '.', all the sets of features representing the category signified by the feature preceded by '.' have to be generated as items. As a result there may be as a many as nine items generated to represent a category.

All the sets of features, representing categories, contain the variable value N, which represents number. The variable value is the same for all sets of features representing all categories, but which could signify singular or plural. The variable value is passed on during the generating of the item sets and building of the parse table. The use of the variable value within the sets of features could be considered as contributing to a meta-interpreting problem as all sets of features representing categories have to be collected and variable value passed on. The variable value is, subsequently, represented in the Parse Table and is utilised during the parsing procedure. During parsing unification is invoked during the matching step whereby the part of the sentence being processed unifies with appropriate match in the parse table. The variable value will also unify and thus the same variable value will apply through out the parse. The building of the item sets and parse table are discussed in more detail below, as is the parsing procedure.

In using an infinite non-terminal domain, as stated, the more conventional methods of parsing may no longer apply. As a result modifications have to be made to the parsing algorithm. Shieber (1985a) investigated the use of a complex-feature based formalism - PATR-II - with an extension of the Earley parsing algorithm.

As stated the main problem with the complex feature based systems, as used with a standard parsing algorithm, is its infinite non-terminal domain. For example in using such formalisms with parsing algorithms where preprocessing of grammar takes place, such as the LR algorithm, failure to terminate may occur. This is discussed below in connection with LParser.

Shieber (1985a) examined the problem with respect to Earley's algorithm, in a chart-parsing guise, which uses top-down prediction to hypothesize the starting points of possible constituents. Prediction determines which categories of constituents can start at a given point in a sentence. However Shieber noted that due to the fact that the majority of information is not in an atomic category symbol, prediction can be useless with many types of constituents being predicted that are never used in a complete parse.

Shieber's extension of the Earley algorithm to deal with PATR-II involved a technique called 'restriction' in performing top-down filtering. The addition of the restriction led to a drastic elimination of chart edges that were never used. The restriction technique involved splitting up the infinite non-terminal domain into a finite set of equivalence classes that can be used for parsing. The splitting into equivalence classes involves taking a quotient of the domain with respect to a restrictor serving as a repository of grammar dependent information. Thus, the prediction step determined which restricted dags (graph structures in the PATR-II formalism) could start at a given point.

As stated above, problems of the infinite non-terminal domain with complex feature based grammar formalisms also occur with the LR algorithm, which had implications for LParser and the DCG formalism used in this research. Amendments had to be made to the LR algorithm to deal with the problem which is discussed below in connection with construction of LParser.

The initial part of the construction LParser involved converting the grammar, the DCG, into a finite-state transition network, which involves constructing item sets or configuration sets from the grammar rules. An item set is comprised of items, dotted grammar rules, representing partial parses, and a lookahead symbol. An example item is [A -> x . B y,a], definitions of components are given below. Each state of a finite-state network has an item set.

The building of item sets is initiated by, firstly, adding another rule to the grammar, which contains a new root symbol and an end symbol. Following this, First Lists are created for each symbol of the grammar. A First List for a non-terminal is created by firstly constructing a list which holds the first symbol of the right-hand side of every grammar rule in which the non-terminal is the left-hand side symbol. The creation of First Lists is completed by examining each non-terminal in the First List already created and adding their own First List to the First List of which they are already a member.

The actual construction of the item sets is usually defined by a Closure function and transitions between item sets by a Goto function. Each function can be explained by the algorithms in Figure 6.1 and 6.2, based on Aho and Johnson (1974), Fischer and Leblanc (1988), and Briscoe (1987), which have been used to build LParser's parse table with added amendments having been made to the standard LALR algorithm to deal with the DCG.

```
Closure(Itemset)
Call (totalitemsets)
Loop
  For A -> x · B y,a   Itemset,
  rule B -> z
     and b   First(ya)
             if (B -> · z, b)  Itemset
                  or (B -> · z, b)   totalitemset
             then add (B -> · z, b) to Itemset
         end if;
  exit when no itemsets can be added;
  end For;
end Loop;
return Itemset;
```

Figure 6.1 Example Algorithm for a Closure Function

```
Goto(Itemset,X)
  if (A -> x · X y,a)   Itemset then
    J can be set of items (A -> x · y,a);
  end if;
return Closure(J);
```

Figure 6.2 Example Algorithm for a Goto Function

```
Build-item-sets(T)
    Itemset = ([S_ --> · S, $]);
    C = Closure(Itemset);
    Loop
    For each item set Itemset   C
      and each grammar symbol X
         if Goto (Itemset,X)   empty and   C
         then add goto(Itemset,X) to C;
         else if ([A --> a · X], L1)
         Goto(Itemset,X) and ([A --> a · X], L2)
         Goto(Itemset0,X) and Itemset0   C
         then let Goto(Itemset1,X)  = ([A --> a · X], L1  L2)
         and add Goto(Itemset1,X) to C;
      end if;
    end if;
    end For;
    end Loop;
    return C.
```

Figure 6.3 Example Algorithm for Building Itemsets

The above functions Closure and Goto are combined to build the item sets which relate to LParser's grammar. They are repeated until all item sets have been built, with a check built in for merging states that differ only in lookahead. This combination has produced a function called Build-item-sets, the algorithm for which is given in Figure

6.3. The amendments to the algorithm to deal with the DCG are added within the 'Build-item-sets' function. A crude form of restriction is imposed so that the non-terminal domain within the DCG will terminate; the main problem that could arise is recursion within the feature system. Restriction is enforced during the building of item sets. As well as checking whether a particular item is already a member of a particular item set, a procedure carried out within the 'Closure' function, an added check within the 'Build-item-sets' function checks whether particular items are members of other itemsets. If certain items are already members of another item set, they are not added to the item set being processed. This is accomplished by storing information on the type of items found in the item sets already processed. As a result recursion within the feature system of a DCG is prevented by restricting processing of the type of item previously processed as a member of another item set. In this instance, the items that are restricted are those developed from rules representing noun-phrases in the grammar.

The parse table is constructed using the following four rules, which involve checking the item set for terminals and non-terminals and certain types of entry. Two of the rules employ the same Goto procedure, as used previously, and all the rules give commands to be placed in the parse table. `State i is built from item set Itemset`

```
1.  If (A -> x · X y,b)  Itemset,
    X is a terminal and Goto(Itemset,X) is J,
    then entry(i,X) must read shift j

2.  If (A -> a · X y,b)  Itemset,
    X is a non-terminal and Goto(Itemset,X) is J,
    then entry(i,X) must read j

3.  If (S_ -> S · $)   Itemset,
    then entry (i,$) must read accept

4.  If (A -> x · a)   Itemset,
    then entry (i,a) must read reduce A -> a.
```

Figure 6.4 Rules for Building a Parse Table

The alphabetic symbols used in the above algorithms represent the following:

A,B = non-terminals.

a,b = terminals.

X,Y = representations of (non)terminal symbols.

x,y,z = representations of sentential forms (sequences of phrases, including null sequences).

i = present state number being executed in Parse Table

j = next state number to be executed in Parse Table

$ = end symbol.

The resulting parse table is a table of states. LParser's parse table has 408 states which are numbered beginning with state 1. More than one state can be represented by a state number, e.g., there are thirty-seven states numbered as state 1. As a result of restricting during the building of item sets The parse table holds three types of commands, which manipulate the stack and buffer. These commands are shift, reduce, and accept which do the following:

shift -	this command removes the left-most constituent of the input string buffer places it on the stack.
reduce -	this command replaces a set of symbols on the stack that correspond to the right-hand side of a grammar rule with the symbol of the left-hand side of the grammar rule.
accept -	this command signifies the end of the parse, when the stack holds the start symbol and the buffer is empty.

Examples from LParser's parse table are the following:

```
1. state(1, det(_,_), s, 8).
2. state(383, $, r1, r_clause(_,_)).
3. state(2, verb_phrase(_,_), 41).
4. state(22, $, a).
```

Each of the above represent the type of states that are found in the parse table. What do these states mean?

1. The contents of example 1 correspond to - number of the state, the left-most element of the buffer if this state is to execute, the command shift to place this element on the stack and the next state that should execute.

2. The contents of example 2 correspond to - number of the state, the lookahead symbol, the command reduce and the result of reducing.

3. The contents of example 3 correspond to - number of the state, the result of reducing (this type of state executes after the type of state in example 2) and the next state that should execute.

4. The contents of example 4 correspond to - number of the state, the lookahead symbol and the command accept signifying the end of the parse.

6.3.2 The Parsing Rules

In this section the interaction of LParser's stack, buffer and parse table, which produces a parse of a sentence, is examined. This examination of the interaction between the three components that make-up LParser involves referring to the series of Prolog rules that represent LParser's parsing rules. These rules follow the commands of the parse table and manipulate the stack and buffer. As with MParser, input strings processed by LParser go through dictionary look-up and morphological analysis prior to entering the buffer, which are discussed in a later section. Below each rule used in the parsing process is described. The Prolog code for LParser appears in Appendix

B.

6.3.2.1 *parse*

The rule *parse* is the top level rule that calls the main rule of the parsing procedure, *match_state*, the rule that controls the interaction between stack, buffer and parse table.

6.3.2.2 *match_state*

As mentioned above *match_state* is the rule that controls the interaction between stack, buffer and parse table. Different versions of the rule deal with shifting constituents from the buffer on to the stack, reducing a set of grammatical structures on the stack to a larger grammatical structure and accepting the structure on the stack headed by the start symbol, signifying that a parse has ended. *match_state* calls a number of rules and checks for certain conditions and grammatical constituents. The rules that *match_state* calls are *state, check_categories, shift, merge, reduce, merge1, reduce1 checking, check_verb, check_verb_prep* and *match_state*. Each of these rules is described individually below.

6.3.2.2.1 *state*

This is the first rule called in each of the rules that form *match_state*. *state* is a state of the parse table and holds information, for example, on shifting or reducing, the contents of the left-most element of the buffer and the next state to be executed. After *state* has executed *match_state* checks the type of command, i.e., *shift* or *reduce*. If the command is s, to shift, *check_categories* is the next rule to be executed.

6.3.2.2.2 *check_categories*

The rule *check_categories* examines the constituent at the left-most end of the buffer. The simple check is to match the constituent in the buffer with that designated by the rule state. If the match fails *state* can be re-executed keeping the same state number and command but with a different constituent to be matched with that in the buffer. Other checks performed by *check-categories* are more complex in that they deal with constituents at the left-most end of the buffer that are compound lexical entries. When this is the case, lookahead is extended by one symbol to allow disambiguation to take place. This situation is discussed in the next section in connection with modifications that have been made to LParser to allow it parse ambiguous natural language grammars.

6.3.2.2.3 *shift*

The rule *shift* is called after *check_categories* has executed successfully. *shift* removes the left-most constituent from the input string buffer and places it on the stack. This rule always executes first in preference to a reduce command because of conflicts that appear in the parse table. The problem with conflicts is discussed in the next section.

6.3.2.2.4 *merge*

If *match_state*, having checked the type of command, after *state* has executed, finds r1, the rule *merge* is called. *merge*, which is part of the reduction process, examines the contents of the stack. If the contents represent the right-hand side of a grammar rule, *merge* returns the left-hand side of the grammar rule to be used by the rule *reduce*.

6.3.2.2.5 *reduce*

The rule *reduce* is called immediately after *merge*. *reduce* checks the contents of the stack. If the contents represent the right-hand side of a grammar rule, *reduce* replaces the contents of the stack with the left-hand side of the grammar rule, passed on from *merge*.

6.3.2.2.6 *merge1*

If *match_state*, having checked the type of command, after *state* has executed, finds r2, the rule *merge1* is called. *merge1*, which is part of the reduction process, examines the contents of the stack. If the contents represent the right-hand side of a grammar rule, *merge1* returns the left-hand side of the grammar rule. The difference between r1 and r2 is that r1 represents the reduction of constituents into noun-phrases, prepositional phrases, relative-clauses, etc, whereas r2 represents the reduction of constituents into verb-phrases and sentences.

6.3.2.2.7 *reduce1*

The rule *reduce1* is called immediately after *merge1*. *reduce1* checks the contents of the stack. If the contents represent the right-hand side of a grammar rule, *reduce* replaces the contents of the stack with the left-hand side of the grammar rule, passed on from *merge1*.

6.3.2.2.8 *checking*

The rule *checking* is called when there is a need for a semantic check of the type of phrases on the stack and the constituent in the left-most element of the buffer. Certain types of phrase on the stack and contents of the buffer signify that there should be a call to *shift* to build a larger grammatical structure on the stack using a state similar to one that has already been executed. As previous state numbers are kept in a list, it is possible to execute these states with state numbers that have already been used or states that have already been used. When one of these states executes, it is possible to call *shift* and the same procedure, as described above, applies.

6.3.2.2.9 *check_verb*

The rule *check_verb* is called when a four argument state rule cannot execute and after a verb is shifted onto the stack. The rule performs a special semantic check that tests for certain types of verb in relation to their use in conjunction with prepositional phrases, i.e., the verb can be part of a verb-phrase with constituents V, NP, PP. *match_state*, subsequently, calls a three argument state of similar state number to that of the four argument state that did not execute. This state number is stored in a list to be used later for the building of a larger grammatical structure. Processing continues until a new state is executed. The execution of the new state allows the whole shift-reduce procedure to start again. This allows parsing of the constituents in the buffer that followed immediately after the verb.

6.3.2.2.10 *check_verb_prep*

The rule *check_verb_prep* is yet another semantic check that tests for certain types of verb and prepositions after the reduction of a noun-phrase, which is preceded by a verb. If the verb and preposition are of a kind that can be part of a verb-phrase with constituents V, NP, PP, then *check_verb_prep* executes. *match_state*, subsequently, prohibits certain states from executing to allow for the longer reduction, since there would be a reduce-reduce conflict in the parse table. The problem with conflicts are discussed in the next section.

6.4 The Grammatical and Linguistic Range of LParser

In this section, the grammatical and linguistic range of LParser is discussed. The discussion includes a description of the modifications that have been made to LParser to allow it to parse ambiguities of natural language grammar. The discussion also focuses on ambiguous and unambiguous sentences, as was the case when examining the grammatical and linguistic range of MParser. LParser parses similar types of sentences as those parsed by MParser. In Appendix C, the results of parses of similar sentences by the two parsers are given. The discussion begins in the next sub-section with a look at the dictionary and the dictionary look-up procedures as used by LParser. This is followed by a look at the modifications that have been made to LParser. The final two sub-sections discuss the processing of unambiguous sentences and the processing of ambiguous sentences. Examples of both types of sentence are given during the discussion.

6.4.1 LParser's Dictionary

As with MParser, the dictionary used by LParser is an important part of the parsing process. The dictionary is made-up of Prolog rules that represent grammatical constituents such as nouns, verbs, pronouns, proper nouns, prepositions, determiners, relative pronouns and wh-pronouns. The dictionary contains approximately one hundred words, which may seem small but still does allow the building of a great many grammatical structures and the handling of ambiguity. This dictionary also contains double entries to represent the polysemous nature of certain words. Examples from the dictionary are the following:

1. verb(_,(persuade)).   4. det(_,(that)).

2. noun(_,(bus)).   5. inf(_,(to)).

3.	rpron(_,(that)).		6. prep(_,(to)).

Each example is self-explanatory, since the definition for each word is given by the rule name.

6.4.1.1 LParser's Dictionary Look-up

After the input string has been read into the system, the dictionary look-up procedure search_word is called, which processes each word of the input string individually. As with MParser's dictionary look-up procedure check, the first procedure that search_word calls is end, which performs morphological analysis. After morphological analysis, which returns the root form of the word and the grammatical number, if the word is a noun, the rule dict_search executes. dict_search calls words, which holds all the grammatical constituents that are in the dictionary. The root form of word is passed into the list of grammatical constituents in words. When one of the grammatical constituents holding the root form of the word matches with a grammatical constituent in the dictionary, the full form of the word is returned by the grammatical constituent held in words. If the word is a noun, the grammatical number will also be held by the grammatical constituent returned by words to dict_search and subsequently to search_word.

When the word being processed by dictionary look-up has two or more entries in the dictionary, search_word returns all entries. All the entries remain in the processed input string until they enter the left-most element of the input buffer. At this stage disambiguation takes by extending the lookahead one symbol. This extension is performed by the rule check_categories. The extension is discussed in more detail below.

6.4.1.2 LParser's Morphological Analyzer

Morphological analysis by LParser is performed by the same method as used by MParser. The rule end is called as the first procedure of dictionary look-up. end checks for a series of endings. On finding an ending end removes it and checks the last one or two remaining letters as the might need to be replaced.

6.4.2 Modifications for Natural Language Processing

As mentioned earlier in this chapter, modifications have to be made to LParser if it is to cope with the ambiguities of natural language. These modifications have to deal with the irregularities or conflicts that appear in LParser's parse table and part of speech ambiguities, i.e., words that can be two or more different grammatical constituents, which requires extending the lookahead. Below the modifications applied to LParser are discussed.

6.4.2.1 Conflicts in the Parse Table

Irregularities appear in LParser's parse table because of the ambiguous nature of natural language. These irregularities are in the form of shift-reduce and reduce-reduce conflicts appearing in the parse table among states with the same number. In order for LParser to be called a competent parser for natural language it has to be able to deal with these conflicts. As mentioned in Chapter 3, Shieber (1983) put forward a method for dealing with these conflicts that has been adopted for this research. This method involves, in the case of shift-reduce conflicts, always performing the shift and in the case of reduce-reduce conflicts performing the longer reduction. Shieber (1983) and Pereira (1985) found that this method could describe the principles of Right Association and Minimal Attachment, and lexical preference, in which can be encapsulated many types of sentence. In Chapter 7, the significance of Right Association and Minimal Attachment for sublanguage is discussed.

6.4.2.2 Extending the Lookahead

Natural language is full of ambiguities that cause problems for any parser; part of speech ambiguity is one of the major problems. The Pereira and Shieber parser deals with the problem of ambiguity by allowing pre-terminal delaying. With this method, if a word can be two or more different grammatical constituents, then the assignment of type of grammatical constituent is delayed until the processing of the constituents following is complete which, then, allows disambiguation to take place. As a modification to the Shieber and Pereira parser, it was decided to experiment with another method of dealing with ambiguity. This method involved extending the lookahead in LParser, but, it has been realised that there are certain ambiguities that extension of the lookahead will not cope with. These ambiguities do not occur in the sample English set used. The relation to the sublanguage is discussed in the next chapter. The procedure for extending the lookahead in LParser is discussed, in detail, below. This topic is discussed in relation to the sublanguage in the next chapter.

At the stage in processing when LParser is prepared to execute the command shift, lookahead is extended to two symbols when the left-most end of the input string buffer contains a compound lexical entry of two or more constituents. A compound lexical entry signifies that a word can be more than one grammatical constituent. The extension of lookahead to two symbols makes it possible to look at the constituent in the buffer following the compound lexical entry, which allows disambiguation to take place between the constituents of the compound lexical entry. For example, with this method LParser would be able to disambiguate between the various meanings of that, as a determiner, complementiser and relative pronoun, respectively, in the following sentences:

I know that boy.

I know that boys should do it.

I told the boy, that I met, the story.

The extension of lookahead is not automatically computed from the grammar. Rather the extension of lookahead to two symbols occurs when there is a compound lexical entry at the left most end of the buffer and the parser is at the stage in processing where the shift command should execute; the presence of the compound lexical entry being the more significant factor in instigating the extension of the lookahead. At all other stages in the processing, the lookahead is one symbol.

How does extending the lookahead effect the legitimacy of LParser with regard to the rules governing deterministic parsers and the LALR(1) algorithm? The extension of lookahead does not break any of Marcus' rules for determinism. LParser does not back-track, all structures created are permanent. LParser does not employ methods of pseudo-parallelism, all structures created for a given input are output. LParser's lookahead facility still remains restricted. As regards the LALR(1) algorithm, LParser extends the lookahead once the parse table has been built, so to some extent it is going beyond the bounds of the algorithm. However, the parse table holds states representing the parsing stages of all the constituents in the compound lexical entry, so LParser is not trying to parse something for which there is no parse table entry. All the constituents are accounted for, but the ambiguous nature of language allows some words to be different grammatical constituents. Ambiguities have to be processed, as LParser has to be a competent parser of natural language. The extension of lookahead does not cause any problem with the rules of determinism, so going beyond the bounds of the algorithm to parse natural language also should not be considered a problem.

Pre-terminal delaying and extension of the lookahead perform the same function in that both procedures aid the processing of ambiguities of the type discussed above. However, if the first two examples immediately above, i.e.,

I know that boy

I know that boys should do it

were amended to include adjectives preceding the nouns, e.g.,

I know that young boy

I know that young boys should do it

extension of lookahead to two cells could not disambiguate between 'that' as a determiner and 'that' as a complementiser. (LParser was not presented with these examples.) Preterminal-delaying could cope with the ambiguities, as the processing of 'that', in both instances, would be delayed until the noun had been processed.

This short-coming in the case of extending the lookahead to two cells could be remedied by extension of the lookahead to three cells, but could go on forever depending on the number of adjectives preceding the noun. Pre-terminal delaying appears to be the best method for dealing with the type of ambiguity as discussed above, but, as stated LParser could parse the ambiguities found within the sample English text used for testing its capabilities.

6.4.3 The Processing of Unambiguous Sentences

As with MParser, the grammatical and linguistic range of LParser is represented by the type of sentence it can process. In this section the unambiguous sentences that

LParser can parse are examined. The unambiguous sentences that LParser can process contain a variety of grammatical constructions which are of the following type:

1. Aux Verbs- to - infinitive, perfective, progressive, modal, do, passive_be.
2. Simple Declarative Sentence.
3. Prepositional Phrase.
4. Verb Complements.
5. That Complements.
6. Passive Constructions.
6. Wh Questions.
8. Seem Constructions.
9. Imperatives.
10. Relative Pronouns.

Some of the above grammatical constructs can hold ambiguities, but unambiguous sentences can be produced. A few example sentences, using the above constructs, are given below, followed by parses of the sentences.

Has the girl killed the boy?

The blind boy loves the beautiful woman.

Who broke the jar?

Is there a meeting scheduled for Friday?

There seems to be a jar broken.

The boy on the bus killed the girl with a gun.

The boy on the bus likes the girl with the gun.

```
| ?- run(S,B).
|: has the girl killed the boy.
S = [sentence(_7962,s(aux1(perf(has)),np(d(the),n(girl,sg)),
              vp(v(killed),np(d(the),n(boy,sg)))))],
B = []
| ?- run(S,B).
|: the blind boy loves the beautiful woman.
S = [sentence(_18039,s(np(d(the),a(blind),n(boy,sg)),
           vp(v(loves),np(d(the),a(beautiful),n(woman,sg)))))],
B = []

| ?- run(S,B).
|: who broke the jar.

S = [sentence(_9799,s(wh_ph(w_c(who)),vp(v(broke),
                              np(d(the),n(jar,sg)))))],
B = []

| ?- run(S,B).

|: is there a meeting scheduled for Friday.

S = [sentence(_23796,s(aux1(prog(is)),there(there(there)),
    np(d(a),n(meeting,sg)),vp(v(scheduled),
                       pp(p(for),np(n('Friday',sg)))))))],
B = []

| ?- run(S,B).

|: there seems to be a jar broken.

S = [sentence(_9242,s(there(there(there),aux(seem(seems))),
             infaux(inf(to),aux(prog(be)),
                  np(d(a),n(jar,sg)),a(broken))))],
B = []

| ?- run(S,B).

|: the boy on the bus killed the girl with a gun.

S = [sentence(_35077,s(np(np(d(the),n(boy,sg)),
             pp(p(on),np(d(the),n(bus,sg)))),vp(v(killed),
                 np(d(the),n(girl,sg)),
                      pp(p(with),np(d(a),n(gun,sg))))))],
B = []

| ?- run(S,B).

|: the boy on the bus likes the girl with the gun.

S = [sentence(_27864,s(np(np(d(the),n(boy,sg)),
           pp(p(on),np(d(the),n(bus,sg)))),
              vp(v(likes),np(np(d(the),n(girl,sg)),
                 pp(p(with),np(d(the),n(gun,sg)))))))],
B = []
```

6.4.4 The Processing of Ambiguous Sentences

As discussed above, LParser can process certain ambiguous sentences, by extending the lookahead, a method that has proved not to be as sophisticated as pre-terminal delaying. The ambiguous sentences that LParser can process are similar to those that are processed by MParser. In Chapter 5, the two types of ambiguity, which produce ambiguous sentences, were discussed; the ambiguities being structural ambiguity and part of speech ambiguity. LParser, as built, cannot deal with structural ambiguity produced by the following, if Have were to be assigned both a main verb and an auxiliary.

Have the students take the exam.

Have the students taken the exam?

However, if LParser had been designed in exactly the same manner as the Pereira and Shieber parser, which used pre-terminal delaying to deal with ambiguity, then it would have been able to process the sentence; the reason being that the assignment of the type of verb to Have could be delayed until the processing of the verbs take or taken. However, it has been decided that LParser being able to process both of these types of construction is not important, especially when the use of Have, as a main verb imperative, is not a common type of British-English grammatical construction.

LParser has similar difficulties to those of the Marcus type parsers in dealing with constructions that exhibit unbounded dependencies, e.g., constituent questions, as stated by Church, cited in Briscoe (1987). To reiterate, the reason for the difficulties in processing is that it is impossible for a deterministic parser trying to solve local ambiguities by using lookahead to find the correct gap in unbounded dependencies; lookahead in a deterministic parser is restricted. In order to resolve the difficulty lookahead would have to be unrestricted. Whether this causes a problem for LParser in parsing sublanguage is discussed in Chapter 6.

As with MParser, LParser (although not programmed to do so) could handle the ambiguities of rightward movement. To recap, Briscoe (1987) notes that rightward movement in the English language is treated:

> " as a case of canonical attachment of the postmodifier to the S node, where semantics remains a matter of more general inference rather than a fully determinate, grammatically defined binding."

This allows deterministic parsing of the rightwardly moved relative clause or prepositional phrase.

How does LParser handle ambiguities it can cope with? As mentioned above, if a word occurs more than once in the dictionary, all entries of the word are passed, at dictionary look-up, to the parsing process. When this group of entries is at the left-most end of the buffer, only one of the entries can be pushed onto the stack. In the previous section, it has been described that in handling some cases of ambiguity, it is necessary to expand the lookahead to two symbols. For example, with the sentences below, to disambiguate to, which can be a preposition or an auxiliary to an infinitive verb, it is necessary to have a lookahead of two symbols.

I went to school.

I want to drive.

There is an added problem to the above disambiguation procedure if the word following *to* is the double entry of noun and verb in the dictionary. Although not implemented within LParser, the problem could be dealt with, to some extent, verb subcategorisation. Verb subcategorisation would distinguish those verbs that could take infinitive complement, thus *to* would be an auxiliary verb starting a VP and the word following disambiguated as a verb. Similarly, verbs could be classified for PPs with preposition *to*. Obviously, if a verb does not precede *to*, it will be difficult to disambiguate the double entry noun and verb following *to*.

If the word to be disambiguated is *block*, there is no need to expand the lookahead to two symbols. The reason for this situation being that the parse table guides the processing closely. For example, with the sentences below, the disambiguation of *block*, which can be a noun or a verb, is guided by the states of the parse table. Having processed *The*, the parse table guides that only a grammatical constituent that follows a determiner will be the next constituent to be processed, so in example 1 *block* can only be a noun. In example 2 *block* can only be a verb because it follows an auxiliary *will* and the parse table guides that the next constituent to be processed is verb.

1. The block is red.
2. He will block the road.

Below examples are given of sentences with ambiguities and their resulting parses. These examples show varying aspects of ambiguity that occur within part of speech ambiguity.

I know that boy likes football.

I know that boys like football.

The boy on the bus that killed the man loves the woman.

The block stands on the table.

I walked to the meeting.

I want to go.

```
| ?- run(S,B).

|: i know that boy likes football.

S = [sentence(_21442,s(np(p_n(i)),vp(v(know),
                  z_c_ph(e_s(np(d(that),n(boy,sg)),
                  vp(v(likes),np(n(football,sg))))))))],
B = []

| ?- run(S,B).
```

```
|: i know that boys like football.

S = [sentence(_12384,s(np(p_n(i)),vp(v(know),
                c_ph(comp(that),e_s(np(n(boys,pl)),
                vp(v(like),np(n(football,sg))))))))],
B = []


| ?- run(S,B).

|: the boy on the bus that killed the man loves the woman.


S = [sentence(_14888,s(np(np(np(d(the),n(boy,sg)),
          pp(p(on),np(d(the),n(bus,sg)))),
          r_c(r_p(that),vp(v(killed),np(d(the),n(man,sg))))),
          vp(v(loves),np(d(the),n(woman,sg)))))],
B = []


| ?- run(S,B).

|: the block stands on the table.

S = [sentence(_16064,s(np(d(the),n(block)),
                vp(v(stands),pp(p(on),
        np(d(the),n(table,sg))))))],
B = []


| ?- run(S,B).

|: i walked to the meeting.

S = [sentence(_14206,s(np(p_n(i)),
               vp(v(walked),pp(p(to),
        np(d(the),n(meeting,sg))))))],
B = []


| ?- run(S,B).

|: i want to go.

S                                                            =
[sentence(_5391,s(np(p_n(i)),vp(v(want),vp2(inf(to),v(go)))
))],

B = []
```

6.5 Summary

LParser has been designed as a modified version of an LALR(1) parser that adheres to Marcus' rules for the deterministic parsing of natural language. The modifications that have been incorporated by LParser allow it to deal with ambiguities of natural language. The ambiguous natural language grammar used by LParser produces conflicts in its parse table. These conflicts are shift-reduce and reduce-reduce conflicts, which LParser handles by shifting in favour of reducing in shift-reduce conflicts and performing a long reduce when there is a reduce-reduce conflict. Another problem with an ambiguous grammar is that it holds part of speech ambiguities. LParser deals with part of speech ambiguities by extending the lookahead to two symbols, if necessary. The extending of the lookahead does not break any of the rules of deterministic parsing.

LParser's grammar is a DCG. Although the DCG is a powerful grammar formalism, which can even perform parsing, it is only the notation of the DCG that is utilised by LParser.

The output produced by LParser is a syntactic structure of the input sentence. The purpose of producing the output is proof of the building of the syntactic structure by the parsing mechanism. As the main aim of building LParser has been to provide a prototype Deterministic Parser that can be used as a model for a parser to be built for processing a sublanguage, the output produced is not used for any other purpose as than that of proof of the working of the parsing process. With some changes to the grammar and parsing rules, LParser is easily modified to become LParserSub, which is discussed in Chapter 7. The output from LParserSub and its purpose is also discussed in Chapter 7.

The performance of LParser has not been measured, as it was built as a prototype. The performance of the parser, LParserSub, which is based on the prototype, to be used as part of a Machine Translation system, is discussed in the next chapter.

In comparing LParser with the Shieber and Pereira parser, on which it is based, both are similar in that both parsers have an input buffer, stack and parse table. The main difference between the parsers is in the type and size of grammar used to create the parse table. The Shieber and Pereira parser used a small categorial type grammar with a few rules, whereas LParser used a Definite Clause Grammar with almost eighty rules. The larger number of grammar rules in LParser resulted in a much larger parse table than that used by the Shieber and Pereira parser; thus LParser could process more and different types of sentences than the Shieber and Pereira parser.

The other difference between the two parsers is the method of dealing with ambiguity. In the Shieber and Pereira parser ambiguity is dealt with by invoking pre-terminal delaying. Pre-terminal delaying causes the processing of the ambiguous word to be delayed until processing of succeeding words takes place which aids disambiguation. LParser uses extension of the lookahead to two cells to deal with ambiguities. However, the extension of lookahead to two cells would not be able to cope with ambiguities where the disambiguating word did not immediately follow the ambiguous word, for example, 'that' in the examples

I know that small boy.

I know that small boys like football.

On comparing the strategies used by both parsers to deal with ambiguity, the pre-terminal delaying strategy suggests itself as the better for dealing with certain ambiguities, although LParser can cope with the all the examples of ambiguity it was presented with, as capably as the Shieber and Pereira parser. As stated, the implications of the extension of lookahead and the sublanguage are discussed in the next chapter.

The main aim of building LParser, as with MParser, has been to provide a prototype deterministic LR parser that can be used as a model for a parser to be built for parsing a sublanguage. The building of this new parser, LParserSub, is discussed in the next chapter.

CHAPTER 7

DETERMINISTIC PARSERS FOR A SUBLANGUAGE

7.1 Introduction

In this chapter the use of Deterministic Parsers on a sublanguage, in particular, the sublanguage of an aircraft maintenance manual, is examined. Both types of Deterministic Parser, discussed in Chapters 5 and 6, have been modified in order to be able to process the aforementioned sublanguage, specifically noun-phrases from that sublanguage. The new parsers are MParserSub, a modified version of MParser, and LParserSub, a modified version of LParser. MParserSub and LParserSub are discussed, in detail, in sections 7.4 and 7.5, respectively. In section 7.2, the discussion focuses on the sublanguage of an aircraft maintenance manual. In section 7.3, the discussion focuses on complex-nouns: a phenomenon found in full natural language and very common in aircraft maintenance manuals.

7.2 The Sublanguage of an Aircraft Maintenance Manual

As stated previously, the sublanguage used in this research is that of an aircraft maintenance manual, namely that found in an extract of a Rolls-Royce aircraft maintenance manual. Therefore, all results of the parsing process can only be attributed to this specific aircraft maintenance manual, but there seems no obvious reason why the results should not be a good general reflection of the parsing of any Rolls Royce aircraft maintenance manual or any English aircraft maintenance manual. The reason for using the aircraft maintenance manual as the sublanguage to be test-parsed by both types of

Deterministic Parser is that the maintenance manual is regarded as a good practical example of the kind of document that can be processed by a Machine Translation system; the main interest of the research is the investigation of the use of a Deterministic Parser on a sublanguage for Machine Translation.

In sub-section 7.2.1, the sublanguage of the aircraft maintenance manual, as a whole, is described. This is followed by a discussion of ambiguities that occur in the sublanguage. Finally, there is a detailed examination of the different types of noun-phrases found in this particular aircraft maintenance manual.

7.2.1 The Aircraft Maintenance Manual

Before looking at the specifics of the sublanguage of the aircraft maintenance manual, it would be of use to reiterate details, outlined in Chapter 4, concerning the majority of sublanguages. Firstly there are several definitions of a sublanguage, three of which are listed below:

1. A sublanguage is an independent system.
2. A sublanguage is identified with a particular semantic domain.
3. Sublanguage texts usually contain some material that does not belong to the sublanguage proper.

All these definitions are applicable to an aircraft maintenance manual.

Other important details concerning sublanguage appertain to vocabulary and grammar rules. Sublanguages do not have the same size of vocabulary or the same number of grammatical rules as the language as a whole. In addition, the vocabulary of the sublanguage can contain words that are restricted to having only a single meaning within the sublanguage, but can have many meanings within the whole language. All this information also can be applied to the aircraft maintenance manual being used in this research.

The type of text found in the Rolls-Royce aircraft maintenance manual sublanguage is both descriptive and imperative, with the imperative text taking up the greater part of the manual. The descriptive text describes various aspects concerned with the maintenance manual, i.e., the TASKs that make-up the manual, TASKs that have been previously referenced, parts of the aircraft and the contents of the manual. Examples of this type of text, representing the four aspects mentioned, are the following:

1. These tasks give the procedure for the removal and installation of the igniter plugs
2. Torque tightening technique.
3. The electrical discharge of the high energy (H.E.) ignition unit is potentially lethal.
4. When reference is made to a part in a different Chapter/Section/Subject, the Fig/item numbers come after the appropriate Chapter/Section/Subject.

The imperative text is concerned, mainly, with parts of the aircraft. Examples of this type of text are.

1. Remove the T26 thermal unit.
2. Install RR289200 protective workmat 1 off in the air intake.
3. Seal the quillshaft with OMat1252 kraft paper tape.

As mentioned above, the Rolls-Royce aircraft maintenance manual is comprised of

TASKs; an example TASK being:

```
Remove/Install the airflow control rpm signal transmitter.
```

Each TASK is broken down into SUBTASKs to enable more detailed instructions to be given about the TASK to be performed. The text covering each TASK is divided into numbered sections, with the majority of the sections being divided into instructions, listed A, B, C, etc. The remaining sections are, generally, descriptive in nature. Text from the first three sections of a TASK is given below.

```
LOW    PRESSURE    (L.P.)    TURBINE    BLADES    -    STAGE    3
INSPECTION/CHECK
TASK 72-52-34-200-000       Examine the stage 3 low pressure
(L.P.) turbine blades
1. Equipment and Material
   A. Standard equipment
      Strong spotlight
   B. Consumable materials - not applicable
   C. Special tools
      RR289200     Protective workmat        1 off
   D. Expendable parts - not applicable
2. General

This TASK gives the procedure for the inspection of the stage

3 L.P. turbine blades.
When   reference   is   made   to   a   part   in   a   different
Chapter/Section/Subject the Fig./item numbers come after the
appropriate Chapter/Section/Subject.
The   engine   in   flight   position   gives   the   positional
relationship.  Detailed radial locations have numbers in a
clockwise direction that start from the engine top position
when you look from the rear, unless otherwise told.
It is advisable to do this TASK at the same time as the
examination of the stage 3 low pressure (L.P.) nozzle guide
vanes, TASK 72-52-20-200-000 (72-52-20, P.B.601).

3. Referenced Procedures
   Low pressure (L.P.) nozzle guide vanes - stage 3, TASK
72-52-20-200-000 (72-52-20, P.B.601).
```

The first three numbered sections of each TASK all have the same headings: Equipment and Material, General, and Referenced Procedures. In every TASK, each of these sections in every TASK is set out as above, but each section does not contain the same information. In section 1, Equipment and Material is categorised into four sub-sections A, B, C and D, each of which is headed by a sub-heading representing specific equipment or materials. If a certain piece of equipment or material is needed by the TASK, it is listed under the specific heading. However, if no equipment or material, as categorised by a certain sub-heading, is needed by the TASK, "not-applicable" appears beside the heading. In section 2, General gives general information appertaining to the whole TASK. In section 3, Referenced Procedures lists other TASKs that need to be used within the TASK to be performed. Sections 4 - 6 are listed below, with discussion about the text following immediately after.

```
SUBTASK 72-52-34-010-001
4. Make Preparation to Examine the Stage 3 L.P. Turbine
Blades
   A. Open relevant circuit breakers to isolate electrical
supply to the engine.
   B. Isolate the thrust reverser system.
   C. Install DO-NOT-OPERATE identifiers.
   D. Install RR289200 protective mat 1 off.
SUBTASK 72-52-34-210-001
5.   Examine   the   Stage   3   L.P.   Turbine   Blades,
Fig.72-52-34-990-001
   WARNING: TAKE CARE, WHEN YOU TURN THE L.P. COMPRESSOR AND
L.P. TURBINE ASSEMBLY, IN ORDER TO PREVENT INJURY TO FINGERS
AND HANDS.
   CAUTION:   TO PREVENT POSSIBLE DAMAGE TO THE L.P.
COMPRESSOR AND L.P. TURBINE BLADES, DO NOT USE METAL BARS OR
SIMILAR EQUIPMENT TO TURN AND/OR LOCK THE L.P. COMPRESSOR OR
TURBINE WHEN YOU EXAMINE THE VANES.
   A. From the front of the engine have someone slowly turn
the L.P. turbine at the rear of the engine.
   B. With the use of a strong spotlight, examine the L.P.
turbine blades at the rear of the engine for impact damage,
cracks or metal deposits.
      (1) The engine is acceptable if there is:
        (a) Light impact damage to the turbine blade
        airfoil, if the metal is not cracked or torn.
        (b) Light impact damage to the turbine blade shroud,
        if the metal is not cracked or torn.
      (2) Do not use the engine if there is:
          (a) Turbine blades which are cracked or torn.
          (b) Turbine blades which have a segment missing.
          (c) Turbine blades with metal deposits.
SUBTASK 72-52-34-410-001
6. Put Engine Back to Normal
   A. Remove RR289200 protective mat 1 off.
   B. Remove DO-NOT-OPERATE identifiers.
```

```
C. Close the relevant circuit breakers.
D. Engage the thrust reverse system.
```

Each of the remaining numbered sections in every TASK represents a SUBTASK. These SUBTASKs contain information on every procedure that has to be performed to fulfil the TASK. The majority of the text, making up each SUBTASK, is of the imperative type, as can be seen from the example above.

The vocabulary of the Rolls-Royce aircraft maintenance manual sublanguage is comprised of nouns, verbs, prepositions, conjunctions, relative pronouns, numerals and reference numbers. The majority of nouns represent parts of machinery, e.g., unit, nuts, blank, turbine. The manual also comprises complex nouns such as the following:

thermal unit joint face.

12th stage air offtake outlet duct assembly.

The complex nouns are more representative of parts of an aircraft, since they have to be exact descriptions. The majority of the verbs represent maintenance actions such as *examine, remove, install, attach,* etc. The reference numbers are a mixture of letters and numerals. The prepositions, conjunctions and relative pronouns are not sublanguage specific and could be found in any sublanguage. Some of the nouns and verbs also could be found in other sublanguages, especially, those of a technical nature.

As stated above, the research has concentrated only on the processing of the noun-phrases of the aircraft maintenance manual. The noun-phrase, of course, can include a variety of different phrases, e.g., prepositional phrases, relative clauses, verb-phrases; the noun-phrases found in the Rolls-Royce aircraft maintenance manual are examined in sub-section 7.2.3.

By concentrating on noun-phrases, it means that sentence types have been ignored.

The majority of sentences are of the imperative type, as shown in examples above. These sentences include the phrase types found in the noun-phrase, such as verb-phrases, relative clauses and prepositional phrases but the make-up of some of these is different to the make-up of those found in the noun-phrases. Below, in Figure 7.0, are examples are given of grammar-type rules representing sentence and phrase types that are not processed by either of the parsers, MParserSub or LParserSub, but which are included to show the grammatical range of the sublanguage.

```
S -> NP, VP                     VP -> ImpV, NP
S -> Letter, PP, VP             VP -> V, NP
S -> Letter, VP                 VP1 -> to, V, comp-phrase
S -> N, NP, VP, conj,VP         VP1 -> to, V, conj, V, NP
S -> Num, Sub-clause, V         VP -> ImpV, NP, PP
```

Figure 7.0 Examples of Grammar Rules of Sentences and Verb-phrases

As the excerpts from the aircraft maintenance show, the aircraft maintenance manual sublanguage contains many of the linguistic structures that were discussed in Chapters 5 and 6 in relation to MParser's and LParser's processing of unambiguous sentences. For example, the sublanguage sample has simple declarative sentences, imperative sentences, relative clauses, prepositional phrases, verb complements, passive constructions and auxiliary verbs. In the following sub-section ambiguity in the Aircraft Maintenance Manual is discussed.

7.2.2 Ambiguity in the Aircraft Maintenance Manual

In Chapters 5 and 6, the ambiguous linguistic structures that the MParser and LParser can handle were discussed. The types of ambiguity examined were structural ambiguity and part of speech ambiguity. These types of ambiguity and whether they occur in the sample of the Aircraft Maintenance Manual are now examined.

An Aircraft Maintenance Manual can be considered as a technical and explanatory sublanguage. Thus, there is a need for the language not to be ambiguous in its meaning. Any sublanguage is considered to be a restricted form of language in that it has a smaller vocabulary, a smaller number of grammatical structures and fewer instances of polysemy, i.e., words tend to only appear in one category. Aircraft Maintenance Manuals, therefore, are examples of language where it is possible to find fewer examples of ambiguous structures than in other samples of language. However, this does not mean that ambiguity does not exist within the Aircraft Maintenance Manual.

When discussing structural ambiguity in Chapters 5 and 6, the examples examined were 'have' in ambiguous structures that can be interpreted as interrogatives or imperatives, unbounded dependencies and rightward movement. It was discussed that deterministic parsers could deal with 'have' in ambiguous structures and rightward movement but not, for example, unbounded dependencies that occur in constituent questions. In the Aircraft Maintenance Manual sample discussed in this research, there are no examples of any of these types of structural ambiguity. However, if the structures contained the ambiguities concerned with 'have' or rightward movement, deterministic parsers could deal with the problems. As stated, deterministic parsers cannot deal with unbounded dependencies, such as occur in constituent questions, but this does not cause problems for this research as this type of construction is not found in technical sublanguages and Aircraft Maintenance manuals in particular (Kittredge, 1982).

The ambiguity that can occur in Aircraft Maintenance Manuals is part of speech ambiguity. Aircraft Maintenance Manuals have a predominance of compound nouns, which can contain categories that are polysemous i.e., they can be nouns or verbs, not withstanding the fact the lexicon would be restricted. An example taken from Kittredge (1982) shows the type of ambiguity that can occur. This example has the added problem of structural ambiguity as a result of containing the coordinate conjunction 'and'.

Disconnect pressure and return lines from pump.

In the sample Aircraft Maintenance Manual, used in this research, there are a few examples of part of speech ambiguity. These examples involve the polysemous words 'lock' and 'access', which in the sample occur as both nouns and verbs. There are, obviously, other example of polysemous categories within the sample sublanguage; however, in the sample they only occur as a single category. Below are listed examples:

box	(noun)	drain	(verb)
return	(noun)	use	(verb)
cover	(noun)	pump	(noun)

The sample of the Aircraft Maintenance manual used in this research also does not contain examples of the structural ambiguity that occur with the coordinate conjunctions 'and' and 'or'. As Kittredge's example above shows, such structures do occur in Aircraft Maintenance Manuals, and therefore would have to be processed. How the parsers developed for this research deal with the sublanguage is discussed in more detail later in this chapter.

7.2.3 The Noun-phrases of the Aircraft Maintenance Manual

As stated above, the parsers MParserSub and LParserSub have been designed to process only the noun-phrases of the aircraft maintenance manual. In total, there are forty-two types of noun-phrase in the manual. The noun-phrases comprise the normal grammatical constituents of determiners, nouns, adjectives and conjunctions, but also comprise the sublanguage specific constituents, such as reference numbers and numerals. The processing of conjunctions within noun-phrases does not pose much problem as the conjunctions are used only as simple connecting devices. In example 8, below, the conjunction is used to connect two noun-phrases related to the determiner. Other examples of conjunctions occurring within noun-phrases are the connecting of nouns, adjectives and adjectives and nouns. The processing of noun-phrases from the aircraft maintenance manual containing conjunctions does not have to deal with ellipsis or any other problematical structures that can occur during the processing of conjunctions; the conjunction connects the phrase or category immediately preceding it with the phrase or category immediately succeeding it.

The noun-phrases also include prepositional phrases, of which there are two types; relative clauses, of which there are three types; verb-phrases called as constituents of the relative clauses, of which there are six types; a subordinate clause called as a constituent of a verb-phrase and auxiliaries called as constituents of the relative clauses, of which there are two types. Examples of typical and non-typical grammar-type rules representing the noun-phrases, which are not specific to either parsing system, are given below:

```
1. NP -> Det, Noun            5. NP -> Ref, Noun, Noun
2. NP -> Det, Noun, Adj       6. NP -> Det, Num, Noun, Num
3. NP -> Pronoun              7. NP -> NP, PP
4. NP -> Num, Noun            8. NP -> Det, NP, Conj, NP
```

Examples of the noun-phrases represented by the above grammar rules are the following:

1.	the gearbox	5.	Omat4/23 anti-seize compound, pure nickel special
2.	a segment missing	6.	the three bolts (1)
3.	you	7.	the inspection of the stage 3 LP turbine blades

4. 4 litres 8. the LP compressor and LP turbine assembly

As shown in examples 5, 7 and 8, complex nouns in the Rolls-Royce aircraft maintenance have been classified as one noun; this is a logical step to take, as complex nouns represent a single entity. It was decided to adopt two approaches to processing complex nouns as they are known to pose problems in natural language processing. Both approaches are discussed in the following section.

7.3 Dealing with Complex Nouns

As stated in the introduction, complex-nouns are very common in aircraft maintenance manuals due to the need for precise descriptions of parts of aircraft. This discussion may seem to veer from the goals of the thesis in that it is not concentrating on the discussion of the use of deterministic parsers in the analysis stage of a Machine Translation System. Rather, the discussion concentrates on the procedure for dealing with complex-nouns in the Rolls-Royce aircraft maintenance manual which involves preprocessing the complex-nouns before parsing begins. The main reason for concentrating on this topic is the fact that the majority of the aircraft maintenance manual is comprised of complex-nouns and they could not be ignored. Therefore, this section can be considered as not being within the goals of thesis but necessary because of the importance of complex-nouns.

In the Rolls-Royce aircraft maintenance manual complex nouns comprise between two and nine constituents. The constituents can be both adjectives and nouns; adjectives are closely linked with nouns to enhance the descriptive qualities of complex nouns found in the aircraft maintenance manual. The greater the number of constituents in the complex noun, the more difficult the processing, as the semantic relationships between constituents and groups of constituents have to be considered.

Other problems are posed by the fact that individual constituents may also appear on their own in the manual or may be grouped with other constituents to form completely different complex nouns. All that is discussed below applies to complex nouns, which are later parsed by both MParserSub and LParserSub.

7.3.1 The Simple Approach to Complex Nouns

As stated above, two approaches have been used to deal with complex nouns. The first approach to be described is the simpler. The dictionary contains complex nouns of between two and three constituents, as well as other grammatical constituents. Each complex noun in the dictionary contains one, two or three constituents that do not appear elsewhere in the manual apart from in the particular complex noun. As complex nouns can be difficult to process, this use of the dictionary provides an easy method of dealing with them. Examples of the dictionary entries, coded in Prolog, for these complex nouns are the following:

```
noun(_,([nose, cone, fairing])).
noun(_,([power, plant])).
noun(_,([petroleum, jelly])).
```

This method, although in some instances it makes things simple, does not provide the full answer to the problem of complex nouns. It would be impossible to store all complex nouns in this manner, as it would create a massive dictionary which would be detrimental to processing speed even, if full advantage was taken of the standard indexing of Prolog procedures. Problems also would occur, as a result of certain nouns appearing in more than one complex noun structure.

7.3.2 The Complex Approach to Complex Nouns

As stated, the above method cannot be used for all complex nouns, even those with only two or three constituents. As mentioned previously, constituents which form part of a complex noun can appear on their own in the manual or with different constituents to produce different complex nouns. In order to deal with the complex nouns that do not appear in the dictionary, predefined syntactic/semantic relationships between groups of nouns and adjectives have been set up. Prior to parsing, when a group of adjectives and nouns are detected, they are preprocessed and checked against the predefined relationships. The theory of relationships and preprocessing of complex nouns are discussed in more detail below.

7.3.2.1 The Theory of Relationships

Before discussing the preprocessing and checking of complex nouns, it is necessary to describe the theory concerning the use of predefined syntactic/semantic relationships. Several researchers have considered the problem of complex nouns. Downing (1977) and Levi (1978) have both looked at the problem of Noun + Noun compounds from a purely linguistic standpoint and both came to the same conclusion about semantic/syntactic relationships. Both also considered Adj + Noun compounds as being complex structures. (References to complex nouns include Adj + Noun structures). For example, Levi (1978) suggested that the syntactic/semantic relationship between Noun + Noun and Adj + Noun compounds could be derived by predicate deletion and predicate nominalization; a larger number of complex nouns being derived by predicate deletions. Levi proposed nine Recoverably Deletable Predicates. Examples of complex nouns derived from the predicates are:

CAUSE(tear gas), HAVE(industrial area), ABOUT(price war).

MAKE(musical clock), USE(steam iron), FROM(olive oil),

BE(soldier ant), IN(marine life), FOR(horse doctor).

Levi (1978) proposes that three types of complex nouns could be derived from four types of predicate nominalization, which she represented in table form.

	Subjective	Objective	Multi-Modifier
Act	parental refusal	dream analysis	city land acquisition
Product	clerical errors	musical critique	student class marks
Agent	-	city planner	-
Patient	student inventions	-	-

Table 7.0 Derivation of Complex Nouns - Predicate Nominalization

Downing (1977) puts forward twelve relationships, which she believes represent a stock set of the most common underlying relationships for deriving complex nouns. The relationships proposed are the following:

WHOLE-PART(duck foot),	HALF-HALF(giraffe-cow),
PART-WHOLE(pendulum clock),	TIME(summer dust),
COMPOSITION(stone furniture),	SOURCE(spring water),
COMPARISON(pumpkin bus),	PLACE(Eastern Oregon Meal),
PRODUCT(honey glands),	USER(flea wheelbarrow),
PURPOSE(hedge hatchet),	OCCUPATION(coffee man).

However, Downing (1977) does not claim that the above relationships are an exhaustive set of compounding relationships.

Lehrberger (1982) reports, briefly, work done at TAUM Aviation on complex nouns from an aircraft maintenance manual and which follows the stance taken by both Downing and Levi that syntactic/semantic relationships can be defined between nouns and also groups of nouns. A set of fifty relationships was derived at TAUM, including several of those derived by Levi and Downing. The full set of the TAUM Aviation relationships was not provided.

In this research a similar approach also has been taken in dealing with complex nouns. A set of nine binary relationships has been derived for the complex nouns from the extract of the Rolls-Royce aircraft maintenance manual. Some of the relationships are similar to those used by the Downing, Levi and Kittredge; the rest are specific to the sublanguage of the Rolls-Royce aircraft maintenance manual, but this does not preclude that they could be used to define relationships in other sublanguages. The relationships are Use, Mod, Obj, Whole-part, Measure, For, Place, Has and Ref. In this work, the binary relationships have a Domain and Range, each of which can represent a singular noun or group of nouns. The examples below are of complex nouns from the Rolls-Royce maintenance manual and the relationships that have been specifically derived for the complex nouns, along with an explanation of the relationship used.

```
Low Pressure compressor
(Domain(Use)
    (Domain(Mod) Low
      Range(Mod)  Pressure)
Range(Use) compressor)
```

The relationship Use represents a deletable instrumental predicate, as the example complex noun can be said to derive from "the compressor uses Low Pressure". This relationship is similar to the definition in Levi (1978). The relationship Mod represents an adjectival modifier, in that the noun Pressure is modified by the adjective Low. This relationship has been derived specifically for the Rolls-Royce maintenance manual, since it has been decided to include adjective as part of complex nouns. However, the relationship could be used in the derivation of complex nouns in other sublanguages or full natural language.

```
turbine blade
(Domain(Whole-part) turbine
Range(Whole-part)  blade)
```

The relationship Whole-part represents the make-up of a particular entity. In this example of the complex noun blade is part of the whole entity turbine. This relationship is similar to the definition in Downing (1977).

```
7th stage outlet connector assembly
(Domain(Ref) 7th stage
Range(Ref) (Domain(Obj)
                (Domain(For) outlet
                Range(For)  connector)
            Range(Obj) assembly)
```

The relationship Ref represents reference, in that "outlet connector assembly" is referenced as type "7th stage". This relationship has been derived specifically for the sublanguage of the Rolls-Royce aircraft maintenance manual. However, as with Mod described above, it could pertain to other complex nouns found in sublanguages or full natural language. The relationship Obj represents object, in that assembly is the whole object, made up of parts appertaining to outlet connector. This relationship is similar to Ref and Mod in that it is specific to the aircraft maintenance manual, but could be used elsewhere. The relationship For represents a deletable purpose predicate, in that connector has a purpose which is connecting the outlet. This relationship For is similar to the relationship defined in Levi (1978).

```
engine top position
(Domain(Place) engine top
Range(Place)  position)
```

The relationship Place represents place, in that the place of position is engine top. This relationship is similar to that used by Downing (1977).

thermal unit mounting face
(Domain(Has) **thermal unit**
Range(Has) **mounting face**)

The relationship Has represents a deletable possessive predicate, in that thermal unit has a mounting face. Thermal unit and mounting face are both complex nouns that appear in the dictionary as complex nouns. This relationship is similar to the Have relationship found in Levi (1978).

rpm indicator generator
(Domain(Measure) **rpm**
Range(Measure) **indicator generator**)

The relationship Measure represents measurement, in that "indicator generator" is measured in "rpm". This relationship is specific to the Rolls-Royce aircraft maintenance manual, but as with other relationships specific to the maintenance manual it could be used elsewhere.

```
7th stage air offtake outlet connector assembly lock ring
(Domain(Ref) 7th stage
Range(Ref)
      (Domain(Has)
          (Domain(Has)
                  (Domain(For) air
                   Range(For)  offtake)
           Range(Has)
                  (Domain(Obj)
                 (Domain(For)outlet
                  Range(For)connector)
                   Range(Obj) assembly)
      Range(Has) lock ring)
```

The relationships shown in the above derivation have the same meanings as defined in the preceding derivations. This is the largest complex noun in the Rolls-Royce aircraft maintenance manual.

Having looked at a theory describing complex nouns, it is necessary to examine how it is put into practice. How do a group of adjectives and nouns get translated into a set of relations. The process is discussed in the next sub-section.

7.3.2.1 The Preprocessing of Complex Nouns

Prior to parsing, if it is found that nouns, perhaps modified by an adjective, are grouped together in the input string, they have to be preprocessed into a state for parsing. This involves producing a set of binary relationships that links the adjectives and nouns within the group. The process of creating the binary relationships occurs in three stages, each of which is discussed below, with reference being made to the Prolog rules used. The method of preprocessing complex nouns is similar for both parsers.

The initial stage of the preprocessing is dictionary look-up. Prior to the processing of other grammatical constituents, the first part of dictionary look-up involves searching the input string for complex nouns that are already stored in the dictionary. After all the predefined complex nouns have been processed, the remaining grammatical constituents of the input string are processed.

Every noun and adjective that can form part of a complex noun or several complex nouns, has in its dictionary entry its binary relationship definitions and the adjective or nouns forming the second constituent of the binary relationship. All relations have been devised by inspection of the Rolls Royce Aircraft Maintenance Manual. Examples of these dictionary entries are given below.

```
noun(_, n([blade,[turbine,rotor,range(whole-part)]])).

noun(_,n([ignition,[[domain(obj),unit],[domain(whole-part),
lead],[domain(for),system]]])).

noun(_,n([lead,[[domain(obj),end],[ignition,range(whole-par
t)],[high,tension,range(use)]]])).

adj(_,a([special,[domain(mod),tool,spanner]])).
```

After dictionary look-up, which works similarly for LParserSub and MParserSub as it does for LParser and MParser, the already predefined complex nouns, adjectives, nouns and the other grammatical constituents are placed back into the input string.

The next stage of the preprocessing of complex nouns is the detection of groups of nouns and adjectives that are not predefined complex nouns. Once detected the adjectives and nouns that appear together in the input string are grouped.

The final stage of the preprocessing is the creation of the binary relationships, which are checked against already validated predefined relationships. Once the binary relationships are created and validated, they can function as nouns, which can be parsed easily by either MParserSub or LParserSub.

7.4 MParserSub - a Marcus Type Parser for a Sublanguage

As stated previously, MParserSub has been developed from MParser, a prototype Marcus type parser, to process the sublanguage of an aircraft maintenance manual, specifically the noun-phrases of the sublanguage. The reason for developing MParserSub is that the sublanguage grammar of the aircraft maintenance manual is quite different from the grammar used by MParser. It was decided that combining the two grammars would not be feasible.

MParserSub has a Pushdown Stack, a two symbol Lookahead Buffer and utilises a grammar of Production Rules. The stack and buffer function in a identical manner to MParser's stack and buffer. Therefore as with MParser, MParserSub's Pushdown Stack is where grammatical structures are stored and the bottom node of the stack, the Current Active Node, is where grammatical structures are built. The processing of grammatical constituents takes place in the first cell of the two cell Lookahead Buffer, with

MParserSub able to examine the contents of second buffer cell. Interaction between MParserSub's stack and buffer is no different from the interaction between MParser's stack and buffer. The differences between the parsers are concerned with the type of grammatical constituents that they can process and the parsing rules that each parser uses to do the processing. In the next sub-section the grammar of MParserSub is examined. This is followed by a look at changes made to the parsing rules.

7.4.1 MParserSub's Grammar

In this sub-section, MParserSub's grammar is examined. The examination of the grammar begins with an evaluation of the use of the theory of phrase structure for the sublanguage. The evaluation of X bar theory is followed by a look at the grammar rules used by MParserSub.

7.4.1.1 X bar Theory and The Sublanguage

The X bar theory of phrase structure is restrictive, by design. As stated in Chapter 5, a phrase in X bar theory is allowed, at most, three branches. The three branches are specifier, head and complement. The specifier corresponds to determiners, quantifiers and/or adjectives. The head, which is the core of a phrase and its most important constituent, corresponds to a noun in a noun-phrase, verb in a verb-phrase and preposition in a prepositional phrase etc. The complement can be an embedded sentence, or a phrase such as a prepositional phrase or a relative clause.

The noun-phrases from the sublanguage of aircraft maintenance manuals correspond to the X bar theory taking into account the use of binary relationship sets:

1. a suitable blank.
2. the engine in flight position.

3. a continuous flow of inhibitor from the LP fuel return outlet.

4. numbers in a clockwise direction that start from an engine top position.

Some noun-phrases occurring in the Rolls-Royce aircraft maintenance manual do not fall precisely into the specifier, head, complement schema of X Bar theory but remain within the bounds of X bar theory. These noun-phrases are discussed below. The grammar rules representing those type of noun-phrases are the following:

1. NP -> Det, Noun, Num

2. NP -> NP, NP

3. NP -> Det, Noun, Num, Conj, Num

4. NP -> Noun, Ref, Noun, Num

Examples of the noun-phrases that are represented by the above are:

1. a new gasket (6a).

2a. impact damage, cracks or metal deposits.

2b. bolts (6), spring washers (7), and spreader washers (8).

3. the bolts 8 and 9.

4. special tool GU28202 special spanner 1 off.

In example 1, the noun-phrase, a new gasket 6a, ends with a number which, it is assumed, is referring to a diagram. As is shown by the grammar rule definition of example 1, the noun-phrase does not conform to the specifier, head, complement rule of X bar theory. However, the number, 6a, can be regarded as being parenthetical and linked to what has preceded it, thus it does not violate X bar theory.

In examples 2a and 2b, the list of noun-phrases separated by a coordinate structure, impact damage, cracks or metal deposits, is considered a noun-phrase, as is bolts (6), spring washers (7), and spreader washers (8). None of these noun-phrases can be considered the individual head of the structure, but X bar theory as applied to GPSG allows description by multiple heads linked by coordinate structure. There is no hierarchy of structure.

Example 3 is another example of coordination within a noun-phrase, but in this instance separating two numbers. The numbers can be considered as parenthetical and linked to the Head of the phrase.

In example 4, the noun-phrase, special tool GU28202 special spanner 1 off, in order to be descriptive, has a very unusual structure. The noun 'special tool' is the head of the phrase. The reference number 'GU28202' is parenthetical and lined to the head. The noun 'special spanner' is also parenthetical to the Head of the phrase. 1 off is a number defining quantity and acts as a complement to the head noun.

In all instances above, the use of more than one head has occurred where a conjunction has appeared in the noun-phrase. There are other examples in the manual where groups of nouns appear together without being separated by a coordinate structure. In these examples also, there is no hierarchical structure and each noun is taken to be a head, which falls within the bounds of X bar theory.

7.4.1.2 MParserSub's Grammar Rules

MParserSub uses a grammar of production rules invoked by a call to 'grammar_rule' by the parsing rules. The modification of MParser's grammar to the grammar used by MParserSub is related to the type of structures found in the sublanguage. MParser's grammar is based on that developed by Marcus to produce a number of test sentences reflecting a variety of grammatical structures. The sublanguage grammar contains a majority of noun-phrase rules. Several of the noun, adjective determiner, preposition and relative-pronoun and noun-phrases and individual verb, verb-phrase, prepositional-phrase and relative-clause rules found in the sublanguage are in MParser's grammar, but the number of similar rules is not sufficient to justify using the same grammar. Moreover, noun-phrase rules would have to have been added to cope with the differing types of noun-phrases found in the sublanguage and many of the grammar rules in MParser's grammar would have been superfluous to MParserSub's grammar. Thus, MParserSub's grammar comprises some of MParser's original rules and extra rules to deal with the various noun-phrases.

The grammar is, once again, a Prolog database of rules; each rule is made up of a grammatical category and a representation of the Pushdown Stack and Lookahead Buffer. The structure of the MParserSub grammar rule has the same structure as the MParser grammar rule. As with a call to grammar_rule under MParser, a call to grammar_rule under MParserSub invokes the attempted matching of the present state of the stack and buffer with one of the rules in the database. If matching succeeds, grammar_rule calls the rule attach which performs the same function previously described for grammar_rule as called under MParser.

Similarly to the grammar rules used by MParser, the MParserSub's grammar rules are grouped by type - specifier, head or complement. Each group varies in size with the head group being the largest. There are seventeen different rules covering grammatical categories, but there are a number of rules with the same grammatical category because of the different conditions that apply to attaching the contents of the first buffer cell to the Current Active Node. Below examples from each group are given, with all rules being described in Appendix D.

7.4.1.2.1 Examples of Specifier Group

attach_det - This is one of the rules for processing determiners.

```
grammar_rule(attach_det,[[xmax,'+n-v+a-p',[spec_, head,
comp]]|R],
[[[FWord,'+n-v-a-p'],[SWord,'+n-v+a+p']]|Rest],NS, NB):-
attach([[xmax,'+n-v+a-p',[spec_, head,comp]]|Reststack],
[[[FWord,'+n-v-a-p'],[SWord,'+n-v+a+p']]|Rest],NS,NB).
```

attach_adj - This is one of the rules for processing adjectives.

```
grammar_rule(attach_adj,[[xmax,S,'+n-v+a-p',[spec_,head
,comp]]|R],
[[[Dom,[FWord, '+n-v+a+p']],[',']]|Rest],NS, NB):-
attach([[xmax, St,'+n-v+a-p', [spec_, head,comp]]|Reststack],
[[[Dom,[FWord, '+n-v+a+p']],[',']]|Rest],NS,NB).
```

7.4.1.2.2 Examples of Head Group

attach_prep - This is one of the rules for processing prepositions.

```
grammar_rule(attach_prep,[[xmax,S,'-n-v',[spec,head_,comp]]
|R],
[[[FWord, '-n-v'], [W, '+n-v+a-p',SP]]|Rest],NS, NB):-
attach([[xmax,S,'-n-v',[spec, head_,comp]]|Reststack],
[[[FWord, '-n-v'],[W, '+n-v+a-p',SP]]|Rest],NS,NB).
```

attach_noun - This is one of the rules for processing nouns.

```
grammar_rule(attach_noun,[[xmax,S,'+n-v+a-p',[spec,head_,
comp]]|R],
[[[Dom,[FWord, '+n-v+a-p',sg]],Second]|Rest],NS, NB):-
attach([[xmax, St, '+n-v+a-p',[spec, head_, comp]]|R],
[[[Dom,[FWord, '+n-v+a-p',sg]],Second]|Rest],NS, NB).
```

7.4.1.2.3 Examples of Complement Group

attach_num - This is one of the rules for processing numbers.

```
grammar_rule(attach_num,[[xmax,S,'+n-v+a-p',[spec,head,
comp_]]|R],
[[[FWord, '+n-v+a-p',num],Second]|Rest],NS, NB):-
attach([[xmax, St, '+n-v+a-p',[spec,head, comp_]]|Reststack],
[[[FWord, '+n-v+a-p',num],Second]|Rest],NS, NB).
```

attach_pp_object - This is a rule for processing objects of a prepositional phrase.

```
grammar_rule(attach_pp_object,[[xmax,S,'-n-v',
[spec,head,comp_]]|R],
[[[xmax,S1,'+n-v+a-p'],Second]|Rest],NS,NB):-
attach([[xmax,S,'-n-v',[spec, head, comp_]]|Reststack],
[[[xmax,S1,'+n-v+a-p'],Second]|Rest],NS, NB).
```

7.4.2 MParserSub's Parsing Rules for Sublanguage

Having discussed the grammar, the discussion turns to MParserSub's parsing rules: These are similar to the rules used by MParser - they have the same names and perform the same functions. However amendments have been made to allow MParserSub to process the various unconventional noun-phrases that appear in the sublanguage of the aircraft maintenance manual. Below the parsing rules, which have been amended, are discussed.

MParserSub's top level rule *input* performs the same function as when called by MParser by calling *enter, prep* and *parse*. *enter* has been amended to place the complex noun - binary relationships in the buffer. The rule *prep* calls the rules *create_max, perc_features, add_template, add_features* and *pre_test*, which prepare the stack for the first constituent from the buffer. *perc_features* and *add_template* have been amended to deal with the grammatical constituents num and ref. *parse* calls *grammar_rule* and *process*, if *grammar_rule* succeeds. If *grammar_rule* fails *parse* calls *drop*. *process* calls *annotate_node, amend_stack, amend_template, act_create_node* and *pre_test*. The rules *amend_template* and *act_create_node* have been amended to deal with grammatical constituents that did not appear in MParser's grammar rules as has the rule *drop*.

7.4.3 MParserSub's Parses from the Sublanguage

In this section example parses from the sublanguage of the aircraft maintenance manual are given. The following noun-phrases are the examples that have been parsed:

cowl doors.

the engine in flight position.

a new seal ring (3).

the bolts and self-locking nuts that attach the 7th stage connector assembly.

The results of parsing, the above, are the following:

```
| ?- start(NP).
|: cowl doors.

NP=

[xmax,[attach_noun,[[cowl,doors],'+n-v+a-p',pl]],'+n-v+a-p']

| ?- start(NP).
|: the engine in flight position.

NP=[xmax,[[[attach_det,[the,'+n-v-a-p']],
          [attach_noun,[engine,'+n-v+a-p',sg]]],
            [attach_pp,[xmax,[[attach_prep,[in,'-n-v']],
            [attach_pp_object,[xmax,
            [attach_noun,[[flight,position],'+n-v+a-p',sg]],
                 '+n-v+a-p']]],'-n-v']]],'+n-v+a-p']

| ?- start(NP).
|: a new seal ring (3).

NP=[xmax,[[[attach_det,[a,'+n-v-a-p']],
    [attach_noun,[[domain(mod),[new,'+n-v+a+p'],range(mod),
              domain(for),[seal,'+n-v+a-p',sg],
         range(for),[ring,'+n-v+a-p',sg]],'+n-v+a-p',sg]]],
              [attach_num,['(3)','+n-v+a-p',num]]],'+n-v+a-p']


| ?- start(NP).
|: the bolts and self-locking nuts that attach the 7th stage
connector assembly.

NP=[xmax,[[[[[attach_det,[the,'+n-v-a-p']],
        [attach_noun,[bolts,'+n-v+a-p',pl]]],
          [attach_conj,[and,'+n-v+a-p',conj]]],
        [attach_noun,[['self-locking',nuts],'+n-v+a-p',pl]]],
        [attach_relative_clause,
          [xmax,[[attach_rpron,[that,'-n-v+a+p']],
          [attach_vp,
          [xmax,[[attach_verb,[attach,'-n+v',-,tense]],
          [attach_object,
          [xmax,[[attach_det,[the,'+n-v-a-p']],
          [attach_noun,
          [[[domain(ref),[['7th',stage],'+n-v+a-p',sg],
            range(ref),
            domain(obj),[connector,'+n-v+a-p',sg],
              range(obj),
            [assembly,'+n-v+a-p',sg]]],'+n-v+a-p',sg]]],
         '+n-v+a-p']]],'-n+v-a+p']]],'-n-v+a+p']]],'+n-v+a-p']
```

The results of the parses prove that MParserSub can produce output. Each output is a syntactic representation of the phrase being parsed. Each output is considered correct with respect to MParserSub's grammar, as the grammar has been defined by the author of the research. It has been assumed that each input phrase is correct as ill-defined input has not been considered within this research. With respect to the grammar and the parser, the output can be considered a true syntactic representation of the input. The results of MParserSub's parses in terms of Machine Translation and in comparison to LParserSub are discussed in sub-section 7.6.

7.4.4 MParserSub and Ambiguity

As stated above in relation to ambiguity in the Aircraft Maintenance Manual, there are relatively few examples of ambiguous structures in the manual. These examples contain part of speech ambiguities which MParserSub can disambiguate, i.e. both 'lock' and 'access' can be disambiguated as verbs or nouns in whatever structure they appear. If, for example, 'lock' was the word being processed, it would first of all be established whether it was part of a complex noun using the method discussed above. On being established that it was part of a complex nominal 'lock' would be disambiguated as a noun belonging to a complex nominal structure. If it was not proven to be part of a complex nominal structure, disambiguation would take place according to methods explained in Chapter 5, with regard to MParser. By having the knowledge of categories already processed and facility to lookahead to the next category, it would be possible to disambiguate between 'lock' as a noun or verb.

It was explained that not many ambiguities appeared in the sample of Aircraft Maintenance Manual used in this research. However all types of ambiguity that MParser can process, as discussed in Chapter 5, MParserSub could also process if its grammar was amended accordingly. As for the type of ambiguity that MParser cannot process, namely constituent questions with unbounded dependencies, it is agreed, as discussed above, that this type structure would not appear in Maintenance Manuals.

7.5 LParserSub - an LR Type Parser for a Sublanguage

LParserSub, like MParserSub, has been developed from a prototype; the prototype being LParser, an LR type parser. LParserSub has been developed to parse the noun-phrases of the Rolls-Royce aircraft maintenance manual extract for the same reasons as given for developing MParserSub - the grammars of LParserSub and LParser are significantly different and it does not seem feasible to combine them.

LParserSub has a stack, input string buffer and a parse table. The parse table has been constructed in exactly the same manner as LParser's parse table, using the same algorithms. LParserSub's parse table has 246 states and as with LParser more than one state can be represented by a state number, e.g., there are eleven states numbered as state 3. The parse table holds the three commands shift, reduce and accept, which manipulate the stack and buffer.

As with LParser, modifications have to be made to LParserSub to allow it to cope with the ambiguities of natural language. Once again shift-reduce conflicts appear in the parse table which are dealt with by shifting when a shift-reduce conflict occurs. There do not seem to be any reduce-reduce conflicts in the parse table, probably due to the fact that there is not a large variety of phrase types in the grammar. The shift-reduce conflicts in the parse table occur because of the variety of noun-phrases, which can have between one and six constituents. At several stages of parsing a noun-phrase, the parser would be confronted with shifting or reducing, but it always obeys the shift command.

Part of speech ambiguity occurs in the aircraft maintenance manual, which has to be dealt with by the parsing process. There are only two words that pose a problem - access and crack which can both be verbs or a nouns. However, in this situation, there is no need to extend the lookahead, as the parse table will guide the choice of verb or noun. In the next sub-section LParserSub's grammar is examined. This followed by a look at the parsing rules and any changes that have been made. Finally the results of LParserSub parsing the same example noun-phrases, as above, are provided.

7.5.1 LParserSub's Grammar

In this sub-section LParserSub's grammar is examined. As with LParser, the grammar is notated in the form of a DCG; this does not provide any difficulties as regards grammatical theory. Yet again it is only the descriptive properties of the DCG that are utilised.

7.5.1.1 LParserSub's Grammar Rules

There are fifty-six rules in LParserSub's grammar. The majority of these rules represent noun-phrases, with the remaining rules covering phrases that can be components of the noun-phrase, such as prepositional phrases, relative clauses, verb-phrases and auxiliaries. The modification of LParser's grammar to the grammar used by LParserSub is related to the type of structures found in the sublanguage. LParser's grammar is based on MParser's, although notated in the form of a DCG. The sublanguage grammar contains a majority of noun-phrase rules. Several of the noun, adjective, determiner, preposition and relative-pronoun and noun-phrase and individual verb, verb-phrase, prepositional-phrase and relative-clause rules found in the sublanguage are in LParser's grammar, but the number of similar rules is not sufficient to justify using the same grammar. Moreover, noun-phrase rules would have to have been added to cope with the differing types of noun-phrases found in the sublanguage and many of the grammar rules in LParser's grammar would have been superfluous to LParserSub's grammar. Thus, LParserSub's grammar, similar to the link between MParser and MParserSub, comprises some of LParser's original rules and extra rules to deal with the various noun-phrases. In the sub-sections below, examples of the grammar rules are given. All LParserSub's grammar rules appear in Appendix D.

7.5.1.1.1 Grammar Rules for Noun-phrases

There are forty-two grammar rules that represent noun-phrases. Examples of the noun-phrases are the following:

```
(noun_phrase(N,np(Det,Noun))-->det(N,Det),noun(N,Noun)).

(noun_phrase(N,np(Ref,Noun))-->ref(N,Ref),noun(N,Noun)).

(noun_phrase(N,np(Noun,Ref,Noun,Num))-->noun(N,Noun),
                 ref(N,Ref),noun(N,Noun),num(N,Num)).

(noun_phrase(N,np(Noun,Noun,Noun,Conj,Noun))-->noun(N,Noun),
        noun(N,Noun),noun(N,Noun),conj(N,Conj),noun(N,Noun)).

(noun_phrase(N,np(NP,Adj,Conj,Adj))-->noun_phrase(N,NP),
              adj(N,Adj),conj(N,Conj),adj(N,Adj)).

(noun_phrase(N,np(Adj,Conj,Noun,Noun))-->adj(N,Adj),conj(N,
              Conj),noun(N,Noun),noun(N,Noun)).
```

7.5.1.1.2 Grammar Rules for Other Phrases

There are fourteen grammar rules that represent the remaining phrases in the grammar. Examples of these phrases are the following:

```
(prep_phrase(N,pp(Prep,Prep,NP))-->prep(N,Prep),prep(N,Prep),
                    noun_phrase(N,NP)).

(r_clause(N,r_c(Rpron,NP))-->rpron(N,Rpron),
                    verb_phrase(N,NP)).

(verb_phrase(N,vp(V,NP,SC))-->verb(N,V),prep_phrase(N,NP),
                    sub_clause(N,SC)).

(sub_clause(N,s_c(Sub,NP,VP,Sub,Adv,VP))-->sub(N,Sub),
    noun_phrase(N,NP),verb_phrase(N,VP),
    sub(N,Sub),adv(N,Adv),verb_phrase(N,VP)).
```

7.5.2 LParserSub's Parsing Rules for Sublanguage

LParserSub's parsing rules are similar to the rules used by LParser. They have the same names and performs the same functions. Amendments have been made to allow LParserSub to parse the variety of noun-phrases that appear in the sublanguage of the aircraft maintenance manual. Below the parsing rules, which have been amended, are discussed.

LParserSub's top level rule *parse* performs a similar function as when called by LParser by calling *match_state*, but also *sing_plural*. *match_state* as called by LParserSub, has different versions to deal with shifting, reducing and accepting. The rules that *match_state* calls are *state, check_categories, shift, merge, reduce, merge1, reduce1, checking* and *match_state*. The rule *state*, in this instance, represent states in LParserSub's parse table. *check_categories* has been amended, since it does not have to process as many part of speech ambiguities. *merge, reduce, merge1* and *reduce1* have been amended to process noun-phrase constituents on the stack. The rule *checking* has been amended to do semantic checks on the constituents of certain noun-phrases, when *state* does not execute. *sing_plural* assigns the grammatical number of the noun-phrase after the parsing process is complete.

7.5.3 LParserSub's Parses from the Sublanguage

In this section example parses from the sublanguage of the aircraft maintenance manual are given. The measurements of time and speed have been calculated in exactly the same way as for MParserSub as described in section 7.4.3. The example parses are those of the noun-phrases in section 7.4.3.

```
| ?- run(S,B).
|: cowl doors.

S = [noun_phrase(pl,np(n([cowl,doors],pl)))],
B = []

| ?- run(S,B).
|: the engine in flight position.

S = [noun_phrase(sg,np(np(d(the),n(engine,sg)),
            pp(p(in),np(n([flight,position],sg)))))],
B = []

| ?- run(S,B).
|: a new seal ring (3).

S=
[noun_phrase(sg,np(d(a),n([domain(mod),adj(_12673,a([new])),
          range(mod),domain(for),noun(_12813,n([seal],sg)),
            range(for),noun(_13606,n([ring],sg))]),
          num('(3)')))],
B = []

| ?- run(S,B).
|: the bolts and self-locking nuts that attach the 7th stage
connector assembly.

S = [noun_phrase(pl,np(np(d(the),n(bolts,pl),
        conj(and),n(['self-locking',nuts],pl)),
        r_c(r_p(that),vp(v(attach),
         np(d(the),
         n([domain(ref),noun(_29655,n(['7th',stage],sg)),
             range(ref),
             domain(obj),noun(_30061,n([connector],sg)),
            range(obj),noun(_30632,n([assembly],sg))])))))))],

B = []
```

The results of the parses also prove that LParserSub can produce output. Once more each output is a syntactic representation of the phrase being parsed. with each particular output being considered correct with respect to MParserSub's grammar, as the grammar has been defined by the author of the research. It has been assumed that each input phrase is correct; ill-defined input has not been considered within this research. With respect to the grammar and the parser, the output can be considered a true syntactic representation of the input. The results of LParserSub's parses in terms of Machine Translation are discussed in sub-section 7.6.

7.5.4 LParserSub and Ambiguity

As with MParserSub, LParserSub can process the ambiguities found in the Aircraft Maintenance Manual. The ambiguities are part of speech ambiguities, both 'lock' and 'access' can be disambiguated by LParserSub as verbs or nouns in whatever structure they appear. If, for example, 'access' was the word being processed, it would first of all be established whether it was part of a complex noun using the method discussed above. On being established that it was part of a complex nominal 'access' would be disambiguated as a noun belonging to a complex nominal structure. If 'lock' was not proven to be part of a complex nominal structure, disambiguation would take place according to methods explained in Chapter 6, with regard to LParser. By having the knowledge of categories already processed and the facility to lookahead to the next category, it would be possible to disambiguate between 'access' as a noun or verb.

As discussed in Chapter 6, all types of ambiguity that LParser can process LParserSub could also process if its grammar was amended accordingly. Similarly to MParser, the type of ambiguity that LParser cannot process, namely constituent questions with unbounded dependencies, would not appear in Maintenance Manuals, thus would not cause problems for LParserSub.

7.6 The Use of MParserSub and LParserSub in Machine Translation

This section concentrates on evaluating the parsers that have been discussed in this chapter; it could be considered as evaluating an experiment. On a superficial level, it could be considered that evaluation of the experiment may have been determining whether the parsers could parse noun-phrases from the Rolls-Royce Aircraft Maintenance Manual. However, the evaluation of the experiment may more correctly be described as determining the possibility of building a deterministic parsing system that is useful for MT. This can be assumed to be the "thesis".

In attempting to prove that the experiment has been successful, it would be useful to consider it from the viewpoint of "equipment" and "data"; "equipment" can be regarded as an MT system and "data" the Rolls-Royce Maintenance Manual. For an ideal experiment, a complete MT system with users to try the results on would be required. As a complete MT system has not been developed, do the parsers developed allow a reasonable judgement to be made of the "thesis"?

If a MT system had been developed, for example Metal, the parser developed would have been constrained to producing output of a particular form, using a vocabulary of category labels specified, in the main, by Metal. The literature describing transfer indirect MT systems, such as Metal, of which the parsers developed for this research would be part, discuss the fact that most outputs from the Analysis Stage are of syntax tree type, sometimes with semantic information attached that does not alter the tree topology. As discussed in Chapter 1, the syntax tree, the result of the parsing of source language is often referred to as the intermediate representation in modular Machine Translation systems. The output in the form of a syntactic representation produced from MParserSub and LParserSub can be considered to be an intermediate representation (IR). The intermediate representation, i.e., the syntactic representation is an abstract structure. As Johnson (1983) states

> " The parser should be capable ideally of yielding for a given source language text a single IR which is unambiguous up to the choice of lexical items in the target language".

This is a statement which can be easily applied to MParserSub and LParserSub. Both parsers are deterministic parsers which produce a single output for a given input. Neither parser produces alternative outputs for the input that has been processed. It is suggested, therefore, that a full practical MT system not being developed is not detrimental to the "thesis", as full indirect MT systems are modular with each stage being considered individually; in this instance, it is the analysis stage. The parsers developed for this research can compare with parsers from the analysis stage of practical full transfer indirect MT systems in that they produce output of similar type, i.e., syntax trees.

Having discussed the "equipment", the "data" used in the experiment is now examined. The "data" used is an extract of a Rolls Royce maintenance manual which can be considered as suitable as it is a sublanguage text. Sublanguage texts have been tried and tested in systems such as TAUM-Aviation, Logos and BSO. Thus, it may be reasonably argued that a representative corpus has been used in building both parsing systems. Johnson (1983) states that a good MT parser will produce only one result to avoid ambiguities occurring in the translation process that were specific to the source language. Both MParserSub and LParserSub perform this task, aided by the source language being a sublanguage, a restricted language with less ambiguous structures and vocabulary.

In concluding the discussion on the experiment in broad terms, it can be stated that it uses an input and produces an output which can both be described as reasonable. Concentrating on specifics of the parsing systems, they can be discussed in terms of the changes made to the non-sublanguage versions to produce the sublanguage versions and by comparing both sublanguage parsers.

The discussion begins with grammar. The grammar used in the MParserSub parsing system differs from that used in MParser; there are amendments to the specifier, head and complement rule of X bar theory and new grammar rules are added. The amendments made to the specifier, head and complement rule relate to the fact that several noun-phrases connected with coordinate structures within the grammar are dominated by more than one head. These type of grammatical structures were not catered for by MParser's grammar. However, the use of more than one head within a phrase does not go beyond the bounds of X bar theory, as discussed in the literature with regard to GPSG (Gazdar *et al*, 1985), which does not rule out its application within Transformational type grammars.

New rules were added to the original grammar designed for MParser with some rules superfluous to the sublanguage deleted. The new rules were specifically noun-phrase rules to deal with the various types of noun-phrases that occurred within the sublanguage of the Rolls-Royce Aircraft Maintenance Manual. As discussed above, there is relatively little ambiguity within the Rolls-Royce Aircraft Maintenance Manual. The changes to the X bar theory and the addition of new rules do not make the grammar any more ambiguous.

The changes made to produce LParserSub's grammar from the original LParser are the addition of rules to cope with the various noun-phrases that occur within the sublanguage. As with MParser, several rules were deleted from LParser as being surplus to requirements. The addition of the new noun-phrase rules were sublanguage specific. With regard to ambiguity within the grammar, the changes made do not make the grammar any more ambiguous.

The changes to both non-sublanguage grammars were made to accommodate the needs of the sublanguage of the Rolls-Royce Aircraft Maintenance Manual. Obviously, if the original grammars had been used they would not have been adequate for the processing of the sublanguage. The removal of superfluous rules helps the understanding of the sublanguage grammars.

The dictionaries of both sublanguage parsing systems were altered radically from the original non-sublanguage versions. Obviously, the text of the Aircraft Maintenance Manuals is much more technical than that used in the non-sublanguage versions. A major change in the type of words held within the dictionaries is the compound nouns. Although some compound nouns are held as compound nouns, other are processed by means of the binary relationship functions. It should be noted that in a full MT system processing of compound nouns in the Transfer Stage would be simpler as compound nouns would be held as compounds within a SL-TL transfer dictionary to get an idiomatic translation. However, this does not detract from the method used during processing in the analysis stage.

The parser software for MParser has not been significantly changed to create MParserSub. The only changes made to create MParserSub were to cope with grammatical constituents held within the stack that MParser did not process. The changes made do not effect processing in any way; thus there is no difference in the parsing process of MParser and MParserSub. Similarly, the parser software for LParser has not been significantly changed to create LParserSub. The changes made are to cope with the various noun-phrases that do not occur with LParser's grammar but do appear in LParserSub's grammar. LParser's method of processing is not different to that of LParserSub.

Finally, a comparison of the parsers, MParserSub and LParserSub is made. The comparison is made on the basis of statistical information produced during the parsing of noun-phrases. The information includes parse times, memory usage and total run times. All parsers were developed in Quintus Prolog, running on a Sun SPARCstation linked to a network file server. As this is a practical experiment, theoretical considerations have not been made regarding times and memory usage, i.e., the statistical information on the parsers has not been gathered in total isolation from everything else that may be running on the network.

Speed and memory usage for each parse by each parser is measured by invoking *statistics* within the program implementation of the parser. The statistics shown relate to memory usage, run time and garbage collection when sentences have been parsed within Quintus Prolog, the language used to implement all the parsers. Memory within Quintus Prolog contains program space, global space which contains global stack and trail, and local stack. The program space contains compiled and interpreted code, recorded items and atoms. The local stack contains all the control information and the variable bindings needed in a Prolog execution. The global stack contains all the data structures constructed in the execution of the program and the trail contains references to all the variables that need to be reset when backtracking occurs. The local stack contains all control information and variable bindings needed in global execution. The memory shown as 'in use' is the sum of the spaces for program, global and local areas.

'parse took' reflects the cpu time since the last call to *statistics*, thus it measures parse time only. 'runtime' measures all procedures within the program. Garbage collection reclaims any inaccessible global stack space reducing the need for global stack expansion. Comparing MParserSub and LParserSub by examining the results of parsing, it is clear that both parsers can parse all noun-phrases from the manual. The example parses above in sections 7.4.3 and 7.5.3 are a selection of the different types of noun-phrases that can be parsed; there are more example parses in Appendix D. Appendix D also contains all the noun-phrases from the Rolls-Royce Aircraft Maintenance Manual. The fact that both parsers can parse all noun-phrases is proof of their effectiveness and goes some way to proving suitability for processing sublanguage.

In the example parses in sections 7.4.3 and 7.5.3 no statistics are given. All statistics regarding the parses generated from MParserSub and LParserSub are presented in table 7.1 and table 7.2, respectively. The statistics presented are the averages gained from 100 sequential repetitions of a parse of each phrase.

The noun-phrases parsed were the following:

1. cowl doors.
2. the engine in flight position.
3. a new seal ring (3).
4. the bolts and self-locking nuts that attach the 7th stage connector assembly.

Noun-phrase	Parse Time (sec.)	Memory (Total) (Bytes)	Program Space (Bytes)	Global Space (Bytes)	Run Time (Sec.)	Local Space (Bytes)
1	0.016	338,744	207,680	63,444	0.316	65,508
2	0.033	338,736	207,672	63,444	0.316	65,508
3	0.018	338,736	207,672	63,444	0.366	65,508
4	0.024	338,752	207,688	63,444	0.438	65,508

Table 7.1 MParserSub - Parser Statistics

Noun-phrase	Parse Time (sec.)	Memory (Total) (Bytes)	Program Space (Bytes)	Global Space (Bytes)	Local Space (Bytes)	Run Time (Sec.)
1	0.017	368,288	237,772	63,444	65,508	0.216
2	0.033	368,836	237,764	63,444	65,508	0.933
3	0.019	369,016	237,952	63,444	65,508	0.950
4	0.026	369,044	237,980	63,444	65,508	2.350

Table 7.2 LParserSub - Parser Statistics

The above statistics are now discussed. All the Prolog code that makes up MParserSub and LParserSub is compiled before the parsing process commences. The results show that parse times from LParserSub and MParserSub are more or less similar. On comparing the memory usage of each parser, it can be seen that LParserSub, on average, uses 30000 bytes more memory than MParserSub. The extra memory used by LParserSub is taken up by program space. There is no difference between the Global Stack space and Local Stack space used by each parser. The differences in full run times are explained by the fact that the times for LParserSub include reading in the data, while those for MParserSub do not. Thus it can be said that in practical terms there is no real difference in speed of the parsers, but, LParserSub does use more program space (memory) while producing similar speeds. For the examples used, in terms of parse times, LParserSub would seem to be slightly more efficient due to the fact that although using more memory it can produce similar parse times to MParserSub. Appendix D contains more example parses which produce similar statistical results.

None of the parsers have been compared with Non-deterministic Parsers in terms of structure, speed and memory usage. However, the sublanguage parsers have demonstrated that they can function similarly to parsers that have been built for practical MT systems, which, in the main, have been Non-Deterministic parsers.

In conclusion, the experiment can be considered as having demonstrated the point that it is possible to build a Deterministic Parsing system that is useful for MT. Although, only noun-phrases have been parsed, they did contain a variety of different structures, such as verb-phrases, prepositional phrases and relative clauses. However, it would have been beneficial to test the parsers on more Rolls-Royce text and perhaps other texts. It would also have been beneficial to plug it into a full MT system.

CHAPTER 8

CONCLUSION AND FUTURE WORK

8.1 Conclusion

In this section of the final chapter, the aim is to give an overall conclusion to the work. Evaluation of the project has been discussed in the previous chapter, the findings of which will be taken into consideration in this section. The evaluation of the project has, in the main, demonstrated that the aims of the project have been met, i.e., deterministic parsers, developed from prototypes, have been built that can parse sublanguage as part of a MT system.

To recap, the evaluation shows that the sublanguage parsers, MParserSub and LParserSub, serve as suitable for MT as they meet criteria for MT parsers in that they use an input and produce an output that is reasonable in MT terms. Having evaluated the sublanguage parsers in terms of MT, the parsers have been compared to the prototypes from which they were developed.

Further, the changes to the non-sublanguage parsing systems to produce the sublanguage versions were related to the needs of the sublanguage. Both grammars used by LParserSub and MParserSub were amended from the originals by the addition of more noun-phrase rules, with several rules that were not relevant being deleted. Changes made to the dictionaries of both original prototypes to produce the dictionaries for the sublanguage parsers related to the addition of vocabulary related to Rolls-Royce Aircraft Maintenance Manuals. The dictionaries also had to store compound nouns and information regarding compound nouns. These dictionaries can be considered source language dictionaries. In a full MT system the Transfer stage of the system would have a separate dictionary which would handle compound nouns differently because of their structure in the target language. The actual parsers developed for processing the sublanguage were changed from the original prototypes only to cope with the structures of the sublanguage.

Finally, the evaluation considered a comparison of both sublanguage parsers. As discussed in the previous chapter this includeded comparing speeds and memory usage. These comparisons showed differences between both parsers in both speed and memory. The more significant difference is the memory usage with LParserSub using 30000 bytes more in memory but still producing comparable parse times. this would indicate that LParserSub is slightly more efficient than MParserSub. The evaluation concluded that the aims had been demonstrated satisfactorily. Although not compared in speed or memory usage with non-deterministic parsers, the deterministic parsers developed demonstrated that they can perform the same function as non-deterministic parser in MT systems.

8.2 Future and Further Work

As a way of concluding the thesis, future and further work will be discussed. This discussion will examine aspects of the work which, it is considered, could be extended.

With regard to future work, it is believed that the research, in general, on the use of Deterministic Parsers on Sublanguage for Machine Translation cannot be expanded. Each aspect of the research has been considered and developed, and it is believed that work that needed to be done has been done. In short, it is considered that no other PhD research topic could be developed directly from this research. However, this does not mean that further work on specific areas within the research topic could not be pursued. Below aspects of the further work that could be undertaken are discussed.

Firstly, the coverage of both parsers could be extended to include the processing of all of the text in the sample Rolls-Royce aircraft maintenance manual. The coverage could be further extended to include a complete Rolls-Royce aircraft maintenance manual. Extension to the coverage would mean increasing the size of the grammar. However, extending the coverage to include a complete manual, would test how good a sample the original sample text was. Overall extension would give a better indication of the qualities of the parsers developed.

Secondly, the coverage of MParserSub and LParserSub could be extended to different sublanguages. Examples of sublanguages to which coverage could be extended are medical records, stock market reports and car hire agency transactions. Extension of coverage is considered viable, since there is interest in using sublanguages within Machine Translation systems, especially, sublanguages of a scientific and technical nature. This could give some further insight into the degree of syntactic limitation of various sublanguages.

Thirdly, the research project could be extended to a full Machine Translation system. This would involve building transfer and generation components. The transfer component would deal immediately with the results of parsing form the analysis stage, with the generation component processing the results for the transfer stage. It is considered that this would be an interesting project which could show the practical value of this research.

Finally, an environment could be developed to build a grammar and a grammar and parse table for MParserSub and LParserSub, respectively. This environment would prove to be aid to building both parsers in that the tedious work could be removed in the building of grammars and a parse table from scratch. An environment would make the whole process simpler and quicker.

8.3 Concluding Remarks

On reaching the end of this thesis, a few final remarks should be made about the research. The research must be considered a success, in that Deterministic Parsers have been built that can parse the quite complex sublanguage of an aircraft maintenance manual. Although, the parsers have been only applied to the noun-phrases of the sublanguage, these noun-phrases include many other phrase-types in their composition. The success of these Deterministic Parsers for sublanguage proves the prototypes, from which they were developed, to be good models. The achievement sought has been attained.

REFERENCES

Aho, A.V.; Johnson, S.C. (1974) LR Parsing. *Computing Surveys* 6(2) pp 99-124.

Aho, A.V.; Ullman, J.D. (1972) *The theory of parsing, translation and compiling, vol 1: parsing*. Prentice-Hall.

Archibald, J.; Hancox, P. (1988) A survey of deterministic parsers. *In: Applied Informatics: proceedings of the IASTED International Symposium, Grindelwald, Switzerland, 16-18 February 1988*. ACTA Press. pp 143-146.

Bachenko, J; Hindle, D; Fitzpatrick, E. (1983) Constraining a deterministic parser. *In:Proceedings of the National Conference on Artificial Intelligence, Washington, DC, 22-26 August 1983*. AAAI. pp 8-11.

Bennett, W.S.; Slocum, J. (1985) The LRC machine translation system. *Computational Linguistics* 11(2-3) pp 11-119.

Berwick, R.C. (1982) *Locality principles and the acquisition of syntactic knowledge*. Unpublished PhD thesis, Department of Electrical Engineering and Computer Science, MIT.

Berwick, R.C. (1985) *The acquisition of syntactic knowledge*. MIT Press.

Briscoe, E.J. (1987) *Modelling human speech comprehension: a computational approach*. Ellis Horwood.

Carter, A.W.; Freiling, M.J. (1984) Simplifying Deterministic Parsing. *In:Proceedings of the 10th International Conference on Computational Linguistics and the 22nd Annual Meeting of the Association for Computational Linguistics, Stanford, 2-6 July 1984*. ACL. pp 239-242.

Chandioux, J. (1976) METEO: An operational system for the translation of public weather forecasts. *In*:Hays, D.G.; Mathias, J. eds Summary proceedings of the FBIS Seminar on Machine Translation, Rosslyn, VA, 8-9 March 1976. *American Journal for Computational Linguistics*. Microfiche 46 pp 27-36.

Charniak, E. (1983) A parser with something for everyone. *In:King, M. ed. Parsing natural language: proceedings of the Second Lugano Tutorial, Lugano, 6-11 July 1983*. Academic Press. pp 117-149.

Chomsky, N. (1957) *Syntactic structures*. Mouton.

Chomsky, N. (1965) *Aspects of the theory of syntax*. MIT Press.

Chomsky, N. (1972) Some empirical issues on the theory of Transformational Grammar. In:Peters, S. ed. *Goals of linguistic theory*. Prentice-Hall. pp 63-130.

Church, K. (1980) *On memory limitations in natural language processing*. Unpublished

Masters thesis, Laboratory for Computer Science, MIT.

Colmerauer, A. (1970) Les systemes Q *Publication Interne* nr. 43, TAUM, Universite de Montreal.

Crystal, D. (1985) *A Dictionary of Linguistics and Phonetics*. Basil Blackwell Ltd.

De Roeck, A. (1983) An underview of parsing. *In*:King, M. ed. *Parsing natural language: proceedings of the Second Lugano Tutorial, Lugano, 6-11 July 1983*. Academic Press. pp 3-17.

Downing, P; (1977) On the creation and use of English compound nouns. *Language* 53(4) pp 810-842.

Earley, J. (1970) An efficient, context-free parsing algorithm. *Communications of the ACM* 6(8) pp 451-455.

Fischer, C.N.; Leblanc, R.J. (1988) *Crafting a compiler*. Benjamin-Cumming.

Fitzpatrick, E; Bachenko, J; Hindle, D. (1986) The status of telegraphic sublanguages. *In*:Grishman, R.; Kittredge, R. *eds. Analyzing language in restricted domains: sublanguage description and processing*. Lawrence Erlbaum. pp 39-53.

Gazdar, G.; Klein, E.; Pullum, G.; Sag, I. (1985) *Generalized Phrase Structure Grammar*. Blackwell.

Hindle, D. (1983) Deterministic parsing of syntactic non-fluencies. *In*: *Proceedings of the 21st Annual Meeting of the Association for Computational Linguistics, Cambridge, Mass, 15-17 June 1983*. ACL. pp 123-128.

Hutchins, W.J. (1982) The evolution of machine translation systems. *In*:Lawson, V. *ed. Practical experience of machine translation: proceedings of a conference, London, 5-6 November 1981*. North-Holland. pp 21-37.

Hutchins, W.J. (1986) *Machine translation: past, present, future*. Ellis Horwood.

Isabelle, P; Bourbeau, L. (1985) TAUM-Aviation: its technical features and some experimental results. *Computational Linguistics* 11(1) pp 18-27.

Jackendoff, R. (1977) *X bar syntax: a study of phrase structure*. MIT Press.

Johnson, R (1983) Parsing - an MT Perspective. *In:*
Sparck-Jones, K.; Wilks, Y., *eds. Automatic Natural Language Parsing*. Ellis Horwood.

Kay, M. (1973) The MIND system. In:Rustin, R. *ed. Natural language processing: Courant Computer Science Symposium 8, 20-21 December 1971*. Algorithmics Press. pp 155-158.

King, M. (1982) Eurotra: an attempt to achieve multilingual MT. *In*:Lawson, V. *ed. Practical experience of machine translation: proceedings of a conference, London, 5-6*

November 1981. North-Holland. pp 139-147.

Kittredge, R.; Lehrberger, J. (1982) Introduction. *In*:Kittredge, R.; Lehrberger, J. *eds. Sublanguage: studies of language in restricted semantic domains*. de Gruyter. pp 1-7.

Kittredge, R. (1982) Variation and homogeneity of sublanguages. *In*:Kittredge, R.; Lehrberger, J. *eds. Sublanguage: studies of language in restricted semantic domains*. de Gruyter. pp 107-137.

Kittredge, R. (1987) The significance of sublanguage for automatic translation. In:Nirenburg, S. ed. Machine translation: theoretical and methodological issues. Cambridge University Press. pp 59-67.

Kwasny, S.C.; Sondheimer, N.K. (1981) Relaxation techniques for parsing grammatically ill-formed input in natural language understanding systems. American Journal of Computational Linguistics 7(2) pp 99-109.

Lehrberger, J. (1982) Automatic translation and the concept of sublanguage. In:Kittredge, R.; Lehrberger, J. *eds. Sublanguage: studies of language in restricted semantic domains*. de Gruyter. pp 81-106.

Lehrberger, J. (1986) Sublanguage analysis *In*:Grishman, R.; Kittredge, R. *eds. Analyzing language in restricted domains: sublanguage description and processing*. Lawrence Erlbaum. pp 19-39.

Lesmo, L; Magnini, D; Torasso, P. (1981) A deterministic analyzer for interpretation of natural language commands. *In: Proceedings of the 7th International Joint Conference on Artificial Intelligence, Vancouver, 24-28 August 1981*. IJCAI. pp 440-442.

Levi, J.N. (1978) *The syntax and semantics of complex nominals*. Academic Press.

Maas, H.D. (1984) The MT system SUSY. *Presented at:ISSCO tutorial on Machine Translation, Lugano, 2-6 April 1984*.

Marcus, M.P. (1980) *A theory of syntactic recognition for natural language*. MIT Press.

Marcus, M.P. (1985) Deterministic parsing and description theory. *In*:Whitelock, P. et al. *eds. Alvey/ICL Workshop on Linguistic Theory and Computer Applications, Manchester, September 1985*. Centre for Computational Linguistics, UMIST. pp 49-75.

Matsumoto, Y. *et al.* (1983) BUP: a bottom-up parser embedded in Prolog. *New Generation Computing* 1(1) pp145-158

Milne, R.W. (1982) Predicting garden path sentences. *Cognitive Science* 6(4) pp 349-373.

Milne, R. (1983) *Resolving lexical ambiguity in a deterministic parser*. Unpublished PhD thesis, Department of Artificial Intelligence, University of Edinburgh.

Milne, R. (1986) Resolving lexical ambiguity in a deterministic parser. *Computational*

Linguistics 12(1) pp 1-12.

Nozohoor-Farshi, R. (1986) On formalization of the Marcus parser. *In:Proceedings of the 11th International Conference on Computational Linguistics and the 24th Annual Meeting of the Association for Computational Linguistics, Bonn, 25-29 August 1986.* ACL. pp 533-535.

Pereira, F.C.N.; Warren, D.H.D. (1980) Definite Clause Grammars for language analysis: a survey of the formalism and a comparison with Augmented Transition Networks. *Artificial Intelligence* 13(3) pp 231-278.

Pereira, F.C.N. (1985) A new characterization of attachment preferences. *In*:Dowty, D.R.; Kartunnen, L; Zwicky, A.M. *eds. Natural language parsing: psychological, computational and theoretical perspectives*. Cambridge University Press. pp 307-319.

Radford, A. (1981) *Transformational Syntax: a Student's Guide to Extended Standard Theory*. Cambridge University Press

Sabah, G; Mohamed, R. (1983) A deterministic syntactic-semantic parser. *In:Proceedings of the 8th International Joint Conference on Artificial Intelligence, Karlsruhe, 8-12 August 1983*. IJCAI. pp 707-703.

Sager, N. (1986) Sublanguage: linguistic phenomenon, computational tool. *In*:Grishman, R.; Kittredge, R. *eds. Analyzing language in restricted domains: sublanguage description and processing*. Lawrence Erlbaum. pp 1-19.

Sells, P. (1985) *Lectures on contemporary syntactic theories*. CSLI.

Shieber, S. (1983) Sentence disambiguation by a shift-reduce parsing technique. *In: Proceedings of the 21st Annual Meeting of the Association for Computational Linguistics, Cambridge, Mass, 15-17 June 1983*. ACL. pp 113-118.

Shieber, S. (1985) An Introduction to Unification-Based Approaches to Grammar. *Presented as a Tutorial Session at the 23rd Annual Meeting of the Association for Computational Linguistics, University of Chicago, July 8, 1985*. ACL.

Shieber, S. (1985a) Using Restriction to Extend Parsing Algorithms for Complex-Feature-Based Formalisms. *In: Proceedings of the 23rd Annual Meeting of the Association for Computational Linguistics, University of Chicago, July 8-11, 1985*. ACL. pp 145-152.

Shipman, D.W; Marcus, M.P. (1979) Towards minimal data structures for deterministic parsing. *In:Proceedings of the 6th International Joint Conference on Artificial Intelligence, Tokyo, 20-23 August 1979*. IJCAI. pp 815-817.

Slocum, J. (1981) A Practical Comparison of Parsing Strategies. *In: Proceedings of the 19th Annual Meeting of the Association for Computational Linguistics, Stanford University, 29June - 1* July. *ACL. pp 1-6.*

Slocum, J. (1984) Metal: The LRC machine translation system. *Presented at:ISSCO* *tutorial on Machine Translation, Lugano, 2-6 April 1984.*

Slocum, J. (1985) A survey of machine translation: its history, current status and future prospects. *Computational Linguistics* 11(1) pp 1-17.

Slocum, J. (1988) *Machine translation systems.* Cambridge University Press.

Tomita, M. (1986) *Efficient parsing for natural language: a fast algorithm for practical systems.* Kluwer Academic.

Tucker, A. (1987) Current strategies in machine translation research and development. *In*:Nirenburg, S. *ed. Machine translation: theoretical and methodological issues.* Cambridge University Press. pp 22-41.

Varile, G. (1983) Charts: a data structure for parsing. *In*:King, M. *ed. Parsing natural language: proceedings of the Second Lugano Tutorial, Lugano, 6-11 July 1983.* Academic Press. pp 73-87.

Winograd, T. (1972) *Understanding natural language.* Edinburgh University Press.

Winograd, T. (1983) *Language as a cognitive process, vol 1: syntax.* Addison-Wesley.

Woods, W.A. (1970) Transition network grammars for natural language analysis. *Communications of the ACM* 13(10) pp 591-606.

APPENDIX A

This appendix contains the code for MParser and its grammar rules. The code below represent the parsing rules.

```
/*Top level rule which invokes the processing of the sentence, calling rules to prepare the
sentence for parsing, initiating the parsing mechanism and returning the result of the parse*/

start(S):-readin(S).

input(Sentence,S):- statistics(runtime,_),
enter(Sentence, Buffer, Stack),
prep(Buffer, Stack, Newbuffer, Newstack),!,
parse(Newstack,Newbuffer,S),
statistics(runtime,[_,T]),
format('parse took ~3d sec.~n',[T]).

/*This rule, called by 'input', enters the first two words of the sentence into the two cell
lookahead buffer.*/

enter([First, Second | R], [[First, Second] | R], _S).

/*This rule, called by 'input', prepares the stack for the parsing of the word in the first buffer
cell.*/

prep(Buffer, Stack,Newbuffer, Newstack):-
create_max(xmax, Stack, NewS),
perc_features(Buffer, NewS, NewS1),
add_template(Buffer, _Template, NewS1,NewS2),
add_features(NewS2,_Features,NewS3),
pre_test(Buffer, NewS3, Newbuffer, Newstack).

/*This rule, called by 'prep', 'act_create_node' and 'drop' places the term 'xmax', which signifies
that the phrase is a maximal, in the stack*/

create_max(Maximal, [], [[Maximal]]).

create_max(Maximal,[Ca | Reststack],[[Maximal],Ca | Reststack]).

/*This rule, called by 'prep', 'act_create_node' and 'drop' percolates the features of the word
in the first buffer cell into the stack, except if the first word is a specifier, eg determiner, then
the features of the head of a phrase beginning with a specifier are placed in the stack, eg
noun.*/

perc_features([[[[_word,'-n+v-a+p',_Pn,tense],[_word,'+n-v+a-p',sg]],_] | _],[[Maximal]],[[Maxi
mal,'-n+v-a+p']]).

perc_features([[[[_word,'+n-v+a-p',sg],[_word,'-n+v+a+p']],[be,'-n+v']] | _],[[Maximal] | R],[[Ma
ximal,'-n+v+a+p'] | R]).

perc_features([[[_word, '+n-v-a-p'],_] | _],[[Maximal]],[[Maximal, '+n-v+a-p']]).
perc_features([[[_word, '+n-v+a+p'],_] | _],[[Maximal]],[[Maximal, '+n-v+a-p']]).

perc_features([[[_word,'+n-v+a-p',pn],_] | _],[[Maximal]],
[[Maximal,'+n-v+a-p']]).

perc_features([[[_word,'+n-v+a-p'],_] | _],[[Maximal]],
[[Maximal,'+n-v+a-p']]).

perc_features([[[_word,Features],_] | _],[[Maximal]],
[[Maximal,Features]]).

perc_features([[[[word,'-n+v','-',tense],[_word,'-n+v+a+p']],
_word1,'-n+v',_PN,_Tense]] | _],[[Maximal]],[[Maximal,'-n+v+a+p']]).

perc_features([[[[_word,'-n+v','-',tense],[_word,'-n+v+a+p'
]], [[_word1,'-n+v',_PN,_Tense],[_word1,'-n+v+a+p']]] | _],  [[Maximal]],[[Maximal,'-n+v+a+p']]).

perc_features([[[_word, '-n+v', _PN, _Tense],
_] | _],[[Maximal]],[[Maximal, '-n+v']]).

perc_features([[[_word,  '-n+v'],  [_word1,  _Features,  _PN,  _Tense]] | _],
[[Maximal] | Reststack],[[Maximal, '-n+v'] | Rests
tack]).

perc_features([[[_word,'-n+v+a+p','+',tense],[_word1,_Features,_PN,_Tense]] | _],[[Maximal] | R
eststack],[[Maximal, '-n+v+a+p'] | Reststack]).

perc_features([[[xmax,_S,_F],  [[Word1,_Features,sg],  [Word1,Features1]]] | _],
[[Maximal]],[[Maximal, Features1]]).

perc_features([[[xmax,_S,_F],  [_word1,  Features,  _PN,  _Tense]] | _],
[[Maximal] | Reststack],[[Maximal, Features] | Reststack]).

perc_features([[[_word,  '+n-v-a-p'],  _Second] | _],   [[Maximal] | Reststack],[[Maximal,
'+n-v+a-p'] | Reststack]).

perc_features([[[[_word,'+n-v-a-p'],[_word,'-n-v+a+p']],[_W,'+n-v+a-p',sg]] | _],[[Maximal] | Res
tstack],[[Maximal, '+n-v+a-p'] | Reststack]).

perc_features([[[[_word,'+n-v-a-p'],[_word,'-n-v+a+p']],[_W,'+n-v+a-p',pl]] | _],[[Maximal] | Res
tstack],[[Maximal,'-n-v+a+p'] | Reststack]).

perc_features([[[[_word,'+n-v-a-p'],[_word,'-n-v+a+p']],_] | _],[[Maximal] | Reststack],[[Maximal,
'-n-v+a+p'] | Reststack]).

perc_features([[[[_word,'+n-v-a-p'],[_word,'-n-v+a+p']],[_W,'+n-v+a-p',pn]] | _],[[Maximal] | Re
ststack],[[Maximal,'-n-v+a+p'] | Reststack]).

perc_features([[[[_word,'+n-v-a-p'],[_word,'-n-v+a+p']],[_W,'+n-v-a-p']] | _],[[Maximal] | Reststa
ck],[[Maximal,'-n-v+a+p'] | Reststack]).

perc_features([[[_word,  '+n-v+a+p'],  _Second] | _],   [[Maximal] | Reststack],[[Maximal,
'+n-v+a-p'] | Reststack]).

perc_features([[[_word,  '+n-v+a-p',_SP],  _Second] | _],   [[Maximal] | Reststack],[[Maximal,
'+n-v+a-p'] | Reststack]).

perc_features([[[_word,  '+n-v+a-p',pn],  _Second] | _],   [[Maximal] | Reststack],[[Maximal,
'+n-v+a-p'] | Reststack]).

perc_features([[[_word, Features], _] | _],
[[Maximal] | Reststack],[[Maximal, Features] | Reststack]).

perc_features([[[_word, Features, _PN, _Tense], _] | _],[[Maximal] | Reststack],[[Maximal,
Features] | Reststack]).

perc_features([[[xmax,[_Rulename,_word],_Features],[[_word1,
_Features1a],[_word1,Features1]]] | _],[[Maximal] | Reststack],[[Maximal, Features1] | Reststack]).

perc_features([[[xmax, [_Rulename, _word],_Features],[_word1,
Features1]] | _],[[Maximal] | Reststack],[[Maximal, Features1] | Reststack]).

/* This rule, called by 'prep','act_create_node' and 'drop' places a template in the stack which
describes the function of the word or phrase in the first cell, eg specifier, head, or complement.
On ocassions it is also necessary toconsider the 2nd buffer cell and/or the stack.*/

add_template([[[_word, '+n-v+a+p'],_] | _], [spec_, head, comp],[[xmax, Features] | Reststack],
[[xmax, Features,[spec_, head, comp]] | Reststack]).

add_template([[[_word, '+n-v-a-p'],_] | _], [spec_, head, comp],[[xmax, Features] | Reststack],
[[xmax, Features,[spec_, head, comp]] | Reststack]).

add_template([[[[_word,'+n-v-a-p'],[_word,'-n-v+a+p']],[_W,'+n-v+a-p',sg]] | _],[spec_, head,
comp],[[xmax, Features] | Reststack], [[xmax, Features,[spec_, head, comp]] | Reststack]).

add_template([[[[_word,'-n+v',_PN,_T],[_word,'-n+v+a+p']],[_W,'-n+v',_PN1,_T1]] | _],[spec,
head_, comp],[[xmax, Features] | Reststack], [[xmax, Features,[spec, head_, comp]] | Reststack]).

add_template([[[[_word,'+n-v+a-p',sg],[_word,'-n+v+a+p']],[be,'-n+v']] | _],[spec, head_,
comp],[[xmax, Features] | Reststack], [[xmax, Features,[spec, head_, comp]] | Reststack]).

add_template([[[[_word,'-n+v',_PN,_T],[_word,'-n+v+a+p']],
[_W,'-n+v',_PN1,_T1],[_W,'-n+v+a+p']]] | _],[spec, head_, comp],[[xmax, Features] | Reststack],
[[xmax, Features,[spec, head_, comp]] | Reststack]).

add_template([[[[_word,'+n-v-a-p'],[_word,'-n-v+a+p']],[_W,'+n-v+a-p',pl]] | _],[spec_, head,
comp],[[xmax, Features] | Reststack], [[xmax, Features,[spec, head_, comp]] | Reststack]).

add_template([[[[_word,'+n-v-a-p'],[_word,'-n-v+a+p']],_] | _],[spec_, head, comp],[[xmax,
Features] | Reststack], [[xmax, Features,[spec, head_, comp]] | Reststack]).

add_template([[[[_word,'+n-v-a-p'],[_word,'-n-v+a+p']],[_W,'+n-v+a-p',pn]] | _],[spec_, head,
comp],[[xmax, Features] | Reststack], [[xmax, Features,[spec, head_, comp]] | Reststack]).

add_template([[[[_word,'+n-v-a-p'],[_word,'-n-v+a+p']],[_W,'+n-v+a-p',pl]] | _],[spec_, head,
comp],[[xmax, Features] | Reststack], [[xmax, Features,[spec_, head, comp]] | Reststack]).

add_template([[[[_word,'-n+v-a+p',_PN,_T],[_word,'+n-v+a-p',sg]],[_W,'+n-v-a-p']] | _],[spec_,
head, comp],[[xmax, Features] | Reststack], [[xmax, Features,[spec_, head, comp]] | Reststack]).

add_template([[[_word, '-n+v', '+', tense], _Second] | _], [spec_, head, comp],[[xmax, Features]],
[[xmax, Features, [spec_, head, comp]]]).

add_template([[[_word, '-n+v', _PN, _Tense],_] | _],[spec_, head, comp],[[xmax,
Features],[xmax,[attach_wh_comp,E],F,T,wh]], [[xmax,
Features,[spec_,head,comp]],[xmax,[attach_wh_comp,E],F,T,wh]]).
```

```
add_template([[[_word, '-n+v', _PN, _Tense],_]|_],[spec, head_, comp],[[xmax,
Features]|Reststack], [[xmax, Features,[spec, head_, comp]]|Reststack]).

add_template([[[_word, '-n+v-a+p', _PN, _Tense],_]|_], [spec_, head, comp],[[xmax, Features]],
[[xmax, Features,[spec_, head, comp]]]).

add_template([[[_word, '-n+v', _PN, _Tense],_]|_],[spec, head_, comp],[[xmax,
Features]|Reststack], [[xmax, Features,[spec, head_, comp]]|Reststack]).

add_template([[[_word, '-n+v-a+p', _PN, _Tense],_]|_], [spec_, head, comp],[[xmax, Features]],
[[xmax, Features,[spec_, head, comp]]]).

add_template([[[_word, '-n+v-a+p', _PN, _Tense],_]|_], [spec, head_, comp],[[xmax,
Features]|Reststack], [[xmax, Features,[spec, head_, comp]]|Reststack]).

add_template([[[_word, '-n+v+a+p', _PN, _Tense],_]|_], [spec_, head, comp],[[xmax, Features]],
[[xmax, Features,[spec_, head, comp]]]).

add_template([[[_word, '-n+v+a+p', _PN, _Tense],_]|_], [spec, head_, comp],[[xmax,
Features]|Reststack], [[xmax, Features,[spec, head_, comp]]|Reststack]).

add_template([[[_word, '-n+v'],[_w1,'+n-v+a-p']]|_], [spec_, head, comp],[[xmax, Features]],
[[xmax, Features,[spec_, head, comp]]]).

add_template([[[_word, '-n+v'],[_w1,'+n-v-a-p']]|_], [spec_, head, comp],[[xmax, Features]],
[[xmax, Features,[spec_, head, comp]]]).

add_template([[[_word,'-n+v+a+p'],[_w1,'+n-v+a-p',_PL]]|_],[spec_, head, comp],[[xmax,
Features]|R], [[xmax, Features,[spec_, head, comp]]|R]).

add_template([[[_word, '-n+v+a+p'],[_w1,'+n-v-a-p']]|_], [spec_, head, comp],[[xmax, Features]],
[[xmax, Features,[spec_, head, comp]]]).

add_template([[[_word, '-n+v'],_]|_], [spec, head_, comp],[[xmax, Features]|Reststack], [[xmax,
Features,[spec, head_, comp]]|Reststack]).

add_template([[[_word, '-n+v+a+p'],_]|_], [spec, head_, comp],[[xmax, Features]|Reststack],
[[xmax, Features,[spec, head_, comp]]|Reststack]).

add_template([[[xmax, _S, '+n-v+a-p'],_Second]|_], [spec_, head,
comp],[[xmax,Features]|Reststack], [[xmax, Features,[spec_, head, comp]]|Reststack]).

add_template([[[xmax,_S, '-n-v+a+p'],[_Word, '-n+v',_PN,_Tense]]|_],[spec_, head, comp],
[xmax,Features]|Reststack], [[xmax, Features,[spec_, head, comp]]|Reststack]).

add_template([[[_word, '-n-v+a+p'],_Second]|_],[spec, head_, comp],[[xmax,Features]|Reststack],
[[xmax, Features,[spec, head_, comp]]|Reststack]).

add_template([[[_word, '-n-v'],_Second]|_],[spec, head_, comp],[[xmax,Features]|Reststack],
[[xmax, Features,[spec, head_, comp]]|Reststack]).

add_template([[[_word, '+n-v+a-p',pn],_]|_], [spec, head_, comp],[[xmax, Features]|Reststack],
[[xmax, Features,[spec, head_, comp]]|Reststack]).

add_template([[[_word, '+n-v+a-p',_SP],_]|_], [spec, head_, comp],[[xmax, Features]|Reststack],
[[xmax, Features,[spec, head_, comp]]|Reststack]).

/*This rule, called by 'prep', amends certain features after percolation which would not be the
```

```
correct descriptions of word or phrase.*/

add_features([[xmax, '-n+v', [spec_, head, comp]] | Reststack], '-n+v', [[xmax, '-n+v+a+p',[spec_,
head, comp]] | Reststack]).

add_features([[xmax, '-n+v-a+p', [spec_, head, comp]] | Reststack], '-n+v-a+p', [[xmax,
'-n+v+a+p',[spec_, head, comp]] | Reststack]).

add_features([[xmax, '-n+v', [spec, head_, comp]] | Reststack], '-n+v', [[xmax, '-n+v-a+p',[spec,
head_, comp]] | Reststack]).

add_features([[xmax, '-n+v', [spec, head, comp_]] | Reststack], '-n+v', [[xmax, '-n+v-a+p',[spec,
head, comp_]] | Reststack]).

add_features([[xmax,'-n-v+a+p',[spec_,head,comp]] | Reststack], '-n-v+a+p',
[[xmax,'-n+v+a+p',[spec_,head,comp]] | Reststack]).

add_features([[xmax, Features, Template] | Reststack],
Features,[[xmax,Features,Template] | Reststack]).add_features([[xmax,Features,Template] | Rests
tack],_Features1,[[xmax,Features,Template] | Reststack]).

/*This recursive rule invokes the parsing mechanism, calling on the 'grammar_rule', which
invokes the grammar, and 'process' which manipulate the stack and buffer. If no grammar rules
fire the rule 'drop' is invoked.*/

parse([], Buffer, Buffer).parse(Stack, Buffer, P):-
grammar_rule(Rulename, Stack, Buffer, News, Newb),
process(Rulename,News,Newb,Newstack,Newbuffer),parse(Newstack, Newbuffer, P),!.

parse(Stack, Buffer,P):-
drop(Stack, Buffer, Newstack, Newbuffer),

parse(Newstack,Newbuffer,P),!.

/*This rule, called by parse, processes further the stack and buffer to be in the proper state for
the parsing of next phrase or word.*/

process(Rulename,Stack,Buffer,Newstack,Newbuffer):-
annotate_node(Rulename, Stack, NewS1),
amend_stack(NewS1,Buffer,NewS2),
amend_template(NewS2, Buffer, NewS3),
act_create_node(NewS3, Buffer, NewS4),
pre_test(Buffer,NewS4,Newbuffer,Newstack).

/*This rule, called by 'process', adds the name of the grammar rule just fired to the word or
phrase in the stack.*/

annotate_node(Rulename, [[xmax, [xmax, Entry,Features1], Features,
Template] | Reststack],[[xmax, [Rulename, [xmax,Entry,Features1]], Features,
Template] | Reststack]).

annotate_node(passive, [[xmax,[Rulename,Entry],Features,Template,Ty] | Reststack],
[xmax,[passive,[Rulename,Entry]],Features,Template,Ty] | Reststack]).

annotate_node(Rulename1,[[xmax,[[Rulename,Entry],Entry1],Features,Template] | Reststack],
[xmax,[[Rulename,Entry],[Rulename1,Entry1]],Features,Template] | Reststack]).

annotate_node(Rulename, [[xmax, Entry, Features, Template] | Reststack],[[xmax, [Rulename,
Entry], Features, Template] | Reststack]).

annotate_node(Rulename1,[[xmax,[[Rulename,Entry],Entry1],Features,Template,Type] | Reststa
ck],   [xmax,[[Rulename,Entry],[Rulename1,Entry1]],Features,Template,Type] | Reststack]).

annotate_node(_Rulename,[[xmax,Features,Template] | Reststack],[[xmax,   Features,
Template] | Reststack]).

annotate_node(_Rulename,[[xmax,S,Features,Template,Ty] | Reststack],[[xmax,   S,Features,
Template,Ty] | Reststack]).

/*This rule, called by 'process', checks types of verbs and wh words and adds description to
the stack.*/

amend_stack([[xmax,[attach_wh_comp,[Word,F]],Feat,Temp] | Reststack],Newbuffer,Newstack):-
check_type(Word,Type),
amend_stack1([[xmax,[attach_wh_comp,[Word,F]],
Feat,Temp] | Reststack],Type,Newstack).

amend_stack([[xmax,[attach_verb,[Word,F,PN,Tense]],Feat,Temp] | Reststack],_Newbuffer,
Newstack):-
check_type(Word,Type),
amend_stack1([[xmax,[attach_verb,[Word,F,PN,Tense]],
Feat,Temp] | Reststack],Type,Newstack).
amend_stack(Stack,_Buffer,Stack).

/*This rule, called by amend_stack checks type of verb or wh word.*/

check_type(Word,Type):-word_type(Word, Type).

/*This rule, called by 'amend_stack' adds description of word to stack.*/

amend_stack1([[xmax,[attach_wh_comp,[Word,F]],Feat,Temp] | Reststack],Type,
[xmax,[attach_wh_comp,[Word,F]],Feat,Temp,Type] | Reststack]).

amend_stack1([[xmax,[attach_verb,[Word,F,PN,Tense]],Feat,Temp] | Reststack],Type,
[xmax,[attach_verb,[Word,F,PN,Tense]],Feat,Temp,Type] | Reststack]).

/*This rule, called by 'process', either moves the template's function word pointer to the
succeeding function word, keeps pointer at same position or moves it from comp to spec, by
looking at stack and buffer.*

/amend_template([[xmax, Structure, Features,[spec_, head, comp]] | Reststack],[[[_Word1,
'+n-v+a-p',_SP], _Second] | _Rest], [[xmax, Structure,Features,[spec, head_, comp]] | Reststack]).

amend_template([[xmax, Structure, Features,          [spec_, head, comp]] | Reststack],[[[[W,
'+n-v+a-p',sg],[W, '-n+v+a+p']], [_W2,'-n+v']] | _Rest],[[xmax, Structure,Features,[spec, head_,
comp]] | Reststack]).

amend_template([[xmax,   Structure,   Features,[spec_,   head,   comp]] | Reststack],[[[[W,
'-n+v-a+p','-',_T],[W,  '+n-v+a-p',_SP]],  [_W2,'-n+v']] | _Rest],[[xmax,  Structure,Features,[spec,
head_, comp]] | Reststack]).

amend_template([[xmax, Structure, Features,          [spec_, head, comp]] | Reststack],[[[[W1,
'-n+v','+',s],[W1, '+n-v+a-p',_SP]], [_W2,'-n+v']] | _Rest], [[xmax, Structure,Features,[spec, head_,
comp]] | Reststack]).

amend_template([[xmax,  Features, [spec_, head, comp]] | Reststack],[[_First, _Second] | _Rest],
[[xmax, Features,[spec_, head, comp]] | Reststack]).

amend_template([[xmax, Features, [spec, head, comp_]] | Reststack],[[_First, _Second] | _Rest],
[[xmax, Features,[spec, head, comp_]] | Reststack]).

amend_template([[xmax,St, Features,
Template,Ty] | Reststack],_Newbuffer,Newstack):-equal('-n+v-a+p',
Features),change_template1([[xmax,St,Features,Template,Ty] | Reststack],Newstack).

amend_template([[xmax,St,Features,Template,Ty] | Reststack],_Newbuffer,Newstack):-
equal('-n-v+a+p',
Features),change_template1([[xmax,St,Features,Template,Ty] | Reststack],Newstack).

amend_template([[xmax, S, Features, Template] | Reststack],_Newbuffer,Newstack):-
equal('+n-v+a+p',Features),change_template1([[xmax,S,Features,Template] | Reststack],Newstack).

amend_template([[xmax, S, Features,Template] | Reststack],_Newbuffer,Newstack):-
equal('+n-v-a-p',Features),change_template1([[xmax,S,Features,Template] | Reststack],Newstack).

amend_template([[xmax, [passive, Entry], Features, Template] | Reststack],_Newbuffer,Newstack):-
equal('-n+v-a+p',Features),
hange_template1([[xmax,[passive,Entry],Features,Template] | Reststack],Newstack).

amend_template([[xmax,[wh_insert,Entry],Features,Template,Ty] | Reststack],_Newbuffer,New
stack):-
equal('-n+v-a+p',Features),
hange_template1([[xmax,[wh_insert,Entry],Features,Template,Ty], | Reststack], Newstack).

amend_template([[xmax, [Rulename, Entry], Features,
Template] | Reststack],_Newbuffer,Newstack):-member('+n-v+a-p', Entry),
hange_template1([[xmax,[Rulename,Entry],Features,Template] | Reststack],Newstack).

amend_template([[xmax, [Rulename, Entry], Features,
Template] | Reststack],_Newbuffer,Newstack):-member('-n+v+a+p', Entry),
hange_template1([[xmax,[Rulename,Entry],Features,Template] | Reststack],Newstack).

amend_template([[xmax,S,Features,Template] | Reststack],_Newbuffer,Newstack):-
equal('-n+v+a+p',Features),change_template1([[xmax,S,Features,Template] | Reststack],Newstack).

amend_template([[xmax, St, Features,
Template] | Reststack],_Newbuffer,Newstack):-equal('-n-v+a+p',
Features),change_template1([[xmax,St,Features,Template] | Reststack],Newstack).

amend_template([[xmax, [Rulename, Entry], Features,
Template] | Reststack],_Newbuffer,Newstack):-member('+n-v-a-p', Entry),
hange_template([[xmax,[Rulename,Entry],Features,Template] | Reststack],Newstack).

amend_template([[xmax, S,
Features,Template] | Reststack],[[[_Word,_F,_PD],_Second] | _R],Newstack):-
equal('+n-v+a-p',Features),change_template([[xmax,S,Features,Template] | Reststack],Newstack).

amend_template([[xmax,S, Features,Template] | Reststack],
[[_Word,'+n-v+a+p'],_Second] | _R],Newstack):-equal('+n-v+a-p', Features),
hange_template([[xmax,S,Features,Template] | Reststack],Newstack).

amend_template([[xmax, [Rulename, Entry], Features,
Template] | Reststack],_Newbuffer,Newstack):-
member('-n+v',Entry),    hange_template1([[xmax,[Rulename,Entry],Features,Template] | Reststack],Newstack).
```

```
amend_template([[xmax, [Rulename, Entry], Features, Template] | Reststack],
_Newbuffer,Newstack):- ember('-n+v-a+p',
Entry),change_template1([[xmax,[Rulename,Entry],Features,Template] | Reststack],ewstack).

amend_template([[xmax, [Rulename, Entry], Features,
Template] | Reststack],_Newbuffer,Newstack):-
member('-n-v',Entry),
change_template1([[xmax,[Rulename,Entry],Features,Template] | Reststack],Newstack).

amend_template([[xmax,S,Features, Template] | Reststack],_Newbuffer,Newstack):-
equal('-n+v-a+p', Features),        change_template1([[xmax,S,Features,Template] | Reststack],
Newstack).

amend_template([[xmax,S,
Features,Template] | Reststack],_Newbuffer,Newstack):-equal('-n+v+a+p',
Features),change_template1([[xmax,S,Features,Template] | Reststack], Newstack).

amend_template([[xmax,S,
Features,Template] | Reststack],_Newbuffer,Newstack):-equal('-n-v+a+p',
Features),change_template1([[xmax,S,Features,Template] | Reststack], Newstack).

amend_template([[xmax,[S,[attach_relative_clause,E]],
Features,Template] | Reststack],_Newbuffer,Newstack):-equal('+n-v+a-p', Features),
hange_template1([[xmax,[S,[attach_relative_clause,E]],Features,Template] | Reststack],Newstack).

amend_template([[xmax,S,
Features,Template] | Reststack],_Newbuffer,Newstack):-equal('+n-v+a-p',
Features),change_template1([[xmax,S,Features,Template] | Reststack], Newstack).

amend_template([[xmax, S, Features, Template] | Reststack],_Newbuffer,Newstack):-
equal('-n-v', Features),
change_template2([[xmax, S,Features,Template] | Reststack], Newstack).

amend_template([[xmax,S, Features, Template] | Reststack],_Newbuffer,Newstack):-
equal('-n+v',Features),change_template1([[xmax,S,Features,Template] | Reststack], Newstack).

/*These rules, called by 'amend_template' check features of the buffer.*/
equal(X, X).
member(X, [[X,_Y] | _L]).
member(X, [[_Y,X] | _L]).
member(X, [X | _Tail]).
member(X, [_Head | Tail]):-member(X, Tail).

/*These rules, called by 'amend_template' change the pointers.*/

change_template([[xmax,S,Features,Template] | Reststack],[[xmax,S,Features,Template] | Reststa
ck]).

change_template1([[xmax,S,Features, [spec_, head, comp]] | Reststack],[[xmax,S,Features, [spec,
head_, comp]] | Reststack]).

change_template1([[xmax,S,Features, [spec_, head, comp],Type] | Reststack],[[xmax,S,Features,
[spec, head_, comp],Type] | Reststack]).

change_template1([[xmax,S,Features,[spec, head_, comp]] | Reststack],[[xmax,S, Features, [spec,
head, comp_]] | Reststack]).

change_template1([[xmax,S,Features,[spec,  head,  comp_]] | Reststack],[[xmax,S,Features,[spec,
head, comp_]]|Reststack]).

change_template1([[xmax,S,Features, [spec, head,
comp_],Type]|Reststack],[[xmax,S,Features,[spec, head, comp_],Type]|Reststack]).

change_template1([[xmax,S,Features, [spec, head_, comp],Type]|Reststack],[[xmax,S,Features,
[spec, head, comp_],Type]|Reststack]).

change_template2([[xmax,S,Features,[spec, head, comp_]]|Reststack],[[xmax,S,Features,[spec_,
head, comp]]|Reststack]).

/*This rule, called by 'process', either keeps nodes of the stack active orcreate new nodes on
the stack depending on the states of the buffer and stack.*/

act_create_node([[xmax,S, '+n-v+a-p',Template]|Reststack], [[[_Word, '-n+v-a+p', '+', ing],
Second]|_Rest],Newstack):-create_max(xmax,[[xmax,S,'+n-v+a-p',Template]|Reststack],NewS2
),      erc_features([[[Word1,'-n+v-a+p','+',ing],Second]|Rest],NewS2,NewS3),
dd_template([[[Word1,'-n+v-a+p','+',ing],Second]|Rest],_Template1,NewS3, Newstack).

act_create_node([[xmax,S, '+n-v+a-p',Template]|Reststack], [[[_Word, '-n+v-a+p', '+', ed],
Second]|_Rest],Newstack):-create_max(xmax,[[xmax,S,'+n-v+a-p',Template]|Reststack],NewS2
),      erc_features([[[Word1,'-n+v-a+p','+',ed],Second]|Rest],NewS2,NewS3),
dd_template([[[Word1,'-n+v-a+p','+',ed],Second]|Rest],_Template1,NewS3, Newstack).

act_create_node([[xmax,S, '+n-v+a-p',Template]|Reststack], [[[Word, '+n-v+a-p'],
[Word1,'-n+v',PN,Tense]]|Rest],Newstack):-last1(S,attach_noun),
reate_max(xmax,[[xmax,S,'+n-v+a-p',Template]|Reststack],NewS2),
erc_features([[[Word,'+n-v+a-p'],[Word1,'-n+v',PN,Tense]]|Rest],NewS2,NewS3),
dd_template([[[Word,'+n-v+a-p'],[Word1,'-n+v',PN,Tense]]|Rest],_Template1,NewS3, Newstack).

act_create_node([[xmax,S, '+n-v+a-p',Template],[xmax,S1,'-n+v',T]|Reststack], [[[Word,
'-n-v+a+p'],
Second]|Rest],Newstack):-create_max(xmax,[[xmax,S,'+n-v+a-p',Template],[xmax,S1,'-n+v',T]
Reststack],NewS2),perc_features([[[Word,'-n-v+a+p'],Second]|Rest],NewS2,NewS3),
dd_template([[[Word,'-n-v+a+p'],Second]|Rest],_Template1, NewS3, Newstack).

act_create_node([[xmax,S,'+n-v+a-p',Template],[xmax,S1,'-n+v-a+p',T,Ty]|Reststack], [[[Word,
'-n-v+a+p'],
Second]|Rest],Newstack):-create_max(xmax,[[xmax,S,'+n-v+a-p',Template],[xmax,S1,'-n+v-a+p
',T,Ty]
Reststack],NewS2),perc_features([[[Word,'-n-v+a+p'],Second]|Rest],NewS2,NewS3),
dd_template([[[Word,'-n-v+a+p'],Second]|Rest],_Template1, NewS3, Newstack).

act_create_node([[xmax,S,'+n-v+a-p',Template],[xmax,S1,'-n+v-a+p',T,Ty]|Reststack], [[[[Word,
'+n-v-a-p'],[Word, '-n-v+a+p']], Second]|Rest],Newstack):-
reate_max(xmax,[[xmax,S,'+n-v+a-p',Template],[xmax,S1,'-n+v-a+p',T,Ty]|Reststack],NewS2),
erc_features([[[[Word,'+n-v-a-p'],[Word,
'-n-v+a+p']],Second]|Rest],NewS2,NewS3),add_template([[[[Word,'+n-v-a-p'],[Word,
'-n-v+a+p']],Second]|Rest],_Template1, NewS3, Newstack).

act_create_node([[xmax,S, '+n-v+a-p',Template]],[[[[Word, '+n-v-a-p'],[Word, '-n-v+a+p']],
Second]|Rest],Newstack):-create_max(xmax,[[xmax,S,'+n-v+a-p',Template]],NewS2),
erc_features([[[[Word,'+n-v-a-p'],[Word,
'-n-v+a+p']],Second]|Rest],NewS2,NewS3),add_template([[[[Word,'+n-v-a-p'],[Word,
'-n-v+a+p']],Second]|Rest],_Template1, NewS3,
Newstack).act_create_node([[xmax,[attach_embedded_subject,Entry],Features,_Template]
Reststack],Newbuffer,Newstack):-create_max(xmax,[[xmax,[attach_embedded_subject,Entry],F
eatures]
```

```
Reststack],NewS2),perc_features(Newbuffer,NewS2,NewS3),add_template(Newbuffer,_Templa
te1,NewS3, NewS4),add_features(NewS4, _Feat,Newstack).

act_create_node([[xmax,S, Features,Template] | Reststack],
[[be,'-n+v','+',T],[Word,'-n+v-a+p',PN,Tense]] | Rest],Newstack):-create_max(xmax,[[xmax,S,Fe
atures,Template] | Reststack],NewS2),perc_features([[[be,'-n+v','+',T],[Word,'-n+v-a+p',PN,Ten
s e ] ] | R e s t ] , N e w S 2 ,
NewS3),add_template([[[be,'-n+v','+',T],[Word,'-n+v-a+p',PN,Tense]] | Rest],_Template1, NewS3,
NewS4),add_feat_spec(NewS4,[[[be,'-n+v','+',T],[Word,'-n+v-a+p',PN,Tense]] | Rest],Newstack).

act_create_node([[xmax,S, Features,Template] | Reststack], [[[be, '-n+v', '+', T],
_Second] | Rest],Newstack):-create_max(xmax,[[xmax,S,Features,Template] | Reststack],NewS2),
erc_features([[[be, '-n+v', '+', T], _Second] | Rest], NewS2, NewS3),add_template([[[be, '-n+v', '+',
T], _Second] | Rest],_Template1, NewS3, Newstack).

act_create_node([[xmax,[attach_subject,Entry], Features,_Template]], [[[Word, '-n+v-a+p', PN,
Tense], _Second] | _Rest],Newstack):-create_max(xmax,[[xmax,[attach_subject,Entry],
Features]],NewS2),perc_features([[[Word,'-n+v-a+p',PN,Tense],_Second] | _Rest],NewS2,NewS
3), dd_template([[[Word,'-n+v-a+p',PN,Tense],_Second] | _Rest],_Template1, NewS3,
NewS4), dd_features(NewS4, '-n+v-a+p', Newstack).

act_create_node([[xmax,[attach_subject,E],Features,_Template] | Reststack], [[[Word,'-n+v'],
[Word1,'-n+v-a+p','+',Tense]] | _Rest],Newstack):-create_max(xmax,[[xmax,[attach_subject,E],
Features] | Reststack],NewS2),perc_features([[[Word, '-n+v'], [Word1, '-n+v-a+p','+',Tense]] | Rest],
NewS2, NewS3),add_template([[[Word, '-n+v'], [Word1, '-n+v-a+p', '+', Tense]] | Rest],
_ T e m p l a t e 1 , N e w S 3 ,
NewS4),add_feat_spec(NewS4,[[[Word,'-n+v'],[Word1,'-n+v-a+p','+',Tense]] | Rest],Newstack).

act_create_node([[xmax,[attach_subject,E], Features,_Template] | Reststack], [[[is, '-n+v', '+',
tense],Second] | _Rest],Newstack):-
reate_max(xmax,[[xmax,[attach_subject,E],Features] | Reststack],NewS2),perc_features([[[is,'-n+
v','+',tense],Second] | Rest],NewS2,NewS3),add_template([[[is,'-n+v','+',tense],Second] | Rest],
Template1,NewS3,Newstack).

act_create_node([[xmax,[attach_subject,E], Features,_Template] | Reststack], [[[was, '-n+v', '+',
tense],Second] | _Rest],Newstack):-
reate_max(xmax,[[xmax,[attach_subject,E],Features] | Reststack],NewS2),perc_features([[[was,'-
n+v','+',tense],Second] | Rest],NewS2,NewS3),add_template([[[was,'-n+v','+',tense],Second] | Re
st], Template1,NewS3,Newstack).act_create_node([[xmax,[attach_subject,E],
Features,_Template] | Reststack], [[[are, '-n+v', '+', tense],Second] | _Rest],Newstack):-
reate_max(xmax,[[xmax,[attach_subject,E],Features] | Reststack],NewS2),perc_features([[[are,'-n
+v','+',tense],Second] | Rest],NewS2,NewS3),add_template([[[are,'-n+v','+',tense],Second] | Rest]
, Template1,NewS3,Newstack).

act_create_node([[xmax,[attach_subject,E], Features,_Template] | Reststack], [[[were, '-n+v', '+',
tense],Second] | _Rest],Newstack):-
reate_max(xmax,[[xmax,[attach_subject,E],Features] | Reststack],NewS2),perc_features([[[were,'-
n+v','+',tense],Second] | Rest],NewS2,NewS3),add_template([[[were,'-n+v','+',tense],Second] | R
est], Template1,NewS3,Newstack).
act_create_node([[xmax,[attach_subject,Entry], Features,_Template]], [[[Word, '-n+v', PN, Tense],
_Second] | _Rest],Newstack):-
c r e a t e _ m a x ( x m a x , [ [ x m a x , [ a t t a c h _ s u b j e c t , E n t r y ] ,
Features]],NewS2),perc_features([[[Word,'-n+v',PN,Tense],_Second] | _Rest],NewS2,NewS3),
dd_template([[[Word,'-n+v',PN,Tense],_Second] | _Rest],_Template1, NewS3,
NewS4),add_features(NewS4, '-n+v', Newstack).

act_create_node([[xmax,[attach_subject,Entry], Features,_Template]], [[[Word, F],
_Second] | _Rest],Newstack):-
```

```
create_max(xmax,[[xmax,[attach_subject,Entry], Features]],NewS2),perc_features([[[Word, F],
_Second] | _Rest], NewS2,NewS3),add_template([[[Word, F], _Second] | _Rest],_Template1, NewS3,
        ewstack).act_create_node([[xmax,[attach_subject,Entry], Features,_Template]], [[[[Word,
F,sg],[Word,F1]], _Second] | _Rest],Newstack):-
create_max(xmax,[[xmax,[attach_subject,Entry], Features]],NewS2),perc_features([[[[Word,
F,sg],[Word,F1]], _Second] | _Rest],NewS2,NewS3),add_template([[[[Word, F,sg],[Word,F1]],
_Second] | _Rest],_Template1, NewS3,Newstack).

act_create_node([[xmax,[attach_subject,Entry], Features,_Template]], [[[Word, '-n+v+a+p', +,
tense], _Second] | _Rest],Newstack):-
create_max(xmax,[[xmax,[attach_subject,Entry], Features]],NewS2),perc_features([[[Word,
'-n+v+a+p', '+', tense],_Second] | _Rest], NewS2, NewS3),add_template([[[Word, '-n+v+a+p', '+',
tense], _Second] | _Rest], _Template1, NewS3, Newstack).

act_create_node([[xmax,Structure, '-n+v',Template] | Reststack], [[[Word, '-n+v-a+p', PN, Tense],
_Second] | _Rest],Newstack):-
create_max(xmax,[[xmax,Structure,'-n+v',Template] | Reststack],NewS2),
erc_features([[[Word,'-n+v-a+p',PN,Tense],_Second] | Rest],NewS2,NewS3),add_template([[[W
ord,'-n+v-a+p',PN,Tense],_Second] | Rest],_Template1, NewS3, Newstack).

act_create_node([[xmax,Structure, '-n+v+a+p',Template] | Reststack], [[[Word, '-n+v-a+p', PN,
Tense], _Second] | _Rest],Newstack):-
create_max(xmax,[[xmax,Structure,'-n+v+a+p',Template] | Reststack],NewS2),
erc_features([[[Word,'-n+v-a+p',PN,Tense],_Second] | Rest],NewS2,NewS3),add_template([[[W
ord,'-n+v-a+p',PN,Tense],_Second] | Rest],_Template1, NewS3, Newstack).

act_create_node([[xmax,[attach_wh_comp,Entry], Features,Template]], [[[Word, '-n+v+a+p', +,
tense], _Second] | _Rest],Newstack):-
c r e a t e _ m a x ( x m a x , [ [ x m a x , [ a t t a c h _ w h _ c o m p , E n t r y ] ,
Features,Template]],NewS2),perc_features([[[Word, '-n+v+a+p'],_Second] | _Rest], NewS2,
NewS3),
add_template([[[Word, '-n+v+a+p'], _Second] | _Rest], _Template1, NewS3, Newstack).

act_create_node([[xmax,[modal,Entry], Features,Template] | Reststack], [[[Word, '-n+v'],[Word1,
'-n+v-a+p' ,PN, Tense]] | Rest],Newstack):-
create_maxxmax,[[xmax,[modal,Entry],Features,Template] | Reststack],NewS2),
erc_features([[[Word,'-n+v'],[Word1,'-n+v-a+p',PN,Tense]] | Rest],NewS2,NewS3),add_templat
e([[[Word, '-n+v'],[Word1, '-n+v-a+p',PN, Tense]] | Rest], _Template1, NewS3, NewS4),
dd_feat_spec(NewS4,[[[Word,'-n+v'],[Word1,'-n+v-a+p',PN,Tense]] | Rest],Newstack).

act_create_node([[xmax,[modal,Entry], Features,Template] | Reststack], [[[Word, '-n+v'],
_Second] | _Rest],Newstack):-
create_maxxmax,[[xmax,[modal,Entry],Features,Template] | Reststack],NewS2),perc_features([[[
Word, '-n+v'],_Second] | _Rest], NewS2, NewS3),add_template([[[Word, '-n+v'], _Second] | _Rest],
_Template1, NewS3, Newstack).

act_create_node([[xmax,[attach_rpron,E],Features,Template] | Reststack], [[[Word,'-n+v'],
[Word1,'-n+v-a+p','+',Tense]] | _Rest],Newstack):-
create_max(xmax,[[xmax,[attach_rpron,E], Features,Template] | Reststack],NewS2),
perc_features([[[Word, '-n+v'], [Word1, '-n+v-a+p','+',Tense]] | Rest], NewS2, NewS3),
add_template([[[Word, '-n+v'], [Word1, '-n+v-a+p', '+', Tense]] | Rest],           _Template1,
N          e          w          S          3          ,
NewS4),add_feat_spec(NewS4,[[[Word,'-n+v'],[Word1,'-n+v-a+p','+',Tense]] | Rest],Newstack).

act_create_node([[xmax,'-n+v-a+p',Template] | Reststack],Newbuffer,Newstack):-
create_max(xmax,[[xmax,'-n+v-a+p',Template] | Reststack],NewS2),
perc_features(Newbuffer, NewS2, NewS3),add_template(Newbuffer, _Template1, NewS3,
Newstack).
```

```
act_create_node([[xmax,Structure,'-n+v-a+p',Template,Ty] | Reststack],[[[Word,'+n-v+a-p',PL],
_Second] | _Rest],Newstack):-
create_max(xmax,[[xmax,Structure,'-n+v-a+p',Template,Ty] | Reststack], NewS2),
perc_features([[[Word,'+n-v+a-p',PL], _Second] | _Rest], NewS2, NewS3),
add_template([[[Word,'+n-v+a-p',PL], _Second] | _Rest], _Template1, NewS3, Newstack).

act_create_node([[xmax,Structure,'-n+v-a+p',Template,Ty] | Reststack],
[[[Word,'+n-v-a-p'],[Word,'-n-v+a+p']],Second] | _Rest],Newstack):-
create_max(xmax,[[xmax,Structure,'-n+v-a+p',Template,Ty] | Reststack], NewS2),
erc_features([[[[Word,'+n-v-a-p'],[Word,'-n-v+a+p']],Second] | _Rest],NewS2,NewS3),add_temp
late([[[[Word,'+n-v-a-p'],[Word,'-n-v+a+p']],Second] | _Rest],_Template1,NewS3, Newstack).
 act_create_node([[xmax,Structure,'-n+v-a+p',Template,Ty] | Reststack],[[[Word,'+n-v+a-p',pn],
_Second] | _Rest],Newstack):-
create_max(xmax, [[xmax,Structure,'-n+v-a+p',Template,Ty] | Reststack], NewS2),
perc_features([[[Word,'+n-v+a-p',pn], _Second] | _Rest], NewS2, NewS3),
add_template([[[Word,'+n-v+a-p',pn], _Second] | _Rest],_Template1,NewS3, Newstack).

act_create_node([[xmax,Structure,'-n+v-a+p',Template,Ty] | Reststack],[[[Word,'+n-v+a+p'],
_Second] | _Rest],Newstack):-
create_max(xmax,[[xmax,Structure,'-n+v-a+p',Template,Ty] | Reststack], ewS2),
perc_features([[[Word,'+n-v+a+p'], _Second] | _Rest], NewS2, NewS3),
add_template([[[Word,'+n-v+a+p'], _Second] | _Rest], _Template1, NewS3, Newstack).

act_create_node([[xmax,Structure,'-n+v-a+p',Template,Ty] | Reststack],[[[Word,'+n-v-a-p'],
_Second] | _Rest],Newstack):-
create_max(xmax,[[xmax,Structure,'-n+v-a+p',Template,Ty] | Reststack], NewS2),
perc_features([[[Word,'+n-v-a-p'], _Second] | _Rest], NewS2, NewS3),
add_template([[[Word,'+n-v-a-p'], _Second] | _Rest], _Template1, NewS3, Newstack).
 act_create_node([[xmax,Structure,'-n+v-a+p',Template,Ty] | Reststack],[[[Word,'-n-v'],
_Second] | _Rest],Newstack):-
create_max(xmax,[[xmax,Structure,'-n+v-a+p',Template,Ty] | Reststack],NewS2),
perc_features([[[Word,'-n-v'], _Second] | _Rest], NewS2, NewS3),
add_template([[[Word,'-n-v'], _Second] | _Rest], _Template1, NewS3, Newstack).

act_create_node([[xmax,Structure,'-n+v-a+p',Template,Ty] | Reststack],[[[Word,'-n-v+a+p'],
_Second] | _Rest],Newstack):-
create_max(xmax,[[xmax,Structure,'-n+v-a+p',Template,Ty] | Reststack], NewS2),
perc_features([[[Word,'-n-v+a+p'], _Second] | _Rest], NewS2,
NewS3),add_template([[[Word,'-n-v+a+p'], _Second] | _Rest], _Template1, NewS3, Newstack).

act_create_node([[xmax,Structure,'-n+v-a+p',Template,Ty] | Reststack],[[[to,'-n+v+a+p'],
_Second] | _Rest],Newstack):-
create_max(xmax,[[xmax,Structure,'-n+v-a+p',Template,Ty] | Reststack], ewS2),
perc_features([[[to,'-n+v+a+p'], _Second] | _Rest], NewS2, NewS3),
add_template([[[to,'-n+v+a+p'], _Second] | _Rest], _Template1, NewS3, Newstack).

act_create_node([[xmax,Structure,'-n+v',Template] | Reststack],
[[Word,'+n-v+a-p'],Second] | Rest],Newstack):-
create_max(xmax,[[xmax,Structure,'-n+v',Template] | Reststack],
NewS2),perc_features([[[Word,'+n-v+a-p'],Second] | Rest], NewS2, NewS3),
add_template([[[Word,'+n-v+a-p'],Second] | Rest], _Template1, NewS3, Newstack).

act_create_node([[xmax,Structure,'-n+v',Template] | Reststack],
[[Word,'+n-v+a+p'],Second] | Rest],Newstack):-
create_max(xmax,[[xmax,Structure,'-n+v',Template] | Reststack],
NewS2),perc_features([[[Word,'+n-v+a+p'],Second] | Rest], NewS2, NewS3),
add_template([[[Word,'+n-v+a+p'],Second] | Rest], _Template1, NewS3, Newstack).
```

```
act_create_node([[xmax,Structure,'-n+v',Template] | Reststack],[[[Word,'+n-v-a-p'],Second] | Res
t],Newstack):-                    create_max(xmax,[[xmax,Structure,'-n+v',Template] | Reststack],
NewS2),perc_features([[[Word,'+n-v-a-p'],Second] | Rest], NewS2, NewS3),
add_template([[[Word,'+n-v-a-p'],Second] | Rest], _Template1, NewS3, Newstack).

act_create_node([[xmax,Structure,'-n+v',Template] | Reststack],[[[Word,'-n-v'],Second] | Rest],N
ewstack):-                        create_max(xmax,[[xmax,Structure,'-n+v',Template] | Reststack],
NewS2),perc_features([[[Word,'-n-v'],Second] | Rest], NewS2, NewS3),
dd_template([[[Word,'-n-v'],Second] | Rest], _Template1, NewS3, Newstack).

act_create_node([[xmax,[passive,[attach_verb,
Entry] | Remain],'-n+v-a+p',Template,Ty] | Reststack], Newbuffer, Newstack):-
create_max(xmax,[[xmax,[passive,[attach_verb,Entry] | Remain],'-n+v-a+p',Template,Ty] | Restst
ack], NewS2),perc_features(Newbuffer, NewS2, NewS3),add_template(Newbuffer, _Template1,
NewS3, Newstack ).

act_create_node([[xmax,S, '+n-v+a-p',Template] | Reststack], [[[Word, '-n-v'],
_Second] | Rest],Newstack):-
reate_max(xmax,[[xmax,S,'+n-v+a-p',Template] | Reststack],NewS2),
erc_features([[[Word,'-n-v'],_Second] | Rest],NewS2,NewS3),add_template([[[Word,'-n-v'],_Seco
nd] | Rest],   Template1,NewS3, Newstack).

act_create_node([[xmax,[attach_prep,Entry],'-n-v',Template] | Reststack], Newbuffer, Newstack):-
create_max(xmax, [[xmax,[attach_prep, Entry],'-n-v',Template] | Reststack],
NewS2),perc_features(Newbuffer, NewS2, NewS3),add_template(Newbuffer, _Template1, NewS3,
Newstack).

act_create_node([[xmax,[attach_comp,  Entry],'-n-v+a+p',Template] | Reststack],Newbuffer,
Newstack):-
create_max(xmax, [[xmax,[attach_comp, Entry],'-n-v+a+p',Template] | Reststack], NewS2),
erc_features(Newbuffer, NewS2, NewS3),add_template(Newbuffer, _Template1, NewS3,
Newstack).

act_create_node([[xmax,[attach_rpron,  Entry],'-n-v+a+p',Template] | Reststack],Newbuffer,
Newstack):-
create_max(xmax, [[xmax,[attach_rpron, Entry],'-n-v+a+p',Template] | Reststack], NewS2),
erc_features(Newbuffer, NewS2, NewS3),add_template(Newbuffer, _Template1, NewS3,
NewS4),     dd_features(NewS4,_Features,Newstack).

act_create_node([[xmax,[attach_wh_comp,  Entry],'-n-v+a+p',Template,wh] | Reststack],
Newbuffer,     Newstack):-                    create_max(xmax, [[xmax,[attach_wh_comp,
Entry],'-n-v+a+p',Template,wh] | Reststack],NewS2),perc_features(Newbuffer, NewS2,
NewS3),add_template(Newbuffer, _Template1, NewS3,
NewS4),add_features(NewS4,'-n+v',Newstack).

act_create_node([[xmax,[modal,Entry], Features,Template] | Reststack], [[[Word, '-n+v', PN,
Tense],
_Second] | Rest],Newstack):-
reate_max(xmax,[[xmax,[modal,Entry],Features,Template] | Reststack],NewS2),
perc_features([[[Word,'-n+v', PN, Tense], _Second] | Rest], NewS2,NewS3),add_template([[[Word,
'-n+v', PN, Tense], _Second] | Rest],_Template1, NewS3, NewS4),add_features(NewS4, '-n+v',
Newstack).

act_create_node([[xmax,[to_infinitive,Entry],Features,Template] | Reststack], [[[Word,'-n+v', PN,
Tense], _Second] | Rest],Newstack):-
create_max(xmax,[[xmax,[to_infinitive,Entry],Features,Template] | Reststack],NewS2),
erc_features([[[Word,'-n+v', PN, Tense], _Second] | Rest], NewS2,NewS3),add_template([[[Word,
'-n+v', PN, Tense], _Second] | Rest],_Template1, NewS3, NewS4),add_features(NewS4, '-n+v',
Newstack).

act_create_node([[xmax,[aux_sai,Entry],Features,Template]|Reststack], [[[Word, '-n+v', PN,
Tense], _Second]|Rest],Newstack):-
create_max(xmax,[[xmax,[aux_sai,Entry],Features,Template]|Reststack],NewS2),perc_features(
[[[Word,'-n+v', PN, Tense], _Second]|Rest], NewS2,NewS3),add_template([[[Word, '-n+v', PN,
Tense], _Second]|Rest],_Template1, NewS3, NewS4),
add_features(NewS4, '-n+v', Newstack).

act_create_node([[xmax,Features,Template]|Reststack], [[[Word,
'+n-v+a-p',pn],[[Word1,'-n+v-a+p','-',tense],[Word2,'+n-v+a-p']]]|Rest],Newstack):-
reate_max(xmax,[[xmax,Features,Template]|Reststack],NewS2),perc_features([[[Word,'+n-v+a-
p',pn],[[Word1,'-n+v-a+p','-',tense],[Word2,'+n-v+a-p']]]|Rest],NewS2,NewS3),
add_template([[[Word,'+n-v+a-p',pn],[[Word1,'-n+v-a+p','-',tense],[Word2,'+n-v+a-p']]]|Rest],
_Template1, NewS3, NewS4),add_features(NewS4, _Features, Newstack).

act_create_node([[xmax,Features,Template]|Reststack], [[[Word,
'+n-v+a-p',pn],[Word1,'-n+v-a+p','-',tense]]|Rest],Newstack):-
create_max(xmax,[[xmax,Features,Template]|Reststack],NewS2),
erc_features([[[Word,'+n-v+a-p',pn],[Word1,'-n+v-a+p','-',tense]]|Rest],NewS2,NewS3),add_te
mplate([[[Word,'+n-v+a-p',pn],[Word1,'-n+v-a+p','-',tense]]|Rest],_Template1,NewS3,NewS4),
add_features(NewS4, _Features, Newstack).

act_create_node([[xmax,[R,Entry],'-n+v',Template]|Reststack], [[[Word, '-n+v', PN, Tense],
_Second]|Rest],Newstack):-
create_max(xmax,[[xmax,[R,Entry],'-n+v',Template]|Reststack],NewS2),
perc_features([[[Word,'-n+v', PN, Tense], _Second]|Rest], NewS2,NewS3),
add_template([[[Word,'-n+v',PN,Tense],_Second]|Rest],_Template1, NewS3, NewS4),
add_features(NewS4, '-n+v', Newstack).

act_create_node(NewS2, _Newbuffer, NewS2).

/*This rule, called by 'act_create_node', checks the buffer positions.*/last1([_X,Y],Y).

last1([_X,[Y,_T]],Y).
last1([_X,[_R,[Y,_T],_z,_s]],Y).
last1([_X|Z],Y):-last1(Z,Y).

/*This rule, called by 'act_create_node' and 'drop3' adds extra features to the features in the
stack depending on the contents of the buffer.*/

add_feat_spec([[xmax,[attach_prep, Entry],'-n-v',Template]|Reststack],[[[xmax, [_Rulename,
_Entry1], _Features1], [to, '-n+v+a+p']]|[_Second]],[[xmax,[attach_prep, Entry],
'-n-v+a+p',Template]|Reststack]).

add_feat_spec([[xmax,'-n+v',Template]|Reststack],[[[_Word, '-n+v'],[_Word1, '-n+v-a+p', _PN,
_Tense]]|_Rest], [[xmax,'-n+v+a+p',Template]|Reststack]).

add_feat_spec([[xmax,'-n+v',Template]|Reststack],[[[_Word, '-n+v',_PN, _T],[_Word1, '-n+v-a+p',
_PN, _Tense]]|_Rest],[[xmax,'-n+v+a+p',Template]|Reststack]).


/*This rule, called by 'parse' if grammar_rule fails, drops a word or phrase from the stack into
the buffer, also can create a new node for stack, depending on the state of buffer and stack.*/

drop([[xmax,S,'+n-v+a-p',_Template],[xmax,[attach_verb,[persuade,F1,PN,Te]],F,T,Ty]|R],[[[to,
'-n+v+a+p'],Second]],[[xmax,[attach_verb,[persuade,F1,PN,Te]],F,T,Ty]|R],[[[xmax, S,
'+n-v+a-p'],[to, '-n+v+a+p']]|[Second]]).
```

```
drop([[xmax,S,'+n-v+a-p',_Template],[xmax,[attach_verb,Entry1],F,T,Ty],[xmax,[attach_rpron,E
],F1,T1]|R],[[[Word,'-n+v',PN,Tense],Second]],[[xmax,[attach_verb,Entry1],F,T,Ty],[xmax,[attac
h_rpron,E],F1,T1]|R],[[[xmax,S,'+n-v+a-p'],[Word,'-n+v',PN,Tense]]|[Second]]).

drop([[xmax,S,'+n-v+a-p',_Template],[xmax,S1,'-n-v',T]|Reststack],[[[Word,'-n+v',PN,Tense],S
econd]],[[xmax,S1,'-n-v',T]|Reststack],[[[xmax,S,'+n-v+a-p'],[Word,'-n+v',PN,Tense]]|[Second]]).

drop([[xmax,S,'+n-v+a-p',_Template],[xmax,S1,F,T],[xmax,[attach_reduce_rel,E],F1,T1]|R],[[[W
ord,'-n+v'],Second]],[[xmax,S1,F,T],[xmax,[attach_reduce_rel,E],F1,T1]|R],[[[xmax,S,'+n-v+a-p']
,[Word,'-n+v']]|[Second]]).

drop([[xmax,S,'+n-v+a-p',_Template],[xmax,S1,F,T],[xmax,[attach_reduce_rel,E],F1,T1]|R],[[[W
ord,'-n+v+a+p'],Second]],[[xmax,S1,F,T],[xmax,[attach_reduce_rel,E],F1,T1]|R],[[[xmax,S,'+n-v
+a-p'],[Word,'-n+v']]|[Second]]).

drop([[xmax,[[attach_rpron,Entry1]|R],F,_T]|Reststack],[[[Word,'-n+v'],Second]],Reststack,[[[x
max,[[attach_rpron,Entry1]|R],F],[Word,'-n+v']]|[Second]]).

drop([[xmax,S,
'+n-v+a-p',_Template],[xmax,F,T]|R],[[First,Second]|Rest],[[xmax,F,T]|R],[[[xmax,S,'+n-v+a-p'
],First]|[Second|Rest]]).drop([[xmax,S,
'+n-v+a-p',_Template],[xmax,S1,'-n-v',T]|R],[[First,Second]|Rest],[[xmax,S1,'-n-v',T]|R],[[[xma
x,S,'+n-v+a-p'],First]|[Second|Rest]]).

drop([[xmax,S,F,_Template]|Reststack],[[First,Second]|Rest],Newstack,Newbuffer):-last1(Rests
tack,attach_comp),drop2([[xmax,S,
F,_Template]|Reststack],[[First,Second]|Rest],Newstack,Newbuffer).

drop([[xmax, S, Features, Template]|Reststack],[[First, Second]|Rest], Newstack,
Newbuffer):-drop1([[xmax, S, Features, Template]|Reststack],[[First, Second]|Rest], NewS1,
Newbuffer),create_max(xmax, NewS1, NewS2),perc_features(Newbuffer, NewS2,
NewS3),add_template(Newbuffer, _Template1, NewS3, NewS4),add_features(NewS4, _Features1,
Newstack).

drop([[xmax, S, Features, Template]|Reststack], [[First, Second]|Rest], Newstack,
Newbuffer):-drop1([[xmax, S, Features, Template]|Reststack],[[First, Second]|Rest], NewS1,
Newbuffer),
create_max(xmax, NewS1, NewS2),perc_features(Newbuffer, NewS2,
NewS3),add_template(Newbuffer, _Template1, NewS3, NewS4),add_features(NewS4, _Features1,
Newstack).

drop([[xmax, S, Features1, Template]|Reststack],[[First, Second]], Newstack,
Newbuffer):-drop1([[xmax, S, Features1, Template]|Reststack],[[First, Second]], NewS1,
Newbuffer),
create_max(xmax, NewS1, NewS2),perc_features(Newbuffer, NewS2,
NewS3),add_template(Newbuffer, _Template1, NewS3, NewS4),add_features(NewS4, _Features2,
Newstack).

drop([[xmax, S, Features,Template]|Reststack],[[First, Second]|Rest], Newstack,
Newbuffer):-drop2([[xmax,S,Features,Template]|Reststack],[[First, Second]|Rest], Newstack,
Newbuffer).

drop([[xmax,S,Features,Template]|Reststack],[[First, Second]], Newstack,
Newbuffer):-drop2([[xmax, S,Features,Template]|Reststack],[[First, Second]], Newstack,
Newbuffer).

drop([[xmax,S,Features,Template,Ty]|Reststack],[[First, Second]], Newstack,
Newbuffer):-drop2([[xmax, S,Features,Template,Ty]|Reststack],[[First, Second]], Newstack,
Newbuffer).

drop([[xmax,S,'-n-v',_Template]|R],[[First,Second]|Rest],R,[[[xmax,S,'-n-v'],First]|[Second|Re
st]]).drop([[xmax,S, Features, _Template]|R],[[First, Second]|[X|L]],R,[[[xmax, S,
Features],First]|[Second,X|L]]).

drop([[xmax,S, Features, _Template]|R], [[First, Second]], R,[[[xmax, S, Features],
First]|[Second]]).

/*These rules, called by drop, remove words or phrases from stack into buffer*/

drop1([[xmax,S, Features, _Template]|[]], [[First, Second]],[],[[[xmax,S, Features],
First]|[Second]]).

drop1([[xmax, S, Features, _Template]|[]], [[First, Second]|[X|L]],[],[[[xmax,S, Features],
First]|[Second, X|L]]).

drop1([[xmax,S, Features1, Template]|Reststack], [[[to, '-n+v+a+p'], Second]], NewS1,
Newbuffer):-drop3([[xmax,S, Features1,Template]|Reststack],[[[to,'-n+v+a+p'], Second]],NewS,
Newbuffer),
add_feat_spec(NewS, Newbuffer, NewS1).

drop1([[xmax, S, Features, _Template]|R], [[First, Second]|[X|L]],R,[[[xmax, S, Features],
First]|[Second, X|L]]).

drop1([[xmax, S, Features, _Template]|R], [[First, Second]],R,[[[xmax, S, Features],
First]|[Second]]).

drop3([[xmax,S, Features1,_Template]|Reststack],[[[to,'-n+v+a+p'], Second]],Reststack,[[[xmax,S,
Features1], [to, '-n+v+a+p']]|[Second]]).

drop2([[xmax,S,Features, Template,Ty],[xmax, S1, Features1]|Reststack], [[First,
Second]|[X|L]],[[xmax, S1, Features1,
Template]|Reststack],[[[xmax,S,Features,Ty],First]|[Second,X|L]]).

drop2([[xmax,S,Features,Template,Ty],[xmax, S1, Features1]|Reststack], [[First,
Second]],[[xmax, S1, Features1,
Template]|Reststack],[[[xmax,S,Features,Ty],First]|[Second]]).drop2([[xmax,S,Features,
_Template,Ty]|Reststack], [[First,
Second]|[X|L]],Reststack,[[[xmax,S,Features,Ty],First]|[Second,X|L]]).

drop2([[xmax,S,Features,_Template,Ty]|Reststack], [[First,
Second]],Reststack,[[[xmax,S,Features,Ty],First]|[Second]]).drop2([[xmax, S,Features,
Template],[xmax,[attach_embedded_subject,Entry],F]|R], [[First,
Second]],[[xmax,[attach_embedded_subject,Entry],F,Template]|R],[[[xmax,S,Features],First]|[S
econd]]).

drop2([[xmax,S,Features1, Template],[xmax, [attach_subject, Entry], Features]|L],[[First, Second]],
    [[xmax, [attach_subject, Entry], Features, Template]|L],[[[xmax,S, Features1],
First]|[Second]]).

drop2([[xmax,S,Features,Template],[xmax,S1, Features1],[xmax,
[attach_comp,Entry1],Features2,Template1]|R], [[First, Second]],[[xmax, S1, Features1,
Template],[xmax, [attach_comp,Entry1],Features2,Template1]|R],[[[xmax,
S,Features],First]|[Second]]).

drop2([[xmax,S,Features,_Template]|Reststack], [[First, Second]|[X|L]],Reststack,[[[xmax,
S,Features],First]|[Second,X|L]]).
```

```
drop2([[xmax,S,Features,_Template] | Reststack], [[First, Second]],Reststack,[[[xmax,S,Features],First] | [Second]]).
```

/*This rule, called by 'input' and 'parse', is used to build a noun-phrase from the contents of 2nd buffer cell if sentence is a Yes/No question or Whquestion with aux verb. The rules it calls are identical in function tosimilarly named counterparts in file parser*/

```
pre_test([[[Word, '-n+v'],[Word1, '+n-v+a-p']] | Rest],[[xmax, Features, [spec_, head, comp]]],Newbuffer, Newstack):-
create_max(xmax,[[xmax,Features,[spec_, head, comp]]], NewS),
perc_feat_s([[[Word,'-n+v'],[Word1, '+n-v+a-p']] | Rest], NewS, NewS1),add_template_s([[[Word,'-n+v'],[Word1,'+n-v+a-p']] | Rest], _Template, NewS1, NewS2),attach_s(NewS2,[[[Word,'-n+v'],[Word1,'+n-v+a-p']] | Rest],NewS3, NewB),
drop_s(NewS3, NewB, Newstack, Newbuffer).

pre_test([[[Word, '-n+v'],[Word1, '+n-v-a-p']] | Rest],[[xmax, Features, [spec_, head, comp]]],Newbuffer, Newstack):-
create_max(xmax,[[xmax,Features,[spec_, head, comp]]], NewS),perc_feat_s([[[Word,'-n+v+a+p'],[Word1, '+n-v-a-p']] | Rest], NewS, NewS1),add_template_s([[[Word,'-n+v'],[Word1,'+n-v-a-p']] | Rest], _Template, NewS1, NewS2),attach_s(NewS2,[[[Word,'-n+v'],[Word1,'+n-v-a-p']] | Rest],NewS3,NewB),attach_s1(NewS3,NewB,NewS4, NewB1),drop_s(NewS4, NewB1, Newstack, Newbuffer).

pre_test([[[Word,'-n+v'],[Word1, '+n-v+a-p',PL]] | Rest],[[xmax,Features,[spec_,head,comp]],[xmax,S,'-n-v+a+p',T,wh]], Newbuffer, Newstack):-
create_max(xmax,[[xmax,Features,[spec_, head, comp]],[xmax,S,'-n-v+a+p',T,wh]], NewS),perc_feat_s([[[Word,'-n+v'],[Word1, '+n-v+a-p',PL]] | Rest], NewS, NewS1),add_template_s([[[Word,'-n+v'],[Word1,'+n-v+a-p',PL]] | Rest], _Template, NewS1, NewS2),attach_s(NewS2,[[[Word,'-n+v'],[Word1,'+n-v+a-p',PL]] | Rest],
NewS3, NewB),
drop_s(NewS3, NewB, Newstack, Newbuffer).

pre_test([[[Word,'-n+v'],[Word1, '+n-v-a-p']] | Rest],
[[xmax,Features,[spec_,head,comp]],[xmax,S,'-n-v+a+p',T,wh] | Reststack],Newbuffer, Newstack):-create_max(xmax,[[xmax,Features,[spec_, head, comp]],
[xmax,S,'-n-v+a+p',T,wh] | Reststack],NewS),perc_feat_s([[[Word,'-n+v'],[Word1, '+n-v-a-p']] | Rest], NewS, NewS1),add_template_s([[[Word,'-n+v'],[Word1,'+n-v-a-p']] | Rest],
        _Template,NewS1,NewS2),attach_s(NewS2,[[[Word,'-n+v'],[Word1,'+n-v-a-p']] | Rest],
          NewS3,NewB),attach_s1(NewS3,NewB,NewS4, NewB1),drop_s(NewS4, NewB1,
Newstack, Newbuffer).

pre_test([[[Word, '-n+v+a+p'],[Word1, '+n-v+a-p',PL]] | Rest],[[xmax, Features, [spec_, head, comp]]],Newbuffer, Newstack):-
create_max(xmax,[[xmax,Features,[spec_, head, comp]]], NewS),
perc_feat_s([[[Word,'-n+v+a+p'],[Word1, '+n-v+a-p',PL]] | Rest], NewS, NewS1),add_template_s([[[Word,'-n+v+a+p'],[Word1,'+n-v+a-p',PL]] | Rest],_Template, NewS1, NewS2),attach_s(NewS2,[[[Word,'-n+v+a+p'],[Word1,'+n-v+a-p',PL]] | Rest],NewS3, NewB),
drop_s(NewS3, NewB, Newstack, Newbuffer).pre_test([[[Word, '-n+v+a+p'],[Word1, '+n-v-a-p']] | Rest],[[xmax, Features, [spec_, head, comp]]],Newbuffer, Newstack):-
create_max(xmax,[[xmax,Features,[spec_, head, comp]]], NewS),
perc_feat_s([[[Word,'-n+v+a+p'],[Word1, '+n-v-a-p']] | Rest], NewS, NewS1),add_template_s([[[Word,'-n+v+a+p'],[Word1,'+n-v-a-p']] | Rest], _Template, N e w S 1 , NewS2),attach_s(NewS2,[[[Word,'-n+v+a+p'],[Word1,'+n-v-a-p']] | Rest],NewS3,NewB),attach_s1(NewS3,NewB,NewS4, NewB1),drop_s(NewS4, NewB1, Newstack, Newbuffer).

pre_test([[[Word,'-n+v+a+p'],[Word1,'+n-v+a-p',PL]] | Rest],[[xmax,Features,[spec_,head,comp]
],[xmax,S,'-n-v+a+p',T,wh]], Newbuffer, Newstack):-create_max(xmax,[[xmax,Features,[spec_,
head, comp]],[xmax,S,'-n-v+a+p',T,wh]], NewS),perc_feat_s([[[Word,'-n+v+a+p'],[Word1,
'+n-v+a-p',PL]]|Rest], NewS,
NewS1),add_template_s([[[Word,'-n+v+a+p'],[Word1,'+n-v+a-p',PL]]|Rest],_Template,NewS1,
NewS2),attach_s(NewS2,[[[Word,'-n+v+a+p'],[Word1,'+n-v+a-p',PL]]|Rest], NewS3, NewB),
drop_s(NewS3, NewB, Newstack, Newbuffer).

pre_test([[[Word,'-n+v+a+p'],[Word1, '+n-v-a-p']]|Rest],
[[xmax,Features,[spec_,head,comp]],[xmax,S,'-n-v+a+p',T,wh]|Reststack],Newbuffer, Newstack):-
create_max(xmax,[[xmax,Features,[spec_, head, comp]],
[xmax,S,'-n-v+a+p',T,wh]|Reststack],NewS),perc_feat_s([[[Word,'-n+v+a+p'],[Word1,
'+n-v-a-p']]|Rest],NewS,NewS1),add_template_s([[[Word,'-n+v+a+p'],[Word1,'+n-v-a-p']]|Rest],
_Template, NewS1,
NewS2),attach_s(NewS2,[[[Word,'-n+v+a+p'],[Word1,'+n-v-a-p']]|Rest],
NewS3,NewB),attach_s1(NewS3,NewB,NewS4, NewB1),
drop_s(NewS4, NewB1, Newstack, Newbuffer).

pre_test(Buffer, Stack, Buffer, Stack).

perc_feat_s([[[_Word,_Feat],[_Word1, '+n-v+a-p',PL]]|_Rest],[[xmax], [xmax, Features, [spec_,
head, comp]]|Reststack], [[xmax, '+n-v+a-p',PL], [xmax, Features, [spec_, head,
comp]]|Reststack]).

perc_feat_s([[[_Word, _Feat],[_Word1,'+n-v-a-p']]|_Rest],[[xmax], [xmax, Features, [spec_, head,
comp]]|Reststack],[[xmax, '+n-v+a-p'], [xmax, Features, [spec_, head, comp]]|Reststack]).

add_template_s([[[_Word,_Feat],[_Word1,'+n-v+a-p',_PL]]|_Rest], [spec, head_, comp], [[xmax,
Features1,_PL1],[xmax, Features, Template]|Reststack],[[xmax, Features1, [spec,head_,comp]],
[xmax, Features, Template]|Reststack]).

add_template_s([[[_Word, _Feat],[_Word1, '+n-v-a-p']]|_Rest], [spec_, head, comp], [[xmax,
'+n-v+a-p',_PL],[xmax, Features, Template]|Reststack],[[xmax, '+n-v+a-p', [spec_,head,comp]],
[xmax, Features, Template]|Reststack]).

attach_s([[xmax,'+n-v+a-p', [spec,head_,comp]],[xmax, Features, Template]|Reststack],[[[Word,
Feat],[Word1, '+n-v+a-p',PL]]|[Third|Nr]],[[xmax,[Word1, '+n-v+a-p',PL], '+n-v+a-p',
[spec_,head,comp]],[xmax, Features, Template]|Reststack],[[[Word, Feat],Third]|Nr]).

attach_s([[xmax,'+n-v+a-p', [spec_,head,comp]],[xmax, Features, Template]|Reststack],[[[Word,
Feat],[Word1, '+n-v-a-p']]|[Third|Nr]],[[xmax,[Word1, '+n-v-a-p'], '+n-v+a-p',
[spec_,head,comp]],[xmax, Features, Template]|Reststack],[[[Word, Feat],Third]|Nr]).

attach_s1([[xmax,Structure,'+n-v+a-p',[spec_,head,comp]],[xmax, Features,
Template]|Reststack],[[[Word, Feat],[Word1, '+n-v+a+p']]|[Third|Nr]],Newstack,
Newbuffer):-attach_s2([[xmax, Structure,'+n-v+a-p',[spec_,head,comp]],[xmax, Features,
Template]|Reststack],[[[Word, Feat],[Word1,'+n-v+a+p']]|[Third|Nr]],NewS1,
NewB),member([_Entry, '+n-v+a+p'], NewB),attach_s1(NewS1, NewB,Newstack, Newbuffer).

attach_s1([[xmax,Structure,'+n-v+a-p',[spec_,head,comp]],[xmax, Features,
Template]|Reststack],[[[Word, Feat],[Word1, '+n-v+a+p']]|[Third|Nr]],Newstack,
Newbuffer):-attach_s2([[xmax, Structure,'+n-v+a-p',[spec_,head,comp]],[xmax,
Features,Template]|Reststack],[[[Word, Feat],[Word1,'+n-v+a+p']]|[Third|Nr]],NewS1,
NewB),attach_s3(NewS1,NewB,Newstack, Newbuffer).

attach_s1([[xmax,Structure,'+n-v+a-p',[spec_,head,comp]],[xmax, Features,
Template]|Reststack],[[[Word, Feat],[Word1,
'+n-v+a-p',PL]]|[Third|Nr]],[[xmax,[Structure,[Word1, '+n-v+a-p',PL]], '+n-v+a-p',
[spec_,head,comp]],[xmax, Features, Template]|Reststack],[[[Word, Feat],Third]|Nr]).
```

```
attach_s1([[xmax,Structure,'+n-v+a-p',[spec_,head,comp]],[xmax, Features, Template]|Reststack],[[[Word, Feat],[Word1, '+n-v+a-p',PL]]|[Third|Nr]], [[xmax,[Structure,[Word1, '+n-v+a-p',PL]], '+n-v+a-p', [spec_,head,comp]],[xmax, Features, Template]|Reststack],[[[Word, Feat],Third]|Nr]).

attach_s2([[xmax,Structure,'+n-v+a-p',[spec_,head,comp]],[xmax, Features, Template]|Reststack],[[[Word, Feat],[Word1, '+n-v+a+p']]|[Third|Nr]],[[xmax,[Structure,[Word1, '+n-v+a+p']], '+n-v+a-p', [spec_,head,comp]],[xmax, Features, Template]|Reststack],[[[Word, Feat],Third]|Nr]).

attach_s3([[xmax,Structure,'+n-v+a-p',[spec_,head,comp]],[xmax, Features, Template]|Reststack],[[[Word, Feat],[Word1, '+n-v+a-p',PL]]|[Third|Nr]],[[xmax,[Structure,[Word1, '+n-v+a-p',PL]], '+n-v+a-p', [spec_,head,comp]],[xmax, Features, Template]|Reststack],[[[Word, Feat],Third]|Nr]).

drop_s([[xmax,Structure, Features1, [spec_,head,comp]],[xmax, Features, Template]|Reststack],[[[Word, Feat],Second]|Nr],[[xmax, Features, Template]|Reststack],[[[Word,Feat],[xmax,Structure, Features1]]|[Second|Nr]]).
```

The rules below constitute the grammar.

```
/*These rules constitute the grammar. The rules unify with the state of thestack and buffer and either attach the contents of the first buffer cell to the stack, switch contents of first and second buffer cells or insert a traceor lexical item into the first buffer cell.*/

grammar_rule(attach_adj, [[xmax, '+n-v+a-p', [spec_, head, comp]]|Reststack],[[[Words, '+n-v+a+p'],[Word,'+n-v+a+p']]|Rest],Newstack, Newbuffer):-attach([[xmax,'+n-v+a-p', [spec_, head, comp]]|Reststack],[[[Words, '+n-v+a+p'],[Word, '+n-v+a+p']]|Rest],Newstack, Newbuffer).

grammar_rule(attach_adj, [[xmax, '+n-v+a-p',[spec_, head, comp]]|Reststack],[[[Words, '+n-v+a+p'],[Word, '+n-v+a-p',PL]]|Rest],Newstack, Newbuffer):-
attach([[xmax, '+n-v+a-p', [spec_, head, comp]]|Reststack],[[[Words, '+n-v+a+p'],[Word,'+n-v+a-p',PL]]|Rest],Newstack,Newbuffer).

grammar_rule(attach_adj, [[xmax,St, '+n-v+a-p',[spec_, head, comp]]|Reststack],[[[Words, '+n-v+a+p'],[Word, '+n-v+a+p']]|Rest],Newstack, Newbuffer):-
attach([[xmax,St, '+n-v+a-p', [spec_, head, comp]]|Reststack],[[[Words,'+n-v+a+p'],[Word,'+n-v+a+p']]|Rest],Newstack, Newbuffer).

grammar_rule(attach_adj, [[xmax,St,'+n-v+a-p', [spec_, head, comp]]|Reststack],[[[Words, '+n-v+a+p'],[Word, '+n-v+a-p',PL]]|Rest],Newstack, Newbuffer):-
attach([[xmax,St,'+n-v+a-p', [spec_, head, comp]]|Reststack],[[[Words,'+n-v+a+p'],[Word,'+n-v+a-p',PL]]|Rest],Newstack,Newbuffer).

grammar_rule(wh_insert,[[xmax,'-n+v+a+p',[spec_,head,comp]],[xmax,S1,F1,T,wh]],[[[Word,'-n+v',PN,T],Second]|Rest],Newstack,Newbuffer):-insert(_Trace,[[xmax,'-n+v+a+p',[spec_,head, comp]],[xmax,S1,F1,T,wh]],[[[Word,'-n+v',PN,T],Second]|Rest],Newstack,Newbuffer).

grammar_rule(wh_insert,[[xmax,'-n+v+a+p',[spec_,head,comp]],[xmax,S1,F1,T,wh]],[[[Word,'-n+v+a+p',PN,T],Second]|Rest],Newstack,
Newbuffer):-insert(_Trace,[[xmax,'-n+v+a+p',[spec_,head,comp]],[xmax,S1,F1,T,wh]],[[[Word,'-n+v+a+p',PN,T],Second]|Rest],Newstack,Newbuffer).

grammar_rule(insert_subjless_np,[[xmax,'-n+v+a+p',[spec_,head,comp]]|Reststack],[[[to,'-n+v+a+p'],[Word,'-n+v','-',tense]]|Rest],Newstack,Newbuffer):-
insert(_Trace,[[xmax,'-n+v+a+p',[spec_,head,comp]]
|Reststack],
[[[to,'-n+v+a+p'],[Word,'-n+v','-',tense]] | Rest],Newstack, Newbuffer).

grammar_rule(attach_det, [[xmax, '+n-v+a-p', [spec_, head, comp]] | Reststack],[[[[FWord,
'+n-v-a-p'],[FWord, '-n-v+a+p']],[Sw,'+n-v+a-p',sg]] | Rest],Newstack, Newbuffer):-
attach([[xmax, '+n-v+a-p', [spec_, head,comp]] | Reststack],[[[[FWord,
'+n-v-a-p'],[FWord,'-n-v+a+p']],
[Sw,'+n-v+a-p',sg]] | Rest],Newstack,Newbuffer).

grammar_rule(attach_det, [[xmax, '+n-v+a-p', [spec_, head, comp]] | Reststack],[[[FWord,
'+n-v-a-p'],Second] | Rest],
Newstack, Newbuffer):-
attach([[xmax, '+n-v+a-p', [spec_, head, comp]] | Reststack],
[[[FWord, '+n-v-a-p'],Second] | Rest],Newstack, Newbuffer).

grammar_rule(attach_possdet, [[xmax, '+n-v+a-p',[spec_,head, comp]] | Reststack],
[[[Words, '+n-v-a-p',PD],Second] | Rest],Newstack, Newbuffer):-
attach([[xmax, '+n-v+a-p', [spec_, head,
comp]] | Reststack],[[[Words,'+n-v-a-p',PD],Second] | Rest],Newstack,Newbuffer).

grammar_rule(attach_possdet,[[xmax,S,'+n-v+a-p',[spec_,head, comp]] | Reststack],[[[Words,
'+n-v-a-p',PD],Second] | Rest],Newstack,Newbuffer):-attach([[xmax,S,'+n-v+a-p', [spec_, head,
comp]] | Reststack],[[[Words, '+n-v-a-p',PD],Second] | Rest],Newstack,Newbuffer).

grammar_rule(attach_subject, [[xmax, '-n+v+a+p', [spec_, head,comp]]],[[[xmax, E,
'-n-v+a+p'],[Word, '-n+v',PN,Tense]] | Rest],Newstack, Newbuffer):-
attach([[xmax, '-n+v+a+p', [spec_, head,
comp]]],[[[xmax,E,'-n-v+a+p'],[Word,'-n+v',PN,Tense]] | Rest],Newstack,Newbuffer).

grammar_rule(attach_subject,[[xmax,'-n+v+a+p',[spec_, head,
comp]]],[[[xmax,E,'+n-v+a-p'],[W,F,PN,T]] | Rest],Newstack,
Newbuffer):-
attach([[xmax,'-n+v+a+p',[spec_,head,comp]]],[[[xmax,E,'+n-v+a-p'],[W,F,PN,T]] | Rest],Newstack,
Newbuffer).

grammar_rule(attach_subject,[[xmax,'-n+v+a+p',[spec_, head,
comp]]],[[[xmax,E,'+n-v+a-p'],[W,'-n+v']] | Rest],Newstack,
Newbuffer):-
attach([[xmax,'-n+v+a+p',[spec_,head,comp]]],[[[xmax,E,'+n-v+a-p'],[W,'-n+v']] | Rest],Newstack,
Newbuffer).

grammar_rule(attach_subject,[[xmax,'-n+v+a+p',[spec_, head,
comp]]],[[[xmax,E,'+n-v+a-p'],[W,'-n+v+a+p']] | Rest],Newstack,
Newbuffer):-
attach([[xmax,'-n+v+a+p',[spec_, head,
comp]]],[[[xmax,E,'+n-v+a-p'],[W,'-n+v+a+p']] | Rest],Newstack,Newbuffer).

grammar_rule(attach_subject,[[xmax,'-n+v+a+p',[spec_, head,
comp]]],[[[xmax,E,'+n-v+a-p'],S] | Rest],Newstack,Newbuffer):-attach([[xmax,'-n+v+a+p',[spec_,
head, comp]]],[[[xmax,E,'+n-v+a-p'],S] | Rest],Newstack,Newbuffer).

grammar_rule(attach_embedded_subject,[[xmax,'-n+v+a+p',[spec_,head, comp]] | Reststack],
[[[xmax,E,'+n-v+a-p'],[W,F,PN,T]] | Rest],Newstack,
Newbuffer):-
attach([[xmax,'-n+v+a+p',[spec_,
head,comp]] | Reststack],[[[xmax,E,'+n-v+a-p'],[W,F,PN,T]] | Rest],Newstack,
Newbuffer).

grammar_rule(attach_embedded_subject,[[xmax,'-n+v+a+p',[spec_, head,
comp]]|Reststack],[[[xmax,E,'+n-v+a-p'],[W,'-n+v']]|Rest],Newstack,
Newbuffer):-
attach([[xmax,'-n+v+a+p',[spec_,
head,comp]]|Reststack],[[[xmax,E,'+n-v+a-p'],[W,'-n+v']]|Rest],Newstack,
Newbuffer).

grammar_rule(attach_embedded_subject,[[xmax,'-n+v+a+p',[spec_, head,
comp]]|Reststack],[[[xmax,E,'+n-v+a-p'],[W,'-n+v+a+p']]|Rest],Newstack,
Newbuffer):-
attach([[xmax,'-n+v+a+p',[spec_,
head,comp]]|Reststack],[[[xmax,E,'+n-v+a-p'],[W,'-n+v+a+p']]|Rest],Newstack,Newbuffer).

grammar_rule(attach_embedded_subject, [[xmax, '-n+v+a+p', [spec_, head, comp]]|Reststack],
[[[Words, '+n-v+a-p'],[Word,'-n+v+a+p']]|Rest],Newstack, Newbuffer):-
attach([[xmax, '-n+v+a+p', [spec_, head,
comp]]|Reststack],[[[Words,'+n-v+a-p'],[Word,'-n+v+a+p']]|Rest], Newstack, Newbuffer).

grammar_rule(attach_embedded_subject, [[xmax, '-n+v+a+p', [spec_, head, comp]]|Reststack],
[[[xmax,E,'+n-v+a-p'],[[Word,'+n-v-a-p'],[Word,'-n-v+a+p']]]|Rest],Newstack, Newbuffer):-
attach([[xmax, '-n+v+a+p', [spec_, head, comp]]|Reststack],
[[[xmax,E,'+n-v+a-p'],[[Word,'+n-v-a-p'],[Word,'-n-v+a+p']]]|Rest], Newstack, Newbuffer).

grammar_rule(attach_embedded_subject, [[xmax, '-n+v+a+p', [spec_, head, comp]],[xmax,S,
'-n-v+a+p',Template1,Ty]|Reststack],[[[Words,'+n-v+a-p'],[Word,'-n+v',PN,T]]|Rest],Newstack,
Newbuffer):-
attach([[xmax, '-n+v+a+p', [spec_, head, comp]],
[xmax,S,'-n-v+a+p',Template1,Ty]|Reststack],[[[Words,'+n-v+a-p'],[Word,'-n+v',PN,T]]|Rest],
Newstack, Newbuffer).

grammar_rule(subj_aux_inv,[[xmax, '-n+v+a+p', [spec_, head, comp]]|Reststack],
[[[Word, F],[xmax, Word1, '+n-v+a-p']]|Rest],Newstack, Newbuffer):-
switch([[xmax, '-n+v+a+p', [spec_, head, comp]]|Reststack],[[[Word, F],[xmax,
Word1,'+n-v+a-p']]|Rest],Newstack, Newbuffer).

grammar_rule(imperative,[[xmax,'-n+v+a+p',[spec_, head, comp]]],
[[[Words, '-n+v-a+p','-', tense],Second]|Rest],Newstack, Newbuffer):-
insert([you,'+n-v+a-p',pn],[[xmax,'-n+v+a+p',[spec_, head,
comp]]],[[[Words,'-n+v-a+p','-',tense],Second]|Rest],Newstack,Newbuffer).

grammar_rule(imperative,[[xmax,'-n+v+a+p',[spec_, head, comp]]],[[[[Words, '-n+v-a+p','-',
tense],[Words,'+n-v+a-p',PL]],Second]|Rest],Newstack, Newbuffer):-
insert([you,'+n-v+a-p',pn],[[xmax,'-n+v+a+p',[spec_, head,
comp]]],[[[[Words,'-n+v-a+p','-',tense],[Words,'+n-v+a-p',PL]],Second]|Rest],Newstack,Newbu
ffer).

grammar_rule(imperative,[[xmax,'-n+v+a+p',[spec_, head, comp]]],[[[[Words, '-n+v-a+p','-',
tense],[Word,'+n-v+a-p']],Second]|Rest],Newstack, Newbuffer):-
insert([you,'+n-v+a-p',pn],[[xmax,'-n+v+a+p',[spec_, head,
comp]]],[[[[Words,'-n+v-a+p','-',tense],[Word,'+n-v+a-p']],Second]|Rest],Newstack,Newbuffer).

grammar_rule(perfective, [[xmax, '-n+v',[spec, head_, comp]]|Reststack],[[[_word, '-n+v'],[Words,
'-n+v','+', en]]|Rest],Newstack, Newbuffer):-
attach([[xmax,'-n+v', [spec, head_,comp]]|Reststack],[[[_word, '-n+v'],[Words, '-n+v',
'+',en]]|Rest],Newstack, Newbuffer).

grammar_rule(perfective, [[xmax, '-n+v',[spec, head_, comp]]|Reststack],[[[_word,
'-n+v'],[Words,'-n+v','+',ed]]|Rest],Newstack, Newbuffer):-attach([[xmax,'-n+v', [spec, head_,
comp]]|Reststack],[[[_word, '-n+v'],[Words, '-n+v', '+', ed]]|Rest],Newstack,
Newbuffer).

grammar_rule(to_infinitive,[[xmax, '-n+v+a+p',[spec, head_, comp]]|Reststack],[[[to,
'-n+v+a+p'],[Words, '-n+v', '-', tense]]|Rest], Newstack,
Newbuffer):-attach([[xmax,'-n+v+a+p',[spec,head_, comp]]|Reststack], [[[to,
'-n+v+a+p'],[Words, '-n+v', '-',tense]]|Rest],
Newstack,Newbuffer).

grammar_rule(progressive,[[xmax,'-n+v',[spec,head_,comp]]|Reststack],[[[_word, '-n+v'],[Words,
'-n+v','+',ing]]|Rest],Newstack, Newbuffer):-attach([[xmax,'-n+v', [spec, head_,
comp]]|Reststack],[[[_word, '-n+v'],[Words, '-n+v','+',ing]]|Rest],Newstack, Newbuffer).

grammar_rule(progressive,[[xmax,'-n+v',[spec,head_,comp]]|Reststack],[[[be, '-n+v', '+',
en],[Words, '-n+v', '+',ing]]|Rest],Newstack, Newbuffer):-
attach([[xmax,'-n+v', [spec, head_, comp]]|Reststack],
[[[be, '-n+v','+',en],[Words,'-n+v','+',ing]]|Rest],Newstack, Newbuffer).

grammar_rule(copula,[[xmax,'-n+v',[spec,head_,comp]]|Reststack],
[[[_word, '-n+v'],Second]|Rest],Newstack,Newbuffer):-attach([[xmax,'-n+v', [spec, head_,
comp]]|Reststack],[[[_word, '-n+v'],Second]|Rest],Newstack, Newbuffer).

grammar_rule(aux_sai, [[xmax, '-n+v+a+p', [spec, head_, comp]]|Reststack],[[[Word,
'-n+v+a+p'],[Words, '-n+v', '-', tense]]|Rest], Newstack, Newbuffer):-attach([[xmax,'-n+v+a+p',
[spec, head_, comp]]|Reststack],[[[Word,'-n+v+a+p'],[Words,'-n+v','-',tense]]|Rest], Newstack,
Newbuffer).

grammar_rule(aux_sai, [[xmax, '-n+v', [spec, head_, comp]]|Reststack],[[[Word, '-n+v'],[Words,
'-n+v', '-', tense]]|Rest], Newstack, Newbuffer):-
attach([[xmax,'-n+v', [spec, head_,
comp]]|Reststack],[[[Word,'-n+v'],[Words,'-n+v','-',tense]]|Rest], Newstack, Newbuffer).

grammar_rule(modal, [[xmax, '-n+v+a+p', [spec, head_, comp]]|Reststack],
[[[Word,'-n+v+a+p'],[Words,'-n+v','-',tense]]|Rest],Newstack,Newbuffer):-
attach([[xmax,'-n+v+a+p', [spec, head_,comp]]|Reststack],[[[Word,'-n+v+a+p'],[Words, '-n+v',
'-',tense]]|Rest], Newstack, Newbuffer).

grammar_rule(modal, [[xmax, '-n+v+a+p', [spec, head_, comp]]|Reststack],[[[Word,
'-n+v+a+p'],[Words, '-n+v']]|Rest], Newstack, Newbuffer):-attach([[xmax,'-n+v+a+p', [spec,
head_, comp]]|Reststack],[[[Word,'-n+v+a+p'],[Words,'-n+v']]|Rest],Newstack,
Newbuffer).

grammar_rule(modal, [[xmax, '-n+v+a+p', [spec, head_,
comp]]|Reststack],[[[[Word,'+n-v+a-p',PL],[Word, '-n+v+a+p']],[be, '-n+v']]|Rest],
Newstack,Newbuffer):-attach([[xmax,'-n+v+a+p', [spec, head_,
comp]]|Reststack],[[[[Word,'+n-v+a-p',PL],[Word,'-n+v+a+p']],[be,
'-n+v']]|Rest],Newstack,Newbuffer).

grammar_rule(do,[[xmax,'-n+v+a+p',[spec,head_comp]],[xmax,Word,'-n+v+a+p']],[[[does,
'-n+v+a+p'],[Words, '-n+v-a+p', '-', tense]]|Rest],Newstack, Newbuffer):-
attach([[xmax,'-n+v+a+p',[spec,head_,comp]],
[xmax,Word,'-n+v+a+p']],[[[does,'-n+v+a+p'],[Words,'-n+v-a+p','-',tense]]|Rest], Newstack,
Newbuffer).


grammar_rule(do, [[xmax,'-n+v+a+p',[spec,head_,comp]],[xmax,Word,'-n+v+a+p']], [[[did,
'-n+v+a+p'],[Words, '-n+v-a+p', '-', tense]]|Rest],
Newstack, Newbuffer):-attach([[xmax,'-n+v+a+p',[spec,head_,comp]],
[xmax,Word,'-n+v+a+p']],
[[[did, '-n+v+a+p'],[Words, '-n+v-a+p','-',tense]] | Rest], Newstack, Newbuffer).

grammar_rule(passive_be, [[xmax, '-n+v+a+p',[spec, head_, comp]] | Reststack],
[[[_word,'-n+v'],[Words,'-n+v-a+p','+',T]] | Rest],Newstack, Newbuffer):-
attach([[xmax, '-n+v+a+p',[spec, head_,comp]] | Reststack],[[[_word, '-n+v'], [Words,
'-n+v-a+p','+',T]] | Rest],Newstack, Newbuffer).

grammar_rule(passive_be, [[xmax, '-n+v+a+p',[spec, head_, comp]] | Reststack],[[[be, '-n+v', '+',
PN], [Words, '-n+v-a+p','+',T]] | Rest],Newstack, Newbuffer):-
attach([[xmax, '-n+v+a+p',[spec, head_, comp]] | Reststack],[[[be, '-n+v', '+', PN], [Words,
'-n+v-a+p','+',T]] | Rest],Newstack, Newbuffer).

grammar_rule(attach_rpron,[[xmax, '-n-v+a+p', [spec, head_,
comp]],[xmax,S,'+n-v+a-p',T] | Reststack],[[[Words, '-n-v+a+p'],Second] | Rest],Newstack,
Newbuffer):-attach([[xmax,'-n-v+a+p',[spec, head_,
comp]],[xmax,S,'+n-v+a-p',T] | Reststack],[[[Words,'-n-v+a+p'],Second] | Rest],Newstack,Newbu
ffer).

grammar_rule(attach_rpron,[[xmax, '-n-v+a+p', [spec, head_,
comp]],[xmax,S,'+n-v+a-p',T] | Reststack],[[[[Words, '+n-v-a-p'],[Words,
'-n-v+a+p']],Second] | Rest],Newstack, Newbuffer):-
attach([[xmax,'-n-v+a+p',[spec, head_,comp]],[xmax,S,'+n-v+a-p',T] | Reststack],
[[[[Words,'+n-v-a-p'],[Words,'-n-v+a+p']],Second] | Rest],Newstack,Newbuffer).

grammar_rule(attach_comp, [[xmax, '-n-v+a+p', [spec, head_, comp]] | Reststack],
[[[[Words,'+n-v-a-p'],[Words,'-n-v+a+p']],Second] | Rest],Newstack, Newbuffer):-
attach([[xmax,'-n-v+a+p',[spec, head_, comp]] | Reststack],
[[[[Words, '+n-v-a-p'],[Words,'-n-v+a+p']],Second] | Rest],
Newstack, Newbuffer).

grammar_rule(attach_comp, [[xmax, '-n-v+a+p', [spec, head_, comp]] | Reststack],
[[[that, '-n-v+a+p'],Second] | Rest],Newstack,Newbuffer):-attach([[xmax,'-n-v+a+p',[spec, head_,
comp]] | Reststack],[[[that, '-n-v+a+p'], Second] | Rest],Newstack,Newbuffer).

grammar_rule(attach_wh_comp,[[xmax,'-n-v+a+p',[spec,head_,comp]] | Reststack],[[[Wh,'-n-v+
a+p'],[W,'+n-v+a-p']] | Rest],Newstack,Newbuffer):-attach([[xmax,'-n-v+a+p',[spec,head_,comp
]] | Reststack],[[[Wh,'-n-v+a+p'],[W,'+n-v+a-p']] | Rest],Newstack,Newbuffer).

grammar_rule(attach_wh_comp,[[xmax,'-n-v+a+p',[spec,head_,comp]] | Reststack],[[[Wh,'-n-v+
a+p'],[W,'-n+v']] | Rest],Newstack,Newbuffer):-attach([[xmax,'-n-v+a+p',[spec,head_,comp]] | R
eststack],[[[Wh,'-n-v+a+p'],[W,'-n+v']] | Rest],Newstack,Newbuffer).

grammar_rule(attach_wh_comp,[[xmax,'-n-v+a+p',[spec,head_,comp]] | Reststack],[[[Wh,'-n-v+
a+p'],[W,'-n+v+a+p']] | Rest],Newstack,Newbuffer):-attach([[xmax,'-n-v+a+p',[spec,head_,comp
]] | Reststack],[[[Wh,'-n-v+a+p'],[W,'-n+v+a+p']] | Rest],Newstack,Newbuffer).

grammar_rule(attach_propnoun,[[xmax,'+n-v+a-p',[spec, head_, comp]] | Reststack],[[[Words,
'+n-v+a-p',pn], Second] | Rest],Newstack, Newbuffer):-
attach([[xmax, '+n-v+a-p', [spec, head_,comp]] | Reststack],[[[Words,
'+n-v+a-p',pn],Second] | Rest],Newstack, Newbuffer).

grammar_rule(attach_propnoun,[[xmax,St,'+n-v+a-p',[spec, head, comp_]] | Reststack],[[[Words,
'+n-v+a-p',pn], Second] | Rest],Newstack, Newbuffer):-
attach([[xmax, St,'+n-v+a-p', [spec, head,comp_]]
| Reststack],
[[[Words, '+n-v+a-p',pn],Second] | Rest],Newstack, Newbuffer).

grammar_rule(attach_noun, [[xmax,St,'+n-v+a-p',[spec, head_, comp]] | Reststack],[[[Words,
'+n-v+a-p',sg],Second] | Rest],Newstack, Newbuffer):- attach([[xmax, St, '+n-v+a-p',[spec, head_,
comp]] | Reststack],[[[Words,'+n-v+a-p',sg],Second] | Rest],Newstack, Newbuffer).

grammar_rule(attach_noun, [[xmax,'+n-v+a-p',[spec, head_, comp]] | Reststack],[[[Words,
'+n-v+a-p',sg],Second] | Rest],Newstack, Newbuffer):- attach([[xmax,'+n-v+a-p',[spec, head_,
comp]] | Reststack],[[[Words, '+n-v+a-p',sg],Second] | Rest],Newstack, Newbuffer).

grammar_rule(attach_noun, [[xmax,St,'+n-v+a-p',[spec, head_, comp]] | Reststack],[[[Words,
'+n-v+a-p',pl],Second] | Rest],Newstack, Newbuffer):- attach([[xmax, St, '+n-v+a-p',[spec, head_,
comp]] | Reststack],[[[Words,'+n-v+a-p',pl],Second] | Rest],Newstack, Newbuffer).

grammar_rule(attach_noun, [[xmax,'+n-v+a-p',[spec, head_, comp]] | Reststack],[[[Words,
'+n-v+a-p',pl],Second] | Rest],Newstack, Newbuffer):- attach([[xmax, '+n-v+a-p',[spec, head_,
comp]] | Reststack],[[[Words, '+n-v+a-p',pl],Second] | Rest],Newstack, Newbuffer).

grammar_rule(attach_noun, [[xmax,St,'+n-v+a-p',[spec, head_, comp]] | Reststack],[[[[Words,
'+n-v+a-p',SP],[Words,'-n+v+a+p']],Second] | Rest],Newstack, Newbuffer):-
attach([[xmax, St, '+n-v+a-p',[spec,head_,comp]]
| Reststack],
[[[[Words,'+n-v+a-p',SP],[Words,'-n+v+a+p']],Second] | Rest],Newstack, Newbuffer).

grammar_rule(attach_noun, [[xmax,St,'+n-v+a-p',[spec, head_,
comp]] | Reststack],[[[[Words,'-n+v-a+p','-',tense],[Words, '+n-v+a-p',SP]],Second] | Rest],Newstack,
Newbuffer):-attach([[xmax, St, '+n-v+a-p',[spec, head_ comp]]
| Reststack],
[[[[Words,'-n+v-a+p','-',tense],[Words,'+n-v+a-p',SP]],Second] | Rest],Newstack, Newbuffer).

grammar_rule(attach_prep, [[xmax, '-n-v', [spec, head_ comp]] | Reststack],
[[[Words, '-n-v'],[W, '+n-v+a-p']] | Rest],Newstack, Newbuffer):-
attach([[xmax,'-n-v',[spec, head_,comp]] | Reststack],[[[Words,'-n-v'],[W,
'+n-v+a-p']] | Rest],Newstack, Newbuffer).

grammar_rule(attach_prep, [[xmax, '-n-v', [spec, head_ comp]] | Reststack],
[[[Words, '-n-v'],[W, '+n-v+a-p',PL]] | Rest],Newstack, Newbuffer):-
attach([[xmax,'-n-v',[spec, head_,comp]] | Reststack],[[[Words,'-n-v'],[W,
'+n-v+a-p',PL]] | Rest],Newstack, Newbuffer).

grammar_rule(attach_prep, [[xmax,S, '-n-v',[spec, head_ comp]] | Reststack],
[[[Words, '-n-v'], [W, '+n-v+a-p']] | Rest],Newstack, Newbuffer):-
attach([[xmax,S,'-n-v',[spec,head_,comp]] | Reststack],[[[Words, '-n-v'],[W,
'+n-v+a-p']] | Rest],Newstack,Newbuffer).

grammar_rule(attach_prep, [[xmax, '-n-v', [spec, head_ comp]] | Reststack],
[[[Words, '-n-v'],[W, '+n-v+a-p',pn]] | Rest],Newstack, Newbuffer):-
attach([[xmax,'-n-v',[spec, head_, comp]] | Reststack],         [[[Words,'-n-v'],[W,
'+n-v+a-p',pn]] | Rest],Newstack, Newbuffer).

grammar_rule(attach_prep, [[xmax,S, '-n-v',[spec, head_ comp]] | Reststack],
[[[Words, '-n-v'], [W, '+n-v+a-p',pn]] | Rest],Newstack, Newbuffer):-
attach([[xmax,S,'-n-v',[spec, head_,comp]] | Reststack],         [[[Words, '-n-v'],[W,
'+n-v+a-p',pn]] | Rest],
Newstack,Newbuffer).

grammar_rule(attach_prep, [[xmax, '-n-v', [spec, head_ comp]] | Reststack],
[[[Words, '-n-v'], [W, '+n-v-a-p']] | Rest],
Newstack, Newbuffer):-
attach([[xmax,'-n-v',[spec, head_,comp]] | Reststack],[[[Words,'-n-v'],[W,
'+n-v-a-p']] | Rest],Newstack, Newbuffer).
```

```
grammar_rule(attach_prep, [[xmax,S, '-n-v',[spec, head_, comp]] | Reststack],
[[[Words, '-n-v'], [W, '+n-v-a-p']] | Rest],Newstack, Newbuffer):-
attach([[xmax,S,'-n-v',[spec,head_,comp]] | Reststack],[[[Words, '-n-v'],[W,
'+n-v-a-p']] | Rest],Newstack,Newbuffer).

grammar_rule(attach_prep, [[xmax, '-n-v', [spec, head_, comp]] | Reststack],
[[[Words, '-n-v'], [W, '+n-v+a+p',PD]] | Rest],Newstack, Newbuffer):-
attach([[xmax,'-n-v',[spec,
head_,comp]] | Reststack],[[[Words,'-n-v'],[W,'+n-v+a+p',PD]] | Rest],Newstack, Newbuffer).

grammar_rule(attach_prep, [[xmax,S, '-n-v',[spec, head_, comp]] | Reststack],
[[[Words, '-n-v'], [W, '+n-v+a+p',PD]] | Rest],Newstack, Newbuffer):-attach([[xmax,S,'-n-v',[spec,
head_,comp]] | Reststack],
[[[Words,'-n-v'],[W, '+n-v+a+p',PD]] | Rest],Newstack,Newbuffer).

grammar_rule(attach_prep, [[xmax, '-n-v', [spec, head_, comp]] | Reststack],[[[Words, '-n-v'], [W,
'+n-v+a+p']] | Rest],Newstack, Newbuffer):-
attach([[xmax,'-n-v',[spec,
head_,comp]] | Reststack],[[[Words,'-n-v'],[W,'+n-v+a+p']] | Rest],Newstack, Newbuffer).

grammar_rule(attach_prep, [[xmax,S, '-n-v',[spec, head_, comp]] | Reststack],
[[[Words, '-n-v'], [W, '+n-v+a+p']] | Rest],Newstack, Newbuffer):-
attach([[xmax,S,'-n-v',[spec,head_,comp]] | Reststack],[[[Words,'-n-v'],[W,
'+n-v+a+p']] | Rest],Newstack,Newbuffer).

grammar_rule(attach_reduce_rel,[[xmax,'-n+v-a+p',[spec,head_,comp]],[xmax,S,'+n-v+a-p',T] |
Reststack],
[[[Words, '-n+v-a+p', PN, Tense],Second] | Rest],Newstack, Newbuffer):-
attach([[xmax,'-n+v-a+p',[spec, head_,
comp]],[xmax,S,'+n-v+a-p',T] | Reststack],[[[Words,'-n+v-a+p',PN,Tense],Second] | Rest],Newsta
ck,Newbuffer).

grammar_rule(attach_verb, [[xmax, '-n+v-a+p', [spec, head_, comp]] | Reststack],
[[[Words, '-n+v', PN, Tense],Second] | Rest],Newstack, Newbuffer):-
attach([[xmax, '-n+v-a+p', [spec, head_,comp]] | Reststack],[[[Words,
'-n+v',PN,Tense],Second] | Rest],Newstack,Newbuffer).

grammar_rule(attach_verb, [[xmax,'-n+v', [spec, head_, comp]] | Reststack],[[[Words, '-n+v', PN,
Tense],Second] | Rest], Newstack, Newbuffer):- attach([[xmax,'-n+v',[spec,
head_,comp]] | Reststack],
[[[Words, '-n+v',PN,Tense],Second] | Rest],Newstack, Newbuffer).

grammar_rule(attach_verb,[[xmax,'-n+v-a+p',[spec, head_, comp]] | Reststack],[[[Words,
'-n+v-a+p', PN, Tense],Second] | Rest],Newstack, Newbuffer):- attach([[xmax,'-n+v-a+p',[spec,
head_, comp]] | Reststack],[[[Words,'-n+v-a+p',PN,Tense],Second] | Rest],Newstack,Newbuffer).

grammar_rule(attach_pred_adj,[[xmax,St, '-n+v',[spec, head, comp_]] | Reststack],
[[[Words, '+n-v+a+p'],Second] | Rest],Newstack, Newbuffer):-attach([[xmax,St, '-n+v', [spec, head,
comp_]] | Reststack],[[[Words, '+n-v+a+p'],Second] | Rest],Newstack,Newbuffer).

grammar_rule(attach_infl,[[xmax,St,'-n+v+a+p',[spec,head,comp_]] | Reststack],[[[xmax,[[to_infi
nitive,[to,'-n+v+a+p']] | Remain],'-n+v+a+p'],[]] | Rest],Newstack,Newbuffer):-attach([[xmax,St,'
-n+v+a+p',[spec,
head,comp_]] | Reststack],[[[xmax,[[to_infinitive,[to,'-n+v+a+p']] | Remain],'-n+v+a+p'],[]] | Rest]
,Newstack,Newbuffer).

grammar_rule(wh_insert,[[xmax,[attach_verb,Entry],
'-n+v-a+p',[spec,head,comp_],Ty],[xmax,S,F,T],[xmax,S1,F1],
[xmax,S2,F2,T2,wh]],[[[],[]]],Newstack,Newbuffer):-insert(_Trace,[[xmax,[attach_verb,Entry],'-n
+v-a+p',[spec,head,comp_],Ty],[xmax,S,F,T],[xmax,S1,F1],[xmax,S2,F2,T2,wh]],[[[],[]]],Newstac
k,Newbuffer).

grammar_rule(attach_vp,[[xmax,[attach_subject,Entry],'-n+v+a+p',[spec,head,comp_]]],[[[xmax,
S,'-n+v'], Second] | Rest],Newstack, Newbuffer):-
attach([[xmax, [attach_subject,Entry],'-n+v+a+p',[spec, head, comp_]]],[[[xmax, S,'-n+v'],
Second] | Rest],Newstack, Newbuffer).

grammar_rule(attach_vp,[[xmax,[attach_subject,Entry],
'-n+v+a+p',[spec,head,comp_]] | Reststack],[[[xmax, S,'-n+v+a+p'], Second] | Rest],Newstack,
Newbuffer):-
attach([[xmax, [attach_subject,Entry], '-n+v+a+p',[spec, head,
comp_]] | Reststack],[[[xmax,S,'-n+v+a+p'], Second] | Rest],Newstack, Newbuffer).

grammar_rule(attach_vp,[[xmax,S,'-n+v+a+p',[spec,head,comp_]] | Reststack],[[[xmax,
S1,'-n+v-a+p',Ty], Second] | Rest],Newstack, Newbuffer):-
attach([[xmax, S, '-n+v+a+p',[spec, head,comp_]]
 | Reststack],[[[xmax, S1,'-n+v-a+p',Ty],Second] | Rest],
Newstack, Newbuffer).

grammar_rule(attach_vp,[[xmax,S,'-n+v',[spec,head,comp_]] | Reststack],[[[xmax,
S1,'-n+v-a+p',Ty], Second] | Rest],Newstack, Newbuffer):-
attach([[xmax, S, '-n+v',[spec, head, comp_]] | Reststack],[[[xmax,S1,'-n+v-a+p',Ty],
Second] | Rest],Newstack, Newbuffer).

grammar_rule(attach_vp,[[xmax,S,'-n-v+a+p',[spec,head,comp_]] | Reststack],[[[xmax,S1,'-n+v-a
+p',Ty], Second] | Rest],Newstack, Newbuffer):- attach([[xmax,S,'-n-v+a+p',[spec, head,
comp_]] | Reststack],[[[xmax, S1,'-n+v-a+p',Ty], Second] | Rest],Newstack, Newbuffer).

grammar_rule(attach_vp,[[xmax,S,'-n-v+a+p',[spec,head,comp_]] | Reststack],[[[xmax,
[[passive_be,E] | Remain],'-n+v+a+p'],Second] | Rest],Newstack, Newbuffer):-
attach([[xmax,S, '-n-v+a+p',[spec,
head,comp_]] | Reststack],[[[xmax,[[passive_be,E] | Remain],'-n+v+a+p'],Second] | Rest],Newstack,
Newbuffer).

grammar_rule(attach_vp,[[xmax,S,'-n+v+a+p',[spec,head,comp_]] | Reststack],[[[xmax,
S1,'-n+v+a+p'], Second] | Rest],Newstack, Newbuffer):-
attach([[xmax, S, '-n+v+a+p',[spec, head,comp_]]
 | Reststack],[[[xmax, S1,'-n+v+a+p'], Second] | Rest],Newstack, Newbuffer).

grammar_rule(attach_vp,[[xmax,S,'-n+v+a+p',[spec,head,comp_]] | Reststack],[[[xmax,
S1,'-n+v+a+p'], Second] | Rest],Newstack, Newbuffer):-
attach([[xmax, S, '-n+v+a+p',[spec, head, comp_]] | Reststack],
[[[xmax, S1,'-n+v+a+p'], Second] | Rest],Newstack, Newbuffer).

grammar_rule(attach_vp,[[xmax,S,'-n+v+a+p',[spec,head,comp_]] | Reststack],[[[xmax,
S1,'-n+v-a+p',Ty], Second] | Rest],Newstack, Newbuffer):-
attach([[xmax,S, '-n+v+a+p',[spec, head,comp_]] | Reststack],[[[xmax, S1,'-n+v-a+p',Ty],
Second] | Rest],Newstack, Newbuffer).

grammar_rule(attach_vp,[[xmax,S,'-n+v+a+p',[spec,head,comp_]] | Reststack],[[[xmax,
S1,'-n+v-a+p',Ty], Second] | Rest],Newstack, Newbuffer):-
attach([[xmax, S, '-n+v+a+p',[spec, head,comp_]] | Reststack],
[[[xmax,S1,'-n+v-a+p',Ty], Second] | Rest],Newstack, Newbuffer).

grammar_rule(attach_succ_aux,[[xmax,S,'-n+v+a+p',[spec,head,comp_]] | Reststack],
[[[xmax, S1, '-n+v'], Second] | Rest],Newstack, Newbuffer):-attach([[xmax,S,'-n+v+a+p',[spec,
head, comp_]] | Reststack],[[[xmax, S1,'-n+v'], Second] | Rest],Newstack, Newbuffer).

grammar_rule(attach_succ_aux,[[xmax,S,'-n+v+a+p',[spec,head,comp_]] | Reststack],[[[xmax,S1,
'-n+v+a+p'], Second] | Rest], Newstack,Newbuffer):-
attach([[xmax,S,'-n+v+a+p',[spec, head,
comp_]] | Reststack],[[[xmax,S1,'-n+v+a+p'],Second] | Rest],Newstack,Newbuffer).

grammar_rule(attach_succ_aux,[[xmax,S,'-n+v',[spec,head,comp_]] | Reststack],[[[xmax,S1,'-n+v'],
Second] | Rest],Newstack, Newbuffer):-
attach([[xmax,S,'-n+v',[spec,head,comp_]] | Reststack],[[[xmax,S1,'-n+v'],Second] | Rest],Newstack,
Newbuffer).

grammar_rule(attach_succ_aux,[[xmax,S,'-n+v',[spec,head,comp_]] | Reststack],
[[[xmax,S1, '-n+v+a+p'], Second] | Rest], Newstack, Newbuffer):-
attach([[xmax,S,'-n+v',[spec,
head,comp_]] | Reststack],[[[xmax,S1,'-n+v+a+p'],Second] | Rest],Newstack,Newbuffer).

grammar_rule(attach_sent, [[xmax,S,'-n-v+a+p', [spec, head,
comp_]] | Reststack],[[[xmax,S1,'-n+v+a+p'],Second] | Rest],Newstack,
Newbuffer):-attach([[xmax,S,'-n-v+a+p',[spec, head,
comp_]] | Reststack],[[[xmax,S1,'-n+v+a+p'],Second] | Rest],Newstack, Newbuffer).

grammar_rule(attach_sent,[[xmax,S,'-n-v+a+p',[spec,head,
comp_],wh] | Reststack],[[[xmax,S1,'-n+v+a+p'],Second] | Rest],Newstack,
Newbuffer):-attach([[xmax,S,'-n-v+a+p',[spec,head,comp_],wh] | Reststack],[[[xmax,S1,'-n+v+a+
p'],Second] | Rest],Newstack, Newbuffer).

grammar_rule(attach_sent, [[xmax,'-n+v-a+p',[spec, head,
comp_]] | Reststack],[[[xmax,S,'-n+v+a+p'],Second] | Rest],Newstack,Newbuffer):-attach([[xmax,
'-n+v-a+p',[spec, head, comp_]] | Reststack],[[[xmax,S, '-n+v+a+p'],Second] | Rest],Newstack,
Newbuffer).

grammar_rule(attach_sent, [[xmax,S,'-n-v+a+p', [spec, head, comp_]] | Reststack],[[[xmax,S1,
'-n+v+a+p'],Second] | Rest],Newstack, Newbuffer):-attach([[xmax,S,'-n-v+a+p',[spec, head,
comp_]] | Reststack],[[[xmax,S1,'-n+v+a+p'],Second] | Rest],Newstack, Newbuffer).

grammar_rule(attach_zcomp_sent,[[xmax,S,'-n+v-a+p',[spec,head,comp_],Ty] | Reststack],[[[xm
ax,S1,'-n+v+a+p'],Second] | Rest],Newstack, Newbuffer):-attach([[xmax,S,'-n+v-a+p',[spec,head,
comp_],Ty] | Reststack],[[[xmax,S1,'-n+v+a+p'],Second] | Rest],Newstack,Newbuffer).

grammar_rule(attach_zrpron_sent, [[xmax,S,'+n-v+a-p',[spec, head,
comp_]] | Reststack],[[[xmax,S1, '-n+v+a+p'],Second] | Rest],Newstack,Newbuffer):-
attach([[xmax,S,'+n-v+a-p', [spec, head,
comp_]] | Reststack],[[[xmax,S1,'-n+v+a+p'],Second] | Rest],Newstack,Newbuffer).

grammar_rule(attach_zrpron_sent, [[xmax,S,'+n-v+a-p',[spec, head, comp_]] | Reststack],
[[[xmax,S1,'-n+v-a+p',Ty],Second] | Rest],Newstack,Newbuffer):- attach([[xmax, S,'+n-v+a-p',[spec,
head, comp_]] | Reststack],[[[xmax,S1,'-n+v-a+p',Ty],Second] | Rest],Newstack,Newbuffer).

grammar_rule(attach_zrpron_sent, [[xmax,S,'+n-v+a-p',[spec, head,
comp_]] | Reststack],[[[xmax,S1, '-n+v-a+p'],Second] | Rest],Newstack,Newbuffer):- attach([[xmax,
S,'+n-v+a-p',[spec, head,
comp_]] | Reststack],[[[xmax,S1,'-n+v-a+p'],Second] | Rest],Newstack,Newbuffer).

grammar_rule(attach_comp_phr, [[xmax, [attach_verb, Entry], '-n+v-a+p',[spec, head,
comp_],Ty] | Reststack],[[[xmax, [[attach_comp,
Entry1] | Remain],'-n-v+a+p'],Second] | Rest],Newstack,Newbuffer):-
attach([[xmax, [attach_verb, Entry], '-n+v-a+p',[spec, head, comp_],Ty] | Reststack],[[[xmax,
[[attach_comp, Entry1] | Remain],'-n-v+a+p'],Second] | Rest],Newstack,Newbuffer).

grammar_rule(attach_relative_clause,[[xmax,S,'+n-v+a-p',[spec, head, comp_]] | Reststack],
[[[xmax,S1,'-n-v+a+p'],Second] | Rest],Newstack,
Newbuffer):-attach([[xmax,S,'+n-v+a-p',[spec,head,comp_]] | Reststack],[[[xmax,S1,'-n-v+a+p'],S
econd] | Rest],Newstack,Newbuffer).

grammar_rule(attach_object,[[xmax, St, '-n+v', [spec, head,
comp_]] | Reststack],[[[xmax,E,'+n-v+a-p'],Second] | Rest],Newstack, Newbuffer):- attach([[xmax,
St, '-n+v', [spec, head, comp_]] | Reststack],[[[xmax, E, '+n-v+a-p'],Second] | Rest],Newstack,
Newbuffer).


grammar_rule(attach_object,[[xmax,St,'-n+v-a+p',[spec,head,comp_],Ty] | Reststack],[[[xmax,E,'
+n-v+a-p'],Second] | Rest],Newstack, Newbuffer):- attach([[xmax, St,
'-n+v-a+p',[spec,head,comp_],Ty]
| Reststack],
[[[xmax,E, '+n-v+a-p'],Second] | Rest],Newstack, Newbuffer).

grammar_rule(attach_object,[[xmax, St, '-n+v', [spec, head,
comp_]] | Reststack],[[[xmax,E,'+n-v+a-p',pn],Second] | Rest],Newstack, Newbuffer):-
attach([[xmax, St, '-n+v', [spec, head,comp_]] | Reststack],[[[xmax, E,
'+n-v+a-p',pn],Second] | Rest],Newstack, Newbuffer).

grammar_rule(attach_object,[[xmax,St,'-n+v-a+p',[spec,head,comp_],Ty] | Reststack],[[[xmax,E,'
+n-v+a-p',pn],Second] | Rest],Newstack, Newbuffer):- attach([[xmax, St,
'-n+v-a+p',[spec,head,comp_],Ty]
| Reststack],
[[[xmax,E, '+n-v+a-p',pn],Second] | Rest],Newstack, Newbuffer).

grammar_rule(attach_pp,[[xmax,S,'-n+v-a+p',[spec, head,comp_],Ty] | Reststack],[[[xmax,
S1,'-n-v'],Second] | Rest],Newstack,Newbuffer):-attach([[xmax,S,'-n+v-a+p',[spec,head,comp_],T
y] | Reststack],[[[xmax, S1, '-n-v'],Second] | Rest], Newstack, Newbuffer).

grammar_rule(attach_pp,[[xmax,S,'-n+v-a+p',[spec, head,comp_]] | Reststack],[[[xmax,
S1,'-n-v'],Second] | Rest],Newstack,Newbuffer):-attach([[xmax,S,'-n+v-a+p',[spec,head,comp_]]
| Reststack],[[[xmax, S1, '-n-v'],Second] | Rest], Newstack, Newbuffer).

grammar_rule(attach_pp,[[xmax,S,'+n-v+a-p',[spec, head,comp_]] | Reststack],[[[xmax,
S1,'-n-v'],Second] | Rest],Newstack,Newbuffer):-attach([[xmax,S,'+n-v+a-p',[spec,head,comp_]]
| Reststack],[[[xmax, S1, '-n-v'],Second] | Rest], Newstack, Newbuffer).

grammar_rule(attach_pp,[[xmax,S,'-n+v',[spec,
head,comp_]] | Reststack],[[[xmax,S1,'-n-v'],Second] | Rest],Newstack,Newbuffer):-attach([[xmax
,S,'-n+v',[spec,head,comp_]] | Reststack],[[[xmax,S1,'-n-v'],Second] | Rest],Newstack,Newbuffer).

grammar_rule(attach_pp,[[xmax,S,'-n+v-a+p',[spec, head,comp_],Ty] | Reststack],
[[[xmax,S1,'-n-v+a+p'],Second] | Rest],Newstack,
Newbuffer):-attach([[xmax,S,'-n+v-a+p',[spec,head,comp_],Ty] | Reststack],[[[xmax,S1,'-n-v+a+
p'],Second] | Rest], Newstack, Newbuffer).

grammar_rule(attach_pp_object,[[xmax,S,'-n-v',[spec,head,comp_]] | Reststack],
[[[xmax,S1,'+n-v+a-p'],Second] | Rest],Newstack,Newbuffer):-attach([[xmax,S,'-n-v',[spec, head,
comp_]] | Reststack],[[[xmax,S1,'+n-v+a-p'],Second] | Rest],Newstack, Newbuffer).

grammar_rule(passive,[[xmax,[attach_verb,F],'-n+v-a+p',[spec,head,comp_],Ty],[xmax,[passive
_be, Entry1],Features,Template] | Reststack],[[First, Second] | Rest],Newstack, Newbuffer):-
insert(_Trace,[[xmax,[attach_verb,F],'-n+v-a+p',[spec,head,comp_],Ty],[xmax,[passive_be,Entry
1],Features,Template]|Reststack],  [[First, Second]|Rest],Newstack, Newbuffer).

grammar_rule(passive,[[xmax,[[attach_verb,F]|R],'-n+v-a+p',[spec,head,comp_],Ty],[xmax,[pas
sive_be, Entry1],Features,Template]|Reststack],[[First, Second]|Rest],Newstack,
Newbuffer):-insert(_Trace,[[xmax,[[attach_verb,F]|R],'-n+v-a+p',[spec,head,comp_],Ty],[xmax,[
passive_be,Entry1],Features,Template]|Reststack],  [[First, Second]|Rest],Newstack, Newbuffer).

/*This rule, called by grammar_rule, attaches item form first buffer cell to the stack.*/

attach([[xmax, Features1,Template]|Reststack],[[[[W,'+n-v-a-p'],
_First1],[S,'+n-v+a-p',sg]]],[[xmax, [W,'+n-v-a-p'], Features1,
Template]|Reststack],[[[S,'+n-v+a-p',sg], []]]).

attach([[xmax,Features1,Template]|Reststack],[[[[W,'+n-v-a-p'],
_First1],[S,'+n-v+a-p',sg]]|[X|L]],[[xmax, [W,'+n-v-a-p'], Features1,
Template]|Reststack],[[[S,'+n-v+a-p',sg], X]|L]).

attach([[xmax, Features1,Template]|Reststack],[[[[W,'-n+v+a+p'],
_First1],[be,'-n+v']]|[X|L]],[[xmax,[W,'-n+v+a+p'], Features1, Template]|Reststack],[[[be,'-n+v'],
X]|L]).

attach([[xmax, Features1, Template]|Reststack],[[[_First,
[W,'-n-v+a+p']],[S,'+n-v+a-p',pl]]|[X|L]],[[xmax, [W,'-n-v+a+p'], Features1,
Template]|Reststack],[[[S,'+n-v+a-p',pl], X]|L]).

attach([[xmax, S,Features1, Template]|Reststack],[[[_First,
[W,'+n-v+a-p',sg]],[W1,'-n+v']]|[X|L]],[[xmax, [S,[W,'+n-v+a-p',sg]], Features1,
Template]|Reststack],[[[W1,'-n+v'], X]|L]).

attach([[xmax,
S,Features1,Template]|Reststack],[[[[W,'+n-v+a-p',sg],_Se],[W1,'-n+v']]|[X|L]],[[xmax,
[S,[W,'+n-v+a-p',sg]], Features1, Template]|Reststack],[[[W1,'-n+v'], X]|L]).

attach([[xmax, Features1, Template]|Reststack],[[[_First,
[W,'-n-v+a+p']],[S,'+n-v+a-p',pn]]|[X|L]],[[xmax, [W,'-n-v+a+p'], Features1,
Template]|Reststack],[[[S,'+n-v+a-p',pn], X]|L]).

attach([[xmax, Features1, Template]|Reststack],[[[_First, [W,'-n-v+a+p']],S]|[X|L]],[[xmax,
[W,'-n-v+a+p'], Features1, Template]|Reststack],[[S,X]|L]).

attach([[xmax, Features1, Template]|Reststack],[[First, Second]|[X|L]],[[xmax, First, Features1,
Template]|Reststack],[[Second, X]|L]).

attach([[xmax, Features1, Template]|Reststack],[[First, Second]],[[xmax, First,
Features1,Template]|Reststack],[[Second, []]]).

attach([[xmax, Structure, Features,Template]|Reststack],[[First, Second]|[X|L]],[[xmax,
[Structure,First], Features, Template]|Reststack],[[Second, X]|L]).attach([[xmax, Structure,
Features, Template]|Reststack],[[First, Second]],[[xmax, [Structure,First], Features,
Template]|Reststack],[[Second, []]]).

attach([[xmax,Structure,Features,Template,Type]|Reststack],[[First, Second]|[X|L]],[[xmax,
[Structure,First], Features, Template,Type]|Reststack],[[Second, X]|L]).

/*This rule, called by 'grammar_rule', switches the contents of the two buffer cells.

*/switch([[xmax, '-n+v+a+p', [spec_,head,comp]]|Reststack],[[First, Second]|Rest],[[xmax,
'-n+v+a+p', [spec_, head, comp]]|Reststack],[[Second, First]|Rest]).
```

```
/*This rule, called by 'grammar_rule' insert either a trace or lexical item into the stack.*/

insert([np_empty,'+n-v+a-p',PL],[[xmax,S,'-n+v-a+p',[spec,head,comp_],Ty],[xmax,[passive_be,
Entry],Features,Template]|Reststack], [[[],
Second]|Rest],[[xmax,S,'-n+v-a+p',[spec,head,comp_],Ty],[xmax,[passive_be,Entry],Features,Te
mplate]|Reststack], [[[np_empty,'+n-v+a-p',PL],Second]|Rest]).

insert([np_empty,'+n-v+a-p',PL],[[xmax,'-n+v+a+p',[spec_,head,comp]]|Reststack],[[[to,
'-n+v+a+p'],[Word,'-n+v','-',tense]]|Rest],
[[xmax,'-n+v+a+p',[spec_,head,comp]]|Reststack],[[[np_empty,'+n-v+a-p',PL],[to,'-n+v+a+p']],
[Word,'-n+v','-',tense]|Rest]).

insert([np_empty,'+n-v+a-p',PL],[[xmax,[R,St],'-n+v-a+p',[spec,head,comp_],Ty],[xmax,S,F,T],[
xmax,S1,F1],[xmax,S2,F2,T2,wh]],[[[],[]]],[[xmax,[R,St],'-n+v-a+p',[spec,head,comp_],Ty],[xmax,
S,F,T],[xmax,S1,F1],[xmax,S2,F2,T2,wh]],[[[np_empty,'+n-v+a-p',PL],[]],[]]).

insert([np_empty,'+n-v+a-p',PL],[[xmax,'-n+v+a+p',[spec_,head,comp]],[xmax,S1,F1,T,wh]|Res
tstack],[[[Word,'-n+v',PN,Tense],Second]|Rest],[[xmax,'-n+v+a+p',[spec_,head,comp]],[xmax,S
1,F1,T,wh]|Reststack],[[[np_empty,'+n-v+a-p',PL],[Word,'-n+v',PN,Tense]]|[Second|Rest]]).

insert([np_empty,'+n-v+a-p',PL],[[xmax,'-n+v+a+p',[spec_,head,comp]],[xmax,S1,F1,T,wh]|Res
tstack],[[[Word,'-n+v+a+p'],Second]|Rest],[[xmax,'-n+v+a+p',[spec_,head,comp]],[xmax,S1,F1,
T,wh]|Reststack],[[[np_empty,'+n-v+a-p',PL],[Word,'-n+v+a+p']]|[Second|Rest]]).

insert([you,'+n-v+a-p',pn],[[xmax,'-n+v+a+p',[spec_,head,comp]]],[[[[Word,
'-n+v-a+p','-',tense],_],Second]|Rest],
[[xmax,'-n+v+a+p',[spec_,head,comp]]],[[[you,'+n-v+a-p',pn],[Word,'-n+v-a+p','-',tense]]|[Seco
nd|Rest]]).

insert([you,'+n-v+a-p',pn],[[xmax,'-n+v+a+p',Template]|Reststack],[[[[Word,'-n+v-a+p','-',tens
e],[Word1,
'+n-v+a-p']],Second]|Rest],[[xmax,Word,'-n+v+a+p',Template]|Reststack],[[[you,'+n-v+a-p',pn
],[[Word,'-n+v-a+p','-',tense],[Word1,'+n-v+a-p']]]|[Second|Rest]]),Entry1],Features,Template]).
```

APPENDIX B

This appendix contains code for the LParser and its grammar. The code below represent the parsing rules.

```
/*This file contains the parsing rule for LParser.*/

/*The top level rule 'run' calls the procedure for reading in text - 'readin'and the rule which
initiates the parsing process 'parse'.*/

run(NStack,Nbuffer):-
readin(P),
statistics(runtime,_),
add_last($,P,P1),
parse([],P1,NStack,Nbuffer),
statistics(runtime,[_,T]),
format('parse took ~3d sec.~n',[T]).

/*The rule 'parse' initiates the parsing procedure by calling the rule'match_state' which controls
the parsing by manipulating the interactionbetween stack, buffer and parse_table.*/

parse(Stack,[Input | RestBuffer],NStack,Nbuffer):-
match_state(_State,_State1,_States,Stack,[Input | RestBuffer]
,NStack,Nbuffer),!.

/*The rule 'match_state' deals with shifting constituents from the buffer,reducing the
constituents on the stack and accepting the result when parsingis complete. It calls 'state' which
represents states in the parse table toaid the interaction between stack and buffer.*/

match_state(State,_State1,[],Stack,[Input | Restbuffer],
Nstack,Nbuffer):-
state(State,A,B,C),
B == s,
(Input=[_H | _T],
check_categories(Input,Restbuffer,A,[Input1,Rest1]);
check_categories1(Input,Restbuffer,A,[Input1,Rest1])),
shift(Stack,[Input1 | Rest1],N1stack,N1buffer),
\+member(State,[]),
conc([State],[],OldStates),    match_state(C,State,OldStates,N1stack,N1buffer,Nstack,
Nbuffer).

match_state(State,_State1,States,Stack,[Input | Restbuffer],
Nstack,Nbuffer):-
state(State,A,B,C),
B  ==  s,          (Input=[_H | _T],check_categories(Input,Restbuffer,A,[Input1,Rest1]);
check_categories1(Input,Restbuffer,A,[Input1,Rest1])),
shift(Stack,[Input1 | Rest1],N1stack,N1buffer),
(member(State,States),OldStates=States;   conc([State],States,OldStates),
\+member(State,States)),    match_state(C,State,OldStates,N1stack,N1buffer,Nstack,
Nbuffer).

match_state(State,_State1,States,Stack,[Input | Restbuffer],
Nstack,Nbuffer):-          (\+state(State,A,B,C);last1(wh_phrase(_,_),Stack);
last(rpron(_,_),Stack);last(aux(_,_),Stack),
se_last(inf(_,_),Stack)),
checking(Stack,Input),!,
member(State2,States),
state(State2,A,B,C),
B  ==  s,          (Input=[_H | _T],check_categories(Input,Restbuffer,A,[Input1,Rest1]);
check_categories1(Input,Restbuffer,A,[Input1,Rest1])),
shift(Stack,[Input1 | Rest1],N1stack,N1buffer),
(member(State,States),OldStates=States;  conc([State],States,OldStates),\+member(State,States)),
 match_state(C,State,OldStates,N1stack,N1buffer,Nstack,
Nbuffer).

match_state(State,State1,States,Stack,[Input | Restbuffer],
Nstack,Nbuffer):-
check_verb_rpron(Stack),   \+state(State,A,B,_C),state(State,A,B),
(member(State,States),OldStates=States;   conc([State],States,OldStates)),
state(State1,_A1,_B1,C1),\+atom(C1),C1=\=State,          \+member(C1,OldStates),
match_state(C1,State1,OldStates,Stack,[Input | Restbuffer],
Nstack,Nbuffer).

match_state(State,_State1,States,Stack,[Input | Restbuffer],
Nstack,Nbuffer):-
state(State,_A,B,C),
B == r1,
merge(Stack,C),
reduce(Stack,C,N1stack),   (check_verb_prep(N1stack,Input),
member(State2,States),           state(State2,C,D),State2\==23,State2\==128,State2\==118;
member(State2,States),
state(State2,C,D)),   match_state(D,State,States,N1stack,[Input | Restbuffer],
Nstack,Nbuffer).

match_state(State,_State1,States,Stack,[Input | Restbuffer],
Nstack,Nbuffer):-
state(State,_A,B,C),
B == r2,
checking1(Stack,Input),
merge1(Stack,C,D),
reduce1(Stack,D,N1stack),
(Input= $, last1(sentence(_,_),N1stack),last(State2,States);  member(State2,States)),
state(State2,D,E),   match_state(E,State,States,N1stack,[Input | Restbuffer],
Nstack,Nbuffer).

match_state(State,_State1,_States,Stack,[Input | Restbuffer],Stack,Restbuffer):-
state(State,$,a)=state(22,$,a),   delete($,[Input | Restbuffer],Restbuffer).

/*The rule 'check_categories' compares the leftmost element of the inputstring with gram
category of executed state.  It also deals with compoundentries which in certain circumstances
leads to lookahead being extended to be able to process them.*/

check_categories1(Input,Rest,A,B):-
mem_eq(Input,A),!,
B=[Input | [Rest]].

check_categories(Input,Rest,A,B):-
Input=[T,_S],
T=passive_be(_,_),          (mem_eq(T,A),first(Z,Rest),Z=verb(_,v(Y)),get_end(Y),!,fail;
mem_eq(T,A),first(Z,Rest),Z=verb(_,v(Y)),\+get_end(Y),
Input1=T),   B=[Input1 | [Rest]].

check_categories(Input,Rest,A,B):-
Input=[_T,S],
S=prog(_,_),   (mem_eq(S,A),first(Z,Rest),Z=verb(_,v(Y)),get_end(Y),
Input1=S;   mem_eq(S,A),first(Z,Rest),(Z=det(_,d(_Y));Z=adj(_,a(_Y))),   Input1=S),
B=[Input1 | [Rest]].

check_categories(Input,Rest,A,B):-
Input=[_T,S],
S=prep(_,_),   (mem_eq(S,A),first(Z,Rest),mem_eq(Z1,Z),verbaux(List),
member(Z1,List),!,fail;
mem_eq(S,A),Input1=S),
B=[Input1 | [Rest]].

check_categories(Input,Rest,A,B):-
Input=[T,_S],
T=inf(_,_),   mem_eq(T,A),first(Z,Rest),mem_eq(Z1,Z),verbaux(List),
member(Z1,List),
Input1=T,
B=[Input1 | [Rest]].

check_categories(Input,Rest,A,B):-
Input=[_T,S],
S=det(_,_),   (mem_eq(S,A),first(Z,Rest),mem_eq(Z1,Z),Z1=noun(_,_),
Input1=S),
B=[Input1 | [Rest]].

check_categories(Input,Rest,A,B):-
Input=[T,_S],
T=pronoun(_,_),
mem_eq(T,A),
Input1=T,
B=[Input1 | [Rest]].

check_categories(Input,Rest,A,B):-
Input=[_T,S],
S=noun(_,_),   mem_eq(S,A),first(Z,Rest),mem_eq(Z1,Z),verbaux(List),
member(Z1,List),
Input1=S,
B=[Input1 | [Rest]].

check_categories(Input,Rest,A,B):-
Input=[T,_S],
T=verb(_,_),   (mem_eq(T,A),first(Z,Rest),mem_eq(Z1,Z),verbaux(List),
member(Z1,List),!,
fail;
mem_eq(T,A),
Input1=T),
B=[Input1 | [Rest]].

check_categories(Input,Rest,A,B):-
Input=[T,S],
cat_clash(List),
member(T,List),
member(S,List),
(mem_eq(T,A),
Input1=T;
mem_eq(S,A),
Input1=S),
B=[Input1 | [Rest]].
```

```
check_categories(Input,Rest,A,B):-
Input=[T,S,U,V],
cat_clash(List),
member(T,List),
member(S,List),
member(U,List),
member(V,List),    (mem_eq(S,A),first(Z,Rest),Z=noun(_,n(_,sg)),
Input1=S;     mem_eq(U,A),first(Z,Rest),(Z=noun(_,n(_,pl));Z=det(_,d(_))),     Input1=U;
mem_eq(V,A),first(Z,Rest),\+Z=noun(_,n(_,pl)),Input1=V;
mem_eq(T,A),first(Z,Rest),(\+Z=noun(_,n(_,_)),
\+Z=det(_,d(_))),Input1=T),
B=[Input1 | [Rest]].
```

/*This rule called by 'match_state' shifts the leftmost constituent of thebuffer on to the stack.*/

```
shift([],[X | L],[X],L).

shift([X | T],[Y | L],L1, L):-
add_last(Y,[X | T],L1).
```

/*This rule, called by 'match_state', does semantic checks on verbs listed in 'spec_verb', to aid in the processing of prepositional phrases. It also aidsin processing of relative clauses.*/

```
check_verb_rpron(Stack):-
last(X,Stack),
X=verb(_,v(Y)),
spec_verb(List),
member(Z,List),
Z=verb(_,v(T)),
get_root(Y,R),
T=R.

check_verb_rpron(Stack):-
se_last(rpron(_,_),Stack).
```

/*This rule called by 'match_state' when certain states need to be reexecuted,as guided by the parse table.*/

```
checking(Stack,Input):-
(last(prep(_,_),Stack);
last1(prep(_,_),Stack);
check_aux(Stack);
last(rpron(_,_),Stack);
last1(verb(_,_),Stack);
se_last(prep_phrase(_,_),Stack),
last(verb_phrase(_,_),Stack);
last(inf(_,_),Stack),Input \== $;   se_last(rpron(_,_),Stack);
se_last(inf(_,_),Stack),Input  \== $;     last(inf(_,_),Stack),\+(mem_eq(verb(_,_),Input));
(member(wh_phrase(_,_),Stack),\+last(verb(_,_),Stack));
(se_last(comp(_,_),Stack),last(noun_phrase(_,_),Stack),
\+mem_eq(prep(_,_),Input),\+mem_eq(rpron(_,_),Input));
(se_last(aux(_,_),Stack),last(noun_phrase(_,_),Stack));
(se_last(aux1(_,_),Stack),last(th_phrase(_,_),Stack));
(th_last(aux1(_,_),Stack),se_last(th_phrase(_,_),Stack),     last(noun_phrase(_,_),Stack));
(th_last(rpron(_,_),Stack),last(noun_phrase(_,_),Stack),
Input  =  $);     (th_last(noun_phrase(_,_),Stack),se_last(verb(_,_),Stack),
last(noun_phrase(_,_),Stack))).
```

```
check_aux(Stack):-
last(X,Stack),
X=aux(_,_).
```

/* This rule is called by 'match_state' when semantic checks have to be madebefore reducing of constituents on the stack.*/

```
checking1(Stack,Input):-            ((mem_eq(prep(_,_),Input);\+th_last(rpron(_,_),Stack),
\+fo_last(rpron(_,_),Stack),            mem_eq(X,Input),verbaux(List),member(X,List);
(th_last(wh_phrase(_,_),Stack),last(verb_phrase(_,_),Stack));
mem_eq(comp(_,_),Input);mem_eq(rpron(_,_),Input);
(se_last(rpron(_,_),Stack),mem_eq(det(_,_),Input))),!,fail;true).
```

/*This rule, called by 'match_state' does semantic checks on verbs and prepositions listed in 'spec_verb' and 'spec_prep_inf'. It deals with thereduce-reduce conflicts produced by the parse_table.*/

```
check_verb_prep(Stack,Input):-
member(X,Stack),
X=verb(_,v(Y)),
spec_verb(List),
member(Z,List),
Z=verb(_,v(T)),
get_root(Y,R),T=R,
spec_prep_inf(List1),
member(Z1,List1),    mem_eq(Z1,Input).
```

/*This rule called by the above gets the root form of a verb.*/

```
get_root(Y,Y):-
name(Y,_R).

get_root(Y,R):-
name(Y,R2),
cutoff(_X,R2,R1),
name(R,R1).

get_root(Y,R):-
name(Y,R3),
cutoff(_X,R3,R2),
cutoff(_Z,R2,R1),
name(R,R1).

get_end(Y):-
name(Y,R3),
cutoff1(_X,R3,R2),
cutoff1(_Z,R2,R1),
cutoff1(_T,R1,_R),
name(ing,[_T,_Z,_X]).

mem_eq([],[]):- fail.

mem_eq(X,[X|_L]):- !.

mem_eq(X,X):- !.

mem_eq(X,[_Y|L]):-
mem_eq(X,L).
```

/*This rule called by check_categories list categories that may clash.*/

```
cat_clash(List):-
List=[wh_comp(_,w_c(who)),
rpron(_,r_p(who)),
rpron(_,r_p(that)),comp(_,comp(that)),det(_,d(that)),do(_,do(do)),
verb(_,v(_)),pronoun(_,_)].

spec_verb(List):- List=[verb(_,v(kill)),verb(_,v(know)),verb(_,v(want)),       verb(_,v(persuade))].

spec_prep_inf(List):- List=[prep(_,p(with)),inf(_,inf(to))].

conc([],L,L).c
conc([X|L],L2,[X|L3]):-    conc(L,L2,L3).
```

/*This rule called by 'match_state' is the first stage of the reduction process. It merges the constituents of the stack into the corresponding constituent of the left-hand side of the grammar rule.*/

```
merge(X,Y):-
last(A,X),
A=noun_phrase(_C,B),
se_last(D,X),
D=prep(_E,F),
Y=prep_phrase(_N,pp(F,B)).

merge(X,Y):-
last(A,X),
A=aux(_C,B),
se_last(D,X),
D=there(_E,F),
Y=th_phrase(_N,there(F,B)).

merge(X,Y):-
last(A,X),
A=there(_C,B),
se_last(D,X),
D=aux1(_E,_F),
Y=th_phrase(_N,there(B)).

merge(X,Y):-
last(A,X),
A=noun_phrase(_C,B),
se_last(D,X),
D=aux(_E,F),
Y=cop_phrase(_N,cp(F,B)).

merge(X,Y):-
last(A,X),
A=adj(_C,B),
se_last(D,X),
D=aux(_E,F),
Y=cop_phrase(_N,cp(F,B)).

merge(X,Y):-
last(A,X),
A=adj(_C,B),
se_last(D,X),
D=aux(_E,F),
th_last(G,X),
G=inf(_H,I),
Y=cop_phrase(_N,cp(I,F,B)).

merge(X,Y):-
last(A,X),
A=verb_phrase(_C,B),
se_last(D,X),
D=noun_phrase(_E,F),
th_last(G,X),
G=wh_phrase(_,_),
Y=e_sentence(_N,e_s(F,B)).

merge(X,Y):-
last(A,X),
A=verb_phrase(_C,B),
se_last(D,X),
D=noun_phrase(_E,F),
th_last(G,X),
G=aux(_,_),
Y=e_sentence(_N,e_s(F,B)).

merge(X,Y):-
last(A,X),
A=verb_phrase(_C,B),
se_last(D,X),
D=noun_phrase(_E,F),
th_last(G,X),
G=verb(_,_),
Y=e_sentence(_N,e_s(F,B)).

merge(X,Y):-
last(A,X),
A=verb_phrase(_C,B),
se_last(D,X),
D=noun_phrase(_E,F),
th_last(G,X),
G=rpron(_,_),
Y=e_sentence(_N,e_s(F,B)).

merge(X,Y):-    last(A,X),
    A=verb_phrase(_C,B),
    se_last(D,X),
    D=noun_phrase(_E,F),
    th_last(G,X),
    G=comp(_,_),
    Y=e_sentence(_N,e_s(F,B)).

merge(X,Y):-    last(A,X),
    A=aux1(_C,B),
    se_last(D,X),
    D=wh_comp(_E,F),
    Y=wh_phrase(_N,wh_ph(F,B)).

merge(X,Y):-    last(A,X),
    A=r_clause(_C,B),
    se_last(D,X),
    D=noun_phrase(_E,F),
    Y=noun_phrase(_N,np(F,B)).

merge(X,Y):-    last(A,X),
    A=verb_phrase(_C,B),
    se_last(D,X),
    D=rpron(_E,F),
    Y=r_clause(_N,r_c(F,B)).

merge(X,Y):-    last(A,X),
    A=cop_phrase(_C,B),
    se_last(D,X),
    D=rpron(_E,F),
    Y=r_clause(_N,r_c(F,B)).

merge(X,Y):-    last(A,X),
    A=e_sentence(_C,B),
    se_last(D,X),
    D=rpron(_E,F),
    Y=r_clause(_N,r_c(F,B)).

merge(X,Y):-    last(A,X),
    A=prog(_C,B),
    se_last(C,X),
    C=passive_be(_D,E),
    th_last(F,X),
    F=perf(_G,H),
    fo_last(I,X),
    I=modal(_J,K),
    Y=aux(_N,aux(K,H,E,B)).

merge(X,Y):-    last(A,X),
    A=passive_be(_C,B),
    se_last(C,X),
    C=perf(_D,E),
    th_last(F,X),
    F=modal(_G,H),
    Y=aux(_N,aux(H,E,B)).

merge(X,Y):-    last(A,X),
    A=prog(_C,B),
    se_last(C,X),
    C=perf(_D,E),
    th_last(F,X),
    F=modal(_G,H),
    Y=aux(_N,aux(H,E,B)).

merge(X,Y):-    last(A,X),
    A=perf(_C,B),
    se_last(C,X),
    C=modal(_D,E),
    Y=aux(_N,aux(E,B)).

merge(X,Y):-    last(A,X),
    A=prog(_C,B),
    se_last(C,X),
    C=modal(_D,E),
    Y=aux(_N,aux(E,B)).
```

```
merge(X,Y):-    last(A,X),
   A=passive_be(_C,B),
   se_last(C,X),
   C=prog(_D,E),
   Y=aux(_N,aux(E,B)).

merge(X,Y):-    last(A,X),
   A=passive_be(_C,B),
   se_last(C,X),
   C=perf(_D,E),
   Y=aux(_N,aux(E,B)).

merge(X,Y):-    last(A,X),
   A=prog(_C,B),
   se_last(C,X),
   C=perf(_D,E),
   Y=aux(_N,aux(E,B)).

merge(X,Y):-    last(A,X),
   A=modal(_C,B),
     (se_last(noun_phrase(_,_),X);se_last(there(_,_),X);
se_last(verb_phrase(_,_),X);se_last(verb_phrase2(_,_),X);    se_last(rpron(_,_),X)),
   Y=aux(_N,aux(B)).

merge(X,Y):-    last(A,X),
   A=seem(_C,B),
     (se_last(noun_phrase(_,_),X);se_last(there(_,_),X);
se_last(verb_phrase(_,_),X);se_last(verb_phrase2(_,_),X);    se_last(rpron(_,_),X)),
   Y=aux(_N,aux(B)).

merge(X,Y):-    last(A,X),
   A=do(_C,B),
     (se_last(noun_phrase(_,_),X);se_last(there(_,_),X);
se_last(verb_phrase(_,_),X);se_last(verb_phrase2(_,_),X);    se_last(rpron(_,_),X)),
   Y=aux(_N,aux(B)).

merge(X,Y):-    last(A,X),
   A=perf(_C,B),
     (se_last(noun_phrase(_,_),X);se_last(there(_,_),X);
se_last(verb_phrase(_,_),X);se_last(verb_phrase2(_,_),X);    se_last(rpron(_,_),X)),
   Y=aux(_N,aux(B)).

merge(X,Y):-    last(A,X),
   A=prog(_C,B),
    (se_last(noun_phrase(_,_),X);se_last(inf(_,_),X);se_last(there(_,_),X);
se_last(verb_phrase(_,_),X);se_last(verb_phrase2(_,_),X);    se_last(rpron(_,_),X)),
   Y=aux(_N,aux(B)).

merge(X,Y):-    last(A,X),
   A=be(_C,B),
    (se_last(noun_phrase(_,_),X);se_last(inf(_,_),X);se_last(there(_,_),X);
se_last(verb_phrase(_,_),X);se_last(verb_phrase2(_,_),X);    se_last(rpron(_,_),X)),
   Y=aux(_N,aux(B)).

merge(X,Y):-    last(A,X),
   A=passive_be(_C,B),
     (se_last(noun_phrase(_,_),X);se_last(there(_,_),X);
se_last(verb_phrase(_,_),X);se_last(verb_phrase2(_,_),X);    se_last(rpron(_,_),X)),
   Y=aux(_N,aux(B)).

merge(X,Y):-    last(A,X),
   A=modal(_C,B),
   se_last(wh_comp(_,_),X),
   Y=aux1(_N,aux1(B)).

merge(X,Y):-    last(A,X),
   A=do(_C,B),
   se_last(wh_comp(_,_),X),
   Y=aux1(_N,aux1(B)).

merge(X,Y):-    last(A,X),
   A=perf(_C,B),
   se_last(wh_comp(_,_),X),
   Y=aux1(_N,aux1(B)).

merge(X,Y):-    last(A,X),
   A=prog(_C,B),
   se_last(wh_comp(_,_),X),
   Y=aux1(_N,aux1(B)).

merge(X,Y):-    last(A,X),
   A=passive_be(_C,B),
   se_last(wh_comp(_,_),X),
   Y=aux1(_N,aux1(B)).

merge(X,Y):-    last1(A,X),
   A=modal(_C,B),
   Y=aux1(_N,aux1(B)).

merge(X,Y):-    last1(A,X),
   A=do(_C,B),
   Y=aux1(_N,aux1(B)).

merge(X,Y):-    last1(A,X),
   A=perf(_C,B),
   Y=aux1(_N,aux1(B)).

merge(X,Y):-    last1(A,X),
   A=prog(_C,B),
   Y=aux1(_N,aux1(B)).

merge(X,Y):-    last1(A,X),
   A=passive_be(_C,B),
   Y=aux1(_N,aux1(B)).

merge(X,Y):-   last(A,X),
  A=adj(_C,B),
  se_last(D,X),
  D=noun_phrase(_E,F),
  th_last(G,X),
  G=aux(_H,I),
  fo_last(J,X),
  J=inf(_M,L),
  Y=infaux(_N,infaux(L,I,F,B)).

merge(X,Y):-   last(A,X),
   A=e_sentence(_C,B),
   se_last(D,X),
   D=aux(_H,I),
   th_last(G,X),
   G=inf(_M,L),
   Y=infaux(_N,infaux(L,I,B)).

merge(X,Y):-   last(A,X),
   A=verb_phrase(_C,B),
   se_last(D,X),
   D=aux(_H,I),
   th_last(G,X),
   G=inf(_M,L),
   Y=infaux(_N,infaux(L,I,B)).

merge(X,Y):-    last(A,X),
    A=verb(_C,B),
    se_last(D,X),
    D=inf(_E,F),
    Y=verb_phrase2(_N,vp2(F,B)).

merge(X,Y):-    last(A1,X),
    A1 = noun_phrase(_S1,H),
    se_last(A,X),
    A=verb(_C,B),
    th_last(D,X),
    D=inf(_E,F),
    Y=verb_phrase2(_N,vp2(F,B,H)).

merge(X,Y):-    last(A,X),
    A=noun(_C,B),
    se_last(D,X),
    D=adj(_E,F),
    th_last(G,X),
    G=det(_H,I),
    Y=noun_phrase(_N,np(I,F,B)).

merge(X,Y):-    last(A,X),
    A=noun(_C,B),
    se_last(D,X),
    D=det(_E,F),
    Y=noun_phrase(_N,np(F,B)).

merge(X,Y):-    last(A,X),
    A=e_sentence(_C,B),
    se_last(D,X),
    D=comp(_E,F),
    Y=comp_phrase(_N,c_ph(F,B)).

merge(X,Y):-    last(A,X),
    A=e_sentence(_C,B),
    se_last(D,X),
    D=verb(_E,_F),
    Y=z_comp_phrase(_N,z_c_ph(B)).

merge(X,Y):-    last(A,X),
    A=prep_phrase(_C,B),
    se_last(D,X),
   D=noun_phrase(_E,F),
   Y=noun_phrase(_N,np(F,B)).

merge(X,Y):-    last(A,X),
   A=verb(_C,B),
   Y=verb_phrase(_N,np1(B,_)).

merge(X,Y):-    last1(A,X),
   A=proper_noun(_C,B),
   Y=noun_phrase(_N,np(B)).

merge(X,Y):-    last1(A,X),
   A=pronoun(_C,B),
   Y=noun_phrase(_N,np(B)).

merge(X,Y):-    last(A,X),
   A=pronoun(_C,B),
   Y=noun_phrase(_N,np(B)).

merge(X,Y):-    last1(A,X),
   A=wh_comp(_C,B),
   Y=wh_phrase(_N,wh_ph(B)).

merge(X,Y):-    last(A,X),
   A=proper_noun(_C,B),
   Y=noun_phrase(_N,np(B)).

merge(X,Y):-    last(A,X),
   A=noun(_C,B),
   (\+se_last(det(_,_),X);\+th_last(det(_,_),X)),
   Y=noun_phrase(_N,np(B)).

/*This rule is the final part of the reduction process, it replaces thecontents of the stack with the
result of 'merge'.*/
reduce(L,C,L4):-    last(noun_phrase(_,_),L),
   delete(noun_phrase(_,_),L,L1),
   del_last(verb(_,_),L1,L2),
   last1(inf(_,_),L2),
   delete(inf(_,_),L2,L3),
   add_last1(C,L3,L4).

reduce(L,C,L3):-  last(prep_phrase(_,_),L),
  delete(prep_phrase(_,_),L,L1),
  last(B,L1),
  delete(B,L1,L2),
  add_last(C,L2,L3).

reduce(L,C,L3):-  last(prep_phrase(_,_),L),
  delete(prep_phrase(_,_),L,L1),
  last1(B,L1),
  delete(B,L1,L2),
  add_last1(C,L2,L3).

reduce(L,C,L4):-    last(noun(_,_),L),
   delete(noun(_,_),L,L1),
   last(adj(_,_),L1),
   delete(adj(_,_),L1,L2),
   last1(D,L2),
    delete(D,L2,L3),
    add_last1(C,L3,L4).

reduce(L,C,L4):-    last(noun(_,_),L),
    delete(noun(_,_),L,L1),
    last(adj(_,_),L1),
    delete(adj(_,_),L1,L2),
    last(D,L2),
    delete(D,L2,L3),
    add_last(C,L3,L4).

reduce(L,C,L3):-    last(noun_phrase(_,_),L),
    delete(noun_phrase(_,_),L,L1),
    last(aux(_,_),L1),
    delete(aux(_,_),L1,L2),
    add_last(C,L2,L3).

reduce(L,C,L4):-    last(adj(_,_),L),
    delete(adj(_,_),L,L1),
    last(aux(_,_),L1),
    delete(aux(_,_),L1,L2),
    last(inf(_,_),L2),
    delete(inf(_,_),L2,L3),
    add_last(C,L3,L4).

reduce(L,C,L3):-    last(adj(_,_),L),
    delete(adj(_,_),L,L1),
    last(aux(_,_),L1),
    delete(aux(_,_),L1,L2),
    add_last(C,L2,L3).

reduce(L,C,L3):-    last(noun(_,_),L),
    delete(noun(_,_),L,L1),
    last1(det(_,_),L1),
    delete(det(_,_),L1,L2),
    add_last1(C,L2,L3).

reduce(L,C,L3):-    last(noun(_,_),L),
    delete(noun(_,_),L,L1),
    last(det(_,_),L1),
    delete(det(_,_),L1,L2),
    add_last(C,L2,L3).

reduce(L,C,L3):-    last(aux1(_,_),L),
    delete(aux1(_,_),L,L1),
    last1(wh_comp(_,_),L1),
    delete(wh_comp(_,_),L1,L2),
    add_last1(C,L2,L3).

reduce(L,C,L3):-    last(aux(_,_),L),
    delete(aux(_,_),L,L1),
    last1(there(_,_),L1),
    delete(there(_,_),L1,L2),
    add_last1(C,L2,L3).

reduce(L,C,L3):-    last(aux(_,_),L),
    delete(aux(_,_),L,L1),
    last(there(_,_),L1),
   delete(there(_,_),L1,L2),
   add_last(C,L2,L3).

reduce(L,C,L2):-    last(there(_,_),L),
   delete(there(_,_),L,L1),
   last1(aux1(_,_),L1),
   add_last(C,L1,L2).

reduce(L,C,L2):-    last(A,L),
   auxs(List),
   member(A,List),
   delete(A,L,L1),
   last1(wh_comp(_,_),L1),
   add_last(C,L1,L2).

reduce(L,C,L3):-    last(noun_phrase(_,_),L),
   delete(noun_phrase(_,_),L,L1),
   last(verb(_,_),L1),
   delete(verb(_,_),L1,L2),
   last(rpron(_,_),L2),
   add_last(C,L2,L3).

reduce(L,C,L3):-    last(verb_phrase(_,_),L),
   delete(verb_phrase(_,_),L,L1),
   last(noun_phrase(_,_),L1),
   delete(noun_phrase(_,_),L1,L2),
   last(rpron(_,_),L2),
   add_last(C,L2,L3).

reduce(L,C,L3):-    last(cop_phrase(_,_),L),
   delete(cop_phrase(_,_),L,L1),
   last(rpron(_,_),L1),
   delete(rpron(_,_),L1,L2),
   add_last(C,L2,L3).

reduce(L,C1,L5):-    last(A,L),
   auxs(List),
   member(A,List),
   delete(A,L,L1),
   last(B,L1),
   member(B,List),
   delete(B,L1,L2),
   last(C,L2),
   member(C,List),
   delete(C,L2,L3),
   last(D,L3),
   member(D,List),
   delete(D,L3,L4),
    (se_last(noun_phrase(_,_),L3);se_last(there(_,_),L3);se_last(inf(_,_),L3);
se_last(verb_phrase(_,_),L);se_last(verb_phrase2(_,_),L);    se_last(rpron(_,_),L)),
   add_last(C1,L4,L5).

reduce(L,C1,L4):-    last(A,L),
   auxs(List),
   member(A,List),
   delete(A,L,L1),
   last(B,L1),
   member(B,List),
   delete(B,L1,L2),
   last(C,L2),
   member(C,List),
   delete(C,L2,L3),
    (se_last(noun_phrase(_,_),L2);se_last(there(_,_),L2);se_last(inf(_,_),L2);
se_last(verb_phrase(_,_),L);se_last(verb_phrase2(_,_),L);    se_last(rpron(_,_),L)),
   add_last(C1,L3,L4).

reduce(L,C,L3):-    last(A,L),
   auxs(List),
   member(A,List),
   delete(A,L,L1),
   last(B,L1),
   member(B,List),
   delete(B,L1,L2),
    (se_last(noun_phrase(_,_),L1);se_last(there(_,_),L1);se_last(inf(_,_),L1);
se_last(verb_phrase(_,_),L);se_last(verb_phrase2(_,_),L);    se_last(rpron(_,_),L)),
   add_last(C,L2,L3).

reduce(L,C,L2):-    last(A,L),
   auxs(List),
   member(A,List),
   delete(A,L,L1),
    (se_last(noun_phrase(_,_),L);se_last(there(_,_),L);se_last(inf(_,_),L);
se_last(verb_phrase(_,_),L);se_last(verb_phrase2(_,_),L);    se_last(rpron(_,_),L)),
   add_last(C,L1,L2).

reduce(L,C,L3):-    last(e_sentence(_,_),L),
   delete(e_sentence(_,_),L,L1),
   last(comp(_,_),L1),
   delete(comp(_,_),L1,L2),
   add_last(C,L2,L3).

reduce(L,C,L2):-    last(e_sentence(_,_),L),
   delete(e_sentence(_,_),L,L1),
   last(verb(_,_),L1),
   add_last(C,L1,L2).

reduce(L,C,L3):-    last(verb_phrase(_,_),L),
   delete(verb_phrase(_,_),L,L1),
   last(noun_phrase(_,_),L1),
   delete(noun_phrase(_,_),L1,L2),
   last1(wh_phrase(_,_),L2),
   add_last(C,L2,L3).

reduce(L,C,L3):-    last(verb_phrase(_,_),L),
   delete(verb_phrase(_,_),L,L1),
   last(noun_phrase(_,_),L1),
   delete(noun_phrase(_,_),L1,L2),
   last(comp(_,_),L2),
   add_last(C,L2,L3).

reduce(L,C,L3):-    last(verb_phrase(_,_),L),
   delete(verb_phrase(_,_),L,L1),
   last(noun_phrase(_,_),L1),
   delete(noun_phrase(_,_),L1,L2),
   last(aux(_,_),L2),
   add_last(C,L2,L3).
```

```
reduce(L,C,L3):-    last(verb_phrase(_,_),L),
    delete(verb_phrase(_,_),L,L1),
    last(noun_phrase(_,_),L1),
    delete(noun_phrase(_,_),L1,L2),
    last(verb(_,_),L2),
    add_last(C,L2,L3).

reduce(L,C,L3):-    last(noun_phrase(_,_),L),
    delete(noun_phrase(_,_),L,L1),
    last(prep(_,_),L1),
    delete(prep(_,_),L1,L2),
    add_last(C,L2,L3).

reduce(L,C,L3):-    last(noun_phrase(_,_),L),
    delete(noun_phrase(_,_),L,L1),
    last1(prep(_,_),L1),
    delete(prep(_,_),L1,L2),
    add_last1(C,L2,L3).

reduce(L,C,L3):-    del_last(verb(_,_),L,L1),
    last(inf(_,_),L1),
    delete(inf(_,_),L1,L2),
    add_last(C,L2,L3).

reduce(L,C,L4):-    last(noun_phrase(_,_),L),
    delete(noun_phrase(_,_),L,L1),
    del_last(verb(_,_),L1,L2),
    last(inf(_,_),L2),
    delete(inf(_,_),L2,L3),
    add_last(C,L3,L4).

reduce(L,C,L3):-    del_last(verb(_,_),L,L1),
    last1(inf(_,_),L1),
    delete(inf(_,_),L1,L2),
    add_last1(C,L2,L3).

reduce(L,C,L3):-    last(verb_phrase(_,_),L),
    delete(verb_phrase(_,_),L,L1),
    last(comp(_,_),L1),
    delete(comp(_,_),L1,L2),
    add_last(C,L2,L3).

reduce(L,C,L2):-    last(noun(_,_),L),
    delete(noun(_,_),L,L1),
    last(comp(_,_),L1),
    add_last(C,L1,L2).

reduce(L,C,L3):-    last(r_clause(_,_),L),
    delete(r_clause(_,_),L,L1),
    last1(B,L1),
    delete(B,L1,L2),
    add_last1(C,L2,L3).

reduce(L,C,L3):-    last(r_clause(_,_),L),
    delete(r_clause(_,_),L,L1),
    last(B,L1),
    delete(B,L1,L2),
    add_last(C,L2,L3).
```

```
reduce(L,C,L3):-    last(verb_phrase(_,_),L),
   delete(verb_phrase(_,_),L,L1),
   last(rpron(_,_),L1),
   delete(rpron(_,_),L1,L2),
   add_last(C,L2,L3).

reduce(L,C,L3):-    last(e_sentence(_,_),L),
   delete(e_sentence(_,_),L,L1),
   last(rpron(_,_),L1),
   delete(rpron(_,_),L1,L2),
   add_last(C,L2,L3).

reduce(L,C,L2):-    last(A,L),
   delete(A,L,L1),
   last(verb(_,_),L1),
   add_last(C,L1,L2).

reduce(L,C1,L4):-    fo_last(prep(_,_),L),
   last(A,L),
   delete(A,L,L1),
   last(B,L1),
   delete(B,L1,L2),
   last(C,L2),
   delete(C,L2,L3),
   add_last(C1,L3,L4).

reduce(L,C1,L5):-    last(adj(_,_),L),
   delete(adj(_,_),L,L1),
   last(noun_phrase(_,_),L1),
   delete(noun_phrase(_,_),L1,L2),
   last(aux(_,_),L2),
   delete(aux(_,_),L2,L3),
   last(inf(_,_),L3),
   delete(inf(_,_),L3,L4),
   add_last(C1,L4,L5).

reduce(L1,C1,L5):-    last(e_sentence(_,_),L1),
   delete(e_sentence(_,_),L1,L2),
   last(aux(_,_),L2),
   delete(aux(_,_),L2,L3),
   last(inf(_,_),L3),
   delete(inf(_,_),L3,L4),
   add_last(C1,L4,L5).

reduce(L1,C1,L5):-    last(verb_phrase(_,_),L1),
   delete(verb_phrase(_,_),L1,L2),
   last(aux(_,_),L2),
   delete(aux(_,_),L2,L3),
   last(inf(_,_),L3),
   delete(inf(_,_),L3,L4),
   add_last(C1,L4,L5).

reduce(L,C,L3):-    th_last(prep(_,_),L),
   last(A,L),
   delete(A,L,L1),
   last(B,L1),
   delete(B,L1,L2),
   add_last(C,L2,L3).
```

```
reduce(L,C,L3):-    last1(A,L),
   delete(A,L,[]),
   add_last1(C,[],L3).

reduce(L,C,L2):-    last(proper_noun(_,_),L),
   delete(proper_noun(_,_),L,L1),
   add_last(C,L1,L2).

reduce(L,C,L2):-    last(noun(_,_),L),
   delete(noun(_,_),L,L1),
   add_last(C,L1,L2).

reduce(L,C,L2):-    last(pronoun(_,_),L),
   delete(pronoun(_,_),L,L1),
   add_last(C,L1,L2).

/*This rule called by 'match_state' performs the same function as 'merge',deals with
verb-phrases and sentences.*/
merge1(X,Y,Z):-    last(A,X),
   A=verb_phrase(_C,A2),
   se_last(B,X),
   B=noun_phrase(_D,B1),
   th_last(C,X),
   C=aux1(_E,C1),
   Y=sentence(_N,s(C1,B1,_A1)),
   Z=sentence(_N,s(C1,B1,A2)).

merge1(X,Y,Z):-    last(A,X),
   A=verb_phrase(_C,A2),
   se_last(B,X),
   B=noun_phrase(_D,B1),
   th_last(C,X),
   C=th_phrase(_E,C1),
   fo_last(D,X),
   D=aux1(_E,D1),
   Y=sentence(_N,s(D1,C1,B1,_A1)),
   Z=sentence(_N,s(D1,C1,B1,A2)).

merge1(X,Y,Z):-    last(A,X),
   A=verb_phrase(_C,A2),
   se_last(B,X),
   B=noun_phrase(_D,B1),
   Y=sentence(_N,s(B1,_A1)),
   Z=sentence(_N,s(B1,A2)).

merge1(X,Y,Z):-    last(A,X),
   A=cop_phrase(_C,A2),
   se_last(B,X),
   B=verb_phrase2(_D,B1),
   Y=sentence(_N,s(B1,_A1)),
   Z=sentence(_N,s(B1,A2)).

merge1(X,Y,Z):-    last(A,X),
   A=infaux(_C,A2),
   se_last(B,X),
   B=th_phrase(_D,B1),
   th_last(C,X),
   C=aux1(_E,C1),
   Y=sentence(_N,s(C1,B1,_A1)),
   Z=sentence(_N,s(C1,B1,A2)).

merge1(X,Y,Z):-    last(A,X),
   A=infaux(_C,A2),
   se_last(B,X),
   B=aux(_D,B1),
   th_last(C,X),
   C=noun_phrase(_E,C1),
   Y=sentence(_N,s(C1,B1,_A1)),
   Z=sentence(_N,s(C1,B1,A2)).

merge1(X,Y,Z):-    last(A,X),
   A=infaux(_C,A2),
   se_last(B,X),
   B=th_phrase(_D,B1),
   Y=sentence(_N,s(B1,_A1)),
   Z=sentence(_N,s(B1,A2)).

merge1(X,Y,Z):-    last(A,X),
   A=prep_phrase(_C,A2),
   se_last(B,X),
   B=noun_phrase(_D,B1),
   th_last(C,X),
   C=verb(_E,C1),
   Y=verb_phrase(_N,vp(C1,B1,_A1)),
   Z=verb_phrase(_N,vp(C1,B1,A2)).

merge1(X,Y,Z):-    last(A,X),
   A=verb_phrase2(_C,A2),
   se_last(B,X),
   B=noun_phrase(_D,B1),
   th_last(C,X),
   C=verb(_E,C1),
   Y=verb_phrase(_N,vp(C1,B1,_A1)),
   Z=verb_phrase(_N,vp(C1,B1,A2)).

merge1(X,Y,Z):-    last(A,X),
   A=noun_phrase(_C,A2),
   se_last(B,X),
   B=verb(_D,B1),
   th_last(_T,X),
   Y=verb_phrase(_N,vp(B1,_A1)),
   Z=verb_phrase(_N,vp(B1,A2)).

merge1(X,Y,Z):-    last(A,X),
   A=noun_phrase(_C,A2),
   se_last(B,X),
   B=verb(_D,B1),
   Y=imp_verb_phrase(_N,ivp(B1,_A1)),
   Z=imp_verb_phrase(_N,ivp(B1,A2)).

merge1(X,Y,Z):-    last(A,X),
   A=prep_phrase(_C,A2),
   se_last(B,X),
   B=noun_phrase(_D,B1),
   th_last(C,X),
   C=verb(_E,C1),
    Y=imp_verb_phrase(_N,ivp(C1,B1,_A1)),
    Z=imp_verb_phrase(_N,ivp(C1,B1,A2)).

merge1(X,Y,Z):-    last(A,X),
    A=z_comp_phrase(_C,A2),
    se_last(B,X),
    B=verb(_D,B1),
    Y=verb_phrase(_N,vp(B1,_A1)),
    Z=verb_phrase(_N,vp(B1,A2)).

merge1(X,Y,Z):-    last(A,X),
    A=prep_phrase(_C,A2),
    se_last(B,X),
    B=verb(_D,B1),
    Y=verb_phrase(_N,vp(B1,_A1)),
    Z=verb_phrase(_N,vp(B1,A2)).

merge1(X,Y,Z):-    last(A,X),
    A=verb(_C,A2),
    Y=verb_phrase(_N,vp(_A1)),
    Z=verb_phrase(_N,vp(A2)).

merge1(X,Y,Z):-    last1(A,X),
    A=imp_verb_phrase(_C,A2),
    Y=sentence(_N,s(_A1)),
    Z=sentence(_N,s(A2)).

merge1(X,Y,Z):-    last(A,X),
    A=comp_phrase(_C,A2),
    se_last(B,X),
    B=verb(_D,B1),
    Y=verb_phrase(_N,vp(B1,_A1)),
    Z=verb_phrase(_N,vp(B1,A2)).

merge1(X,Y,Z):-    last(A,X),
    A=verb_phrase2(_C,A2),
    se_last(B,X),
    B=verb(_D,B1),
    Y=verb_phrase(_N,vp(B1,_A1)),
    Z=verb_phrase(_N,vp(B1,A2)).

merge1(X,Y,Z):-    last(A,X),
    A=verb_phrase(_C,A2),
    se_last(B,X),
    B=wh_phrase(_D,B1),
    Y=sentence(_N,s(B1,_A1)),
    Z=sentence(_N,s(B1,A2)).

merge1(X,Y,Z):-    last(A,X),
    A=e_sentence(_C,A2),
    se_last(B,X),
    B=wh_phrase(_D,B1),
    Y=sentence(_N,s(B1,_A1)),
    Z=sentence(_N,s(B1,A2)).

merge1(X,Y,Z):-    last(A,X),
    A=cop_phrase(_C,A2),
    se_last(B,X),
    B=noun_phrase(_D,B1),
    Y=sentence(_N,s(B1,_A1)),
    Z=sentence(_N,s(B1,A2)).

merge1(X,Y,Z):-    last(A,X),
    A=cop_phrase(_C,A2),
    se_last(B,X),
    B=verb_phrase(_D,B1),
    th_last(C,X),
    C=prep_phrase(_E,C1),
    Y=sentence(_N,s(C1,B1,_A1)),
    Z=sentence(_N,s(C1,B1,A2)).

merge1(X,Y,Z):-    last(A,X),
    A=verb_phrase(_C,A2),
    se_last(B,X),
    B=aux(_D,B1),
    th_last(C,X),
    C=noun_phrase(_F,C1),
    Y=sentence(_N,s(C1,B1,_A1)),
    Z=sentence(_N,s(C1,B1,A2)).

merge1(X,Y,Z):-    last(A,X),
    A=verb_phrase(_C,A2),
    se_last(B,X),
    B=noun_phrase(_D,B1),
    th_last(C,X),
    C=th_phrase(_F,C1),
    fo_last(D,X),
    D=aux1(_G,D1),
    Y=sentence(_N,s(D1,C1,B1,_A1)),
    Z=sentence(_N,s(D1,C1,B1,A2)).

merge1(X,Y,Z):-    last(A,X),
    A=cop_phrase(_C,A2),
    se_last(B,X),
    B=aux(_D,B1),
    th_last(C,X),
    C=noun_phrase(_F,C1),
    Y=sentence(_N,s(C1,B1,_A1)),
    Z=sentence(_N,s(C1,B1,A2)).

merge1(X,Y,Z):-    last(A,X),
    A=noun(_C,A2),
    se_last(B,X),
    B=noun(_D,B1),
    Y=noun_phrase(_N,np(B1,_A1)),
    Z=noun_phrase(_N,np(B1,A2)).

merge1(X,Y,Z):-    last1(A,X),
    A=noun(_C,A2),
    Y=noun_phrase(_N,np(_A1)),
    Z=noun_phrase(_N,np(A2)).

/*This rule called by 'match_state' performs the same function as 'reduce',deals with
verb-phrases and sentences.*/
reduce1(L,C,L2):-    last(X,L),
    delete(X,L,L1),X=verb(_,_),
   add_last(C,L1,L2).

reduce1(L,C,L3):-    last(A,L),
   delete(A,L,L1),
   del_last(verb(_,_),L1,L2),
   add_last(C,L2,L3).

reduce1(L,C,L4):-    last(A,L),
   delete(A,L,L1),
   last(B,L1),
   delete(B,L1,L2),
   del_last(verb(_,_),L2,L3),
   add_last(C,L3,L4).

reduce1(L,C,L5):-    last(A,L),
   delete(A,L,L1),
   last(B,L1),
   delete(B,L1,L2),
   last(C1,L2),
   delete(C1,L2,L3),
   last1(D,L3),
   delete(D,L3,L4),
   add_last1(C,L4,L5).

reduce1(L,C,L4):-    last(A,L),
   delete(A,L,L1),
   last(B,L1),
   delete(B,L1,L2),
   last1(C1,L2),
   delete(C1,L2,L3),
   add_last1(C,L3,L4).

reduce1(L,C,L3):-    last(A,L),
   delete(A,L,L1),
   last1(B,L1),
   delete(B,L1,L2),
   add_last1(C,L2,L3).

reduce1(L,C,L2):-    last1(A,L),
   delete(A,L,L1),
   add_last1(C,L1,L2).
```

/*The following rules are called by the rules above to perform checks andfunctions as representedby the name of the rule.*/

```
first(X,[X]):-!.first(X,[X | _L]):-!.second(Y,[_X,Y]):-!.second(Y,[_X,Y | _L]):-!.last1(X,[X]):-!.last(X,[_
Y,X]):-!.last(X,[_Y | L]):-    last(X,L).

delete(X,[X | L],L).

delete(X,[Y | L],[Y | L1]):-    delete(X,L,L1).

del_last(X,[Y,X],[Y]).

del_last(X,[Y | L],[Y | L1]):-    del_last(X,L,L1).

del_first(X,[X | L],L).
```

```
add_last(C,[Y],[Y,C]).

add_last(C,[Y|L],[Y|L1]):-    add_last(C,L,L1).

se_last(X,[X,_Y]).

se_last(X,[_Z|Y]):-    se_last(X,Y).

th_last(X,[X,_Y,_Z]).

th_last(X,[_Z|Y]):-    th_last(X,Y).

fo_last(X,[X,_Y,_Z,_T]).

fo_last(X,[_Z|Y]):-    fo_last(X,Y).

/*This rule called by 'reduce' list auxilliaries.*/

auxs(List):-    List=[modal(_,_),
perf(_,_),
do(_,_),
prog(_,_),
passive_be(_,_),
seem(_,_),
be(_,_)].

/*This rule, called by 'checking1' and 'check_categories', lists verbs and auxilliaries.*/

v   e   r   b   a   u   x   (   L   i   s   t   )   :   -
List=[verb(_,_),modal(_,_),perf(_,_),do(_,_),prog(_,_),passive_be(_,_),seem(_,_),be(_,_)].
```

The code below represents the grammar rules of LParser.

```
/*This is the grammar for LParser.*/

gram:-assert((sentence_-->sentence(N,_S))),
assert((sentence(N,s(NP,VP))-->noun_phrase(N1,NP),
verb_phrase(N,VP))),
assert((sentence(N,s(NP,Aux,VP))-->noun_phrase(N1,NP),
aux(N,Aux),
    verb_phrase(N,VP))),
assert((sentence(N,s(NP,CP))-->noun_phrase(N1,NP),
cop_phrase(N1,CP))),
assert((sentence(N,s(Wh_phrase,VP))-->wh_phrase(N,Wh_phrase),
                                      verb_phrase(N,VP))),
assert((sentence(N,s(Wh_phrase,E_Sent))-->wh_phrase(N,Wh_phrase),
                                      e_sentence(N,E_Sent))),
assert((sentence(N,s(Aux1,NP,VP))-->aux1(N,Aux1),
noun_phrase(N1,NP),
verb_phrase(N,VP))),
assert((sentence(N,s(PP,NP,VP))-->prep_phrase(N,PP),
noun_phrase(N1,NP),
verb_phrase(N,VP))),
assert((sentence(N,s(PP,VP,CP))-->prep_phrase(N,PP),
verb_phrase(N,VP),
cop_phrase(N,CP))),
assert((sentence(N,s(VP2,CP))-->verb_phrase2(N,VP2),
cop_phrase(N,CP))),
assert((sentence(N,s(Aux1,Th_phrase,NP,VP))-->aux1(N,Aux1),
        th_phrase(N,Th_phrase),
noun_phrase(N1,NP),
verb_phrase(N,VP))),
assert((sentence(N,s(Aux1,Th_phrase,InfAux))-->aux1(N,Aux1),
th_phrase(N,Th_phrase),
 infaux(N,InfAux))),
assert((sentence(N,s(Aux1,NP,VP))-->aux1(N,Aux1),
noun_phrase(N1,NP),
prep_phrase(N,PP))),
assert((sentence(N,s(Th_phrase,InfAux))-->th_phrase(N,Th_phrase),
infaux(N,InfAux))),
assert((sentence(N,s(Th_phrase,E_Sent))-->th_phrase(N,Th_phrase),
e_sentence(N,E_Sent))),
assert((sentence(N,s(NP,Aux,InfAux))-->noun_phrase(N,NP),
aux(N,Aux),
infaux(N,InfAux))),
assert((e_sentence(N,e_s(NP,VP))-->noun_phrase(N1,NP),
verb_phrase(N,VP))),
assert((imp_verb_phrase(N,vp(V,NP))-->verb(N,V),
noun_phrase(N,NP))),
assert((imp_verb_phrase(N,vp(V,NP,PP))-->verb(N,V),
noun_phrase(N,NP),
prep_phrase(N,PP))),
assert((verb_phrase(N,vp(V,NP))-->verb(N,V),
noun_phrase(N,NP))),
assert((verb_phrase(N,vp(V,WP1))-->verb(N,V),
wh_phrase1(N,WP1))),
assert((verb_phrase(N,vp(V,PP))-->verb(N,V),
prep_phrase(N,PP))),
assert((verb_phrase(N,vp(V,NP,PP))-->verb(N,V),
noun_phrase(N,NP),
prep_phrase(N,PP))),
assert((verb_phrase(N,vp(V,VP2))-->verb(N,V),
verb_phrase2(N,VP2))),
assert((verb_phrase(N,vp(V,NP,VP2))-->verb(N,V),
noun_phrase(N,NP),
verb_phrase2(N,VP2))),
assert((verb_phrase(N,vp(V,CP))-->verb(N,V),
comp_phrase(N,CP))),
assert((verb_phrase(N,vp(V,ZCP))-->verb(N,V),
z_comp_phrase(N,ZCP))),
assert((verb_phrase(N,vp(V,VP))-->verb(N,V))),
assert((verb_phrase2(N,vp(Inf,V))-->inf(N,Inf),
verb(N,V))),
assert((verb_phrase2(N,vp(Inf,V,NP))-->inf(N,Inf),
verb(N,V),
noun_phrase(N,NP))),
assert((infaux(N,infaux(Inf,Aux,VP))-->inf(N,Inf),
aux(N,Aux),
verb_phrase(N,V))),
assert((infaux(N,infaux(Inf,Aux,E_Sent))-->inf(N,Inf),
aux(N,Aux),
e_sentence(N,E_S))),
assert((infaux(N,infaux(Inf,Aux,Adj))-->inf(N,Inf),
aux(N,Aux),
adj(N,Adj))),
assert((infaux(N,infaux(Inf,Aux,NP,Adj))-->inf(N,Inf),
aux(N,Aux),
noun_phrase(N,NP),
adj(N,Adj))),
assert((comp_phrase(N,c_ph(Comp,E_Sent))-->comp(N,Comp),
e_sentence(N,E_Sent))),
assert((z_comp_phrase(N,z_c_ph(E_Sent))-->e_sentence(N,E_Sent))),
assert((noun_phrase(singular,np(Name))-->proper_noun(singular,Name))),
assert((noun_phrase(plural,np(Noun))-->noun(plural,Noun))),
assert((noun_phrase(N1,np(Pronoun))-->pronoun(N1,Pronoun))),
assert((noun_phrase(N1,np(Det,Noun))-->det(N1,Det),
noun(N1,Noun))),
assert((noun_phrase(N1,np(Det,Adj,Noun))-->det(N1,Det),
adj(N1,Adj),
noun(N1,Noun))),
assert((noun_phrase(N1,np(NP,PP))-->noun_phrase(N1,NP),
prep_phrase(N1,PP))),
assert((noun_phrase(N1,np(NP,RC))-->noun_phrase(N1,NP),
r_clause(N1,RC))),
assert((prep_phrase(N1,pp(Prep,NP))-->prep(N1,Prep),
noun_phrase(N1,NP))),
assert((cop_phrase(N1,cp(Aux,NP))-->aux(N1,Aux),
noun_phrase(N1,NP))),
assert((cop_phrase(N1,cp(Aux,Adj))-->aux(N1,Aux),
adj(N1,Adj))),
assert((r_clause(N1,r_c(RPron,VP))-->rpron(N1,RPron),
verb_phrase(N1,VP))),
assert((r_clause(N1,r_c(RPron,CP))-->rpron(N1,RPron),
cop_phrase(N1,CP))),
assert((r_clause(N1,r_c(RPron,Aux,VP))-->rpron(N1,RPron),
aux(N,Aux),
verb_phrase(N1,VP))),
assert((r_clause(N1,r_c(RPron,VP))-->rpron(N1,RPron),
e_sentence(N,E_Sent))),
assert((wh_phrase(N,wh_ph(Wh_comp,Aux1))-->wh_comp(N,Wh_comp),
aux1(N,Aux1))),
assert((wh_phrase(N,wh_ph(Wh_comp))-->wh_comp(N,Wh_comp))),
assert((wh_phrase1(N,wh_ph(Wh_comp,E_S))-->wh_comp(N,Wh_comp),
e_sentence(N,E_S))),
assert((wh_phrase1(N,wh_ph(Wh_comp,VP))-->wh_comp(N,Wh_comp),
verb_phrase(N,VP))),
assert((wh_phrase1(N,wh_ph(Wh_comp,VP2))-->wh_comp(N,Wh_comp),
 verb_phrase2(N,VP2))),
assert((th_phrase(N,there(There,Aux))-->there(N,There),
aux(N,Aux))),
assert((th_phrase(N,there(There))-->there(N,There))),
assert((aux(N,aux(Perf))-->seem(N,seem))),
assert((aux(N,aux(Be))-->be(N,Be))),
assert((aux(N,aux(Perf))-->perf(N,Perf))),
assert((aux(N,aux(Modal))-->modal(N,Modal))),
assert((aux(N,aux(Prog))-->prog(N,Prog))),
assert((aux(N,aux(Do))-->do(N,Do))),
assert((aux(N,aux(P_Be))-->passive_be(N,P_Be))),
assert((aux1(N,aux1(Perf))-->perf(N,Perf))),
assert((aux1(N,aux1(Modal))-->modal(N,Modal))),
assert((aux1(N,aux1(Prog))-->prog(N,Prog))),
assert((aux1(N,aux1(Do))-->do(N,Do))),
assert((aux(N,aux(Modal,Perf))-->modal(N,Modal),
perf(N,Perf))),
assert((aux(N,aux(Modal,Prog))-->modal(N,Modal),
prog(N,Prog))),
assert((aux(N,aux(Modal,Perf,Prog))-->modal(N,Modal),
perf(N,Perf),
prog(N,Prog))),
assert((aux(N,aux(Perf,Prog))-->perf(N,Perf),
prog(N,Prog))),
assert((aux(N,aux(Prog,Passive_be))-->prog(N,Prog),
passive_be(N,Passive_be))),
assert((aux(N,aux(Perf,Passive_be))-->perf(N,Perf),
passive_be(N,Passive_be))),
assert((aux(N,aux(Modal,Perf,Passive_be))-->modal(N,Modal),
perf(N,Prog),passive_be(N,Passive_be))),
assert((aux(N,aux(Modal,Perf,Passive_be,Prog))-->modal(N,Modal),
perf(N,Perf),passive_be(N,Passive_be),prog(N,Prog))).
```

The code below represents the parse table for LParser.

```
/*This is the parse table for LParser*/


state(1, noun_phrase(_144656,_144647), 3).
state(1, det(_144656,_144673), s, 8).
state(1, noun_phrase(_144769,_144760), 4).
state(1, det(_144769,_144786), s, 9).
state(1, noun_phrase(_144769,_144786), 2).
state(1, pronoun(_144769,_144948), s, 10).
state(1, noun(_,_144948), s, 11).
state(1, noun_phrase(_145108,_145098), 2).
state(1, noun_phrase(_145108,_145130), 2).
state(1, proper_noun(singular,_145211), s, 12).
state(1, wh_phrase(_145283,_145286), 2).
state(1, wh_comp(_145283,_145312), s, 5).
state(1, wh_phrase(_145355,_145358), 2).
state(1, wh_comp(_145355,_145384), s, 13).
state(1, verb_phrase2(_145421,_145424), 2).
state(1, inf(_145421,_145450), s, 15).
state(1, inf(_145528,_145531), s, 14).
state(1, th_phrase(_145615,_145618), 2).
state(1, there(_145615,_145644), s, 6).
state(1, there(_145722,_145725), s, 16).
state(1, th_phrase(_145797,_145800), 2).
state(1, noun_phrase(_145823,_145826), 2).
state(1, imp_verb_phrase(_145855,_145858), 2).
state(1, verb(_145855,_145878), s, 7).
state(1, verb(_145901,_145904), s, 7).
state(1, aux1(_145933,_145936), 2).
state(1, perf(_145933,_145968), s, 17).
state(1, modal(_145933,_145968), s, 18).
state(1, prog(_145933,_145968), s, 19).
state(1, do(_145933,_145968), s, 20).
state(1, prep_phrase(_146085,_146088), 2).
state(1, prep(_146085,_146120), s, 21).
state(1, prep_phrase(_146198,_146201), 2).
state(1, aux1(_146230,_146233), 2).
state(1, aux1(_146268,_146271), 2).
state(1, aux1(_146300,_146303), 2).
state(1, sentence(_146338,_146339), 22).
state(2, verb_phrase(144644,_144648), 41).
state(2, verb(_146379,_146382), s, 33).
state(2, verb(_146411,_146414), s, 42).
state(2, verb(_146443,_146446), s, 43).
state(2, verb(_146469,_146472), s, 23).
state(2, verb(_146495,_146498), s, 25).
state(2, verb(_146521,_146524), s, 26).
state(2, verb(_146547,_146550), s, 27).
state(2, verb(_146573,_146576), s, 24).
state(2, verb(_146599,_146602), s, 44).
state(2, cop_phrase(_144769,cp(_146626,_146627)), 46).
state(2, verb_phrase(_145283,_145287), 53).
state(2, e_sentence(_145355,e_s(_146684,_146685)), 56).
state(2, cop_phrase(_145421,cp(_146716,_146717)), 58).
state(2, infaux(_145615,_), 61).
state(2, e_sentence(_145797,_145801), 64).
state(2, $, r2, sentence(_145855,s(_145858))).
state(2, aux(_145095,_), 49).
state(2, aux(_145823,_), 37).
state(2, noun_phrase(_145951,np(_146905,_146906,_146907)), 68).
state(2, noun_phrase(_146103,_), 57).
state(2, verb_phrase(_146198,_146217), 35).
state(2, th_phrase(_145823,there(_147015,_147016)), 38).
state(2, noun_phrase(_146318,np(_147053,_)), 36).
state(2, th_phrase(_146230,_146234), 73).
state(2, inf(_147122,_147125), s, 39).
state(2, inf(_145615,_146748), s, 39).
state(2, inf(_147192,_147195), s, 39).
state(2, inf(_147224,_147227), s, 39).
state(2, aux(_144769,_146626), 40).
state(2, noun_phrase(_147294,_146684), 70).
state(2, aux(_145421,_146716), 40).
state(2, there(_145615,_145644), s, 6).
state(2, there(_145722,_145725), s, 16).
state(2, det(_144656,_144673), s, 8).
state(2, det(_144769,_144786), s, 9).
state(2, noun_phrase(_145108,_145130), 4).
state(2, noun_phrase(_144769,_144786), 3).
state(2, proper_noun(singular,_145211), s, 12).
state(2, noun(_,_144948), s, 11).
state(2, pronoun(_144769,_144948), s, 10).
state(2, modal(_148045,_148048), s, 75).
state(2, modal(_148113,_148116), s, 79).
state(2, modal(_148175,_148178), s, 77).
state(2, modal(_145615,_146867), s, 65).
state(2, modal(_145095,_146829), s, 50).
state(2, perf(_148322,_148325), s, 83).
state(2, prog(_148378,_148381), s, 85).
state(2, perf(_148434,_148437), s, 87).
state(2, passive_be(_148490,_148493), s, 101).
state(2, prog(_148540,_148543), s, 97).
state(2, perf(_148590,_148593), s, 93).
state(2, do(_148640,_148643), s, 99).
state(2, modal(_148690,_148693), s, 95).
state(2, seem(_148740,_), s, 89).
state(3, r_clause(_144769,_), 48).
state(3, rpron(_144769,_148800), s, 28).
state(3, rpron(_148964,_148967), s, 28).
state(3, rpron(_149045,_149048), s, 28).
state(3, rpron(_149126,_149129), s, 28).
state(4, prep_phrase(_145108,pp(_149223,_149224)), 51).
state(4, prep(_144807,_146120), s, 21).
state(5, aux1(_145283,aux1(_149397)), 54).
state(5, perf(_145283,_149397), s, 55).
state(5, modal(_144644,_145968), s, 18).
state(5, prog(_144644,_145968), s, 19).
state(5, do(_144644,_145968), s, 20).
state(6, aux(_145615,_), 62).
state(6, modal(_145615,_149583), s, 63).
state(6, modal(_148045,_148048), s, 75).
state(6, modal(_148175,_148178), s, 77).
state(6, modal(_148113,_148116), s, 79).
state(6, modal(_149939,_149942), s, 81).
state(6, modal(_149939,_149942), s, 81).
state(6, perf(_148322,_148325), s, 83).
state(6, prog(_148378,_148381), s, 85).
state(6, perf(_148434,_148437), s, 87).
state(6, seem(_148740,_), s, 89).
state(6, be(_150269,_150272), s, 91).
state(6, be(_150269,_150272), s, 91).
state(6, perf(_148590,_148593), s, 93).
state(6, modal(_148690,_148693), s, 95).
state(6, prog(_148540,_148543), s, 97).
state(6, do(_148640,_148643), s, 99).
state(6, passive_be(_3903,_3906), s, 101).
state(7, noun_phrase(_1268,np(_6042,_6043)), 66).
state(7, det(_1268,_6042), s, 67).
state(7, proper_noun(singular,_6099), s, 109).
state(7, noun(_,_6119), s, 111).
state(7, pronoun(_6136,_6139), s, 113).
state(7, noun_phrase(_599,_6159), 30).
state(7, noun_phrase(_6182,_6185), 32).
state(7, det(_6208,_6211), s, 107).
state(7, noun_phrase(_599,_1318), 34).
state(8, adj(_69,_87), s, 45).
state(9, noun(_182,_), s, 47).
state(10,[verb(_86922,_86923),inf(_86927,_86928),modal(_86932,_86933),perf(_86937,_86938),pass
ive_be(_86942,_86943),prog(_86947,_86948),do(_86952,_86953),prep(_86957,_86958),rpron(_8696
2,_86963),det(_86967,_86968),noun(_86972,_86973)],r1, noun_phrase(_182,np(_361))).
state(11,[verb(_87340,_87341),inf(_87345,_87346),modal(_87350,_87351),perf(_87355,_87356),pass
ive_be(_87360,_87361),prog(_87365,_87366),do(_87370,_87371),prep(_87375,_87376),rpron(_8738
0,_87381),det(_87385,_87386),noun(_87390,_87391)], r1, noun_phrase(_,_)).
state(12,[verb(_87758,_87759),inf(_87763,_87764),modal(_87768,_87769),perf(_87773,_87774),pass
ive_be(_87778,_87779),prog(_87783,_87784),do(_87788,_87789),prep(_87793,_87794),rpron(_8779
8,_87799),det(_87803,_87804),noun(_87808,_87809)],r1,noun_phrase(singular,np(_624))).
s    t    a    t    e    (    1    3    ,
[verb(_88176,_88177),noun(_88181,_88182),det(_88186,_88187),proper_noun(_88191,_88192)], r1,
wh_phrase(_768,wh_ph(_797))).
state(14, verb(_834,_864), s, 59).
state(15, verb(_941,_945), s, 60).
state(16,[verb(_92728,_92729),inf(_92733,_92734),modal(_92738,_92739),perf(_92743,_92744),pass
ive_be(_92748,_92749),prog(_92753,_92754),do(_92758,_92759),prep(_92763,_92764),rpron(_9276
8,_92769),det(_92773,_92774),noun(_92778,_92779)],r1,th_phrase(_1135,there(_1138))).
state(17, [det(_93426,_93427),proper_noun(_93431,_93432)], r1, aux1(_57,aux1(_1381))).
state(18, [det(_93744,_93745),proper_noun(_93749,_93750)], r1, aux1(_57,aux1(_1381))).
state(19, [det(_94062,_94063),proper_noun(_94067,_94068)], r1, aux1(_57,aux1(_1381))).
state(20, [det(_94380,_94381),proper_noun(_94385,_94386)], r1, aux1(_57,aux1(_1381))).
state(21, noun_phrase(_220,_1534), 72).
state(22, $, a).
state(23, noun_phrase(_1856,_), 133).
state(23, det(_1268,_6042), s, 67).
state(23, noun_phrase(_599,_6042), 30).
state(23, noun_phrase(_599,_6185), 32).
state(23, det(_6208,_6211), s, 107).
state(23, proper_noun(singular,_6099), s, 109).
state(23, noun(_,_6119), s, 111).
state(23, pronoun(_6136,_6139), s, 113).
state(24, wh_phrase1(_1882,wh_ph(_7442,_7443)), 135).
state(24, wh_comp(_1882,_7442), s, 117).
state(24, wh_comp(_1882,_7442), s, 117).
state(24, wh_comp(_1882,_7442), s, 117).
state(25, prep_phrase(_1908,pp(_7558,_7559)), 136).
state(25, prep(_1908,_7558), s, 137).
state(26, verb_phrase2(_1934,_), 138).
state(26, inf(_1934,_7622), s, 204).
state(26, inf(_7676,_7679), s, 139).
state(27, comp_phrase(_1882,c_ph(_7718,_7719)), 140).
state(27, comp(_1882,_7718), s, 118).
state(28, verb_phrase(_182,_), 146).
state(28, verb(_182,_7782), s, 125).
state(28, verb(_182,_7782), s, 203).
state(28, verb(_182,_7782), s, 202).
state(28, verb(_182,_7782), s, 128).
state(28, verb(_182,_7782), s, 124).
state(28, verb(_182,_7782), s, 122).
state(28, verb(_8351,_8354), s, 121).
state(28, verb(_8438,_8441), s, 123).
state(28, verb(_8525,_8528), s, 201).
state(28, cop_phrase(_4377,cp(_8607,_8608)), 194).
state(28, e_sentence(_4475,_4476), 195).
state(28, aux(_4557,_4543), 172).
state(28, noun_phrase(_8868,_8859), 196).
state(28, aux(_4377,_8607), 131).
state(28, aux(_9018,_9021), 131).
state(29, prep_phrase(_220,pp(_1533,_1534)), 51).
state(29, prep(_220,_1533), s, 52).
state(30, prep_phrase(_599,pp(_9283,_9284)), 190).
state(30, prep(_599,_7558), s, 137).
state(31, r_clause(_182,_), 48).
state(31, rpron(_182,_4213), s, 126).
state(31, rpron(_9511,_9514), s, 126).
state(31, rpron(_9592,_9595), s, 126).
state(31, rpron(_9673,_9676), s, 126).
state(32, r_clause(_599,r_c(_9770,_9771)), 193).
state(32, rpron(_599,_9770), s, 127).
state(32, rpron(_9824,_9827), s, 127).
state(32, rpron(_9850,_9853), s, 127).
state(32, rpron(_9876,_9879), s, 127).
state(33, z_comp_phrase(_1986,z_c_ph(_9918)), 141).
state(33, e_sentence(_1986,e_s(_9949,_9950)), 142).
state(33, noun_phrase(_2707,_2097), 57).
state(34, prep_phrase(_599,pp(_10007,_10008)), 158).
state(34, prep(_599,_7558), s, 137).
state(35, cop_phrase(_1611,cp(_10077,_10078)), 164).
state(35, aux(_182,_2039), 40).
state(35, aux(_182,_2039), 40).
state(36, prep_phrase(_599,pp(_10204,_10205)), 168).
state(36, prep(_599,_7558), s, 137).
state(37, infaux(_1028,infaux(_10255,_10256,_10257)), 155).
state(37, inf(_1028,_2161), s, 39).
state(37, inf(_2637,_2640), s, 39).
state(37, inf(_2605,_2608), s, 39).
state(37, inf(_2535,_2538), s, 39).
state(38, infaux(_1028,infaux(_2161,_2162,_2163)), 166).
state(38, inf(_1028,_2161), s, 39).
state(38, inf(_2637,_2640), s, 39).
state(38, inf(_2605,_2608), s, 39).
state(38, inf(_2535,_2538), s, 39).
state(39, aux(_1028,_2162), 116).
state(39, aux(_2535,_2539), 116).
state(39, aux(_2637,_2641), 116).
state(39, aux(_2605,_2609), 116).
state(40, noun_phrase(_182,_2040), 145).
state(40, adj(_182,_2040), s, 151).
state(41, $, r2, sentence(_57,_)).
state(42, noun_phrase(_1792,_1796), 114).
state(43, noun_phrase(_1824,_1828), 115).
state(44, $, r2, verb_phrase(_2012,_)).
state(45, noun(_69,_88), s, 144).
state(46, $, r2, sentence(_170,s(_173,cp(_2039,_2040)))).
state(47,[verb(_373178,_373179),inf(_373183,_373184),modal(_373188,_373189),perf(_373193,_373
194),passive_be(_373198,_373199),prog(_373203,_373204),do(_373208,_373209),prep(_373213,_37
3214),rpron(_373218,_373219),det(_373223,_373224),noun(_373228,_373229)], r1,
noun_phrase(_182,_)).
state(48,[verb(_373600,_373601),inf(_373605,_373606),modal(_373610,_373611),perf(_373615,_373
616),passive_be(_373620,_373621),prog(_373625,_373626),do(_373630,_373631),prep(_373635,_37
3636),rpron(_373640,_373641),det(_373645,_373646),noun(_373650,_373651)], r1,
noun_phrase(_182,_)).
state(49, verb_phrase(_508,_513), 147).
state(50, perf(_508,_2243), s, 148).
state(51,[verb(_381249,_381250),inf(_381254,_381255),modal(_381259,_381260),perf(_381264,_381
265),passive_be(_381269,_381270),prog(_381274,_381275),do(_381279,_381280),prep(_381284,_38
1285),rpron(_381289,_381290),det(_381294,_381295),noun(_381299,_381300)], r1,
noun_phrase(_220,_)).
state(52, noun_phrase(_220,_1534), 72).
state(53, $, r2, sentence(_57,s(_699,_700))).
state(54,
[verb(_383052,_383053),noun(_383057,_383058),det(_383062,_383063),proper_noun(_383067,_383
068)], r1, wh_phrase(_57,wh_ph(_725,aux1(_1381)))).
state(55,[det(_383380,_383381),proper_noun(_383385,_383386)], r1, aux1(_57,aux1(_1381))).
state(56, $, r2, sentence(_768,s(_771,e_s(_2097,_2098)))).
state(57, verb_phrase(_768,_2098), 150).
state(58, $, r2, sentence(_182,s(_837,cp(_2039,_2040)))).
state(59,[verb(_387558,_387559),inf(_387563,_387564),modal(_387568,_387569),perf(_387573,_387
574),passive_be(_387578,_387579),prog(_387583,_387584),do(_387588,_387589),prep(_387593,_38
7594),rpron(_387598,_387599),det(_387603,_387604),noun(_387608,_387609)], r1,
verb_phrase2(_182,vp2(_863,_864))).
state(60, noun_phrase(_941,_946), 152).
state(61, $, r2, sentence(_1028,_)).
state(62,[verb(_391671,_391672),inf(_391676,_391677),modal(_391681,_391682),perf(_391686,_391
687),passive_be(_391691,_391692),prog(_391696,_391697),do(_391701,_391702),prep(_391706,_39
1707),rpron(_391711,_391712),det(_391716,_391717),noun(_391721,_391722)],    r1,
th_phrase(_1028,_)).
state(63, perf(_1028,_4997), s, 154).
state(64, $, r2, sentence(_1210,s(_1213,_1214))).
state(65, prog(_1028,_2281), s, 156).
state(66, $, r2, imp_verb_phrase(_599,ivp(_1291,_))).
state(67, noun(_599,_), s, 157).
state(68, verb_phrase(_57,_1351), 160).
state(69, adj(_69,_87), s, 161).
state(70, verb_phrase(_220,_1503), 162).
state(71, noun(_182,_), s, 163).
state(72,[verb(_81105,_81106),inf(_81110,_81111),modal(_81115,_81116),perf(_81120,_81121),pass
ive_be(_81125,_81126),prog(_81130,_81131),do(_81135,_81136),prep(_81140,_81141),rpron(_8114
5,_81146),det(_81150,_81151),noun(_81155,_81156)],r1,prep_phrase(_220,pp(_1533,_1534))).
state(73, noun_phrase(_1667,_1648), 165).
state(74, aux(_1028,aux(_4996,_4997)), 167).
state(75, perf(_3458,_3462), s, 173).
state(76, perf(_3458,_3462), s, 173).
state(77, perf(_3588,_3592), s, 175).
state(78, perf(_3588,_3592), s, 175).
state(79, perf(_3526,_3545), s, 177).
state(80, perf(_3526,_3545), s, 177).
state(81, prog(_5352,_5356), s, 179).
state(82, prog(_5352,_5356), s, 179).
state(83, prog(_3735,_3739), s, 181).
state(84, prog(_3735,_3739), s, 181).
state(85, passive_be(_3791,_3795), s, 183).
state(86, passive_be(_3791,_3795), s, 183).
state(87, passive_be(_3847,_3851), s, 185).
state(88, passive_be(_3847,_3851), s, 185).
state(89,[verb(_139181,_139182),modal(_139186,_139187),perf(_139191,_139192),passive_be(_139
196,_139197),prog(_139201,_139202),do(_139206,_139207)],r1,aux(_4153,aux(_4156))).
state(90,[verb(_139519,_139520),modal(_139524,_139525),perf(_139529,_139530),passive_be(_139
534,_139535),prog(_139539,_139540),do(_139544,_139545)],r1,aux(_4153,aux(_4156))).
state(91,[verb(_139857,_139858),modal(_139862,_139863),perf(_139867,_139868),passive_be(_139
872,_139873),prog(_139877,_139878),do(_139882,_139883)],r1,aux(_5682,aux(_5685))).
state(92,[verb(_140195,_140196),modal(_140200,_140201),perf(_140205,_140206),passive_be(_140
210,_140211),prog(_140215,_140216),do(_140220,_140221)],r1,aux(_5682,aux(_5685))).
state(93,[verb(_140533,_140534),modal(_140538,_140539),perf(_140543,_140544),passive_be(_140
548,_140549),prog(_140553,_140554),do(_140558,_140559)],r1,aux(_4003,aux(_4006))).
state(94,[verb(_140871,_140872),modal(_140876,_140877),perf(_140881,_140882),passive_be(_140
886,_140887),prog(_140891,_140892),do(_140896,_140897)],r1,aux(_4003,aux(_4006))).
state(95,[verb(_141209,_141210),modal(_141214,_141215),perf(_141219,_141220),passive_be(_141
224,_141225),prog(_141229,_141230),do(_141234,_141235)],r1,aux(_4103,aux(_4106))).
state(96,[verb(_141547,_141548),modal(_141552,_141553),perf(_141557,_141558),passive_be(_141
562,_141563),prog(_141567,_141568),do(_141572,_141573)],r1,aux(_4103,aux(_4106))).
state(97,[verb(_141885,_141886),modal(_141890,_141891),perf(_141895,_141896),passive_be(_141
900,_141901),prog(_141905,_141906),do(_141910,_141911)],r1,aux(_3953,aux(_3956))).
state(98,[verb(_142223,_142224),modal(_142228,_142229),perf(_142233,_142234),passive_be(_142
238,_142239),prog(_142243,_142244),do(_142248,_142249)],r1,aux(_3953,aux(_3956))).
state(99,[verb(_142561,_142562),modal(_142566,_142567),perf(_142571,_142572),passive_be(_142
576,_142577),prog(_142581,_142582),do(_142586,_142587)],r1,aux(_4053,aux(_4056))).
state(100,[verb(_142899,_142900),modal(_142904,_142905),perf(_142909,_142910),passive_be(_14
2914,_142915),prog(_142919,_142920),do(_142924,_142925)],r1,aux(_4053,aux(_4056))).
state(101,[verb(_143237,_143238),modal(_143242,_143243),perf(_143247,_143248),passive_be(_14
3252,_143253),prog(_143257,_143258),do(_143262,_143263)],r1,aux(_3903,aux(_3906))).
state(102,[verb(_143575,_143576),modal(_143580,_143581),perf(_143585,_143586),passive_be(_14
3590,_143591),prog(_143595,_143596),do(_143600,_143601)],r1,aux(_3903,aux(_3906))).
```

```
state(103,[det(_143913,_143914),proper_noun(_143918,_143919)],r1,aux1(_14400,aux1(_14403))).
state(104,[det(_144231,_144232),proper_noun(_144236,_144237)],r1,aux1(_14434,aux1(_14437))).
state(105,[det(_144549,_144550),proper_noun(_144554,_144555)],r1,aux1(_14468,aux1(_14471))).
state(106,[verb(_144867,_144868),inf(_144872,_144873),modal(_144877,_144878),perf(_144882,_14
4883),passive_be(_144887,_144888),prog(_144892,_144893),do(_144897,_144898),prep(_144902,_1
44903),rpron(_144907,_144908),det(_144912,_144913),noun(_144917,_144918)],  r1,
th_phrase(_14502,there(_14505))).
state(107, adj(_6208,_6212), s, 189).
state(108,[verb(_148624,_148625),inf(_148629,_148630),modal(_148634,_148635),perf(_148639,_14
8640),passive_be(_148644,_148645),prog(_148649,_148650),do(_148654,_148655),prep(_148659,_1
48660),rpron(_148664,_148665),det(_148669,_148670),noun(_148674,_148675)],  r1,
noun_phrase(singular,np(_14620))).
state(109, $, r1, noun_phrase(singular,np(_6099))).
state(110,[verb(_149253,_149254),inf(_149258,_149259),modal(_149263,_149264),perf(_149268,_14
9269),passive_be(_149273,_149274),prog(_149278,_149279),do(_149283,_149284),prep(_149288,_1
49289),rpron(_149293,_149294),det(_149298,_149299),noun(_149303,_149304)],  r1,
noun_phrase(_,np(_14723))).
state(111, $, r1, noun_phrase(_,np(_6119))).
state(112,[verb(_149882,_149883),inf(_149887,_149888),modal(_149892,_149893),perf(_149897,_14
9898),passive_be(_149902,_149903),prog(_149907,_149908),do(_149912,_149913),prep(_149917,_1
49918),rpron(_149922,_149923),det(_149927,_149928),noun(_149932,_149933)],  r1,
noun_phrase(_14823,np(_14826))).
state(113, $, r1, noun_phrase(_6136,np(_6139))).
state(114, prep_phrase(_1792,_1797), 222).
state(114, prep(_599,_7558), s, 137).
state(115, verb_phrase2(_1824,_1829), 225).
state(115, inf(_1824,_7622), s, 204).
state(115, inf(_1824,_7622), s, 139).
state(116, e_sentence(_2605,_2629), 244).
state(116, noun_phrase(_15132,_15123), 247).
state(116, adj(_2637,_2642), s, 246).
state(116, noun_phrase(_2535,_2540), 143).
state(116, verb_phrase(_1028,_2597), 230).
state(117, e_sentence(_768,vp(_15236,_15237)), 219).
state(117, noun_phrase(_15132,_15123), 143).
state(117, verb_phrase(_768,vp(_15236,_15237)), 270).
state(117, verb_phrase2(_768,vp2(_15236,_15237)), 271).
state(117, inf(_1824,_7622), s, 204).
state(117, inf(_768,_7622), s, 139).
state(118, e_sentence(_768,_7719), 227).
state(118, noun_phrase(_15132,np(_15358,_15359)), 143).
state(118, det(_15132,_15358), s, 290).
state(118, proper_noun(singular,_15468), s, 292).
state(118, pronoun(_15540,_15543), s, 296).
state(118, noun(_,_15618), s, 294).
state(118, noun_phrase(_15690,_15693), 209).
state(118, noun_phrase(_15771,_15774), 207).
state(118, det(_15852,_15855), s, 288).
state(119, prep_phrase(_599,pp(_7558,_7559)), 190).
state(119, prep(_599,_7558), s, 137).
state(120, r_clause(_599,r_c(_9770,_9771)), 193).
state(120, rpron(_599,_9770), s, 127).
state(120, rpron(_599,_9770), s, 127).
state(120, rpron(_599,_9770), s, 127).
state(120, rpron(_599,_9770), s, 127).
state(121, wh_phrase1(_182,vp(_16653,_16654)), 259).
state(121, wh_comp(_182,_16235), s, 205).
state(121, wh_comp(_182,_16235), s, 205).
state(121, wh_comp(_182,_16235), s, 205).
state(122, prep_phrase(_182,vp(_16653,_16654)), 260).
state(122, prep(_182,_16565), s, 171).
state(123, verb_phrase2(_182,vp2(_16653,_16654)), 262).
state(123, inf(_182,_16653), s, 263).
state(123, inf(_182,_16653), s, 300).
state(124, comp_phrase(_182,vp(_16653,_16654)), 264).
state(124, comp(_182,_16994), s, 206).
state(125, z_comp_phrase(_182,vp(_16653,_16654)), 265).
state(125, e_sentence(_182,_17162), 266).
state(125, noun_phrase(_8868,_8859), 196).
state(126, e_sentence(_182,_4476), 195).
state(126, noun_phrase(_8868,_8859), 196).
state(126, aux(_4557,_4543), 172).
state(126, verb_phrase(_182,_), 146).
state(126, cop_phrase(_4377,cp(_8607,_8608)), 194).
state(126, aux(_9018,_9021), 131).
state(126, aux(_4377,_8607), 131).
state(127, e_sentence(_9841,_9842), 285).
state(127, noun_phrase(_15132,np(_15358,_15359)), 143).
state(127, aux(_9894,_9880), 132).
state(127, verb_phrase(_599,_), 251).
state(127, cop_phrase(_599,cp(_18022,_18023)), 282).
state(127, aux(_18048,_18051), 280).
state(127, aux(_182,_2039), 132).
state(128, noun_phrase(_182,_), 258).
state(128, proper_noun(singular,_15468), s, 292).
state(128, det(_15852,_15855), s, 288).
state(128, noun(_,_15618), s, 294).
state(128, pronoun(_15540,_15543), s, 296).
state(128, det(_15132,_15358), s, 290).
state(128, noun_phrase(_15771,_15774), 207).
state(128, noun_phrase(_15690,_15693), 209).
state(129, aux(_1028,_2162), 153).
state(129, aux(_2535,_2539), 188).
state(129, aux(_2637,_2641), 187).
state(129, aux(_18850,_18854), 214).
state(130, aux(_1028,_2162), 153).
state(130, aux(_2535,_2539), 188).
state(130, aux(_2637,_2641), 187).
state(130, aux(_18850,_18854), 214).
state(131, adj(_9018,_9022), s, 268).
state(131, noun_phrase(_4377,_8608), 252).
state(132, adj(_182,_2040), s, 151).
state(132, noun_phrase(_182,_2040), 145).
state(133, $, r2, verb_phrase(_182,vp(_1859,_))).
state(134, noun(_182,_), s, 157).
state(135, $, r2, verb_phrase(_768,_)).
state(136, $, r2, verb_phrase(_182,_)).
state(137, noun_phrase(_182,_7559), 221).
state(138, $, r2, verb_phrase(_768,_)).
state(139, verb(_768,_7623), s, 224).
state(140, $, r2, verb_phrase(_768,vp(_1963,c_ph(_7718,_7719)))).
state(141, $, r2, verb_phrase(_768,vp(_1989,z_c_ph(_9918)))).
state(142, $, r1, z_comp_phrase(_768,z_c_ph(e_s(_2097,_2098)))).
state(143, verb_phrase(_768,_2098), 150).
state(144,[verb(_107689,_107690),inf(_107694,_107695),modal(_107699,_107700),perf(_107704,_10
7705),passive_be(_107709,_107710),prog(_107714,_107715),do(_107719,_107720),prep(_107724,_1
```

```
07725),rpron(_107729,_107730),det(_107734,_107735),noun(_107739,_107740)],    r1,
noun_phrase(_69,np(_86,_87,_88))).
state(145, $, r1, cop_phrase(_182,cp(_2039,_2040))).
state(146,[verb(_108326,_108327),inf(_108331,_108332),modal(_108336,_108337),perf(_108341,_10
8342),passive_be(_108346,_108347),prog(_108351,_108352),do(_108356,_108357),prep(_108361,_1
08362),rpron(_108366,_108367),det(_108371,_108372),noun(_108376,_108377)],r1,r_clause(_182,_)).
state(147, $, r2, sentence(_508,s(_511,_,_513))).
state(148,[verb(_109037,_109038),modal(_109042,_109043),perf(_109047,_109048),passive_be(_10
9052,_109053),prog(_109057,_109058),do(_109062,_109063)],r1,aux(_508,aux(_2242,_2243))).
state(149,[verb(_109379,_109380),inf(_109384,_109385),modal(_109389,_109390),perf(_109394,_10
9395),passive_be(_109399,_109400),prog(_109404,_109405),do(_109409,_109410),prep(_109414,_1
09415),rpron(_109419,_109420),det(_109424,_109425),noun(_109429,_109430)],    r1,
prep_phrase(_182,pp(_19893,_19894))).
state(150, $, r1, e_sentence(_768,e_s(_,_2098))).
state(151, $, r1, cop_phrase(_182,cp(_2039,_2040))).
state(152,[verb(_110263,_110264),inf(_110268,_110269),modal(_110273,_110274),perf(_110278,_11
0279),passive_be(_110283,_110284),prog(_110288,_110289),do(_110293,_110294),prep(_110298,_1
10299),rpron(_110303,_110304),det(_110308,_110309),noun(_110313,_110314)],    r1,
verb_phrase2(_941,vp2(_944,_945,_946))).
state(154,[verb(_113753,_113754),modal(_113758,_113759),perf(_113763,_113764),passive_be(_11
3768,_113769),prog(_113773,_113774),do(_113778,_113779)],r1,aux(_1028,aux(_4996,_4997))).
state(155, $, r2, sentence(_1028,s(_1239,aux(_2280,_2281),infaux(_2161,_2162,_2163)))).
state(156,[verb(_114328,_114329),modal(_114333,_114334),perf(_114338,_114339),passive_be(_11
4343,_114344),prog(_114348,_114349),do(_114353,_114354)],r1,aux(_1028,aux(_2280,_2281))).
state(157, $, r1, noun_phrase(_182,_)).
state(158, $, r2, imp_verb_phrase(_182,ivp(_1317,_1318,pp(_7558,_7559)))).
state(159, noun_phrase(_182,_7559), 221).
state(160, $, r2, sentence(_57,s(_1349,_,_1351))).
state(161, noun(_69,_88), s, 232).
state(162, $, r2, sentence(_182,s(_1501,_,_1503))).
state(163,[verb(_120697,_120698),inf(_120702,_120703),modal(_120707,_120708),perf(_120712,_12
0713),passive_be(_120717,_120718),prog(_120722,_120723),do(_120727,_120728),prep(_120732,_1
20733),rpron(_120737,_120738),det(_120742,_120743),noun(_120747,_120748)],    r1,
noun_phrase(_182,_)).
state(164, $, r2, sentence(_182,s(_1614,_1615,cp(_2039,_2040)))).
state(165, verb_phrase(_1643,_1649), 234).
state(166, $, r2, sentence(_1028,_)).
state(167,[verb(_124283,_124284),inf(_124288,_124289),modal(_124293,_124294),perf(_124298,_12
4299),passive_be(_124303,_124304),prog(_124308,_124309),do(_124313,_124314),prep(_124318,_1
24319),rpron(_124323,_124324),det(_124328,_124329),noun(_124333,_124334)],    r1,
th_phrase(_1028,there(_1057,aux(_4996,_4997)))).
state(168, $, r2, sentence(_182,s(_1716,_,_1718))).
state(169, noun_phrase(_182,_7559), 221).
state(170,[verb(_127330,_127331),inf(_127335,_127336),modal(_127340,_127341),perf(_127345,_12
7346),passive_be(_127350,_127351),prog(_127355,_127356),do(_127360,_127361),prep(_127365,_1
27366),rpron(_127370,_127371),det(_127375,_127376),noun(_127380,_127381)],    r1,
noun_phrase(_182,_)).
state(171, noun_phrase(_182,_16566), 236).
state(172, verb_phrase(_4539,_4544), 237).
state(173, passive_be(_3458,_3463), s, 238).
state(174, passive_be(_3458,_3463), s, 238).
state(175, prog(_3588,_3593), s, 240).
state(176, prog(_3588,_3593), s, 240).
state(177, passive_be(_3526,_3531), s, 242).
state(178, passive_be(_3526,_3531), s, 242).
state(179,[verb(_148550,_148551),modal(_148555,_148556),perf(_148560,_148561),passive_be(_14
8565,_148566),prog(_148570,_148571),do(_148575,_148576)],r1,aux(_5352,aux(_5355,_5356))).
state(180,[verb(_148892,_148893),modal(_148897,_148898),perf(_148902,_148903),passive_be(_14
8907,_148908),prog(_148912,_148913),do(_148917,_148918)],r1,aux(_5352,aux(_5355,_5356))).
state(181,[verb(_149234,_149235),modal(_149239,_149240),perf(_149244,_149245),passive_be(_14
9249,_149250),prog(_149254,_149255),do(_149259,_149260)],r1,aux(_3735,aux(_3738,_3739))).
state(182,[verb(_149576,_149577),modal(_149581,_149582),perf(_149586,_149587),passive_be(_14
9591,_149592),prog(_149596,_149597),do(_149601,_149602)],r1,aux(_3735,aux(_3738,_3739))).
state(183,[verb(_149918,_149919),modal(_149923,_149924),perf(_149928,_149929),passive_be(_14
9933,_149934),prog(_149938,_149939),do(_149943,_149944)],r1,aux(_3791,aux(_3794,_3795))).
state(184,[verb(_150260,_150261),modal(_150265,_150266),perf(_150270,_150271),passive_be(_15
0275,_150276),prog(_150280,_150281),do(_150285,_150286)],r1,aux(_3791,aux(_3794,_3795))).
state(185,[verb(_150602,_150603),modal(_150607,_150608),perf(_150612,_150613),passive_be(_15
0617,_150618),prog(_150622,_150623),do(_150627,_150628)],r1,aux(_3847,aux(_3850,_3851))).
state(186,[verb(_150944,_150945),modal(_150949,_150950),perf(_150954,_150955),passive_be(_15
0959,_150960),prog(_150964,_150965),do(_150969,_150970)],r1,aux(_3847,aux(_3850,_3851))).
state(189, noun(_6208,_6213), s, 248).
state(190, $, r1, noun_phrase(_182,_)).
state(191, noun_phrase(_182,_7559), 221).
state(192,[verb(_160005,_160006),inf(_160010,_160011),modal(_160015,_160016),perf(_160020,_16
0021),passive_be(_160025,_160026),prog(_160030,_160031),do(_160035,_160036),prep(_160040,_1
60041),rpron(_160045,_160046),det(_160050,_160051),noun(_160055,_160056)],   r1,
noun_phrase(_22286,_)).
state(193, $, r1, noun_phrase(_182,_)).
state(194,[verb(_160642,_160643),inf(_160647,_160648),modal(_160652,_160653),perf(_160657,_16
0658),passive_be(_160662,_160663),prog(_160667,_160668),do(_160672,_160673),prep(_160677,_1
60678),rpron(_160682,_160683),det(_160687,_160688),noun(_160692,_160693)],r1,r_clause(_4377,_)).
state(195,[verb(_161120,_161121),inf(_161125,_161126),modal(_161130,_161131),perf(_161135,_16
1136),passive_be(_161140,_161141),prog(_161145,_161146),do(_161150,_161151),prep(_161155,_1
61156),rpron(_161160,_161161),det(_161165,_161166),noun(_161170,_161171)],r1,r_clause(_4458,_)).
state(196, verb_phrase(_182,_8860), 253).
state(197, $, r1, noun_phrase(singular,np(_22677))).
state(198, $, r1, noun_phrase(_,np(_22701))).
state(199, $, r1, noun_phrase(_22722,np(_22725))).
state(200, adj(_22746,_22750), s, 257).
state(201,[verb(_166868,_166869),inf(_166873,_166874),modal(_166878,_166879),perf(_166883,_16
6884),passive_be(_166888,_166889),prog(_166893,_166894),do(_166898,_166899),prep(_166903,_1
66904),rpron(_166908,_166909),det(_166913,_166914),noun(_166918,_166919)],   r2,
verb_phrase(_8525,vp(_8528,_8529))).
state(202, noun_phrase(_8351,_8355), 212).
state(203, noun_phrase(_8438,_8442), 213).
state(204, verb(_768,_7680), s, 287).
state(205, e_sentence(_182,vp(_23331,_23332)), 315).
state(205, noun_phrase(_23175,_23166), 267).
state(205, verb_phrase(_182,vp(_23331,_23332)), 360).
state(205, verb_phrase2(_182,vp2(_23331,_23332)), 361).
state(205, inf(_182,_16653), s, 300).
state(205, inf(_182,_16653), s, 263).
state(206, e_sentence(_182,_16995), 319).
state(206, noun_phrase(_23175,_23166), 267).
state(207, prep_phrase(_15771,_15775), 348).
state(207, prep(_182,_23838), s, 261).
state(208, prep_phrase(_182,pp(_23926,_23927)), 348).
state(208, prep(_182,_23838), s, 261).
state(209, r_clause(_15690,r_c(_24100,_24101)), 352).
state(209, rpron(_15690,_24100), s, 301).
state(209, rpron(_24264,_24267), s, 301).
state(209, rpron(_15690,_24100), s, 301).
state(209, rpron(_24426,_24429), s, 301).
state(210, r_clause(_15690,r_c(_24100,_)), 352).
state(210, rpron(_15690,_24100), s, 302).
state(210, rpron(_24264,_24267), s, 301).
state(210, rpron(_15690,_24100), s, 301).
state(210, rpron(_24426,_24429), s, 301).
state(211, e_sentence(_768,_9842), 285).
state(211, noun_phrase(_25004,_24995), 220).
state(211, verb_phrase(_182,cp(_2039,_2040)), 251).
state(211, cop_phrase(_182,cp(_2039,_2040)), 282).
state(211, aux(_9894,_9880), 217).
state(211, aux(_182,_2039), 280).
state(211, aux(_25134,_25137), 217).
state(212, prep_phrase(_8351,_8356), 322).
state(212, prep(_182,_23838), s, 261).
state(213, verb_phrase2(_8438,_8443), 324).
state(213, inf(_182,_16653), s, 263).
state(214, e_sentence(_768,_2629), 244).
state(214, noun_phrase(_25004,_24995), 220).
state(215, e_sentence(_768,e_s(_24995,_24996)), 244).
state(215, noun_phrase(_25004,_24995), 220).
state(216, adj(_9018,_9022), s, 268).
state(216, noun_phrase(_4377,_8608), 252).
state(217, adj(_25134,_25138), s, 269).
state(217, noun_phrase(_182,_2040), 233).
state(218, $, r1, noun_phrase(_25869,np(_25872,_25873))).
state(219, $, r1, wh_phrase1(_768,_)).
state(220, verb_phrase(_768,_24996), 229).
state(221, $, r1, prep_phrase(_182,pp(_7558,_7559))).
state(222, $, r2, verb_phrase(_182,vp(_1795,_1796,_1797))).
state(223, noun_phrase(_182,_26029), 231).
state(224, $, r1, verb_phrase2(_768,vp2(_7622,_7623))).
state(225, $, r2, verb_phrase(_768,vp(_1827,_1828,_1829))).
state(226, verb(_768,_26125), s, 306).
state(227, $, r1, comp_phrase(_768,c_ph(_7718,_7719))).
state(228, verb_phrase(_768,_24996), 229).
state(229, $, r1, e_sentence(_768,e_s(_24995,_24996))).
state(230, $, r1, infaux(_1028,infaux(_2161,_2162,_2163))).
state(231, $, r1, prep_phrase(_182,pp(_26028,_26029))).
state(232,[verb(_301849,_301850),inf(_301854,_301855),modal(_301859,_301860),perf(_301864,_30
1865),passive_be(_301869,_301870),prog(_301874,_301875),do(_301879,_301880),prep(_301884,_3
01885),rpron(_301889,_301890),det(_301894,_301895),noun(_301899,_301900)],  r1,
noun_phrase(_69,np(_86,_87,_88))).
state(233, $, r1, cop_phrase(_182,cp(_2039,_2040))).
state(234, $, r2, sentence(_1643,_)).
state(235, $, r1, prep_phrase(_182,pp(_26473,_26474))).
state(236,[verb(_302938,_302939),inf(_302943,_302944),modal(_302948,_302949),perf(_302953,_30
2954),passive_be(_302958,_302959),prog(_302963,_302964),do(_302968,_302969),prep(_302973,_3
02974),rpron(_302978,_302979),det(_302983,_302984),noun(_302988,_302989)],  r1,
prep_phrase(_182,pp(_16565,_16566))).
state(237,[verb(_303392,_303393),inf(_303397,_303398),modal(_303402,_303403),perf(_303407,_30
3408),passive_be(_303412,_303413),prog(_303417,_303418),do(_303422,_303423),prep(_303427,_3
03428),rpron(_303432,_303433),det(_303437,_303438),noun(_303442,_303443)],  r1,
r_clause(_4539,r_c(_4542,_4543,_4544))).
state(238, prog(_3458,_3464), s, 308).
state(239, prog(_3458,_3464), s, 308).
state(240,[verb(_309318,_309319),modal(_309323,_309324),perf(_309328,_309329),passive_be(_30
9333,_309334),prog(_309338,_309339),do(_309343,_309344)],r1,aux(_3588,aux(_3591,_3592,_3593))).
state(241,[verb(_309664,_309665),modal(_309669,_309670),perf(_309674,_309675),passive_be(_30
9679,_309680),prog(_309684,_309685),do(_309689,_309690)],r1,aux(_3588,aux(_3591,_3592,_3593))).
state(242,[verb(_310010,_310011),modal(_310015,_310016),perf(_310020,_310021),passive_be(_31
0025,_310026),prog(_310030,_310031),do(_310035,_310036)],r1,aux(_3526,aux(_3529,_3530,_3531))).
state(243,[verb(_310356,_310357),modal(_310361,_310362),perf(_310366,_310367),passive_be(_31
0371,_310372),prog(_310376,_310377),do(_310381,_310382)],r1,aux(_3526,aux(_3529,_3530,_3531))).
state(244, $, r1, infaux(_768,infaux(_2608,_2609,_2610))).
state(245, verb_phrase(_768,_27124), 304).
state(246, $, r1, infaux(_2637,infaux(_2640,_2641,_2642))).
state(247, adj(_2535,_2541), s, 311).
state(248, $, r1, noun_phrase(_6208,np(_6211,_6212,_6213))).
state(249, $, r1, prep_phrase(_27264,pp(_27267,_27268))).
state(250,[verb(_316477,_316478),inf(_316482,_316483),modal(_316487,_316488),perf(_316492,_31
6493),passive_be(_316497,_316498),prog(_316502,_316503),do(_316507,_316508),prep(_316512,_3
16513),rpron(_316517,_316518),det(_316522,_316523),noun(_316527,_316528)], r1,
r_clause(_27294,r_c(_27297,_27298))).
state(251, $, r1, r_clause(_182,r_c(_9770,_))).
state(252,[verb(_317170,_317171),inf(_317175,_317176),modal(_317180,_317181),perf(_317185,_31
7186),passive_be(_317190,_317191),prog(_317195,_317196),do(_317200,_317201),prep(_317205,_3
17206),rpron(_317210,_317211),det(_317215,_317216),noun(_317220,_317221)], r1,
cop_phrase(_4377,cp(_8607,_8608))).
state(253,[verb(_317844,_317845),inf(_317849,_317850),modal(_317854,_317855),perf(_317859,_31
7860),passive_be(_317864,_317865),prog(_317869,_317870),do(_317874,_317875),prep(_317879,_3
17880),rpron(_317884,_317885),det(_317889,_317890),noun(_317894,_317895)], r1,
e_sentence(_182,e_s(_8859,_8860))).
state(254, $, r1, noun_phrase(_27579,np(_27582,_27583))).
state(255, noun_phrase(_27579,_27613), 305).
state(256, $, r1, noun_phrase(_182,_)).
state(257, noun(_22746,_22751), s, 314).
state(258,[verb(_323127,_323128),inf(_323132,_323133),modal(_323137,_323138),perf(_323142,_32
3143),passive_be(_323147,_323148),prog(_323152,_323153),do(_323157,_323158),prep(_323162,_3
23163),rpron(_323167,_323168),det(_323172,_323173),noun(_323177,_323178)], r2,
verb_phrase(_182,_)).
state(259,[verb(_323675,_323676),inf(_323680,_323681),modal(_323685,_323686),perf(_323690,_32
3691),passive_be(_323695,_323696),prog(_323700,_323701),do(_323705,_323706),prep(_323710,_3
23711),rpron(_323715,_323716),det(_323720,_323721),noun(_323725,_323726)], r2,
verb_phrase(_182,_)).
state(261, noun_phrase(_182,_23839), 317).
state(262,[verb(_327003,_327004),inf(_327008,_327009),modal(_327013,_327014),perf(_327018,_32
7019),passive_be(_327023,_327024),prog(_327028,_327029),do(_327033,_327034),prep(_327038,_3
27039),rpron(_327043,_327044),det(_327048,_327049),noun(_327053,_327054)], r2,
verb_phrase(_182,_)).
state(263, verb(_182,_16654), s, 318).
state(264,[verb(_329958,_329959),inf(_329963,_329964),modal(_329968,_329969),perf(_329973,_32
9974),passive_be(_329978,_329979),prog(_329983,_329984),do(_329988,_329989),prep(_329993,_3
29994),rpron(_329998,_329999),det(_330003,_330004),noun(_330008,_330009)], r2,
verb_phrase(_182,_)).
state(265,
[verb(_330506,_330507),inf(_330511,_330512),modal(_330516,_330517),perf(_330521,_330522),pas
sive_be(_330526,_330527),prog(_330531,_330532),do(_330536,_330537),prep(_330541,_330542),rp
ron(_330546,_330547),det(_330551,_330552),noun(_330556,_330557)], r2, verb_phrase(_182,_)).
state(266,
[verb(_331036,_331037),inf(_331041,_331042),modal(_331046,_331047),perf(_331051,_331052),pas
sive_be(_331056,_331057),prog(_331061,_331062),do(_331066,_331067),prep(_331071,_331072),rp
ron(_331076,_331077),det(_331081,_331082),noun(_331086,_331087)], r1,
z_comp_phrase(_182,z_c_ph(_17162))).
state(267, verb_phrase(_182,_23167), 321).
state(268,
[verb(_333882,_333883),inf(_333887,_333888),modal(_333892,_333893),perf(_333897,_333898),pas
sive_be(_333902,_333903),prog(_333907,_333908),do(_333912,_333913),prep(_333917,_333918),rp
ron(_333922,_333923),det(_333927,_333928),noun(_333932,_333933)], r1,
cop_phrase(_9018,cp(_9021,_9022))).
state(269, $, r1, cop_phrase(_25134,cp(_25137,_25138))).
state(270, $, r1, wh_phrase1(_768,wh_ph(_7442,vp(_7622,_7623)))).
state(271, $, r1, wh_phrase1(_768,wh_ph(_7442,vp(_7622,_7623)))).
state(272, verb(_768,_7623), s, 306).
state(273, verb_phrase(_28772,_28796), 327).
state(274, verb_phrase(_28772,_28796), 327).
state(275, adj(_28844,_28849), s, 333).
state(276, adj(_28844,_28849), s, 333).
state(277, noun_phrase(_28916,_28921), 335).
state(278, noun_phrase(_28916,_28921), 335).
state(279, verb_phrase(_29000,_29005), 337).
state(280, verb_phrase(_182,_9881), 338).
state(281,[verb(_355382,_355383),inf(_355387,_355388),modal(_355392,_355393),perf(_355397,_35
5398),passive_be(_355402,_355403),prog(_355407,_355408),do(_355412,_355413),prep(_355417,_3
55418),rpron(_355422,_355423),det(_355427,_355428),noun(_355432,_355433)],    r1,
r_clause(_4377,r_c(_29130,_))).
state(282, $, r1, r_clause(_182,r_c(_9770,_))).
state(283,[verb(_356075,_356076),inf(_356080,_356081),modal(_356085,_356086),perf(_356090,_35
6091),passive_be(_56095,_356096),prog(_356100,_356101),do(_356105,_356106),prep(_356110,_35
6111),rpron(_356115,_356116),det(_356120,_356121),noun(_356125,_356126)],    r1,
r_clause(_29254,r_c(_29257,_29258))).
state(284, verb_phrase(_182,_23167), 321).
state(285, $, r1, r_clause(_182,r_c(_9770,_))).
state(286, verb_phrase(_768,_27124), 304).
state(287, noun_phrase(_768,_7681), 343).
state(288, adj(_15852,_15856), s, 344).
state(289, adj(_15852,_15856), s, 344).
state(290, noun(_2707,_15359), s, 346).
state(291, noun(_2707,_15359), s, 346).
state(292,[verb(_370852,_370853),inf(_370857,_370858),modal(_370862,_370863),perf(_370867,_37
0868),passive_be(_370872,_370873),prog(_370877,_370878),do(_370882,_370883),prep(_370887,_3
70888),rpron(_370892,_370893),det(_370897,_370898),noun(_370902,_370903)],    r1,
noun_phrase(singular,np(_15468))).
state(293,[verb(_371270,_371271),inf(_371275,_371276),modal(_371280,_371281),perf(_371285,_37
1286),passive_be(_371290,_371291),prog(_371295,_371296),do(_371300,_371301),prep(_371305,_3
71306),rpron(_371310,_371311),det(_371315,_371316),noun(_371320,_371321)],    r1,
noun_phrase(singular,np(_15468))).
state(294,[verb(_371688,_371689),inf(_371693,_371694),modal(_371698,_371699),perf(_371703,_37
1704),passive_be(_371708,_371709),prog(_371713,_371714),do(_371718,_371719),prep(_371723,_3
71724),rpron(_371728,_371729),det(_371733,_371734),noun(_371738,_371739)],    r1,
noun_phrase(_,np(_15618))).
state(295,[verb(_372106,_372107),inf(_372111,_372112),modal(_372116,_372117),perf(_372121,_37
2122),passive_be(_372126,_372127),prog(_372131,_372132),do(_372136,_372137),prep(_372141,_3
72142),rpron(_372146,_372147),det(_372151,_372152),noun(_372156,_372157)],    r1,
noun_phrase(_,np(_15618))).
state(296,[verb(_372524,_372525),inf(_372529,_372530),modal(_372534,_372535),perf(_372539,_37
2540),passive_be(_372544,_372545),prog(_372549,_372550),do(_372554,_372555),prep(_372559,_3
72560),rpron(_372564,_372565),det(_372569,_372570),noun(_372574,_372575)],    r1,
noun_phrase(_15540,np(_15543))).
state(297,[verb(_372942,_372943),inf(_372947,_372948),modal(_372952,_372953),perf(_372957,_37
2958),passive_be(_372962,_372963),prog(_372967,_372968),do(_372972,_372973),prep(_372977,_3
72978),rpron(_372982,_372983),det(_372987,_372988),noun(_372992,_372993)],    r1,
noun_phrase(_15540,np(_15543))).
state(298, verb(_30346,_30350), s, 363).
state(299, verb(_30346,_30350), s, 363).
state(300, verb(_182,_16821), s, 365).
state(301, e_sentence(_24281,_24282), 393).
state(301, noun_phrase(_30604,_30595), 316).
state(301, aux(_24444,_24430), 390).
state(301, verb_phrase(_182,_), 381).
state(301, cop_phrase(_182,cp(aux(_231,_24786,_24787,_24788),_24783)), 382).
state(301, aux(_30928,aux(_30934,_30935,_30936,_30937)), 367).
state(301, aux(_182,aux(_231,_24786,_24787,_24788)), 367).
state(302, e_sentence(_182,e_s(_30595,_30596)), 393).
state(302, noun_phrase(_30604,_30595), 316).
state(302, aux(_24444,_24430), 390).
state(302, verb_phrase(_182,vp(_302,_24613,_24614)), 397).
state(302, cop_phrase(_182,cp(aux(_231,_24786,_24787,_24788),_24783)), 382).
state(302, aux(_30928,aux(_30934,_24786,_24787,_24788)), 367).
state(302, aux(_182,aux(_231,_24786,_24787,_24788)), 367).
state(303, adj(_31720,_31724), s, 359).
state(303, noun_phrase(_31748,_31752), 340).
state(304, $, r1, e_sentence(_768,e_s(_27123,_27124))).
state(305, $, r1, prep_phrase(_27579,pp(_27612,_27613))).
state(306, $, r1, verb_phrase2(_768,vp2(_7622,_7623))).
state(307, $, r1, e_sentence(_31868,e_s(_31871,_31872))).
state(308,[verb(_156109,_156110),modal(_156114,_156115),perf(_156119,_156120),passive_be(_15
6124,_156125),prog(_156129,_156130),do(_156134,_156135)],r1,aux(_3458,aux(_3461,_3462,_3463,
_3464))).
state(309,[verb(_156459,_156460),modal(_156464,_156465),perf(_156469,_156470),passive_be(_15
6474,_156475),prog(_156479,_156480),do(_156484,_156485)],r1,aux(_3458,aux(_3461,_3462,_3463,
_3464))).
state(310, $, r1, e_sentence(_32042,e_s(_32045,_32046))).
state(311, $, r1, infaux(_2535,infaux(_2538,_2539,_2540,_2541))).
state(312, $, r1, prep_phrase(_32114,pp(_32117,_32118))).
state(313, $, r1, r_clause(_32144,r_c(_32147,_32148))).
state(314, $, r1, noun_phrase(_22746,np(_22749,_22750,_22751))).
state(315,[verb(_157888,_157889),inf(_157893,_157894),modal(_157898,_157899),perf(_157903,_15
7904),passive_be(_157908,_157909),prog(_157913,_157914),do(_157918,_157919),prep(_157923,_1
57924),rpron(_157928,_157929),det(_157933,_157934),noun(_157938,_157939)],   r1,
wh_phrase1(_182,wh_ph(_16235,vp(_23331,_23332)))).
state(316, verb_phrase(_182,_23167), 321).
state(317,[verb(_159376,_159377),inf(_159381,_159382),modal(_159386,_159387),perf(_159391,_15
9392),passive_be(_159396,_159397),prog(_159401,_159402),do(_159406,_159407),prep(_159411,_1
59412),rpron(_159416,_159417),det(_159421,_159422),noun(_159426,_159427)],   r1,
prep_phrase(_182,pp(_23838,_23839))).
state(318,[verb(_159826,_159827),inf(_159831,_159832),modal(_159836,_159837),perf(_159841,_15
9842),passive_be(_159846,_159847),prog(_159851,_159852),do(_159856,_159857),prep(_159861,_1
59862),rpron(_159866,_159867),det(_159871,_159872),noun(_159876,_159877)],   r1,
verb_phrase2(_182,vp2(_16653,_16654))).
state(319,[verb(_160388,_160389),inf(_160393,_160394),modal(_160398,_160399),perf(_160403,_16
0404),passive_be(_160408,_160409),prog(_160413,_160414),do(_160418,_160419),prep(_160423,_1
60424),rpron(_160428,_160429),det(_160433,_160434),noun(_160438,_160439)],   r1,
comp_phrase(_182,c_ph(_16994,_16995))).
state(320, verb_phrase(_182,_23167), 321).
state(321,[verb(_161616,_161617),inf(_161621,_161622),modal(_161626,_161627),perf(_161631,_16
1632),passive_be(_161636,_161637),prog(_161641,_161642),do(_161646,_161647),prep(_161651,_1
61652),rpron(_161656,_161657),det(_161661,_161662),noun(_161666,_161667)],   r1,
e_sentence(_182,e_s(_23166,_23167))).
state(322,[verb(_162140,_162141),inf(_162145,_162146),modal(_162150,_162151),perf(_162155,_16
2156),passive_be(_162160,_162161),prog(_162165,_162166),do(_162170,_162171),prep(_162175,_1
62176),rpron(_162180,_162181),det(_162185,_162186),noun(_162190,_162191)],   r2,
verb_phrase(_182,vp(_8354,_8355,_8356))).
state(323, noun_phrase(_182,_32900), 371).
state(324,[verb(_164684,_164685),inf(_164689,_164690),modal(_164694,_164695),perf(_164699,_16
4700),passive_be(_164704,_164705),prog(_164709,_164710),do(_164714,_164715),prep(_164719,_1
64720),rpron(_164724,_164725),det(_164729,_164730),noun(_164734,_164735)], r2,
verb_phrase(_182,vp(_8441,_8442,_8443))).
state(325, verb(_182,_33076), s, 372).
state(326, $, r1, verb_phrase2(_33157,vp2(_33160,_33161))).
state(327, $, r1, infaux(_28772,infaux(_28775,_28776,_28777))).
state(328, $, r1, infaux(_28772,infaux(_28775,_28776,_28777))).
state(329, $, r1, infaux(_33259,infaux(_33262,_33263,_33264))).
state(330, verb_phrase(_33259,_33299), 342).
state(331, $, r1, infaux(_33259,infaux(_33262,_33263,_33264))).
state(332, verb_phrase(_33259,_33299), 342).
state(333, $, r1, infaux(_28844,infaux(_28847,_28848,_28849))).
state(334, $, r1, infaux(_28844,infaux(_28847,_28848,_28849))).
state(335, adj(_28916,_28922), s, 375).
state(336, adj(_28916,_28922), s, 375).
state(337,[verb(_174382,_174383),inf(_174387,_174388),modal(_174392,_174393),perf(_174397,_17
4398),passive_be(_174402,_174403),prog(_174407,_174408),do(_174412,_174413),prep(_174417,_1
74418),rpron(_174422,_174423),det(_174427,_174428),noun(_174432,_174433)], r1,
r_clause(_29000,r_c(_29003,_29004,_29005))).
state(338, $, r1, r_clause(_182,r_c(_9770,_9880,_9881))).
state(339,[verb(_175079,_175080),inf(_175084,_175085),modal(_175089,_175090),perf(_175094,_17
5095),passive_be(_175099,_175100),prog(_175104,_175105),do(_175109,_175110),prep(_175114,_1
75115),rpron(_175119,_175120),det(_175124,_175125),noun(_175129,_175130)], r1,
cop_phrase(_33677,cp(_33680,_33681))).
state(340, $, r1, cop_phrase(_31748,cp(_31751,_31752))).
state(341,[verb(_175968,_175969),inf(_175973,_175974),modal(_175978,_175979),perf(_175983,_17
5984),passive_be(_175988,_175989),prog(_175993,_175994),do(_175998,_175999),prep(_176003,_1
76004),rpron(_176008,_176009),det(_176013,_176014),noun(_176018,_176019)], r1,
e_sentence(_33792,e_s(_33795,_33796))).
state(342, $, r1, e_sentence(_33259,e_s(_33298,_33299))).
state(343, $, r1, verb_phrase2(_768,vp2(_7622,_7680,_7681))).
state(344, noun(_15852,_15857), s, 377).
state(345, noun(_15852,_15857), s, 377).
state(346,[verb(_180000,_180001),inf(_180005,_180006),modal(_180010,_180011),perf(_180015,_18
0016),passive_be(_180020,_180021),prog(_180025,_180026),do(_180030,_180031),prep(_180035,_1
80036),rpron(_180040,_180041),det(_180045,_180046),noun(_180050,_180051)], r1,
noun_phrase(_2707,np(_15358,_15359))).
state(347,[verb(_180422,_180423),inf(_180427,_180428),modal(_180432,_180433),perf(_180437,_18
0438),passive_be(_180442,_180443),prog(_180447,_180448),do(_180452,_180453),prep(_180457,_1
80458),rpron(_180462,_180463),det(_180467,_180468),noun(_180472,_180473)], r1,
noun_phrase(_2707,np(_15358,_15359))).
state(348,[verb(_180844,_180845),inf(_180849,_180850),modal(_180854,_180855),perf(_180859,_18
0860),passive_be(_180864,_180865),prog(_180869,_180870),do(_180874,_180875),prep(_180879,_1
80880),rpron(_180884,_180885),det(_180889,_180890),noun(_180894,_180895)], r1,
noun_phrase(_182,_)).
state(349, noun_phrase(_182,_32900), 371).
state(350,[verb(_182478,_182479),inf(_182483,_182484),modal(_182488,_182489),perf(_182493,_18
2494),passive_be(_182498,_182499),prog(_182503,_182504),do(_182508,_182509),prep(_182513,_1
82514),rpron(_182518,_182519),det(_182523,_182524),noun(_182528,_182529)], r1,
noun_phrase(_182,_)).
state(351, noun_phrase(_182,_32900), 371).
state(352,[verb(_184052,_184053),inf(_184057,_184058),modal(_184062,_184063),perf(_184067,_18
4068),passive_be(_184072,_184073),prog(_184077,_184078),do(_184082,_184083),prep(_184087,_1
84088),rpron(_184092,_184093),det(_184097,_184098),noun(_184102,_184103)], r1,
noun_phrase(_182,_)).
state(353,[verb(_184474,_184475),inf(_184479,_184480),modal(_184484,_184485),perf(_184489,_18
4490),passive_be(_184494,_184495),prog(_184499,_184500),do(_184504,_184505),prep(_184509,_1
84510),rpron(_184514,_184515),det(_184519,_184520),noun(_184524,_184525)], r1,
noun_phrase(_182,_)).
state(354, verb_phrase(_34823,_34828), 383).
state(355, $, r1, r_clause(_31748,r_c(_34862,_))).
state(356, $, r1, r_clause(_34895,r_c(_34898,_34899))).
state(357, verb_phrase(_34910,_34929), 373).
state(358,[verb(_187902,_187903),inf(_187907,_187908),modal(_187912,_187913),perf(_187917,_18
7918),passive_be(_187922,_187923),prog(_187927,_187928),do(_187932,_187933),prep(_187937,_1
87938),rpron(_187942,_187943),det(_187947,_187948),noun(_187952,_187953)], r1,
cop_phrase(_34955,cp(_34958,_34959))).
state(359, $, r1, cop_phrase(_31720,cp(_31723,_31724))).
state(360,[verb(_188791,_188792),inf(_188796,_188797),modal(_188801,_188802),perf(_188806,_18
8807),passive_be(_188811,_188812),prog(_188816,_188817),do(_188821,_188822),prep(_188826,_1
88827),rpron(_188831,_188832),det(_188836,_188837),noun(_188841,_188842)], r1,
wh_phrase1(_182,wh_ph(_16235,vp(_23331,_23332)))).
state(361,[verb(_189521,_189522),inf(_189526,_189527),modal(_189531,_189532),perf(_189536,_18
9537),passive_be(_189541,_189542),prog(_189546,_189547),do(_189551,_189552),prep(_189556,_1
89557),rpron(_189561,_189562),det(_189566,_189567),noun(_189571,_189572)], r1,
wh_phrase1(_182,wh_ph(_16235,vp(_23331,_23332)))).
state(362, verb(_182,_23332), s, 372).
state(363, noun_phrase(_30346,_30351), 387).
state(364, noun_phrase(_30346,_30351), 387).
state(365, noun_phrase(_182,_16822), 389).
state(366, verb(_35494,_35498), s, 392).
state(367, adj(_182,_24783), s, 406).
state(367, noun_phrase(_182,_24783), 400).
state(368, adj(_182,_24783), s, 406).
state(368, noun_phrase(_182,_24783), 400).
state(369,[verb(_203403,_203404),inf(_203408,_203409),modal(_203413,_203414),perf(_203418,_20
3419),passive_be(_203423,_203424),prog(_203428,_203429),do(_203433,_203434),prep(_203438,_2
03439),rpron(_203443,_203444),det(_203448,_203449),noun(_203453,_203454)], r1,
e_sentence(_35941,e_s(_35944,_35945))).
state(370,[verb(_203909,_203910),inf(_203914,_203915),modal(_203919,_203920),perf(_203924,_20
3925),passive_be(_203929,_203930),prog(_203934,_203935),do(_203939,_203940),prep(_203944,_2
03945),rpron(_203949,_203950),det(_203954,_203955),noun(_203959,_203960)], r1,
e_sentence(_36026,e_s(_36029,_36030))).
state(371,[verb(_204415,_204416),inf(_204420,_204421),modal(_204425,_204426),perf(_204430,_20
4431),passive_be(_204435,_204436),prog(_204440,_204441),do(_204445,_204446),prep(_204450,_2
04451),rpron(_204455,_204456),det(_204460,_204461),noun(_204465,_204466)], r1,
prep_phrase(_182,pp(_32899,_32900))).
state(372,[verb(_204865,_204866),inf(_204870,_204871),modal(_204875,_204876),perf(_204880,_20
4881),passive_be(_204885,_204886),prog(_204890,_204891),do(_204895,_204896),prep(_204900,_2
04901),rpron(_204905,_204906),det(_204910,_204911),noun(_204915,_204916)], r1,
verb_phrase2(_182,vp2(_23331,_23332))).
state(373, $, r1, e_sentence(_34910,e_s(_34928,_34929))).
state(374, $, r1, e_sentence(_34910,e_s(_34928,_34929))).
state(375, $, r1, infaux(_28916,infaux(_28919,_28920,_28921,_28922))).
state(376, $, r1, infaux(_28916,infaux(_28919,_28920,_28921,_28922))).
state(377,[verb(_206307,_206308),inf(_206312,_206313),modal(_206317,_206318),perf(_206322,_20
6323),passive_be(_206327,_206328),prog(_206332,_206333),do(_206337,_206338),prep(_206342,_2
06343),rpron(_206347,_206348),det(_206352,_206353),noun(_206357,_206358)], r1,
noun_phrase(_15852,np(_15855,_15856,_15857))).
state(378,[verb(_206733,_206734),inf(_206738,_206739),modal(_206743,_206744),perf(_206748,_20
6749),passive_be(_206753,_206754),prog(_206758,_206759),do(_206763,_206764),prep(_206768,_2
06769),rpron(_206773,_206774),det(_206778,_206779),noun(_206783,_206784)], r1,
noun_phrase(_15852,np(_15855,_15856,_15857))).
state(379,[verb(_207155,_207156),inf(_207160,_207161),modal(_207165,_207166),perf(_207170,_20
7171),passive_be(_207175,_207176),prog(_207180,_207181),do(_207185,_207186),prep(_207190,_2
07191),rpron(_207195,_207196),det(_207200,_207201),noun(_207205,_207206)], r1,
prep_phrase(_36607,pp(_36610,_36611))).
state(380,[verb(_207605,_207606),inf(_207610,_207611),modal(_207615,_207616),perf(_207620,_20
7621),passive_be(_207625,_207626),prog(_207630,_207631),do(_207635,_207636),prep(_207640,_2
07641),rpron(_207645,_207646),det(_207650,_207651),noun(_207655,_207656)], r1,
prep_phrase(_36607,pp(_36610,_36611))).
state(381,[verb(_208055,_208056),inf(_208060,_208061),modal(_208065,_208066),perf(_208070,_20
8071),passive_be(_208075,_208076),prog(_208080,_208081),do(_208085,_208086),prep(_208090,_2
08091),rpron(_208095,_208096),det(_208100,_208101),noun(_208105,_208106)], r1,
r_clause(_182,r_c(_24100,_))).
state(382,[verb(_208533,_208534),inf(_208538,_208539),modal(_208543,_208544),perf(_208548,_20
8549),passive_be(_208553,_208554),prog(_208558,_208559),do(_208563,_208564),prep(_208568,_2
08569),rpron(_208573,_208574),det(_208578,_208579),noun(_208583,_208584)], r1,
r_clause(_182,r_c(_24100,_,_24783))).
state(383, $, r1, r_clause(_34823,r_c(_34826,_34827,_34828))).
state(384, $, r1, cop_phrase(_36989,cp(_36992,_36993))).
state(385, $, r1, e_sentence(_37019,e_s(_37022,_37023))).
state(386,[verb(_209660,_209661),inf(_209665,_209666),modal(_209670,_209671),perf(_209675,_20
9676),passive_be(_209680,_209681),prog(_209685,_209686),do(_209690,_209691),prep(_209695,_2
09696),rpron(_209700,_209701),det(_209705,_209706),noun(_209710,_209711)], r1,
verb_phrase2(_37049,vp2(_37052,_37053))).
state(387, $, r1, verb_phrase2(_30346,vp2(_30349,_30350,_30351))).
state(388, $, r1, verb_phrase2(_30346,vp2(_30349,_30350,_30351))).
state(389,[verb(_210664,_210665),inf(_210669,_210670),modal(_210674,_210675),perf(_210679,_21
0680),passive_be(_210684,_210685),prog(_210689,_210690),do(_210694,_210695),prep(_210699,_2
10700),rpron(_210704,_210705),det(_210709,_210710),noun(_210714,_210715)], r1,
verb_phrase2(_182,vp2(_16653,_16821,_16822))).
state(390, verb_phrase(_24426,_24431), 401).
state(391, verb_phrase(_24426,_24431), 401).
state(392, noun_phrase(_35494,_35499), 403).
state(393,[verb(_213984,_213985),inf(_213989,_213990),modal(_213994,_213995),perf(_213999,_21
4000),passive_be(_214004,_214005),prog(_214009,_214010),do(_214014,_214015),prep(_214019,_2
14020),rpron(_214024,_214025),det(_214029,_214030),noun(_214034,_214035)], r1,
r_clause(_24264,r_c(_24267,_24268))).
state(394, verb_phrase(_182,_37659), 404).
state(395,[verb(_215370,_215371),inf(_215375,_215376),modal(_215380,_215381),perf(_215385,_21
5386),passive_be(_215390,_215391),prog(_215395,_215396),do(_215400,_215401),prep(_215405,_2
15406),rpron(_215410,_215411),det(_215415,_215416),noun(_215420,_215421)], r1,
r_clause(_24264,r_c(_24267,_24268))).
state(396, verb_phrase(_182,_37659), 404).
state(397,[verb(_216696,_216697),inf(_216701,_216702),modal(_216706,_216707),perf(_216711,_21
6712),passive_be(_216716,_216717),prog(_216721,_216722),do(_216726,_216727),prep(_216731,_2
16732),rpron(_216736,_216737),det(_216741,_216742),noun(_216746,_216747)],r1,r_clause(_182,_)).
state(398,[verb(_217174,_217175),inf(_217179,_217180),modal(_217184,_217185),perf(_217189,_21
7190),passive_be(_217194,_217195),prog(_217199,_217200),do(_217204,_217205),prep(_217209,_2
17210),rpron(_217214,_217215),det(_217219,_217220),noun(_217224,_217225)], r1,
r_clause(_182,r_c(_38009,_))).
state(399, $, r1, cop_phrase(_38107,cp(_38110,_38111))).
state(400,[verb(_217867,_217868),inf(_217872,_217873),modal(_217877,_217878),perf(_217882,_21
7883),passive_be(_217887,_217888),prog(_217892,_217893),do(_217897,_217898),prep(_217902,_2
17903),rpron(_217907,_217908),det(_217912,_217913),noun(_217917,_217918)], r1,
cop_phrase(_182,cp(aux(_231,_24786,_24787,_24788),_24783))).
state(401,[verb(_218545,_218546),inf(_218550,_218551),modal(_218555,_218556),perf(_218560,_21
8561),passive_be(_218565,_218566),prog(_218570,_218571),do(_218575,_218576),prep(_218580,_2
18581),rpron(_218585,_218586),det(_218590,_218591),noun(_218595,_218596)], r1,
r_clause(_24426,r_c(_24429,_24430,_24431))).
state(402,[verb(_219027,_219028),inf(_219032,_219033),modal(_219037,_219038),perf(_219042,_21
9043),passive_be(_219047,_219048),prog(_219052,_219053),do(_219057,_219058),prep(_219062,_2
19063),rpron(_219067,_219068),det(_219072,_219073),noun(_219077,_219078)], r1,
r_clause(_24426,r_c(_24429,_24430,_24431))).
state(403,[verb(_219509,_219510),inf(_219514,_219515),modal(_219519,_219520),perf(_219524,_21
9525),passive_be(_219529,_219530),prog(_219534,_219535),do(_219539,_219540),prep(_219544,_2
19545),rpron(_219549,_219550),det(_219554,_219555),noun(_219559,_219560)], r1,
verb_phrase2(_35494,vp2(_35497,_35498,_35499))).
state(404,[verb(_220071,_220072),inf(_220076,_220077),modal(_220081,_220082),perf(_220086,_22
0087),passive_be(_220091,_220092),prog(_220096,_220097),do(_220101,_220102),prep(_220106,_2
20107),rpron(_220111,_220112),det(_220116,_220117),noun(_220121,_220122)], r1,
e_sentence(_182,e_s(_37658,_37659))).
state(405,[verb(_220577,_220578),inf(_220582,_220583),modal(_220587,_220588),perf(_220592,_22
0593),passive_be(_220597,_220598),prog(_220602,_220603),do(_220607,_220608),prep(_220612,_2
20613),rpron(_220617,_220618),det(_220622,_220623),noun(_220627,_220628)], r1,
e_sentence(_182,e_s(_37658,_37659))).
state(406,[verb(_221083,_221084),inf(_221088,_221089),modal(_221093,_221094),perf(_221098,_22
1099),passive_be(_221103,_221104),prog(_221108,_221109),do(_221113,_221114),prep(_221118,_2
21119),rpron(_221123,_221124),det(_221128,_221129),noun(_221133,_221134)], r1,
cop_phrase(_182,cp(aux(_231,_24786,_24787,_24788),_24783))).
state(407,[verb(_221757,_221758),inf(_221762,_221763),modal(_221767,_221768),perf(_221772,_22
1773),passive_be(_221777,_221778),prog(_221782,_221783),do(_221787,_221788),prep(_221792,_2
21793),rpron(_221797,_221798),det(_221802,_221803),noun(_221807,_221808)], r1,
cop_phrase(_38760,cp(aux(_38766,_38767,_38768,_38769),_38764))).
state(408,[verb(_222431,_222432),inf(_222436,_222437),modal(_222441,_222442),perf(_222446,_22
2447),passive_be(_222451,_222452),prog(_222456,_222457),do(_222461,_222462),prep(_222466,_2
22467),rpron(_222471,_222472),det(_222476,_222477),noun(_222481,_222482)], r1,
cop_phrase(_38855,cp(_38858,_38859))).
```

APPENDIX C

This appendix contains further examples of the types of sentences and parses of these sentences that MParser and LParser can process. Some of the sentences are parsed by both parsers, the others are parsed by either MParser or LParser. The example sentences are the following:

I told that boy that boys should do it.

I wanted John to do it.

John bought a pear for Mary to eat.

I want to do it.

Sally knows Bill kissed Jane.

Sally has been forgotten.

I know John to be a fool.

Who did Sally kiss.

I know the boy that likes Sue.

I know the boy Sue likes.

The boy's cat is dead.

Block the road.

I know the boy that Sally likes.

The jar seems broken.

Schedule a meeting for Friday.

There seems to have been a meeting scheduled for Friday.

The examples below are results of parsing with MParser.

```
| ?- start(S).|: I told that boy that boys should do it.
S =[[[xmax,[[attach_subject,[xmax,
[attach_propnoun,['I','+n-v+a-p',pn]],'+n-v+a-p']],
    [attach_vp,[xmax,[[attach_verb,[told,'-n+v',-,tense]],
    [attach_object,
    [xmax,[[[attach_det,[that,'+n-v-a-p']],
[attach_noun,[boy,'+n-v+a-p',sg]]],
    [attach_relative_clause,[xmax,
    [[attach_rpron,[that,'-n-v+a+p']],
    [attach_sent,[xmax,[[attach_embedded_subject,[xmax,
        [attach_noun,[boys,'+n-v+a-p',pl]],'+n-v+a-p']],
        [attach_vp,[xmax,[[aux_sai,[should,'-n+v+a+p']],
        [attach_vp,[xmax,[[attach_verb,[do,'-n+v',-,tense]],
         [attach_object,[xmax,
        [attach_noun,[it,'+n-v+a-p',sg]],
        '+n-v+a-p']]],'-n+v-a+p',subjless]]],'-n+v+a+p']]],
        '-n+v+a+p']]],'-n-v+a+p']]],'+n-v+a-p']]],
        '-n+v-a+p',subj]],],'-n+v+a+p'],[]],[]]

| ?- start(S).|: I wanted John to do it.

S    =    [[[xmax,[[attach_subject,[xmax,
    [attach_propnoun,['I','+n-v+a-p',pn]],'+n-v+a-p']],
    [attach_vp,[xmax,[[attach_verb,[want,'-n+v',+,ed]],
    [attach_zcomp_sent,[xmax,[[attach_embedded_subject,[xmax,
      [attach_propnoun,['John','+n-v+a-p',pn]],'+n-v+a-p']],
      [attach_infl,[xmax,[[to_infinitive,[to,'-n+v+a+p']],
        [attach_vp,[xmax,[[attach_verb,[do,'-n+v',-,tense]],
        [attach_object,[xmax,
        [attach_noun,[it,'+n-v+a-p',sg]],'+n-v+a-p']]],
        '-n+v-a+p',subjless]]],'-n+v+a+p']]],'-n+v+a+p']]],
        '-n+v-a+p',subj_inf]]],'-n+v+a+p'],[]],[]]

| ?- start(S).|: I want to do it.

S    =    [[[xmax,[[attach_subject,[xmax,
    [attach_propnoun,['I','+n-v+a-p',pn]],'+n-v+a-p']],
      [attach_vp,[xmax,[[attach_verb,[want,'-n+v',-,tense]],
    [attach_zcomp_sent,[xmax,
    [[to_infinitive,[to,'-n+v+a+p']],
    [attach_vp,[xmax,[[attach_verb,[do,'-n+v',-,tense]],
        [attach_object,[xmax,
        [attach_noun,[it,'+n-v+a-p',sg]],'+n-v+a-p']]],
        '-n+v-a+p',subjless]]],'-n+v+a+p']]],
        '-n+v-a+p',subj_inf]]], '-n+v+a+p'],[]],[]]

| ?- start(S).|: John bought a pear for Mary to eat.S =

[[[xmax,[[attach_subject,[xmax,
    [attach_propnoun,['John','+n-v+a-p',pn]],'+n-v+a-p']],
    [attach_vp,[xmax,[[attach_verb,[bought,'-n+v',-,tense]],
      [attach_zcomp_sent,[xmax,[[attach_embedded_subject,
        [xmax,[[[attach_det,[a,'+n-v-a-p']],
        [attach_noun,[pear,'+n-v+a-p',sg]]],
        [attach_pp,[xmax,[[attach_prep,[for,'-n-v']],
            [attach_pp_object,[xmax,
            [attach_propnoun,['Mary','+n-v+a-p',pn]],
            '+n-v+a-p']]],'-n-v']]],'+n-v+a-p']],
            [attach_infl,[xmax,
            [[to_infinitive,[to,'-n+v+a+p']],
          [attach_vp,[xmax,
          [attach_verb,[eat,'-n+v',-,tense]],'-n+v-a+p',
          obj]]],'-n+v+a+p']]],'-n+v+a+p']]],'-n+v-a+p',
          obj]]],'-n+v+a+p'],[]],[]]

| ?- start(S).|: Sally knows Bill kissed Jane.

S   =   [[[xmax,[[attach_subject,[xmax,
   [attach_noun,['Sally','+n-v+a-p',sg]],'+n-v+a-p']],
       [attach_vp,[xmax,[[attach_verb,[know,'-n+v',+,s]],
   [attach_zcomp_sent,[xmax,[[attach_embedded_subject,
   [xmax,
   [attach_noun,['Bill','+n-v+a-p',sg]],'+n-v+a-p']],
   [attach_vp,[xmax,[[attach_verb,[kiss,'-n+v',+,ed]],
   [attach_object,[xmax,
   [attach_noun,['Jane','+n-v+a-p',sg]],'+n-v+a-p']]],
       '-n+v-a+p',subj]]],'-n+v+a+p']]],'-n+v-a+p',subj]]],
          '-n+v+a+p'],[]],[]]
| ?- start(S).|: Sally has been forgotten.

S   =   [[[xmax,[[attach_subject,[xmax,
   [attach_noun,['Sally','+n-v+a-p',sg]],'+n-v+a-p']],
       [attach_vp,[xmax,[[perfective,[has,'-n+v']],
   [attach_succ_aux,[xmax,[[passive_be,[be,'-n+v',+,en]],
     [attach_vp,[xmax,[[passive,
   [attach_verb,[forgott,'-n+v-a+p',+,en]]],
   [attach_object,[xmax,[attach_propnoun,
   [np_empty,'+n-v+a-p',pn]],'+n-v+a-p']]],'-
   n+v+a+p',subj]]],
   '-n+v+a+p']]],'-n+v']]],'-n+v+a+p'],[]],[]]

| ?- start(S).|: I know John to be a fool.

S   =   [[[xmax,[[attach_subject,[xmax,
   [attach_propnoun,['I','+n-v+a-p',pn]],'+n-v+a-p']],
     [attach_vp,[xmax,[[attach_verb,[know,'-n+v',-,tense]],
   [attach_zcomp_sent,[xmax,[[attach_embedded_subject,
   [xmax,
   [attach_propnoun,['John','+n-v+a-p',pn]],'+n-v+a-p']],
     [attach_infl,[xmax,[[to_infinitive,[to,'-n+v+a+p']],
       [attach_vp,[xmax,[[attach_verb,[be,'-n+v',-,tense]],
       [attach_object,[xmax,[[attach_det,[a,'+n-v-a-p']],
            [attach_noun,[fool,'+n-v+a-p',sg]]],
            '+n-v+a-p']]],  '-n+v-a+p',prog]]],'-n+v+a+p']]],
            '-n+v+a+p']]]'-n+v-a+p',subj]]],
            '-n+v+a+p'],[]],[]]

| ?- start(S).|: who did Sally kiss.

S  =  [[[xmax,[[attach_wh_comp,[who,'-n-v+a+p']],
   [attach_sent,[xmax,[[attach_embedded_subject,
   [xmax,['Sally','+n-v+a-p',sg],'+n-v+a-p']],
   [attach_vp,[xmax,[[aux_sai,[did,'-n+v+a+p']],
   [attach_vp,[xmax,[[attach_verb,[kiss,'-n+v',-,tense]],
         [attach_object,[xmax,
       [attach_propnoun,[np_empty,'+n-v+a-p',pn]],
       '+n-v+a-p']]],'-n+v-a+p',subj]]],'-n+v+a+p']]],
            '-n+v+a+p']]],'-n-v+a+p',wh],[]],[]]
```

```
| ?- start(S).|: I know the boy that likes Sue.

S    =    [[[xmax,[[attach_subject,[xmax,
    [attach_propnoun,['I','+n-v+a-p',pn]],'+n-v+a-p']],
    [attach_vp,
    [xmax,[[attach_verb,[know,'-n+v',-,tense]],
    [attach_object,[xmax,[[[attach_det,[the,'+n-v-a-p']],
        [attach_noun,[boy,'+n-v+a-p',sg]]],
        [attach_relative_clause,[xmax,
        [[attach_rpron,[that,'-n-v+a+p']],
        [attach_vp,[xmax,[[attach_verb,[like,'-n+v',+,s]],
        [attach_object,[xmax,
        [attach_noun,['Sue','+n-v+a-p',sg]],
        '+n-v+a-p']]],'-n+v-a+p',subj]]],'-n-v+a+p']]],
        '+n-v+a-p']]],'-n+v-a+p',subj]]],'-n+v+a+p'],[]],[]]

| ?- start(S).|: I know the boy Sue likes.

S    =    [[[xmax,[[attach_subject,[xmax,
    [attach_propnoun,['I','+n-v+a-p',pn]],'+n-v+a-p']],
        [attach_vp,[xmax,
        [[attach_verb,[know,'-n+v',-,tense]],
        [attach_zcomp_sent,[xmax,[[attach_embedded_subject,
            [xmax,[[[attach_det,[the,'+n-v-a-p']],
            [attach_noun,[boy,'+n-v+a-p',sg]]],
            [attach_noun,['Sue','+n-v+a-p',sg]]],
            '+n-v+a-p']],
            [attach_vp,[xmax,[attach_verb,[like,'-n+v',+,s]],
            '-n+v-a+p',subj]]],'-n+v+a+p']]],
            '-n+v-a+p',subj]]], '-n+v+a+p'],[]],[]]

| ?- start(S).|: The boy's cat is dead.

S    =    [[[xmax,[[attach_subject,[xmax,
    [[[attach_det,['The','+n-v-a-p']],
    [attach_adj,['boy''s','+n-v+a+p']]],
    [attach_noun,[cat,'+n-v+a-p',sg]]],'+n-v+a-p']],
    [attach_vp,[xmax,[[copula,[is,'^^n+v']],
    [attach_pred_adj,[dead,'+n-v+a+p']]],'-n+v']]],
    '-n+v+a+p'],[]],[]]

| ?- start(S).|: block the road.

S    =    [[[xmax,[[attach_subject,[xmax,
    [attach_propnoun,[you,'+n-v+a-p',pn]],'+n-v+a-p']],
    [attach_vp,[xmax,
    [[attach_verb,[block,'-n+v-a+p',-,tense]],
    [attach_object,
    [xmax,[[attach_det,[the,'+n-v-a-p']],
    [attach_noun,[road,'+n-v+a-p',sg]]],'+n-v+a-p']]],
    '-n+v-a+p',obj]]],'-n+v+a+p'],[]],[]]
```

The examples below are results of parsing with LParser.

```
| ?- run(S,B).|: I wanted John to do it.

S
```

```
[sentence(_22366,s(np(p_n('I')),vp(v(wanted),np(p_n('John')),
              vp2(inf(to),v(do),np(pronoun(it)))))))],
B = []

| ?- run(S,B).|: I want to do it.

S  =  [sentence(_7505,s(np(p_n('I')),vp(v(want),
     vp2(inf(to),v(do),np(pronoun(it))))))],
B = []

| ?- run(S,B).|: Sally knows Bill kissed Jane.

S                                                                   =
[sentence(_21075,s(np(p_n('Sally')),vp(v(knows),z_c_ph(e_s(
np(p_n('Bill')),         vp(v(kissed),np(p_n('Jane')))))))))],
B = []

| ?- run(S,B).|: Sally has been forgotten.

S   =   [sentence(_12342,s(np(p_n('Sally')),
     aux(perf(has),prog(been)),vp(v(forgotten)))))],

B = []

| ?- run(S,B).|: who did Sally kiss.

S = [sentence(_12263,s(wh_ph(w_c(who),aux1(do(did))),
 e_s(np(p_n('Sally')),vp(v(kiss)))))],
B = []

| ?- run(S,B).|: I know the boy that likes Sally.

S                                                                   =
[sentence(_26279,s(np(p_n('I')),vp(v(know),np(np(d(the),n(b
o        y        ,        s        g        )        )        ,
     r_c(r_p(that),vp(v(likes),np(p_n('Sally')))))))))],

B = []

| ?- run(S,B).|: I know the boy that Sally likes.

S                                                                   =
[sentence(_27345,s(np(p_n('I')),vp(v(know),np(np(d(the),n(b
o        y        ,        s        g        )        )        ,
     r_c(r_p(that),e_s(np(p_n('Sally')),vp(v(likes))))))))))],

B = []

| ?- run(S,B).|: the jar seems broken.

S  =  [sentence(_10717,s(np(d(the),n(jar,sg)),
     cp(aux(seem(seems)),a(broken)))))],

B = []

| ?- run(S,B).|: there seems to have been a meeting scheduled
for Friday.
```

```
S = [sentence(_36425,s(there(there(there),aux(seem(seems))),
            infaux(inf(to),aux(perf(have),prog(been)),
    e_s(np(d(a),n(meeting,sg)),vp(v(scheduled),
    pp(p(for),np(n('Friday',sg))))))))],

B = []

| ?- run(S,B).|: schedule a meeting for Friday.

S     =     [sentence(_14885,s(ivp(v(schedule),

np(np(d(a),n(meeting,sg)),pp(p(for),np(n('Friday',sg)))))))
],B = []
```

APPENDIX D

This appendix contains the grammars for MParserSub and LParserSub, example noun-phrases and example parses from both parsers. The grammar below is the grammar used by LParserSub.

```
gramnp:-assert((noun_phrase_-->noun_phrase(N,_NP))),
assert((noun_phrase(N,np(Noun))-->noun(N,Noun))),
assert((noun_phrase(N,np(Noun,Noun))-->noun(N,Noun),noun(N,Noun))),
assert((noun_phrase(N,np(Pronoun))-->pronoun(N,Pronoun))),
assert((noun_phrase(N,np(Proper_noun))->proper_noun(N,Proper_noun))),
assert((noun_phrase(N,np(Det,Noun))-->det(N,Det),noun(N,Noun))),
assert((noun_phrase(N,np(Ref,Noun))-->ref(N,Ref),noun(N,Noun))),
assert((noun_phrase(N,np(Noun,Ref,Noun,Num))-->noun(N,Noun),
ref(N,Ref),noun(N,Noun),num(N,Num))),
assert((noun_phrase(N,np(Noun,Noun,Noun,Conj,Noun))-->noun(N,Noun),noun(N,Noun),no
un(N,Noun),conj(N,Conj),noun(N,Noun))),
assert((noun_phrase(N,np(Noun,Noun,Conj,Noun))-->noun(N,Noun),noun(N,Noun),conj(N,C
onj),noun(N,Noun))),
assert((noun_phrase(N,np(Det,Adj,Conj,Noun,Noun))-->det(N,Det),adj(N,Adj),conj(N,Conj),no
un(N,Noun),noun(N,Noun))),
assert((noun_phrase(N,np(Det,Adj,Conj,Noun))-->det(N,Det),adj(N,Adj),conj(N,Conj),noun(N,
Noun))),
assert((noun_phrase(N,np(Det,Noun,Adj))-->det(N,Det),noun(N,Noun),adj(N,Adj))),
assert((noun_phrase(N,np(Noun,Num,Noun,Num,Noun))-->noun(N,Noun),num(N,Num),nou
n(N,Noun),num(N,Num),noun(N,Noun))),
assert((noun_phrase(N,np(Ref,Noun,Conj,Noun,Noun))-->ref(N,Ref),noun(N,Noun),conj(N,Co
nj),noun(N,Noun),noun(N,Noun))),
assert((noun_phrase(N,np(NP,Adj,Conj,Adj))-->noun_phrase(N,NP),adj(N,Adj),conj(N,Conj),a
dj(N,Adj))),
assert((noun_phrase(N,np(Adj,Conj,Noun,Noun))-->adj(N,Adj),conj(N,Conj),noun(N,Noun),no
un(N,Noun))),
assert((noun_phrase(N,np(Adj,Conj,Noun))-->adj(N,Adj),conj(N,Conj),noun(N,Noun))),
assert((noun_phrase(N,np(Adj,Noun))-->adj(N,Adj),noun(N,Noun))),
assert((noun_phrase(N,np(Adj,Adj,Noun))-->adj(N,Adj),adj(N,Adj),noun(N,Noun))),
assert((noun_phrase(N,np(Det,Adj,Adj,Conj,Noun))-->det(N,Det),adj(N,Adj),adj(N,Adj),conj(N
,Conj),noun(N,Noun))),
assert((noun_phrase(N,np(Ref,Noun,Noun))-->ref(N,Ref),noun(N,Noun),noun(N,Noun))),
assert((noun_phrase(N,np(Ref,Noun,Num))-->ref(N,Ref),noun(N,Noun),num(N,Num))),
assert((noun_phrase(N,np(Det,Noun,Num))-->det(N,Det),noun(N,Noun),num(N,Num))),
assert((noun_phrase(N,np(Det,Noun,Num))-->det(N,Det),noun(N,Noun),num(N,Num),noun(
N,Noun))),
assert((noun_phrase(N,np(Det,Num,Noun))-->det(N,Det),num(N,Num),noun(N,Noun))),
assert((noun_phrase(N,np(Adj,Det,Noun))-->adj(N,Adj),
det(N,Det),noun(N,Noun))),
assert((noun_phrase(N,np(Det,Num,Noun,Num))-->det(N,Det),num(N,Num),noun(N,Noun),n
um(N,Num))),
assert((noun_phrase(N,np(Det,Noun,Num,Conj,Num))-->det(N,Det),noun(N,Noun),num(N,N
um),conj(N,Conj),num(N,Num))),
assert((noun_phrase(N,np(Noun,Num,Conj,Num))-->noun(N,Noun),num(N,Num),conj(N,Con
j),num(N,Num))),
assert((noun_phrase(N,np(Noun,Num))-->noun(N,Noun),num(N,Num))),
assert((noun_phrase(N,np(Num,Noun))-->num(N,Num),noun(N,Noun))),
assert((noun_phrase(N,np(Det,Noun,Conj,Noun))-->det(N,Det),noun(N,Noun),conj(N,Conj),no
un(N,Noun))),
assert((noun_phrase(N,np(Noun,Conj,Noun))-->noun(N,Noun),conj(N,Conj),noun(N,Noun))),
assert((noun_phrase(N,np(Det,Noun,Noun,Noun,Conj,Noun))-->det(N,Det),noun(N,Noun),
noun(N,Noun),noun(N,Noun),conj(N,Conj),noun(N,Noun))),
assert((noun_phrase(N,np(Noun,Noun,Adj,Adj))-->noun(N,Noun),noun(N,Noun),adj(N,Adj),a
dj(N,Adj))),
assert((noun_phrase(N,np(Noun,Noun,Noun))-->noun(N,Noun),noun(N,Noun),noun(N,Noun))),
assert((noun_phrase(N,np(NP,Conj,NP))-->noun_phrase(N,NP),conj(N,Conj),noun_phrase(N,
NP))),
assert((noun_phrase(N,np(NP,NP,Conj,NP))-->noun_phrase(N,NP),noun_phrase(N,NP),conj(N
,Conj),noun_phrase(N,NP))),
assert((noun_phrase(N,np(NP,NP,Conj,NP))-->noun_phrase(N,NP),noun_phrase(N,NP),noun_
phrase(N,NP),conj(N,Conj),noun_phrase(N,NP))),
assert((noun_phrase(N,np(NP,PP))-->noun_phrase(N,NP),
prep_phrase(N,PP))),
assert((noun_phrase(N,np(NP,RC))-->noun_phrase(N,NP),
r_clause(N,RC))),
assert((prep_phrase(N,pp(Prep,NP))-->prep(N,Prep),noun_phrase(N,NP))),
assert((prep_phrase(N,pp(Prep,Prep,NP))-->prep(N,Prep),
prep(N,Prep),noun_phrase(N,NP))),
assert((r_clause(N,r_c(Rpron,NP))-->rpron(N,Rpron),verb_phrase(N,NP))),
assert((r_clause(N,r_c(Rpron,Aux,NP))-->rpron(N,Rpron),aux(N,Aux),noun_phrase(N,NP))),
assert((r_clause(N,r_c(Rpron,Aux,VP))-->rpron(N,Rpron),aux(N,Aux),verb_phrase(N,NP))),
assert((verb_phrase(N,vp(V,NP))-->verb(N,V),noun_phrase(N,NP))),
assert((verb_phrase(N,vp(V,NP))-->verb(N,V),
prep_phrase(N,NP))),
assert((verb_phrase(N,vp(V))-->verb(N,V))),
assert((verb_phrase(N,vp(V,Conj,V))-->verb(N,V),conj(N,Conj),verb(N,V))),
assert((verb_phrase(N,vp(V,NP,PP))-->verb(N,V),noun_phrase(N,NP),
prep_phrase(N,PP))),
assert((verb_phrase(N,vp(V,NP,SC))-->verb(N,V),
prep_phrase(N,NP),
sub_clause(N,SC))),
assert((sub_clause(N,s_c(Sub,NP,VP,Sub,Adv,VP))-->sub(N,Sub),noun_phrase(N,NP),verb_phr
ase(N,VP),sub(N,Sub),adv(N,Adv),verb_phrase(N,VP))),
assert((aux(N,aux(Perf))-->perf(N,Perf))),
assert((aux(N,aux(Prog))-->prog(N,Prog))).
```

The grammar below is the grammar used by MParserSub.

```
grammar_rule([], [S|Reststack],
[[[',',],[SWord, '+n-v+a-p',SP]]|Rest],NS, NB):-

not_attach([S|Reststack],
[[[',',],[SWord, '+n-v+a-p',SP]]|Rest],NS,NB).

grammar_rule([], [S|Reststack],
[[[',',],[Ran,[SWord, '+n-v+a-p',SP]]]],NS, NB):-
not_attach([S|Reststack],
[[[',',],[Ran,[SWord, '+n-v+a-p',SP]]]],NS,NB).

grammar_rule([], [S|Reststack],[[[',',],[Dom,[SWord, '+n-v+a+p']]]|Rest],NS, NB):-
not_attach([S|Reststack],[[[',',],[Dom,[SWord, '+n-v+a+p']]]|Rest],NS,NB).

grammar_rule(attach_adj, [[xmax, '+n-v+a-p', [spec_, head, comp]]|Reststack],
[[[FWord, '+n-v+a+p'],[SWord, '+n-v+a+p']]|Rest],NS, NB):-
attach([[xmax,'+n-v+a-p', [spec_, head, comp]]|Reststack],
[[[FWord, '+n-v+a+p'],[SWord, '+n-v+a+p']]|Rest],NS,NB).
```

```
grammar_rule(attach_adj, [[xmax, '+n-v+a-p', [spec_, head, comp]] | Reststack],
[[[Dom,[FWord, '+n-v+a+p']],[',']] | Rest],NS, NB):-
attach([[xmax,'+n-v+a-p', [spec_, head, comp]] | Reststack],
 [[[Dom,[FWord, '+n-v+a+p']],[',']] | Rest],NS,NB).

grammar_rule(attach_adj, [[xmax, St,'+n-v+a-p', [spec_, head, comp]] | Reststack],
[[[Dom,[FWord, '+n-v+a+p']],[',']] | Rest],NS, NB):-
attach([[xmax, St,'+n-v+a-p', [spec_, head, comp]] | Reststack],
 [[[Dom,[FWord, '+n-v+a+p']],[',']] | Rest],NS,NB).

grammar_rule(attach_adj, [[xmax,St, '+n-v+a-p',[spec_, head, comp]] | Reststack],

[[[FWord,'+n-v+a+p'],[SWord,'+n-v+a+p']] | Rest],NS, NB):-attach([[xmax,St,'+n-v+a-p', [spec_,
head, comp]] | Reststack],
[[[FWord, '+n-v+a+p'],[SWord, '+n-v+a+p']] | Rest],NS,NB).

grammar_rule(attach_adj, [[xmax,'+n-v+a-p', [spec_, head, comp]] | Reststack],
[[[FWord, '+n-v+a+p'],[SWord, '+n-v+a-p']] | Rest],NS, NB):-
attach([[xmax,'+n-v+a-p', [spec_, head, comp]] | Reststack],
[[[FWord, '+n-v+a+p'],[SWord, '+n-v+a-p',conj]] | Rest],NS,NB).

grammar_rule(attach_adj, [[xmax,St,'+n-v+a-p', [spec_, head, comp]] | Reststack],
[[[FWord, '+n-v+a+p'],[SWord, '+n-v+a-p',conj]] | Rest],NS, NB):-
attach([[xmax,St,'+n-v+a-p', [spec_, head, comp]] | Reststack],
[[[FWord, '+n-v+a+p'],[SWord, '+n-v+a-p',conj]] | Rest],NS,NB).

grammar_rule(attach_adj, [[xmax,St,'+n-v+a-p', [spec_, head, comp]] | Reststack],
[[[D,[FWord, '+n-v+a+p']],[SWord, '+n-v+a-p',conj]] | Rest],NS, NB):-
attach([[xmax,St,'+n-v+a-p', [spec_, head, comp]] | Reststack],
[[[D,[FWord, '+n-v+a+p']],[SWord, '+n-v+a-p',conj]] | Rest],NS,NB).

grammar_rule(attach_adj, [[xmax,St, '+n-v+a-p',[spec, head, comp_]] | Reststack],
[[[FWord, '+n-v+a+p'],Second] | Rest],NS, NB):-
attach([[xmax,St, '+n-v+a-p', [spec, head, comp_]] | Reststack],

[[[FWord, '+n-v+a+p'],Second] | Rest],NS,NB).

grammar_rule(attach_det, [[xmax,'+n-v+a-p', [spec_, head, comp]] | Reststack],
[[[FWord, '+n-v-a-p'],[SWord, '+n-v+a-p',SP]] | Rest],NS, NB):-
attach([[xmax, '+n-v+a-p',[spec_,head,comp]] | Reststack],
[[[FWord, '+n-v-a-p'],[SWord, '+n-v+a-p',SP]] | Rest],NS,NB).

grammar_rule(attach_det, [[xmax,'+n-v+a-p', [spec_, head, comp]] | Reststack],
[[[FWord, '+n-v-a-p'],[SWord, '+n-v-a-p']] | Rest],NS, NB):-
attach([[xmax, '+n-v+a-p',[spec_,head,comp]] | Reststack],
[[[FWord, '+n-v-a-p'],[SWord, '+n-v-a-p']] | Rest],NS,NB).

grammar_rule(attach_det, [[xmax,St,'+n-v+a-p', [spec_, head, comp]] | Reststack],
[[[FWord, '+n-v-a-p'],[SWord, '+n-v+a-p',SP]] | Rest],NS, NB):-
attach([[xmax, St,'+n-v+a-p',[spec_,head,comp]] | Reststack],
[[[FWord, '+n-v-a-p'],[SWord, '+n-v+a-p',SP]] | Rest],NS,NB).

grammar_rule(attach_det, [[xmax,St,'+n-v+a-p', [spec_, head, comp]] | Reststack],
[[[FWord, '+n-v-a-p'],[SWord, '+n-v-a-p']] | Rest],NS, NB):-
attach([[xmax, St,'+n-v+a-p',[spec_,head,comp]] | Reststack],
[[[FWord, '+n-v-a-p'],[SWord, '+n-v-a-p']] | Rest],NS,NB).

grammar_rule(attach_det,[[xmax,  '+n-v+a-p',  [spec_, head, comp]] | Reststack],
[[[FWord,'+n-v-a-p'],[D,[SWord,'+n-v+a+p']]] | Rest],NS, NB):-
attach([[xmax,'+n-v+a-p',[spec_, head, comp]] | Reststack],
[[[FWord,'+n-v-a-p'],[D,[SWord,'+n-v+a+p']]] | Rest],NS,NB).

grammar_rule(attach_det,[[xmax,  '+n-v+a-p',  [spec_, head, comp]] | Reststack],
[[[FWord,'+n-v-a-p'],[SWord,'+n-v+a+p']] | Rest],NS, NB):-
attach([[xmax,'+n-v+a-p',[spec_, head, comp]] | Reststack],
[[[FWord,'+n-v-a-p'],[SWord,'+n-v+a+p']] | Rest],NS,NB).

grammar_rule(attach_np,[[xmax,'+n-v+a-p',[spec, head_, comp]]],
[[[xmax,E,'+n-v+a-p'],Second] | Rest],NS,NB):-
attach([[xmax,'+n-v+a-p',[spec, head_, comp]]],
[[[xmax,E,'+n-v+a-p'],Second] | Rest],NS,NB).

grammar_rule(attach_np,[[xmax,St,'+n-v+a-p',[spec, head_, comp]]],
[[[xmax,E,'+n-v+a-p'],Second] | Rest],NS,NB):-
attach([[xmax,St,'+n-v+a-p',[spec, head_, comp]]],
[[[xmax,E,'+n-v+a-p'],Second] | Rest],NS,NB).

grammar_rule(attach_np,[[xmax,'+n-v+a-p',[spec, head_, comp]]],
[[[xmax,E,'+n-v+a-p'],Second] | Rest],NS,NB):-
attach([[xmax,'+n-v+a-p',[spec, head_, comp]]],
[[[xmax,E,'+n-v+a-p'],Second] | Rest],NS,NB).

grammar_rule(attach_comp,[[xmax, '-n-v+a+p', [spec, head_, comp]] | Reststack],
[[[when, '-n-v+a+p'],Second] | Rest],NS, NB):-
attach([[xmax,'-n-v+a+p',[spec, head_, comp]] | Reststack],
[[[when,'-n-v+a+p'],Second] | Rest],NS,NB).

grammar_rule(attach_comp,[[xmax, '-n-v+a+p', [spec, head_, comp]] | Reststack],
[[[unless, '-n-v+a+p'],Second] | Rest],NS, NB):-
attach([[xmax,'-n-v+a+p',[spec, head_, comp]] | Reststack],
[[[unless,'-n-v+a+p'],Second] | Rest],NS,NB).

grammar_rule(attach_rpron,[[xmax, '-n-v+a+p', [spec, head_, comp]],
[xmax,S,'+n-v+a-p',T] | Reststack],
[[[FWord, '-n-v+a+p'],Second] | Rest],NS, NB):-
attach([[xmax,'-n-v+a+p',[spec, head_, comp]],
[xmax,S,'+n-v+a-p',T] | Reststack],
[[[FWord,'-n-v+a+p'],Second] | Rest],NS,NB).

grammar_rule(attach_rpron,[[xmax, '-n-v+a+p', [spec, head_, comp]],
[xmax,S,'+n-v+a-p',T] | Reststack],
[[[[FWord, '-n-v+a+p'],[FWord, '+n-v-a-p']],Second] | Rest],NS, NB):-
attach([[xmax,'-n-v+a+p',[spec, head_, comp]],
[xmax,S,'+n-v+a-p',T] | Reststack],
[[[[FWord,'-n-v+a+p'],[FWord,'+n-v-a-p']],Second] | Rest],NS,NB).

grammar_rule(attach_propnoun,[[xmax,'+n-v+a-p',[spec, head_, comp]] | Reststack],
[[[FWord, '+n-v+a-p',pn], Second] | Rest],NS, NB):-
attach([[xmax, '+n-v+a-p',  [spec, head_, comp]] | Reststack],
[[[FWord, '+n-v+a-p',pn],Second] | Rest],NS, NB).

grammar_rule(attach_propnoun,[[xmax,St,'+n-v+a-p',[spec,head_,comp]] | Reststack],
[[[FWord, '+n-v+a-p',pn], Second] | Rest],NS, NB):-
attach([[xmax, St,'+n-v+a-p',  [spec, head_, comp]] | Reststack],
[[[FWord, '+n-v+a-p',pn],Second] | Rest],NS, NB).
```

```
grammar_rule(attach_pron, [[xmax,St,'+n-v+a-p',[spec, head_, comp]] | Reststack],
[[[FWord, '+n-v+a-p',pron],Second] | Rest],NS, NB):-
 attach([[xmax, St, '+n-v+a-p',[spec, head_, comp]] | Reststack],
[[[FWord, '+n-v+a-p',pron],Second] | Rest],NS, NB).

grammar_rule(attach_pron, [[xmax,'+n-v+a-p',[spec, head_, comp]] | Reststack],
[[[FWord, '+n-v+a-p',pron],Second] | Rest],NS, NB):-
 attach([[xmax, '+n-v+a-p',[spec, head_, comp]] | Reststack],
[[[FWord, '+n-v+a-p',pron],Second] | Rest],NS, NB).

grammar_rule(attach_noun, [[xmax,St,'+n-v+a-p',[spec, head_, comp]] | Reststack],
[[[FWord, '+n-v+a-p',sg],Second] | Rest],NS, NB):-
 attach([[xmax, St, '+n-v+a-p',[spec, head_, comp]] | Reststack],
[[[FWord, '+n-v+a-p',sg],Second] | Rest],NS, NB).

grammar_rule(attach_noun, [[xmax,St,'+n-v+a-p',[spec, head_, comp]] | Reststack],
[[[Range,FWord, '+n-v+a-p',sg],Second] | Rest],NS, NB):-
 attach([[xmax, St, '+n-v+a-p',[spec, head_, comp]] | Reststack],
[[[Range,FWord, '+n-v+a-p',sg],Second] | Rest],NS, NB).

grammar_rule(attach_noun, [[xmax,St,'+n-v+a-p',[spec, head_, comp]] | Reststack],
[[[Dom,[FWord, '+n-v+a-p',sg]],Second] | Rest],NS, NB):-
 attach([[xmax, St, '+n-v+a-p',[spec, head_, comp]] | Reststack],
[[[Dom,[FWord, '+n-v+a-p',sg]],Second] | Rest],NS, NB).

grammar_rule(attach_noun, [[xmax,'+n-v+a-p',[spec_, head, comp]] | Reststack],
[[[[Dom,F,R,S],'+n-v+a-p',sg],[SWord,'+n-v+a+p',ref]] | Rest],NS, NB):-
 attach([[xmax,'+n-v+a-p',[spec_, head, comp]] | Reststack],
[[[[Dom,F,R,S],'+n-v+a-p',sg],[SWord,'+n-v+a+p',ref]] | Rest],NS, NB).

grammar_rule(attach_noun, [[xmax,'+n-v+a-p',[spec, head_, comp]] | Reststack],
[[[FWord, '+n-v+a-p',sg],Second] | Rest],NS, NB):-
 attach([[xmax,'+n-v+a-p',[spec, head_, comp]] | Reststack],
[[[FWord, '+n-v+a-p',sg],Second] | Rest],NS, NB).

grammar_rule(attach_noun, [[xmax,St,'+n-v+a-p',[spec, head_, comp]] | Reststack],
[[[FWord, '+n-v+a-p',pl],Second] | Rest],NS, NB):-
 attach([[xmax, St, '+n-v+a-p',[spec, head_, comp]] | Reststack],
[[[FWord, '+n-v+a-p',pl],Second] | Rest],NS, NB).

grammar_rule(attach_noun, [[xmax,St,'+n-v+a-p',[spec, head_, comp]] | Reststack],
[[[Range,FWord, '+n-v+a-p',pl],Second] | Rest],NS, NB):-
 attach([[xmax, St, '+n-v+a-p',[spec, head_, comp]] | Reststack],
[[[Range,FWord, '+n-v+a-p',pl],Second] | Rest],NS, NB).

grammar_rule(attach_noun, [[xmax,'+n-v+a-p',[spec, head_, comp]] | Reststack],
[[[FWord, '+n-v+a-p',pl],Second] | Rest],NS, NB):-
 attach([[xmax,'+n-v+a-p',[spec, head_, comp]] | Reststack],
[[[FWord, '+n-v+a-p',pl],Second] | Rest],NS, NB).

grammar_rule(attach_num, [[xmax,St,'+n-v+a-p',[spec, head, comp_]] | Reststack],
[[[FWord, '+n-v+a-p',num],Second] | Rest],NS, NB):-
 attach([[xmax, St, '+n-v+a-p',[spec, head, comp_]] | Reststack],
[[[FWord, '+n-v+a-p',num],Second] | Rest],NS, NB).

grammar_rule(attach_num, [[xmax,'+n-v+a-p',[spec, head_, comp]] | Reststack],
[[[FWord, '+n-v+a-p',num],Second] | Rest],NS, NB):-
 attach([[xmax,'+n-v+a-p',[spec, head_, comp]] | Reststack],
[[[FWord, '+n-v+a-p',num],Second] | Rest],NS, NB).

grammar_rule(attach_num, [[xmax,St,'+n-v+a-p',[spec_, head, comp]] | Reststack],
[[[FWord, '+n-v+a-p',num],Second] | Rest],NS, NB):-
 attach([[xmax,St,'+n-v+a-p',[spec_, head, comp]] | Reststack],
[[[FWord, '+n-v+a-p',num],Second] | Rest],NS, NB).

grammar_rule(attach_ref, [[xmax,St,'+n-v+a-p',[spec, head_, comp]] | Reststack],
[[[FWord, '+n-v+a+p',ref],Second] | Rest],NS, NB):-
 attach([[xmax, St, '+n-v+a-p',[spec, head_, comp]] | Reststack],
[[[FWord, '+n-v+a+p',ref],Second] | Rest],NS, NB).

grammar_rule(attach_ref, [[xmax,'+n-v+a-p',[spec_, head, comp]] | Reststack],
[[[FWord, '+n-v+a+p',ref],Second] | Rest],NS, NB):-
 attach([[xmax,'+n-v+a-p',[spec_, head, comp]] | Reststack],
[[[FWord, '+n-v+a+p',ref],Second] | Rest],NS, NB).

grammar_rule(attach_conj, [[xmax,St,'+n-v+a-p',[spec_, head, comp]] | Reststack],  [[[FWord,
'+n-v+a-p',conj],[SWord,'+n-v+a-p',SP]] | Rest],NS, NB):-  attach([[xmax, St, '+n-v+a-p',[spec_,
head, comp]] | Reststack],  [[[FWord, '+n-v+a-p',conj],[SWord,'+n-v+a-p',SP]] | Rest],NS, NB).

grammar_rule(attach_conj, [[xmax,St,'+n-v+a-p',[spec, head_, comp]] | Reststack],  [[[FWord,
'+n-v+a-p',conj],[SWord,'+n-v+a-p',SP]] | Rest],NS, NB):-  attach([[xmax, St, '+n-v+a-p',[spec,
head_, comp]] | Reststack],  [[[FWord, '+n-v+a-p',conj],[SWord,'+n-v+a-p',SP]] | Rest],NS, NB).

grammar_rule(attach_conj, [[xmax,St,'+n-v+a-p',[spec, head_, comp]] | Reststack],  [[[FWord,
'+n-v+a-p',conj],[Range,SWord,'+n-v+a-p',SP]] | Rest],NS,  NB):-   attach([[xmax,  St,
'+n-v+a-p',[spec, head_, comp]] | Reststack],  [[[FWord,
'+n-v+a-p',conj],[Range,SWord,'+n-v+a-p',SP]] | Rest],NS, NB).

grammar_rule(attach_conj, [[xmax,St,'+n-v+a-p',[spec, head_, comp]] | Reststack],  [[[FWord,
'+n-v+a-p',conj],[SWord,'+n-v+a+p']] | Rest],NS, NB):-  attach([[xmax, St, '+n-v+a-p',[spec, head_,
comp]] | Reststack],  [[[FWord, '+n-v+a-p',conj],[SWord,'+n-v+a+p']] | Rest],NS, NB).

grammar_rule(attach_conj, [[xmax,St,'+n-v+a-p',[spec, head_, comp]] | Reststack],  [[[FWord,
'+n-v+a-p',conj],[SWord,'+n-v-a-p']] | Rest],NS, NB):-  attach([[xmax, St, '+n-v+a-p',[spec, head_,
comp]] | Reststack],  [[[FWord, '+n-v+a-p',conj],[SWord,'+n-v-a-p']] | Rest],NS, NB).

grammar_rule(attach_conj, [[xmax,St,'-n+v-a+p',[spec, head_, comp],Ty] | Reststack],  [[[FWord,
'+n-v+a-p',conj],[SWord,'-n+v-a+p',PN,T]] | Rest],NS, NB):-  attach([[xmax, St, '-n+v-a+p',[spec,
head_, comp],Ty] | Reststack],  [[[FWord, '+n-v+a-p',conj],[SWord,'-n+v-a+p',PN,T]] | Rest],NS,
NB).

grammar_rule(attach_prep, [[xmax, '-n-v', [spec, head_, comp]] | Reststack],
[[[FWord, '-n-v'],[W, '-n-v']] | Rest],NS, NB):-
attach([[xmax,'-n-v',[spec, head_, comp]] | Reststack],
[[[FWord,'-n-v'],[W, '-n-v']] | Rest],NS, NB).

grammar_rule(attach_prep, [[xmax, '-n-v', [spec, head_, comp]] | Reststack],
[[[FWord, '-n-v'],[W, '+n-v+a-p',SP]] | Rest],NS, NB):-
attach([[xmax,'-n-v',[spec, head_, comp]] | Reststack],
[[[FWord,'-n-v'],[W, '+n-v+a-p',SP]] | Rest],NS, NB).

grammar_rule(attach_prep, [[xmax,S, '-n-v',[spec, head_, comp]] | Reststack],
[[[FWord, '-n-v'], [W,'+n-v+a-p',SP]] | Rest],NS, NB):-
attach([[xmax,S,'-n-v',[spec, head_,comp]] | Reststack],
[[[FWord, '-n-v'],[W, '+n-v+a-p',SP]] | Rest],NS,NB).
```

```
grammar_rule(attach_prep, [[xmax, '-n-v', [spec, head_, comp]] | Reststack],
[[[FWord, '-n-v'],[W, '+n-v+a-p',pn]] | Rest],NS, NB):-
attach([[xmax,'-n-v',[spec, head_, comp]] | Reststack],
   [[[FWord,'-n-v'],[W, '+n-v+a-p',pn]] | Rest],NS, NB).

grammar_rule(attach_prep, [[xmax,S, '-n-v',[spec, head_, comp]] | Reststack],          [[[FWord,
'-n-v'], [W, '+n-v+a-p',pn]] | Rest],NS, NB):-
attach([[xmax,S,'-n-v',[spec, head_,comp]] | Reststack],
   [[[FWord, '-n-v'],[W, '+n-v+a-p',pn]] | Rest],NS,NB).

grammar_rule(attach_prep, [[xmax, '-n-v', [spec, head_, comp]] | Reststack],
[[[FWord, '-n-v'], [W, '+n-v-a-p']] | Rest],NS, NB):-
attach([[xmax,'-n-v',[spec, head_, comp]] | Reststack],
[[[FWord,'-n-v'],[W, '+n-v-a-p']] | Rest],NS, NB).

grammar_rule(attach_prep, [[xmax,S, '-n-v',[spec, head_, comp]] | Reststack],
[[[FWord, '-n-v'], [W, '+n-v-a-p']] | Rest],NS, NB):-
attach([[xmax,S,'-n-v',[spec, head_,comp]] | Reststack],
[[[FWord, '-n-v'],[W, '+n-v-a-p']] | Rest],NS,NB).

grammar_rule(attach_prep, [[xmax, '-n-v', [spec, head_, comp]] | Reststack],
[[[FWord, '-n-v'], [W, '+n-v+a+p',PD]] | Rest],NS, NB):-
attach([[xmax,'-n-v',[spec, head_, comp]] | Reststack],
[[[FWord,'-n-v'],[W,'+n-v+a+p',PD]] | Rest],NS, NB).

grammar_rule(attach_prep, [[xmax,S, '-n-v',[spec, head_, comp]] | Reststack],
[[[FWord, '-n-v'], [W, '+n-v+a+p',PD]] | Rest],NS, NB):-
attach([[xmax,S,'-n-v',[spec, head_,comp]] | Reststack],
[[[FWord,'-n-v'],[W, '+n-v+a+p',PD]] | Rest],NS,NB).

grammar_rule(attach_prep, [[xmax, '-n-v', [spec, head_, comp]] | Reststack],
[[[FWord, '-n-v'], [W, '+n-v+a+p']] | Rest],NS, NB):-
attach([[xmax,'-n-v',[spec, head_, comp]] | Reststack],
[[[FWord,'-n-v'],[W,'+n-v+a+p']] | Rest],NS, NB).

grammar_rule(attach_prep, [[xmax,S, '-n-v',[spec, head_, comp]] | Reststack],
[[[FWord, '-n-v'], [W, '+n-v+a+p']] | Rest],NS, NB):-
attach([[xmax,S,'-n-v',[spec, head_,comp]] | Reststack],
[[[FWord,'-n-v'],[W, '+n-v+a+p']] | Rest],NS,NB).

grammar_rule(attach_vp,[[xmax,S,'-n-v+a+p',[spec,head,comp_]] | Reststack],
[[[xmax,S1,'-n+v-a+p'], Second] | Rest],NS, NB):-
attach([[xmax,S,'-n-v+a+p',[spec, head, comp_]] | Reststack],
[[[xmax, S1,'-n+v-a+p'], Second] | Rest],NS, NB).

grammar_rule(attach_adverb, [[xmax, '-n+v', [spec_,head, comp]] | Reststack],
[[[FWord, '-n+v'],[SWord, '-n+v',PN,Tense]] | Rest],NS, NB):-
attach([[xmax, '-n+v', [spec_, head, comp]] | Reststack],
[[[FWord, '-n+v'],[SWord, '-n+v', PN, Tense]] | Rest],NS,NB).

grammar_rule(attach_verb, [[xmax, '-n+v-a+p', [spec, head_, comp]] | Reststack],
[[[FWord, '-n+v', PN, Tense],Second] | Rest],NS, NB):-
attach([[xmax, '-n+v-a+p', [spec, head_, comp]] | Reststack],
[[[FWord, '-n+v', PN, Tense],Second] | Rest],NS,NB).

grammar_rule(attach_verb, [[xmax,St,'-n+v-a+p',[spec, head_, comp],Ty] | Reststack],
[[[FWord, '-n+v-a+p', PN, Tense],Second] | Rest],NS, NB):-
attach([[xmax, St, '-n+v-a+p', [spec, head_, comp],Ty] | Reststack],
[[[FWord, '-n+v-a+p', PN, Tense],Second] | Rest],NS,NB).

grammar_rule(attach_verb, [[xmax,'-n+v', [spec, head_, comp]] | Reststack],
[[[FWord, '-n+v', PN, Tense],Second] | Rest], NS, NB):-
attach([[xmax,'-n+v',[spec, head_, comp]] | Reststack],
[[[FWord, '-n+v',PN,Tense],Second] | Rest],NS, NB).

grammar_rule(attach_verb, [[xmax,St,'-n+v-a+p',[spec, head_, comp]] | Reststack],
[[[FWord, '-n+v', PN, Tense],Second] | Rest], NS, NB):-
attach([[xmax,St,'-n+v-a+p',[spec, head_, comp]] | Reststack],
[[[FWord, '-n+v',PN,Tense],Second] | Rest],NS, NB).

grammar_rule(attach_verb,[[xmax,'-n+v-a+p',[spec, head_, comp]] | Reststack],
[[[FWord, '-n+v-a+p', PN, Tense],Second] | Rest],NS, NB):-
attach([[xmax,'-n+v-a+p',[spec, head_, comp]] | Reststack],
[[[FWord,'-n+v-a+p',PN,Tense],Second] | Rest],NS,NB).

grammar_rule(attach_relative_clause,[[xmax,S,'+n-v+a-p',[spec, head, comp_]]
 | Reststack],
[[[xmax,S1,'-n-v+a+p'],Second] | Rest],NS, NB):-
attach([[xmax,S,'+n-v+a-p',[spec,head,comp_]] | Reststack],
[[[xmax,S1,'-n-v+a+p'],Second] | Rest],NS,NB).

grammar_rule(attach_object,[[xmax, St, '-n+v', [spec, head, comp_]] | Reststack],
[[[xmax,E,'+n-v+a-p'],Second] | Rest],NS, NB):-
attach([[xmax, St, '-n+v', [spec, head, comp_]] | Reststack],
[[[xmax, E, '+n-v+a-p'],Second] | Rest],NS, NB).

grammar_rule(attach_object,[[xmax,St,'-n+v-a+p',[spec,head,comp_],Ty]
 | Reststack],
[[[xmax,E,'+n-v+a-p'],Second] | Rest],NS, NB):-
attach([[xmax,  St, '-n+v-a+p', [spec, head, comp_],Ty] | Reststack],
[[[xmax,E, '+n-v+a-p'],Second] | Rest],NS, NB).

grammar_rule(attach_object,[[xmax,St,'-n+v-a+p',[spec,head,comp_]]
 | Reststack],
[[[xmax,E,'+n-v+a-p'],Second] | Rest],NS, NB):-
attach([[xmax,  St, '-n+v-a+p', [spec, head, comp_]] | Reststack],
[[[xmax,E, '+n-v+a-p'],Second] | Rest],NS, NB).

grammar_rule(attach_object,[[xmax, St, '-n+v', [spec, head, comp_]] | Reststack],
[[[xmax,E,'+n-v+a-p',pn],Second] | Rest],NS, NB):-
attach([[xmax, St, '-n+v', [spec, head, comp_]] | Reststack],
[[[xmax, E, '+n-v+a-p',pn],Second] | Rest],NS, NB).

grammar_rule(attach_object,[[xmax,St,'-n+v-a+p',[spec,head,comp_],Ty]
 | Reststack],
[[[xmax,E,'+n-v+a-p',pn],Second] | Rest],NS, NB):-
attach([[xmax,  St, '-n+v-a+p', [spec, head, comp_],Ty] | Reststack],
[[[xmax,E, '+n-v+a-p',pn],Second] | Rest],NS, NB).

grammar_rule(attach_pp,[[xmax,S,'-n+v-a+p',[spec, head,comp_],Ty] | Reststack],
[[[xmax, S1,'-n-v'],Second] | Rest],NS,NB):-
attach([[xmax,S,'-n+v-a+p',[spec,head,comp_],Ty] | Reststack],
[[[xmax, S1, '-n-v'],Second] | Rest], NS, NB).

grammar_rule(attach_pp,[[xmax,S,'-n+v-a+p',[spec, head,comp_]] | Reststack],
[[[xmax, S1,'-n-v'],Second] | Rest],NS,NB):-
```

```
attach([[xmax,S,'-n+v-a+p',[spec,head,comp_]] | Reststack],
[[[xmax, S1, '-n-v'],Second] | Rest], NS, NB).

grammar_rule(attach_pp,[[xmax,S,'+n-v+a-p',[spec, head,comp_]] | Reststack],
[[[xmax, S1,'-n-v'],Second] | Rest],NS,NB):-
attach([[xmax,S,'+n-v+a-p',[spec,head,comp_]] | Reststack],
[[[xmax, S1, '-n-v'],Second] | Rest], NS, NB).

grammar_rule(attach_pp,[[xmax,S,'-n+v',
[spec, head,comp_]] | Reststack],
[[[xmax,S1,'-n-v'],Second] | Rest],NS,NB):-
attach([[xmax,S,'-n+v',[spec,head,comp_]] | Reststack],
[[[xmax,S1,'-n-v'],Second] | Rest],NS, NB).

%grammar_rule(attach_pp,[[xmax,S,'-n+v-a+p',[spec,  head,comp_],Ty] | Reststack],
%[[[xmax,S1,'-n-v+a+p'],Second] | Rest],NS, NB):-
%attach([[xmax,S,'-n+v-a+p',[spec,head,comp_],Ty] | Reststack],
%[[[xmax,S1,'-n-v+a+p'],Second] | Rest], NS, NB).

grammar_rule(attach_pp_object,[[xmax,S,'-n-v',[spec,head,comp_]] | Reststack],
[[[xmax,S1,'+n-v+a-p'],Second] | Rest],NS,NB):-
attach([[xmax,S,'-n-v',[spec, head, comp_]] | Reststack],
[[[xmax,S1,'+n-v+a-p'],Second] | Rest],NS, NB).

grammar_rule(passive_be,  [[xmax,  '-n+v+a+p',[spec,  head_,  comp]] | Reststack],
[[[_word,'-n+v'],[Words,'-n+v-a+p','+',T]] | Rest],Newstack, Newbuffer):-
attach([[xmax, '-n+v+a+p',[spec, head_, comp]] | Reststack],
[[[_word, '-n+v'], [Words, '-n+v-a+p','+',T]] | Rest],Newstack,
Newbuffer).

/*This rule, called by grammar_rule, attaches item form first buffer cell to the stack.*/

attach([[xmax, Features1, Template] | Reststack],
[[[[_word,'-n-v+a+p'],_F1], Second] | [X | L]],
[[xmax, [_word,'-n-v+a+p'], Features1, Template] | Reststack],
[[Second, X] | L]).

attach([[xmax, Features1, Template] | Reststack],
[[First, Second] | [X | L]],
[[xmax, First, Features1, Template] | Reststack],
[[Second, X] | L]).

attach([[xmax, Features1, Template] | Reststack],
[[First, Second]],
[[xmax, First, Features1, Template] | Reststack],
[[Second, []]]).

attach([[xmax, Structure, Features, Template] | Reststack],
[[First, Second] | [X | L]],
[[xmax, [Structure,First], Features, Template] | Reststack],
[[Second, X] | L]).

attach([[xmax, Structure, Features, Template] | Reststack],
[[First, Second]],
[[xmax, [Structure,First], Features, Template] | Reststack],
[[Second, []]]).

attach([[xmax, Structure, Features, Template,Type] | Reststack],
[[First, Second] | [X | L]],
[[xmax, [Structure,First], Features, Template,Type] | Reststack],
[[Second, X] | L]).

attach([[xmax, Structure, Features,Template,Type] | Reststack],
[[First, Second] | L],
[[xmax, [Structure,First],Features,Template,Type] | Reststack],
[[Second, L]]).

not_attach([S | Reststack],[[_First, Second] | [X | L]],
[S | Reststack],
[[Second, X] | L]).

not_attach([S | Reststack],
[[_First, Second] | []],
[S | Reststack],
[[Second, []]]).
```

The examples below are taken from the Rolls-Royce aircraft maintenance manual. The parsers MParserSub and LParserSub can parse all these noun-phrases.

Equipment and Material	Standard equipment
Consumable materials	relevant circuit breakers
OMat 4/23 Anti-seize compound, pure nickel special	
Overhaul Materials Manual (OMat)	Special tools
Expendable parts	General
these TASKS	a part in a different Chapter/Section/Subject
the Fig./item numbers	the appropriate Chapter/Section/Subject.
the engine in flight position	the positional relationship
Detailed radial locations	Referenced Procedures
cowl doors Torque tightening technique	connection of electrical plugs the igniter plugs
the Igniter Plugs	item numbers in parentheses in the text
this TASK	the Removal of the Igniter Plugs
the low tension electrical connector	the high tension leads or igniter plugs
the H.E. ignition unit	one minute
any stored energy	electrical supply to ignition system.
DO-NOT-OPERATE identifiers.	the high energy ignition unit.
access panel or panels	the right of the drains tank on the by-pass duct
the access panel	number 4 combustion liner.
fuel cooled oil cooler (F.C.O.C.)	number 8 combustion liner.
the Associated Parts	the high energy (H.E.) ignition leads (1).
the igniter plugs (2)	number 4 and/or number 8 combustion liners.
blanks apertures	a suitable blank
the lead end of the igniter.	diffuser case.
strong spotlightRR289200 Protective workmat 1 off	
the L.P. compressor and L.P. turbine assembly	
electrical supply	L.P. compressor and L.P. turbine blades
fingers and hands	metal bars or similar equipment
the L.P. compressor or turbine	the vanes
the front of the engine	the L.P. turbine at the rear of the engine.
the L.P. turbine blades	impact damage, cracks or metal deposits.
the rear of the engine	light impact damage to the turbine blade airfoil
the engine	light impact damage to the turbine blade shroud
the metal turbine blades which are cracked or torn	
OMat 1031 Lubricating oil	turbine blades which have a segment missing.
turbine blades with metal deposits.	the thrust reverse system.
OMat 238 Lockwire	spacers and distance pieces.
the removal of the T26 thermal unit	the T26 thermal unit

the electrical connector (2)
the low speed (L.S.) gearbox.
the bleed valve control rod (7).
the thermocouple/thermophial probe
the 4 off bolts
the T26 thermal unit (6)
the bolts
the blanking plate
the regulator
the intermediate case
the thermal unit mounting face
OMat 1031 lubricating oil
the probe a new gasket (6A)
the T26 thermal unit (6)
the thermocouple/thermophial tube
the bleed valve control rod (7).
a new gasket (1A).
the low speed gearbox
the electrical connector (2)
clean receptacle
fluid container
fine mesh filter
OMat 1252 Kraft paper tape
OMat 101 Kerosine
OMat 1020 Liquid paraffin
the air intake
the nose cone fairing (3)
the temporary marker
KU37366 socket wrench 1 off
the three bolts (1) washers (2)
the numbered locations
the rotor spinner assembly
the nine bolt locations
the airflow control-r.p.m. signal transmitter
the quillshaft
a work surface.
the correct preservation procedure
all orifices
ingress of foreign matter
the signal pressure outlet
the H.P. fuel inlet.
the signal pressure outlet
several times.
the L.P. fuel return outlet
all the internal parts.
a dry suface
new blanks
the bare patches
OMat 1047A preservation compound
OMat 1252 kraft paper tape
transit box
strong, dry and undamaged box
the package document
OMat 1238 polythene bag
a new seal ring (9A)
the seal ring.
plain washers and spring washers.
the fuel drain tube connector (7)

the L.P. r.p.m. indicator generator
the gasket.the I.G.V. control rod (8)
the securing clips (4) and (5).
the unit
airflow control regulator and actuator
the T26 thermocouple/thermophial assembly (3)
the T26 assembly.
the thermal unit mounting face
a suitable blank
the installation of the T26 thermal unit
the T26 thermocouple/thermophial probe (3)
bolts, spreader washers and spring washers
the thermal unit joint face the regulator
4 off bolts
the clips (4) and (5)
the I.G.V. control rod (8)
the L.P. r.p.m. indicator generator (1)
bolts, spring washers and flat washers
the L.P. r.p.m. indicator generator;
minimum capacity 4 Litres (1 gallon)
OMat 1003 Mineral oil (fuel system inhibitor)
OMat 1047A Preservation compound
OMat 1238 Polythene bag
OMat 1011 Engine lubricating oil (synthetic)
OMat 262 felt or fibre tip temporary marker
a mark
the rotor spinner assembly (7)
the three bolt locations
a suitable ratchet handle,
the bolts and washers
a piece of cardboard.
the fuel cut off front ring (8)
the spinner assembly.
the transmitter
the bolts (8) and (9)the gearbox,
the seal ring
a maximum of 48 hours
all fuel
OMat 1003 mineral oil (fuel system inhibitor)
free flow
an approved tapered blanking plug
the quillshaft
a continuous flow of inhibitor
an approved blank.
the exterior of the transmitter clean and dry
30 minutes
a suitable container for storage
the housing and quillshaft
the approved quillshaft transportation cover
OMat 1238 polythene bag.
between padded, wooden, clamping pieces
the relevant storage details
the airflow control r.p.m. signal transmitter
the H.P. r.p.m. signal transmitter
OMat 1011 engine lubricating oil
the bolts (8) and (9)
a new gasket (7A)
the 2 bolts

the drain tube.
the H.P. r.p.m. signal tube (10).
OMat 1020 liquid paraffin B.P.
the L.P. fuel return tube (11)
a new seal ring (12A)
the connection (12)
new seal rings (1A) and (1B)
new seal rings (4A)
new seal rings (5A)
the H.P. fuel pump
the transfer tube (4)
bonding leads
the Tay engine
the period of self-bleeding
an increase in vibration
clean OMat 1011 engine oil
power plant
air offtake blanks.
the unit
the by-pass duct
the adjusting spacer (2) special tool GU28208
special spanner 1 off
the 12th stage air offtake outlet duct assembly (1)
the lock ring
a new lock ring
the locking plate (1)
OMat 402 petroleum jelly
the Corrujoint gasket (4)
the locating ring face.
the dogs
the cover locking plate
the blanking cover assembly
the outlet connector assembly bore
the locating ring
a vee-clamp assembly (5)
a new seal ring (3)
the by-pass duct.
the 12th stage inner cover assembly
the seal ring (2).
the 7th stage manifold cover (5)
the 7th stage duct seat assembly (1).
the 7th stage manifold cover
the 7th stage sealing cover nuts (1), spreader washer (5) and bolts (6
the sealing cover (4)
the inner cover assembly (2)
the inner cover assembly.
the sealing cover
the 7th stage inner cover assembly
the 7th stage air offtake sealing cover
the sealing cover (4)
the seal ring (3).
the vee-clamp assembly (5)
the blanking cover assembly
12th stage air offtake sealing cover
the air outlet duct locating ring.
the locking plate (1)
the blanking cover (2).
the Corrujoint gasket (4).

a new seal ring (10A)
bolts, spring washers and flat washers
new seal rings (11A) and (11B)
liquid paraffin
the connector (12)
the end of tube (11)
the H.P. shut-off valve elbow connector
the H.P. fuel transfer tube (4)
the H.P. fuel tube (5)
bolts, spring washers and a flat washer
the clip (6)
the fuel system
a continuous bleed system
a facility for manual bleeding
all threads and abutment faces
fitting vee band coupling clamps
the 7th and 12th stage compressor air ducts
the 7th and 12th stage air offtake blanks
the 7th and 12th stage air offtakes
the 12th stage air offtake outlet duct assembly
the seal ring (3)
special tool AB12345 extractor 1 off
the 12th stage air offtake seal carrier assembly (4).
the 12th stage air offtake seal carrier assembly
the 12th stage offtake blanking cover
the 12th stage offtake blanking cover (2)
the 12th stage outlet cover locating ring (3)
the air outlet duct threaded connector.
special tool AB23456 torque spanner 1 off
the air outlet duct locating ring
the blanking cover assembly
the 12th stage air offtake sealing cover
the 12th stage sealing cover (4) seal groove
the 12th stage inner cover assembly (2)
the 12th stage sealing cover
the 7th stage air offtake outlet duct assembly
the 7th stage air offtake outlet duct assembly (1)
special tool GU28208 special spanner 1 off,
special tool AB12345 extractor 1 off
the 7th stage air offtake seal carrier assembly (3).
the 7th stage outlet connector retainer (4)
the 7th stage outlet connector assembly (5)
the outlet adjusting spacer (6)
the 7th staqe air offtake blanking cover
the 7th stage outlet spherical seat (7).
the stage 7 sealing cover (4) seal groove
the 7th stage inner cover assembly (2)
nuts (1),spreader washer (5) and bolts (6)
the 7th stage air offtake sealing cover
the bolts (6),spreader washers (5) and nuts (1)
the 7th stage air offtake blanking cover (5)
the 7th stage air offtake blanking cover
the bolts (6),spreader washers (5) and nuts (1)
the 12th stage offtake blanking cover
special tool AB23456 torque spanner 1 off

the 12th stage outlet securing ring	the 12th stage outlet locating ring (3)
the adjusting spacer (6)	the air outlet duct threaded connector (4).
the 7th stage outlet spherical seat (7)	the 7th stage offtake outlet duct assembly
the 7th stage outlet spherical seat	OMat 4/47 jointing compound
the adjusting spacer	the 7th stage outlet connector assembly
the spherical connection	the 7th stage outlet connector retainer (4)
the outlet spherical seat	the outlet connector assembly (5)
the outlet connector retainer self-locking nuts	
the 7th stage air offtake seal carrier	the 7th stage air offtake seal carrier seal groove
a new seal ring (2)	the 7th stage air offtake seal carrier (3)
the 7th stage outlet connector.	the 7th stage air outlet duct (1) threads
the 7th stage air outlet duct	special tool GU28208 special spanner 1 off
the lock ring (5)	the 7th stage air offtake outlet connector assembly
a new seal ring (2)	12th stage air offtake seal carrier seal groove
the 12th stage outlet connector	the 12th stage air offtake seal carrier (4)
the adjusting washer	the bottom of the 12th stage air offtake seal carrier
the 7th and 12th stage air offtakes	the 12th stage air offtake outlet connector assembly

the procedure for the removal and installation of the igniter plugs
numbers in a clockwise direction that start from an engine top position when you look from the rear unless otherwise told
the electrical discharge of the high energy (H.E.) ignition unit
the procedure for the inspection of the stage 3 L.P.turbineblades.
the examination of the stage 3 low pressure (L.P.) nozzle guide vanes
the procedure for the removal and installation of the T26 thermal unit.
all flexible seal rings, gaskets, keywashers and split pins.
for identification, lubrication and installation of flexible seal rings
the bi-hex head bolts that attach the L.p. r.p.m. indicator generator (1)
the regulator and the flange of the intermediate compressor case
the flexible section of the T26 thermocouple/thermophial probe

a test of the high pressure (H.P.) compressor control system
the procedures for the removal of the L.P. compressor rotor blade(s)
the installation of the airflow control r.p.m. signal transmitter
its location on the rear face of the high speed (H.S.) gearbox
a test of the high pressure (H.P.) compressor control system
the 12th stage air offtake outlet connector assembly lock ring
bolts (6),spring washers (7),and spreader washers (8).
the bolts and self locking nuts that attach the 7th stage connector assembly (3).
the 7th stage air offtake outlet connector assembly lock ringstud retaining rings (2),
studs (3),retaining screws (4),spreader washers (6) and nuts (7)
the instructions for the removal of the 7th and 12th stage air offtake blanks the installation of the 7th and 12th stage compressor air ducts
the nuts (7),spreader washers (6),retaining screws (4) and stud retaining rings (2) the bolts (6), spring washers (7) and spreader washers (8)

These example parses have been parsed by MparserSub.

```
| ?- start(NP).

|: equipment and material.
parse took 0.012 sec.

memory (total)       338,736 bytes
    program space    207,672 bytes:  207,672 in use  0 free
    global space      63,444 bytes:   21,436 in use  42,008
free
      global stack                    21,372 bytes
```

```
        trail                                 64  bytes
    local stack       65,508 bytes:          848 in use 64,660
                                                          free
0.000  sec.  for  0  program,  0  global  and  ?  local  space
overflows.
0.000 sec. for 0 garbage collections which collected 0 bytes.
0.316 sec. runtime. 0 pagefaults.

NP=
[xmax,[[[attach_noun,[equipment,'+n-v+a-p',sg]],
[attach_conj,[and,'+n-v+a-p',conj]]],
[attach_noun,[material,'+n-v+a-p',sg]]],'+n-v+a-p']

| ?- start(NP).

|: standard equipment.
parse took 0.016 sec.

memory (total)        338,736 bytes
    program space     207,672 bytes:  207,672 in use  0 free
    global space       63,444 bytes:   20,164 in use  43,280
                                                          free
        global stack                   20,164 bytes
        trail                               0 bytes
    local stack        65,508 bytes:      324 in use 65,184
                                                          free
0.000  sec.  for  0  program,  0  global  and  ?  local  space
overflows.
0.000 sec. for 0 garbage collections which collected 0 bytes.
0.266 sec. runtime. 0 pagefaults.
NP=
[xmax,[attach_noun,[[standard,equipment],'+n-v+a-p',sg]],
  '+n-v+a-p']

| ?- start(NP).

|: consumable materials.
parse took 0.013 sec.
memory (total)        338,736 bytes
    program space     207,672 bytes:  207,672 in use  0 free
    global space       63,444 bytes:   20,448 in use  42,996
                                                          free
        global stack                   20,448 bytes
        trail                               0 bytes
    local stack        65,508 bytes:      324 in use 65,184
                                                          free
0.000  sec.  for  0  program,  0  global  and  ?  local  space
overflows.
0.000 sec. for 0 garbage collections which collected 0 bytes.
0.233 sec. runtime. 0 pagefaults.
NP=


[xmax,[attach_noun,[[consumable,materials],'+n-v+a-p',pl]],
'+n-v+a-p']

| ?- start(NP).
```

```
|: OMat4/23 anti-seize compound, pure nickel special.
parse took 0.016 sec.
memory (total)       338,736 bytes
    program space    207,672 bytes:  207,672 in use  0 free
    global space      63,444 bytes:   21,976 in use  41,468
                                                          free
        global stack                  21,904 bytes
        trail                             72 bytes
    local stack       65,508 bytes:      936 in use 64,572
                                                          free
0.000  sec.  for  0  program,  0  global  and  ?  local  space
overflows.
0.000 sec. for 0 garbage collections which collected 0 bytes.
0.316 sec. runtime. 0 pagefaults.
NP=
[xmax,[[[attach_ref,['OMat4/23','+n-v+a+p',ref]],
[attach_noun,[['anti-seize',compound],'+n-v+a-p',sg]]],

[attach_noun,[[pure,nickel,special],'+n-v+a-p',sg]]],'+n-v+
a-p']

| ?- start(NP).

|: the procedure for the removal and installation of the
igniter plugs.
parse took 0.033 sec.
memory (total)       338,744 bytes
    program space    207,680 bytes:  207,680 in use  0 free
    global space      63,444 bytes:   29,420 in use  34,024
                                                          free
        global stack                  29,256 bytes
        trail                            164 bytes
    local stack       65,508 bytes:    2,224 in use 63,284
                                                          free
0.000  sec.  for  0  program,  0  global  and  ?  local  space
overflows.
0.000 sec. for 0 garbage collections which collected 0 bytes.
0.583 sec. runtime. 0 pagefaults.
NP=

[xmax,[[[attach_det,[the,'+n-v-a-p']],[attach_noun,[procedu
re, '+n-v+a-p', sg]]],
[attach_pp,[xmax,[[attach_prep,[for,'-n-v']],
[attach_pp_object,[xmax,[[[[[attach_det,[the,'+n-v-a-p']],
[attach_noun,[removal,'+n-v+a-p',sg]]],[attach_conj,[and,'+
n-v+a-p', conj]]],
[attach_noun,[installation,'+n-v+a-p',sg]]],
[attach_pp,[xmax,[[attach_prep,[of,'-n-v']],
[attach_pp_object,[xmax,[[attach_det,[the,'+n-v-a-p']],
 [attach_noun,[[domain(for),
[igniter,'+n-v+a-p',sg],range(for),
[plugs,'+n-v+a-p',pl]],'+n-v+a-p',pl]]],'+n-v+a-p']]],'-n-v
']]],'+n-v+a-p']]],         '-n-v']]],'+n-v+a-p']

| ?- start(NP).

|: item numbers in parentheses in the text.
parse took 0.033 sec.
```

```
memory (total)        338,764 bytes
    program space     207,700 bytes:  207,700 in use  0 free
    global space       63,444 bytes:   26,836 in use  36,608
                                                        free
        global stack                   26,712 bytes
        trail                             124 bytes
    local stack        65,508 bytes:    1,612 in use 63,896
                                                        free
0.000 sec. for 0 program, 0 global and ? local space
overflows.
0.000 sec. for 0 garbage collections which collected 0 bytes.
0.483 sec. runtime. 0 pagefaults.
NP=
 [xmax,[[attach_noun,[[domain(obj),
[item,'+n-v+a-p',sg],range(obj),
[numbers,'+n-v+a-p',pl]],'+n-v+a-p',pl]],
[attach_pp,[xmax,[[attach_prep,[in,'-n-v']],[attach_pp_obje
ct,[xmax,  [[attach_noun,[parentheses,'+n-v+a-p',pl]],
[attach_pp,[xmax,[[attach_prep,[in,'-n-v']],
[attach_pp_object,[xmax,[[attach_det,[the,'+n-v-a-p']],
[attach_noun,[text,'+n-v+a-p',sg]]],'+n-v+a-p']]],'-n-v']]],
   '+n-v+a-p']]],'-n-v']]],'+n-v+a-p']

| ?- start(NP).

|: referenced procedures.
parse took 0.013 sec.
memory (total)        338,752 bytes
    program space     207,688 bytes:  207,688 in use  0 free
    global space       63,444 bytes:   20,488 in use  42,956
                                                        free
        global stack                   20,488 bytes
        trail                               0 bytes
    local stack        65,508 bytes:      324 in use 65,184
                                                        free
0.000 sec. for 0 program, 0 global and ? local space
overflows.
0.000 sec. for 0 garbage collections which collected 0 bytes.
0.300 sec. runtime. 0 pagefaults.

NP=
 [xmax,[attach_noun,[[referenced,procedures],'+n-v+a-p',pl]],
  '+n-v+a-p']

| ?- start(NP).

|: connection of electrical plugs.
parse took 0.017 sec.
memory (total)        338,744 bytes
    program space     207,680 bytes:  207,680 in use  0 free
    global space       63,444 bytes:   23,740 in use  39,704
                                                        free
        global stack                   23,652 bytes
        trail                              88 bytes
    local stack        65,508 bytes:    1,244 in use 64,264
                                                        free
0.000 sec. for 0 program, 0 global and ? local space
overflows.
```

```
0.000 sec. for 0 garbage collections which collected 0 bytes.
0.333 sec. runtime. 0 pagefaults.

NP=
 [xmax,[[attach_noun,[connection,'+n-v+a-p',sg]],
[attach_pp,[xmax,[[attach_prep,[of,'-n-v']],[attach_pp_obje
ct,[xmax, [attach_noun,[[domain(mod),
[electrical,'+n-v+a+p'],range(mod),
[plugs,'+n-v+a-p',pl]],'+n-v+a-p',pl]],'+n-v+a-p']]],'-n-v'
]]],'+n-v+a-p']

| ?- start(NP).

|: the removal of igniter plugs.
parse took 0.017 sec.
memory (total)        338,744 bytes
    program space     207,680 bytes:  207,680 in use  0 free
    global space       63,444 bytes:   24,196 in use  3 9 , 7 0 4
                                                           free
        global stack                   24,104 bytes
        trail                              92 bytes
    local stack        65,508 bytes:    1,336 in use 64,172
                                                           free
0.000  sec.  for  0  program,  0  global  and  ?  local  space
overflows.
0.000 sec. for 0 garbage collections which collected 0 bytes.
0.400 sec. runtime. 0 pagefaults.
NP=

[xmax,[[[attach_det,[the,'+n-v-a-p']],[attach_noun,[removal
, ' + n - v + a - p ' , s g ] ] ] ,
[attach_pp,[xmax,[[attach_prep,[of,'-n-v']],[attach_pp_obje
c    t    ,    [    x    m    a    x    ,
[attach_noun,[[domain(for),[igniter,'+n-v+a-p',sg],range(fo
r),
[plugs,'+n-v+a-p',pl]],'+n-v+a-p',pl]],'+n-v+a-p']]],'-n-v'
]]],'+n-v+a-p']

| ?- start(NP).

|: the electrical discharge of the high energy (HE) ignition
unit.
parse took 0.033 sec.
memory (total)        338,736 bytes
    program space     207,672 bytes:  207,672 in use  0 free
    global space       63,444 bytes:   26,592 in use  3 6 , 8 5 2
                                                           free
        global stack                   26,436 bytes
        trail                             156  bytes
    local stack        65,508 bytes:    2,124 in use 63,384
                                                           free
0.000  sec.  for  0  program,  0  global  and  ?  local  space
overflows.
0.000 sec. for 0 garbage collections which collected 0 bytes.
0.566 sec. runtime. 0 pagefaults.
NP=
  [ x m a x , [ [ [ a t t a c h _ d e t , [ t h e , ' + n - v - a - p ' ] ] ,
[attach_noun,[[domain(mod),
[electrical,'+n-v+a+p'],range(mod),[discharge,'+n-v+a-p',sg
] ] , ' + n - v + a - p ' , s g ] ] ] ,
[attach_pp,[xmax,[[attach_prep,[of,'-n-v']],
[attach_pp_object,[xmax,[[attach_det,[the,'+n-v-a-p']],
[attach_noun,[[[domain(use),domain(mod),[high,'+n-v+a+p'],r
ange(mod),
[energy,'+n-v+a-p',sg],['(HE)','+n-v+a-p',sg],range(use),do
main(obj),
[ignition,'+n-v+a-p',sg],range(obj),
[unit,'+n-v+a-p',sg]]],'+n-v+a-p',sg]]],'+n-v+a-p']]],'-n-v
']]],'+n-v+a-p']

| ?- start(NP).

|: the high tension leads or igniter plugs.
parse took 0.025 sec.
memory (total)      338,744 bytes
    program space   207,680 bytes:  207,680 in use  0 free
    global space     63,444 bytes:   24,088 in use  36,852
                                                         free
        global stack                 23,964 bytes
        trail                           124  bytes
    local stack      65,508 bytes:    1,664 in use 63,844
                                                         free
0.000  sec.  for  0  program,  0  global  and  ?  local  space
overflows.
0.000 sec. for 0 garbage collections which collected 0 bytes.
0.450 sec. runtime. 0 pagefaults.
NP=

[xmax,[[[[attach_det,[the,'+n-v-a-p']],[attach_noun,[[[doma
in(use),
domain(mod),[high,'+n-v+a+p'],range(mod),[tension,'+n-v+a-p
',sg],range(use),
[leads,'+n-v+a-p',pl]]],'+n-v+a-p',pl]]],[attach_conj,[or,'
+ n - v + a - p ' , c o n j ] ] ] ,
[attach_noun,[[domain(for),[igniter,'+n-v+a-p',sg],range(fo
r),[plugs,'+n-v+a-p',pl]],'+n-v+a-p',pl]]],'+n-v+a-p']

| ?- start(NP).

|: the low tension electrical connector.
parse took 0.017 sec.
memory (total)      338,736 bytes
    program space   207,672 bytes:  207,672 in use  0 free
    global space     63,444 bytes:   22,296 in use  36,852
                                                         free
        global stack                 22,216 bytes
        trail                            80 bytes
    local stack      65,508 bytes:    1,164 in use 63,344
                                                         free
0.000  sec.  for  0  program,  0  global  and  ?  local  space
overflows.
0.000 sec. for 0 garbage collections which collected 0 bytes.
0.383 sec. runtime. 0 pagefaults.
NP=

[xmax,[[attach_det,[the,'+n-v-a-p']],[attach_noun,[[[domain
```

```
(use),
domain(mod),[low,'+n-v+a+p'],range(mod),[tension,'+n-v+a-p'
,sg],range(use),
domain(mod),[electrical,'+n-v+a+p'],range(mod),[connector,'
+n-v+a-p',sg]]],
'+n-v+a-p',sg]]],'+n-v+a-p']

| ?- start(NP).

|: relevant circuit breakers.
parse took 0.017 sec.
memory (total)       338,736 bytes
    program space    207,672 bytes:  207,672 in use  0 free
    global space      63,444 bytes:   24,188 in use  3 6 , 8 5 2
                                                          free
        global stack                  24,080 bytes
        trail                            108 bytes
    local stack       65,508 bytes:    1,588 in use 63,920
                                                          free
0.000  sec.  for  0  program,  0  global  and  ?  local  space
overflows.
0.000 sec. for 0 garbage collections which collected 0 bytes.
0.450 sec. runtime. 0 pagefaults.
NP=

[xmax,[attach_noun,[[domain(mod),[relevant,'+n-v+a+p'],rang
e(mod),
[[circuit,breakers],'+n-v+a-p',pl]],'+n-v+a-p',pl]],'+n-v+a
-p']

| ?- start(NP).

|: electrical supply to ignition system.
parse took 0.017 sec.
memory (total)       338,736 bytes
    program space    207,672 bytes:  207,672 in use  0 free
    global space      63,444 bytes:   24,188 in use  3 6 , 8 5 2
                                                          free
        global stack                  24,080 bytes
        trail                            108 bytes
    local stack       65,508 bytes:    1,588 in use 64,920
                                                          free
0.000  sec.  for  0  program,  0  global  and  ?  local  space
overflows.
0.000 sec. for 0 garbage collections which collected 0 bytes.
0.450 sec. runtime. 0 pagefaults.
NP=

[xmax,[[attach_noun,[[domain(mod),[electrical,'+n-v+a+p'],r
ange(mod),
[supply,'+n-v+a-p',sg]],'+n-v+a-p',sg]],
[attach_pp,[xmax,[[attach_prep,[to,'-n-v']],[attach_pp_obje
c  t  ,   [  x  m  a  x  ,
[attach_noun,[[domain(for),[ignition,'+n-v+a-p',sg],range(f
or),
[system,'+n-v+a-p',sg]],'+n-v+a-p',sg]],'+n-v+a-p']]],'-n-v
']]],'+n-v+a-p']
```

```
| ?- start(NP).

|: the nine bolt locations.
parse took 0.017 sec.
memory (total)       338,736 bytes
    program space    207,672 bytes:  207,672 in use  0 free
    global space      63,444 bytes:   22,304 in use  36,852
                                                        free
        global stack                  22,224 bytes
        trail                             80 bytes
    local stack       65,508 bytes:    1,188 in use 64,320
                                                        free
0.000  sec.  for  0  program,  0  global  and  ?  local  space
overflows.
0.000 sec. for 0 garbage collections which collected 0 bytes.
0.300 sec. runtime. 0 pagefaults.
NP=

[xmax,[[[attach_det,[the,'+n-v-a-p']],[attach_num,[nine,'+n
- v + a - p ' , n u m ] ] ] ,
[attach_noun,[[domain(obj),[bolt,'+n-v+a-p',sg],range(obj),
[locations,'+n-v+a-p',pl]],'+n-v+a-p',pl]]],'+n-v+a-p']

| ?- start(NP).

|: the  removal  of  the  7th  and  12th  stage  compressor  air
ducts.
parse took 0.017 sec.
memory (total)       338,736 bytes
    program space    207,672 bytes:  207,672 in use  0 free
    global space      63,444 bytes:   27,352 in use  36,092
                                                        free
        global stack                  27,196 bytes
        trail                            156 bytes
    local stack       65,508 bytes:    2,004 in use 63,504
                                                        free
0.000  sec.  for  0  program,  0  global  and  ?  local  space
overflows.
0.000 sec. for 0 garbage collections which collected 0 bytes.
0.550 sec. runtime. 0 pagefaults.
NP=
[xmax,[[[attach_det,[the,'+n-v-a-p']],[attach_noun,[removal
, ' + n - v + a - p ' , s g ] ] ] ,
[attach_pp,[xmax,[[attach_prep,[of,'-n-v']],
[attach_pp_object,[xmax,[[[[attach_det,[the,'+n-v-a-p']],[a
ttach_adj,[domain(ref),
['7th','+n-v+a+p']]]]],[attach_conj,[and,'+n-v+a-p',conj]]],
[attach_noun,[[domain(ref),[['12th',stage],'+n-v+a-p',sg],r
ange(ref),
domain(has),[compressor,'+n-v+a-p',sg],range(has),
domain(for),[air,'+n-v+a-p',sg],
range(for),[ducts,'+n-v+a-p',pl]],'+n-v+a-p',pl]]]],
'+n-v+a-p']]],'-n-v']]],'+n-v+a-p']
```

These example parses have been parsed by LparserSub.

```
| ?- run(S,B).
```

```
|: equipment and material.
parse took 0.013 sec.
memory (total)       368,828 bytes
    program space    237,764 bytes:  237,764 in use  0 free
    global space      63,444 bytes:   19,276 in use  44,168
                                                          free
        global stack                  19,160 bytes
        trail                            116 bytes
    local stack       65,508 bytes:    1,612 in use     63,896
                                                          free
0.000  sec.  for  0  program,  0  global  and  ?  local  space
overflows.
0.000 sec. for 0 garbage collections which collected 0 bytes.
0.566 sec. runtime. 0 pagefaults.
S=

[noun_phrase(sg,np(n(equipment,sg),conj(and),n(material,sg)
))],
B= []

| ?- run(S,B).

|: standard equipment.
parse took 0.017 sec.
memory (total)       368,836 bytes
    program space    237,772 bytes:  237,772 in use  0 free
    global space      63,444 bytes:   18,520 in use  44,924
                                                          free
        global stack                  18,500 bytes
        trail                             20 bytes
    local stack       65,508 bytes:      468 in use     65,040
                                                          free
0.000  sec.  for  0  program,  0  global  and  ?  local  space
overflows.
0.000 sec. for 0 garbage collections which collected 0 bytes.
0.566 sec. runtime. 0 pagefaults.
S=
 [noun_phrase(sg,np(n([standard,equipment],sg)))],
B= []

| ?- run(S,B).

|: consumable materials.
parse took 0.013 sec.
memory (total)       368,828 bytes
    program space    237,764 bytes:  237,772 in use  0 free
    global space      63,444 bytes:   18,728 in use  44,716
                                                          free
        global stack                  18,708 bytes
        trail                             20 bytes
    local stack       65,508 bytes:      468 in use     65,040
                                                          free
0.000  sec.  for  0  program,  0  global  and  ?  local  space
overflows.
0.000 sec. for 0 garbage collections which collected 0 bytes.
0.216 sec. runtime. 0 pagefaults.
S=
 [noun_phrase(pl,np(n([consumable,materials],pl)))],
B= []

| ?- run(S,B).

|: OMat4/23 anti-seize compound pure nickel special.
parse took 0.100 sec.
memory (total)       368,964 bytes
    program space    237,900 bytes:  237,900 in use  0 free
    global space      63,444 bytes:   24,664 in use  38,780
                                                           free
        global stack                  24,272 bytes
        trail                            392 bytes
    local stack       65,508 bytes:    5,068  in use      60,440
                                                           free
0.000  sec.  for  0  program,  0  global  and  ?  local  space
overflows.
0.000 sec. for 0 garbage collections which collected 0 bytes.
2.716 sec. runtime. 0 pagefaults.
S=

[noun_phrase(sg,np(ref('OMat4/23'),n(['anti-seize',compound
],sg),
      n([pure,nickel,special],sg)))],
B= []

| ?- run(S,B).

|: the procedure for the removal and installation of the
igniter plugs.
parse took 0.050 sec.
memory (total)       368,964 bytes
    program space    237,900 bytes:  237,900 in use  0 free
    global space      63,444 bytes:   24,664 in use  38,780
                                                           free
        global stack                  24,272 bytes
        trail                            392 bytes
    local stack       65,508 bytes:    5,068  in use      60,440
                                                           free
0.000  sec.  for  0  program,  0  global  and  ?  local  space
overflows.
0.000 sec. for 0 garbage collections which collected 0 bytes.
2.716 sec. runtime. 0 pagefaults.
S=
 [noun_phrase(sg,np(np(np(d(the),n(procedure,sg)),
      pp(p(for),np(d(the),n(removal,sg),
      conj(and),n(installation,sg)))),
    pp(p(of),np(d(the),n([domain(for),
    noun(_20658,n([igniter],sg)),
      range(for),noun(_21154,n([plugs],pl))]))))))],
B= []

| ?- run(S,B).

|: item numbers in parentheses in the text.
parse took 0.050 sec.
memory (total)       368,992 bytes
    program space    237,928 bytes:  237,928 in use  0 free
    global space      63,444 bytes:   22,808 in use  40,636
```

```
                                                                    free
        global stack                          22,496 bytes
        trail                                    312 bytes
    local stack        65,508 bytes:         3,572 in use          61,936
                                                                    free
0.000  sec.  for  0  program,  0  global  and  ?  local  space
overflows.
0.000 sec. for 0 garbage collections which collected 0 bytes.
1.716 sec. runtime. 0 pagefaults.
S=

[noun_phrase(pl,np(np(np(n([domain(obj),noun(_7599,n([item]
,sg)),
range(obj),noun(_22618,n([numbers],pl))]))),
pp(p(in),np(n(parentheses,pl)))),
pp(p(in),np(d(the),n(text,sg)))))],
B= []

| ?- run(S,B).

|: referenced procedures.
parse took 0.014 sec.
memory (total)        368,828 bytes
    program space     237,764 bytes:  237,764 in use  0 free
    global space       63,444 bytes:   18,768 in use  44,676
                                                                    free
        global stack                          18,748 bytes
        trail                                     20 bytes
    local stack        65,508 bytes:             468 in use        65,040
                                                                    free
0.000  sec.  for  0  program,  0  global  and  ?  local  space
overflows.
0.000 sec. for 0 garbage collections which collected 0 bytes.
0.233 sec. runtime. 0 pagefaults.

S=
 [noun_phrase(pl,np(n([referenced,procedures],pl)))],
B= []

| ?- run(S,B).

|: connection of electrical plugs.
parse took 0.033 sec.
memory (total)        368,960 bytes
    program space     237,896 bytes:  237,896 in use  0 free
    global space       63,444 bytes:   21,380 in use  42,064
                                                                    free
        global stack                          21,120 bytes
        trail                                    260 bytes
    local stack        65,508 bytes:         2,686 in use          62,820
                                                                    free
0.000  sec.  for  0  program,  0  global  and  ?  local  space
overflows.
0.000 sec. for 0 garbage collections which collected 0 bytes.
1.133 sec. runtime. 0 pagefaults.
S=
 [noun_phrase(pl,np(np(n(connection,sg)),
      pp(p(of),np(n([domain(mod),adj(_7444,a([electrical])),
        range(mod),noun(_7822,n([plugs],pl))]))))],
B= []

| ?- run(S,B).

|: the removal of the igniter plugs.
parse took 0.017 sec.
memory (total)       368,964 bytes
    program space    237,900 bytes:  237,900 in use  0 free
    global space      63,444 bytes:   21,968 in use  41,476
                                                      free
        global stack                  21,692 bytes
        trail                            276 bytes
    local stack       65,508 bytes:    3,240 in use     62,268
                                                      free
0.000  sec.  for  0  program,  0  global  and  ?  local  space
overflows.
0.000 sec. for 0 garbage collections which collected 0 bytes.
1.450 sec. runtime. 0 pagefaults
S=
[noun_phrase(sg,np(np(d(the),n(removal,sg)),
pp(p(of),np(d(the),n([domain(for),noun(_10007,n([igniter],s
g)),
range(for),noun(_10503,n([plugs],pl))]))))],
B= []

| ?- run(S,B).

|: the high tension leads or igniter plugs.
parse took 0.016 sec.
memory (total)       369,152 bytes
    program space    237,088 bytes:  237,088 in use  0 free
    global space      63,444 bytes:   23,388 in use  40,056
                                                      free
        global stack                  22,988 bytes
        trail                            400 bytes
    local stack       65,508 bytes:    4,048 in use     61,460
                                                      free
0.000  sec.  for  0  program,  0  global  and  ?  local  space
overflows.
0.000 sec. for 0 garbage collections which collected 0 bytes.
1.816 sec. runtime. 0 pagefaults
S=
 [noun_phrase(pl,np(d(the),n([domain(use),
      domain(mod),adj(_10806,a([high])),
      range(mod),noun(_10908,n([tension],sg)),
      range(use),noun(_18048,n([leads],pl))]),
      conj(or),n([domain(for),noun(_13675,n([igniter],sg)),
      range(for),noun(_18066,n([plugs],pl))])))],
B= []

| ?- run(S,B).

|: the low tension electrical connector.
parse took 0.017 sec.
memory (total)       369,068 bytes
    program space    238,004 bytes:  238,004 in use  0 free
    global space      63,444 bytes:   21,384 in use  42,060
```

```
                                                              free
       global stack                         21,112 bytes
       trail                                   272 bytes
    local stack       65,508 bytes:       2,780 in use       62,728
                                                              free
0.000  sec.  for  0  program,  0  global  and  ?  local  space
overflows.
0.000 sec. for 0 garbage collections which collected 0 bytes.
1.233 sec. runtime. 0 pagefaults
S=
 [noun_phrase(sg,np(d(the),n([domain(use),
       domain(mod),adj(_7007,a([low])),
       range(mod),noun(_7109,n([tension],sg)),
       range(use),
    domain(mod),adj(_7484,a([electrical])),
       range(mod),noun(_8055,n([connector],sg))]))))],
B= []

| ?- run(S,B).

|: relevant circuit breakers.
parse took 0.000 sec.
memory (total)        369,980 bytes
    program space     237,916 bytes:  237,916 in use  0 free
    global space       63,444 bytes:   20,136 in use  43,308
                                                              free
       global stack                         20,004 bytes
       trail                                   132 bytes
    local stack       65,508 bytes:       1,456 in use       64,052
                                                              free
0.000  sec.  for  0  program,  0  global  and  ?  local  space
overflows.
0.000 sec. for 0 garbage collections which collected 0 bytes.
0.516 sec. runtime. 0 pagefaults
S=
 [noun_phrase(pl,np(n([domain(mod),adj(_5307,a([relevant])),
     range(mod),noun(_7902,n([circuit,breakers],pl))])))],
B= []

| ?- run(S,B).

|: electrical supply to ignition system.
parse took 0.016 sec.
memory (total)        368,108 bytes
    program space     238,044 bytes:  238,044 in use  0 free
    global space       63,444 bytes:   22,636 in use  40,808
                                                              free
       global stack                         22,340 bytes
       trail                                   296 bytes
    local stack       65,508 bytes:       3,160 in use       62,348
                                                              free
0.000  sec.  for  0  program,  0  global  and  ?  local  space
overflows.
0.000 sec. for 0 garbage collections which collected 0 bytes.
1.266 sec. runtime. 0 pagefaults
S=

[noun_phrase(sg,np(np(n([domain(mod),adj(_4268,a([electrica
l])),
     range(mod),noun(_4557,n([supply],sg))])),
     pp(p(to),np(n([domain(for),noun(_6606,n([ignition],sg)),
     range(for),noun(_6917,n([system],sg))]))))))],
B= []

| ?- run(S,B).

|: the nine bolt locations.
parse took 0.000 sec.
memory (total)       368,964 bytes
    program space    237,900 bytes:  237,900 in use  0 free
    global space      63,444 bytes:   20,776 in use  42,668
                                                          free
        global stack                  20,568 bytes
        trail                            208 bytes
    local stack       65,508 bytes:    2,408 in use     63,100
                                                          free
0.000  sec.  for  0  program,  0  global  and  ?  local  space
overflows.
0.000 sec. for 0 garbage collections which collected 0 bytes.
0.966 sec. runtime. 0 pagefaults
S=
[noun_phrase(pl,np(d(the),num(nine),n([domain(obj),noun(_12
518,n([bolt],sg)),
    range(obj),noun(_12952,n([locations],pl))])))],

B=[]
```