Contents

Table of Figures

Introduction

I have always been interested in language since my high school days, when I learned French and German. Unfortunately I didn't continue my language studies as I progressed to A levels and my degree. Recently my interest in language has been fired again, with my increasing use of the Internet, I found speakers of a multitude of tongues, from Hebrew, Portuguese, Finnish, Norwegian, Icelandic, French and German to name a few.

I am fascinated by the number of languages spoken in the world and wish to understand them better. When this project was suggested by Craig Duffy I immediately looked at it as an opportunity to increase my knowledge and understanding in an area that interests me.

Aims

In undertaking this project I knew that I would be expected to Implement some form of NLP program and comment on what I had learned from this project. My Aims therefore are:

- Learn about linguistics in general
- Learn of Chomsky's theories and how they might be implemented on a computer
- learn about Natural Language Processing techniques and appropriate programming languages for use in Natural Language Processing

I will of course not be able to produce a complete Natural Language Processing system in the restricted time that I have available. I have decided to implement a few modules that could make up part of a larger Natural Language Processing system.

Background

Early work on Natural Language Processing assumed that the syntactic information in the sentence, along with the meaning of a finite set of words was sufficient to perform certain language tasks. In particular answering questions posed in English.

Generally, these early programs were limited to dialogues about restricted domains in simple English and ignored the problems faced when using unrestricted English. These programs bridge the gap between the early machine translation attempts of the 1950s and current semantic-based natural language systems. Some of these programs are shown below, showing their successes and some of their faults.

SAD-SAM

SAD-SAM stands for Syntactic Appraiser and Diagrammer - Semantic Analysing Machine. SAD-SAM was programmed by Robert Lindsay in 1963, it was written in IPL the list processing language.

The program accepts a vocabulary of Basic English (approximately 1,700 words) and follows a simple context-free grammar. The SAD module parses the input from left to right, builds a derivation tree structure and passes this structure on to SAM, which extracts the semantically relevant (kinship related) information in order to build the family tree and find the answers to the questions. Though the subset of English processed by SAD is quite impressive in extent and complexity of structure, only kinship relations are considered by

SAM; all other semantic information is ignored SAM does not depend on the order of the input for the building of the family tree. For example if SAM is first told that:

> Adele and Rachel are the daughters of Marvin and later that Laurie and Edra are the daughters of Eva.

Two different family units will be built, but they will be collapsed into one if we later find that Rachel and Edra are sisters (multiple marriages are illegal).

SAM cannot handle certain ambiguities. The sentence Joe plays in his Aunt Jane's yard could mean that Jane is either the sister, or sister-in-law of Joe's father but SAM can only assign one connection at a time and cannot use the ambiguous information: SAM permits storing definite links but not possible inferences.

SHRDLU

SHRDLU written by Terry Winograd, caused a tremendous stir in 1972, because of the fairly successful way it modelled human language, and is now seen as a classic example of a modular model of language understanding.

SHRDLU was an early attempt to introduce different types of knowledge into a computer program dealing with language. SHRDLU was modular because it proposed separate modules for syntax, semantics, and world knowledge. SHRDLU was also an interactive

model because it was possible for the modules to call up information from other modules when necessary to contribute to the final interpretation of a sentence.

All the input sentences were instructions for moving different types of blocks around in small, simplified world, in which the blocks resided. This world was displayed on the computer screen. The blocks could be moved around by a pointer, changing the position of the blocks on the screen.

The figure below shows the three modules incorporated into SHRDLU. The Syntax Module contains syntactic rules formulated as augmented transition networks. The Semantics Module contains a lexicon and semantic selection rules for building up semantic representations, and the Blocks Module contains general knowledge about the position of the blocks at any one time.

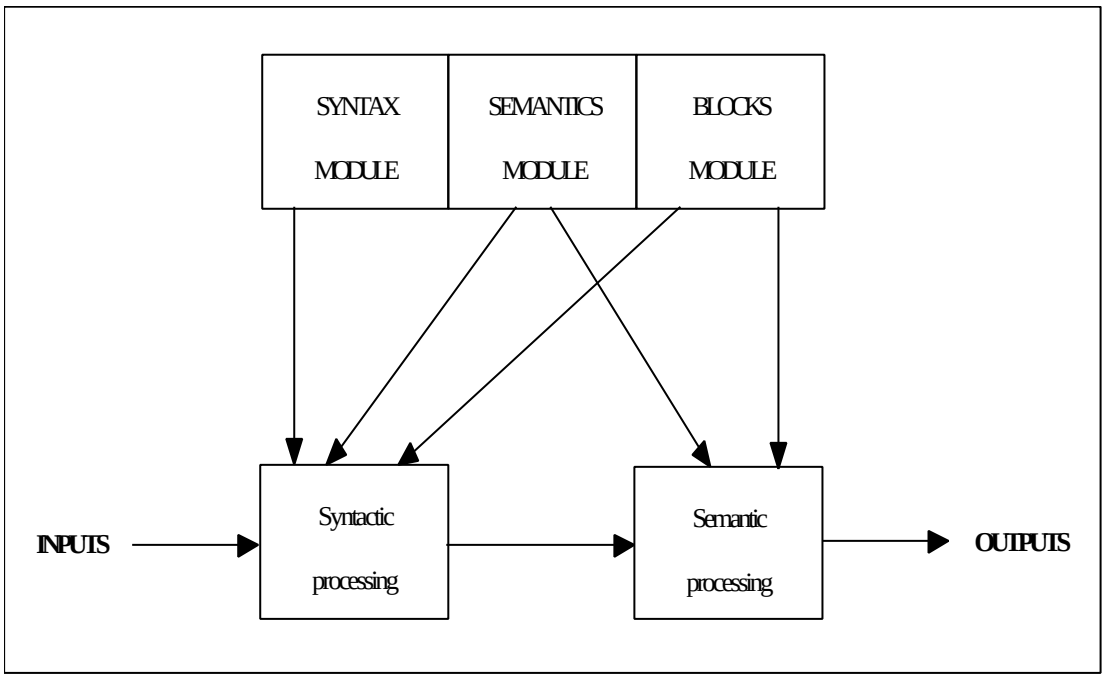

Figure 1 -- SHRDLU modules

Winograd argues that the modules must be able to pass information to each other when required. It is easy to see the need to consult different modules by considering what could happen when the computer received the command *Put the green pyramid on the block in the box*. There are two possible ways to group the noun phrases in the sentence, as indicated by the position of the brackets:

1. Put (the green pyramid) on (the block in the box), i.e. move the green pyramid, and place it on top of the block which is already in the box.
2. Put (the green pyramid on the block) in (the box), i.e. move the green pyramid currently on top of a block and place it on the floor of the box.

In order to deal with potentially ambiguous sentences like *Put the green pyramid on the block in the box*, the Syntax Module in SHRDLU can call on the Semantics Module and the Block Module to settle which of the two possible syntactic analyses is correct. For instance, the Semantics Module would confirm that the verb put, when combined with the names of two blocks can be constructed as a meaningful sentence taking the meaning of placing one block on top of another. At this point reference would be made to the Blocks Module. Using it's knowledge of the Block World, it would report that either interpretation is possible because a green pyramid is currently on top of a red block, and so could be moved from there, to the floor of the box; equally, the green pyramid could be moved from it's current position and place on top of the block in the box. SHRDLU would then ask the operator which was the intended meaning of the sentence.

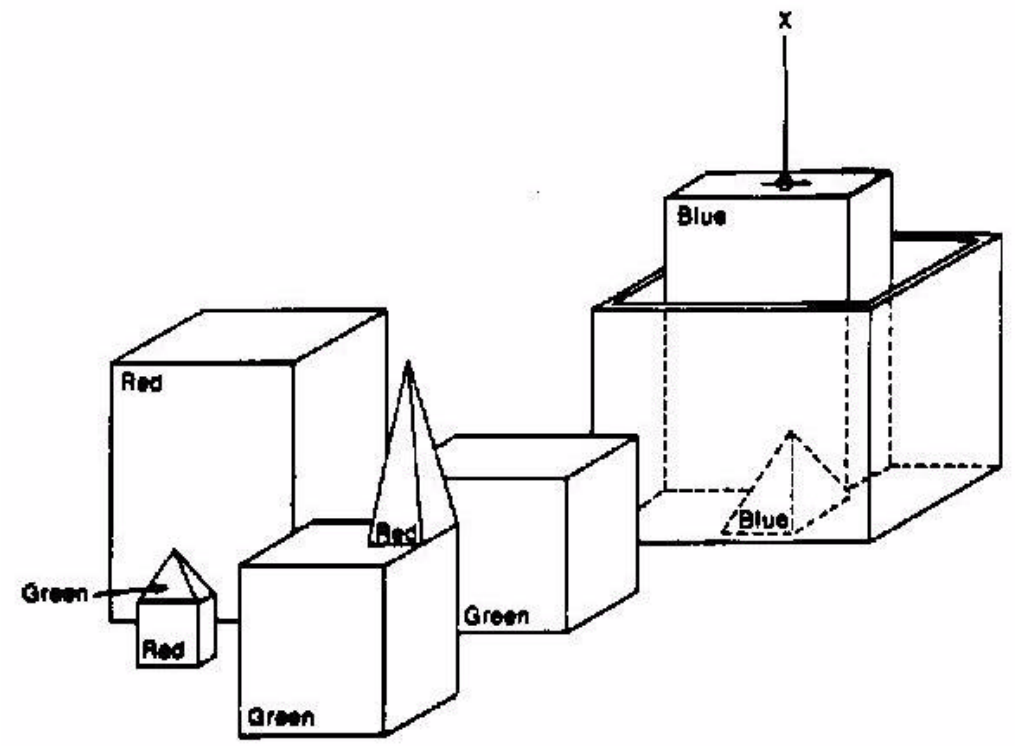

Figure 2 -- A Screen Shot from SHRDLU.

While the Syntax Module is attempting to build up a syntactic structure for a sentence, it can call in the Semantics and Blocks Modules in order to resolve potential ambiguities. Thus, rather than relying solely on the syntactic back up and look forward procedures, interaction with semantic processing and even with the Blocks Module, it is possible to short circuit semantic interpretations.

SHRDLU is a model in which all the modules are equally important and on the same level, compared with strictly hierarchical models. SHRDLU has some of the characteristics of independent processing modules and some of the characteristics of an interactive model. The most important characteristic of a heterarchical model is that processing can be interrupted in order to consult other processing modules. In later versions of Winograd's

theory there was an even greater emphasis on the need to understand how individual sentences relate to other sentences, that had been entered before, or perhaps might be entered after the current sentence.

Current Systems

There are many areas where computational linguistics has been implemented some in unlikely places, others in more traditional areas. On the following pages I hope to show some of the well-know and some of the less well-know NLP systems that are in development and use today.

Xerox Polyglot Photocopier

From next March Xerox intend to display a polyglot photocopier at their Grenoble labs. The photocopier draws from research into language engineering, including Information Extraction. Monica Beltrammetti of Xerox research was quoted as saying

> "You will be able to take a copy of a French newspaper and simply highlight or circle the words you don't understand. You will put it on the copier, press the 'translate' button, and it will return a copy with those words translated and indexed."[1]

Aventinus

Professor Yorick Wilks of Sheffield University is currently working with Europol on a project called Aventinus. Aventinus translates and processes police records and reports. The main aim of the project is for all European police forces to be able to search through each others records in search of criminals possible aliases, or previously committed crimes.

[1] The word crunchers. The Guardian On-line supplement November 27 1997 Page 3

Policespeak

Researchers at British Telecom are investigating a bilingual spoken communication system for use by the French and British police forces, involving research on the sub-language of police messages. The idea is that this system would convey messages about possible problems, or criminals to police forces in the other country. Particularly useful now, with the growing use of the channel tunnel.

Ntran

Ntran is an English-Japanese system which uses active disambiguation as a translation technique. It was developed at UMIST (University of Manchester Institute of Science and Technology). The particularly interesting thing the way in which active disambiguation is used by Ntran. Active disambiguation is used on the English source text. When further ambiguities arise during the lexical transfer between English and Japanese, the system asks the user to choose certain phrases, based on English paraphrases of the Japanese text. This means that a person with no knowledge of Japanese at all could quite easily translate English to Japanese.

Statistics-based Machine Translation at IBM

Research at IBM's labs in Yorktown heights, New York, has shown a modest amount of success, using statistical analysis of the text. Many systems use statistical data to guide and help in rule writing and the formulation of routines. Some systems however use the statistical data in lexical selection, and disambiguation. IBM's project however uses statistical data as the only tool for analysing the text and generating the output. The method involved matching

sentences in both English and French. The probabilities were estimated by matching bigrams (two consecutive words).

Two sets of probabilities were calculated. One for each individual English word and its corresponding two French words. For example *the* corresponds to the French *le* with a probability of .610, to *la* with .178, to *l'* with 0.83 to *les* with 0.23, to *ce* with 0.13, to *il* with 0.12 and so on. The other set of probabilities were based on whether two , one or zero French words correspond to a single word. For example *the* corresponds to one French word with a probability of .871, to zero French words with .124 and to two French words with a probability of .004. The translation system were tested using 73 new French sentences. A quite limited vocabulary of the 1000 most frequently used English words and the corresponding 1700 most frequently used French words. The results were classified as either: (a) exact ,(b) alternative (same meaning but in slightly different words), (c) different (legitimate translation but not conveying the same meaning), (d) wrong (intelligible result, but not the translation of the French), (e) ungrammatical (no sense conveyed).

Only 5% of the translations came into the 'exact' category; however, translations were considered reasonable if the fell into the first three categories (exact, alternate and different). When this criterion is used the system performs with a 48% success rate. Improvements are expected when a larger body of text is used for the initial probabilities, by probabilistic segmentation of sentences into phrases, by using trigrams as well as bigrams, and including data on inflectional morphology.

Statistical based techniques used in conjunction with linguistic methods will be a feature of many machine translation systems to come.

Theory

Chomskian Linguistics

In this century the most famous argument that language is an instinct comes from Noam Chomsky.

In the 1950s the social sciences were dominated by behaviourism, the school of thought popularised by John Watson and B. F. Skinner. Mental terms like “know” and “think” were thought to be unscientific, and language was thought to be explained by a few simple laws of stimulus-response learning.

Chomsky disagreed and argued his point using two fundamental facts about language. First, most sentences a person utters or understands are brand-new combinations of words appearing for the first time ever. Hence a language cannot simply be a list of responses to stimulus. The brain must have a method of building an infinite set of sentences from a finite set of words. This method of producing sentences could be said to be a mental grammar. For example:

> The man clutched at his umbrella and looked anxiously at his watch, and then outside at the darkening sky.

The above sentence is probably unfamiliar to you, it is likely you have never heard or read it before; however you are able to understand the sentence. If I asked the question Who, or what do you think the man clutching the umbrella is waiting for? (Another unfamiliar sentence, probably never written before) You could undoubtedly reply with a sensible response even though you had never seen these sentences before.

The second fundamental fact Chomsky argues is that children develop these complex grammars rapidly and without formal instruction, and grow up to give consistent interpretations to new sentence constructions that they have never seen before. Chomsky argued then, that children must innately be equipped with a plan common to all languages, a Universal Grammar. Chomsky is ultimately aiming to model this universal grammar in his theories.

Syntactic Structures

The Concept Of Structure

It is impossible to give a definition of syntactic structure without first constructing a theory of syntax. Many linguists have attempted to show that there is order in the way language speakers create sentences.

Chomsky has had a great influence on the theories about the role of syntax in sentence understanding. Chomsky's theories are intended to formalise the rules which constitute linguistic competence, that is the knowledge that enables language speakers to identify certain sequences of words as grammatical and others as ungrammatical. For example, what

makes a reader look at these two sequences of words differently, one is seen as grammatical, and one is seen as ungrammatical.

Colourless green ideas sleep furiously.

* Ideas green furiously colourless sleep.

One important point Chomsky made is that it is impossible to list all the sentences that could possibly be spoken in a particular language.

Before Chomsky started his work, psychologists had mainly concentrated on the processing of single words, simply because they had no method of representing larger structures like sentences and texts. Chomsky's demonstration that people are able to identify grammatical or ungrammatical sequences of words was interpreted as supporting evidence that human language includes the parsing of sentences into grammatical categories.

Chomsky's theories are particularly suited to modelling on computers because they are based on discrete mathematics, and hence are easy to implement as computer programs.

Generative Grammar

Chomsky's theory of grammar takes the form of rules for generating sentences. His use of the term generative has sometimes caused confusion, and some people hold the mistaken view that Chomsky is claiming that these are the actual rules that language speakers follow to produce sentences. A simple example of the rules in Chomsky's (1957) grammar is shown in the table on the following page.

1	S (sentence)	→	NP (noun phrase) + VP (verb phrase)
2	NP	→	N (noun)
3	NP	→	article + N
4	NP	→	adjective + N
5	NP	→	pronoun
6	VP	→	V (verb) + NP
7	VP	→	V + adjective
8	N	→	Jane, boy, girl, apples
9	V	→	likes, hit, was hit, was, are cooking, are
10	adjective	→	good, unfortunate, cooking
11	article	→	a, the
12	pronoun	→	he, she, they

These rules are also known as rewrite rules because they can be used to rewrite a sentence into it's component parts. Rule 1 states that the symbol for sentence S can be rewritten into symbols standing for noun phrase (NP) and verb phrase (VP). This can be interpreted as saying that English sentences consist of a noun phrase, and a verb phrase. Rules 2 - 5 say that a noun phrase can be rewritten as any of the following: a single noun, an article followed by a noun, an adjective followed by a noun, or a single pronoun. Rules 6 and 7 show that a verb phrase can be rewritten as: a verb followed by an noun phrase, or a verb followed by an adjective. Rules 8 - 12 allow the symbols to be rewritten as words. These rewrite rules can be used to produce syntactic trees which define the syntactic structure of sentences.

An example of how the rules can be used to generate a syntactic tree structure for a particular sentence is shown in figure 3 below.

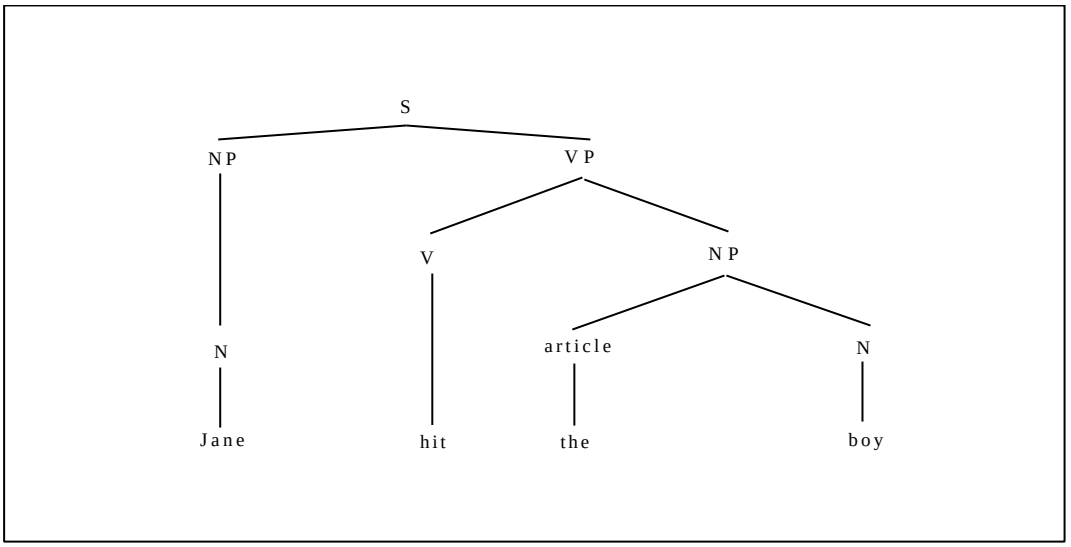

Figure 3 -- A syntactic tree structure

Syntactic tree structures show which of the rules have been used to generate a sentence. The rules are applied, and re applied, until all the symbols, e.g. NP, VP, N, V are rewritten as actual words. The syntactic tree for the phrase *Jane hit the boy* was generated using the rules 1, 2, 6, 3, 8, 9, 11, 8.

Syntactic trees are extremely useful when dealing with syntactically ambiguous structures. Syntactic trees can be used to specify different syntactic structures for sentences that are syntactically ambiguous.

The sentence They are cooking apples can be analysed in two different ways, one refers to a particular type of apple (cooking apples), and the other sentence refers to the fact that some people are cooking some apples. Figure 4 on the following page shows the two distinct syntactic trees for this syntactically ambiguous sentence.

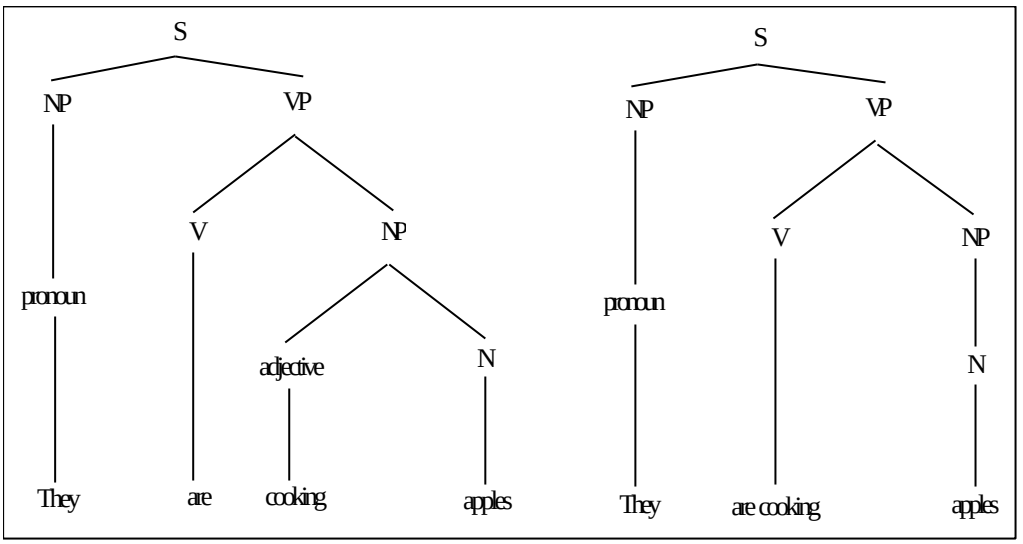

Figure 4 -- Syntactic ambiguity

Phrase structure can also be represented in box diagrams as in Figure 5 below:

NOUN PHRASE				
Article	Noun	Prepositional Phrase		
		Preposition	Noun Phrase	
			Article	Noun
the	people	in	the	room

Figure 5 --Phrase structure box diagram

Embedding

It is interesting to note that often grammar rules will allow the embedding of one rule within another. This means for instance that a noun phrase might consist of an article, a noun and a prepositional phrase. A prepositional phrase consists of a preposition, and may itself contain

a noun phrase. It is therefore possible to build an infinitely long NP as well as an infinite number of distinct NPs.

Transformational Rules

Chomsky (1957) proposed that the simplest way to generate more complex sentences like passives is to use transformational rules for rearranging the order of words in a sentence. For example an active sentence like *Jane hit the boy* would be generated directly by rewriting rules to generate the syntactic tree as show in Figure 3. This tree would then be transformed by reordering the words in order to produce the passive *The boy was hit by Jane*. The syntactic tree is shown in the figure 6 below.

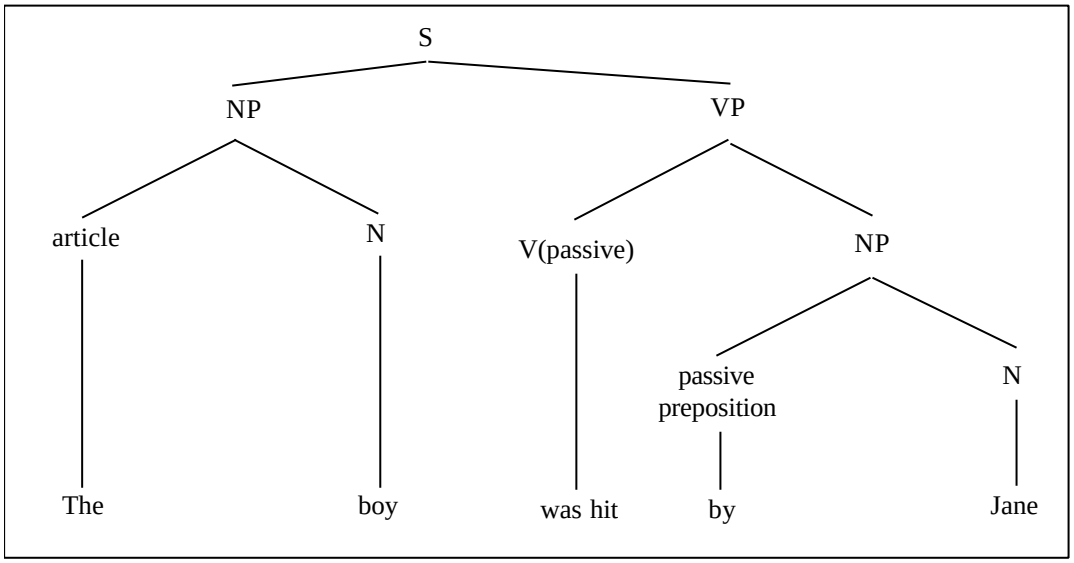

Figure 6 -- Active to passive transformation tree

A passive transformation rule might state that, when a sentence is transformed, the passive form of the verb (*was hit*) and the preposition by should be selected to produce the transformed sentence (*The boy was hit by Jane*).

Chomsky's transformational rules are given as operations on a tree structure. I will show below how a rule would be created using an example of particle movement. Consider the following sentences:

She will stand up her date.

She will stand her date up.

Particle Movement

SD:	X	-	Verb	-	Particle	-	NP	-	Y
	1		2		3		4		5
SC:	1		2		0		4 + 3		5

A formal transformational rule consists of an input: a structural description (SD), which is an instruction to analyse a phrase marker into a sequence of constituents (in this case a verb followed by a particle followed by a noun phrase). The *X* and *Y* variables indicate that anything preceding the verb and anything coming after the noun phrase are irrelevant to this transformation. In order for a transformation to be applied the analysis of a phrase marker must satisfy the SD of the particular transformation. Figure 7 on the following page shows how this structure tree can be analysed in a way that matches the SD of the particle movement transformation.

The second part of the transformational rule is the output or structural change (SC), which in the case of particle movement is an instruction to modify the SD by shifting term 3 (the particle) immediately to the right of term 4 (NP). The '+' sign indicates that these two constituents are to be attached under the same node (VP). The symbol '0' indicates that nothing remains in the slot where the particle had been.

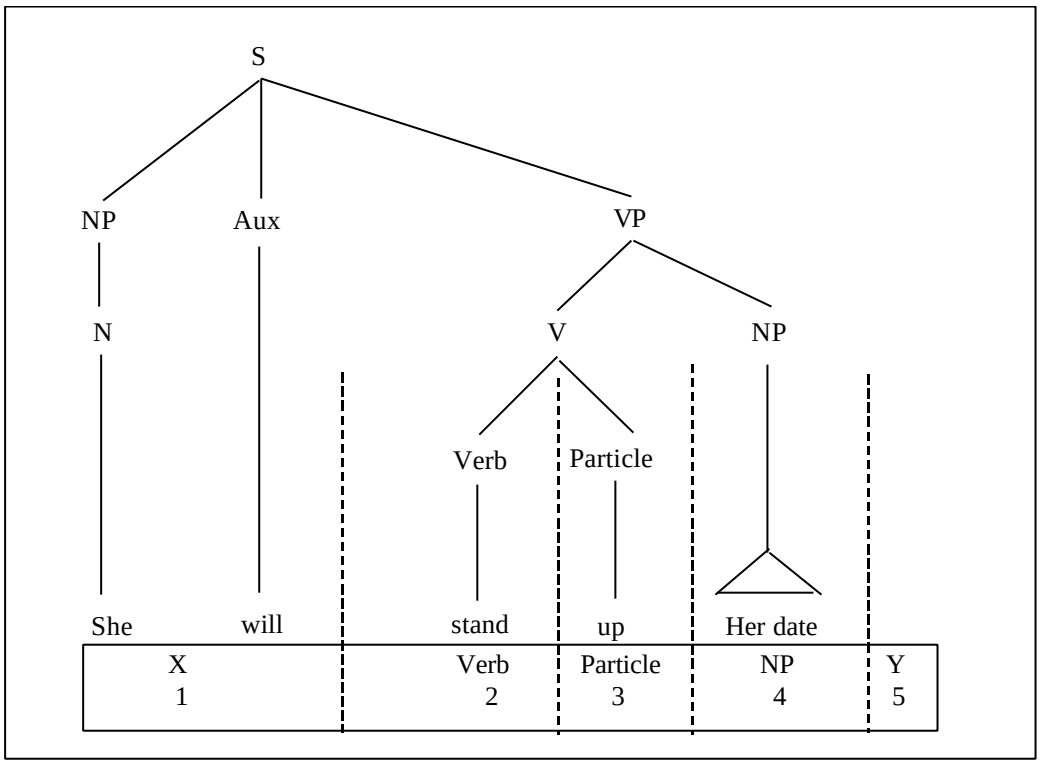

Figure 7 -- Transformational Tree (Particle Movement)

Auxiliary Verbs

Auxiliary Verbs in English include the following forms

a) Forms of the verb *be* (is, am, are, was, were)

b) Forms of the verb *have* (have, has, had)

c) Forms of the verb *do* (do, does, did)

The verbs *can*, *could*, *will*, *would*, *shall*, *should*, *may*, *might*, *must*, and a few other modal verbs. Verbs that are included in this group are called modal auxiliaries. Modals are "helping verbs" that usually refer to concepts such as possibility, necessity, and obligation.

Throughout this section I will use the standard method of showing ungrammatical sentences, an asterisk(*).

The difference between auxiliary verbs and main verbs is clear in several grammatical processes in English, among which are the following:

1. Auxiliary verbs, are fronted in forming questions, main verbs are not.

a) John *is* running.	*Is* John running?
b) They *have* left.	*Have* they left?
c) I *can* sing.	*Can* I sing?
d) Mary speaks French.	*Speaks Mary French?

When a sentence doesn't contain an auxiliary verb but only has a main verb, the auxiliary verb *do* is used to form questions.

a) You know those women. Do you know those women?

b) Mary left early. Did Mary leave early?

c) They went to Phoenix. Did they go to Phoenix?

2. The contracted negative form n't can be attached to auxiliary verbs:

a) John *is* running. John *isn't* running.

b) They *have* left. They *haven't* left.

c) I *can* sing. I *can't* sing.

Main verbs cannot be negated in this way, instead the auxiliary verb *do* is used to form the negative.

a) You know those women. *You known't those women.

b) Mary left early. *Mary leftn't early.

c) You know those women. You *don't* know those women.

d) Mary left early. Mary *didn't* leave early.

Auxiliary verbs may also be followed by the uncontracted form of the negative not. Main verbs cannot be followed by the uncontracted negative not, in current spoken international

English: when it is used in sentences such as "They know not what they do" and "Ask not what your country can do for you" it is used in stylised versions of English, in which an archaic flavour has been preserved (as might be seen in religious preaching, or formal public speaking).

3. Auxiliary verbs can appear in tags, but main verbs cannot. A tag occurs at the end of a sentence and contains a repetition of the Auxiliary verb found earlier in the sentence. Tags are usually used to add emphasis, or to request confirmation of the information given in the preceding sentence.

    a) Herman *is* threatening to leave, *is* he!

    b) Herman *is* threatening to leave, *isn'*t he?

If the auxiliary verb of the main sentence is positive in it's positive form, the auxiliary verb in the tag may be either in positive or negative form. If the auxiliary verb in the main sentence is in it's negative form, then the auxiliary verb in the tag will always be in the positive form:

    a) Herman *isn't* threatening to leave, *is* he?

    b) *Herman *isn't* threatening to leave, *isn't* he?

Where a sentence contains no auxiliary verb, and only has a main verb, it is possible to tag the sentence by using the auxiliary verb *do*.

    a) *You know those women, know you?

    b) You know those women, do you!

c) You know those women, don't you?

What happens if there is more than one auxiliary verb in a sentence?

a) John will have left.

b) *Have John will left?

c) Galen has been studying very hard.

d) *Been Galen has studying very hard?

From these examples it looks as if the auxiliary verb that fronts to form the question should be the first auxiliary verb reached; forming sentences like:

a) Will John have left?

b) Has Galen been studying very hard?

It is however not as simple as this. Considering the following examples it is clear that it is not always the first auxiliary verb that fronts, in fact the verb that fronts doesn't correspond to any number.

The auxiliary verb used to front the sentence could be the first, second, third, forth or indeed any number.

a) The people who are standing in the room will leave soon.

b) *Are the people who standing in the room will leave soon?

c) The people who were saying that John is sick will leave soon.

d) *Were the people saying that John is sick will leave soon?

e) *Is the people who were saying that John sick will leave soon?

f) The people who were saying that Pat has told Mary to make Terry stop trying to persuade Dave that mathematicians are thought to be odd will leave soon.

g) *Are the people who were saying that Pat has told Mary to make Terry stop trying to persuade Dave that mathematicians thought to be odd will leave soon.

In each case the auxiliary verb *will* is the correct verb to use to front the sentence.

The auxiliary verb that should be moved to produce the question is the auxiliary verb that immediately follows an intuitively natural grouping of words referred to as the subject of the sentence.

The subject in the following sentences is underlined:

a) John will have left.

b) The people who are standing in the room will leave soon.

c) The people who were saying that Pat has told Mary to make Terry stop trying to persuade Dave that mathematicians are thought to be odd will leave soon.

d) Yesterday, John could lift five hundred pounds.

e) Around this time last year I was ill.

The last two sentences suggest that the appropriate auxiliary verb of the sentence should be placed immediately to the left of the subject, and not merely at the beginning of the sentence.

We now have a rule for translating a declarative sentence into a questioning sentence:

locate the first auxiliary verb that follows the subject of the sentence and place it immediately to the left of the subject.

Finding the Subject of a declarative sentence

Subjects not only play a role in transformation rules, they play an important part in other grammatical processes. It is unfortunate then, that the notion of a subject has never been precisely defined, despite its significant role in linguistic analysis. Finding the subject of a sentence is generally left to intuition, something that is very hard to give a computer program.

The classic example of a subject comes from simple sentences with action verbs such as *the farmer fed the piglet*, in which the subject, in this case the farmer, is understood as the agent ("the doer") of the action, and the object, in this case the piglet, is understood as that which undergoes the action. Not every subject is an agent; in the sentence *Mary resembles her Aunt Bettina*, Mary is the subject, but no action is involved. Trying to characterise subject in terms of meaning is a very difficult and complex task, if it is possible at all.

Tests for finding the subject

Seeing as we cannot say for definite if any sequence of words is the subject of a sentence or not we will have to rely on a few tests that will tell us if a sequence of words is the likely subject of a sentence.

a) The subject of a declarative sentence generally precedes the auxiliary and main verb in linear order.
b) It forms the constituent around which an auxiliary is fronted in forming a question.
c) It is the constituent with which a pronoun in a tag agrees in terms of person, number and gender.

Using these test we can be fairly certain that we have the subject of a sentence.

Grammatical Relations

In English the subject of a sentence can structurally be defined as the particular NP in the structural configuration immediately dominated by S and preceding Aux VP as shown in the figure 8 below:

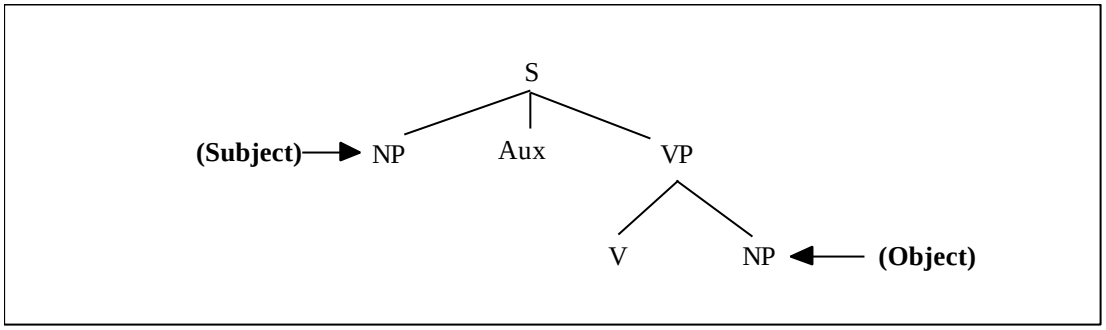

Figure 8 -- Structural definition of subject and object

The object of a main verb can be structurally defined as the NP in the structural configuration that is immediately dominated by VP.

X-bar Theory

The phrase structure of a sentence is a hierarchy, with each constituent successively consisting of other constituents, until only non-expandable items are left . The 'consists of' relationship can be expressed as re-write rules (A ⑤ B C). An item that comes above another item in the tree and is not on a separate branch is said to dominate it.

With Principles & Parameters theory phrase structure is a comparatively simple system derived from a few principles and the setting of certain parameters. X-bar syntax replaces

large numbers of individual re-write rules with general principles; it captures properties of all phrases, not just those of a certain type; it bases syntax on lexical categories.

X-bar claims every phrase conforms to certain requirements. Particularly that phrases must be endocentric, that is a phrase always contains at least a head as well as other possible constituents. For example a NP such as *the bird* has a head *bird*; a VP *sees the cat* has a head *sees*. The essentials of X-bar syntax is that the head of the phrases belong to a particular lexical category related to the type of phrase. Consider the re-write rules below

$NP \rightarrow N$

$VP \rightarrow V$

X-bar can replace these two re-write rules with one:

$XP \rightarrow X$

In X-bar there are four lexical phrases:

- VP (verb phrase)
- NP (noun phrase)
- AP (Adjective phrase)
- PP (prepositional phrase)

X-bar claims the phrase level, i.e. XP where X stands for any of the categories are not sufficient to capture all the details of the phrase structure: an intermediate level is needed. Consider the following sentence:

The education minister will resign her post on Tuesday.

The VP of this sentence will require a tree as shown in figure 9:

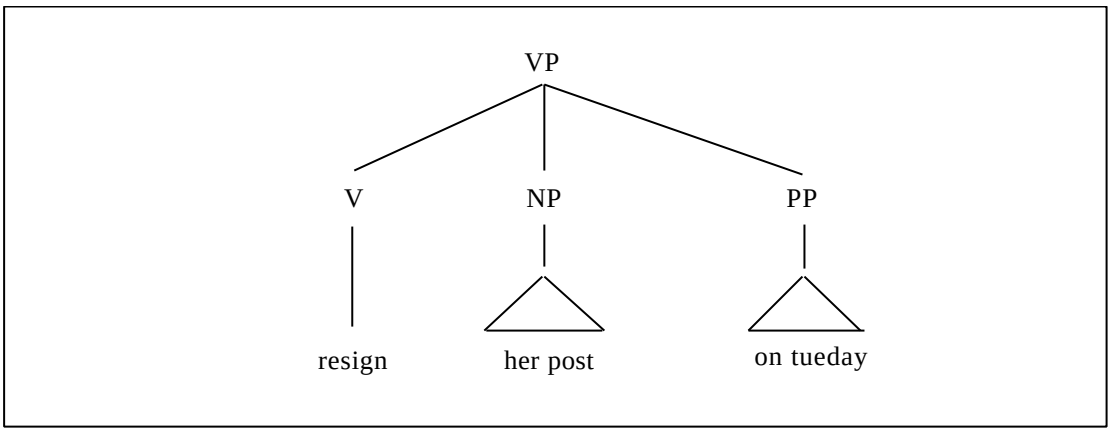

Figure 9 --Three branched verb phrase

This atypically requires three branches rather than the more usual two. This tree fails to distinguish the relationship between V and the NP, and the relationship between V and the PP. It would be convenient if the structure of the VP distinguished optional elements from those that are compulsory. Defining the compulsory elements as V' gives the tree shown in figure 10 on the following page.

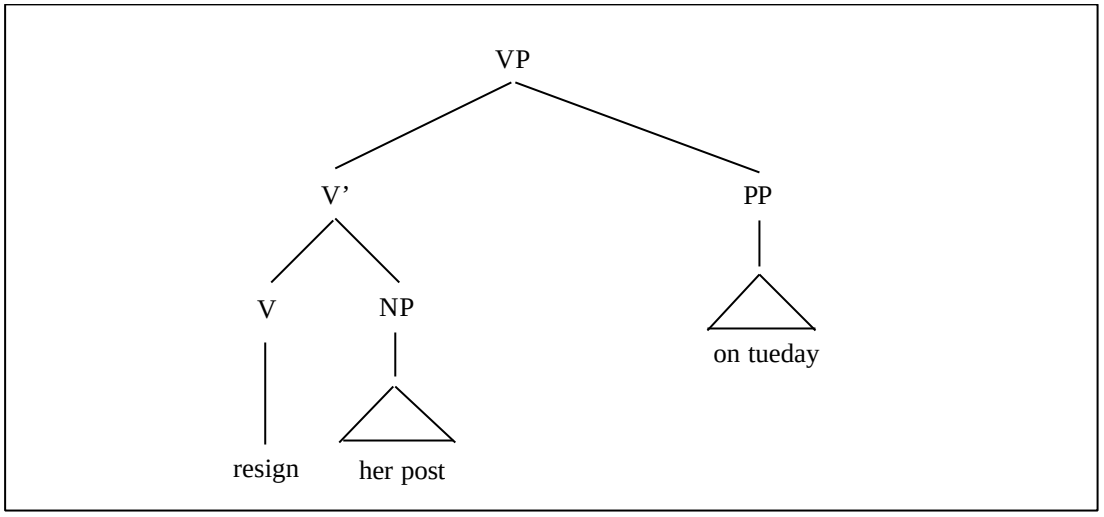

Figure 10 --X-bar verb phrase

The VP now has levels of structure and the PP can now be detached from the head V. The close relationship between V and object NP can be expressed through sisterhood in the V' phrase. X-bar allows all other phrases to be solved in the same way; they all have an intermediate level.

q ***theory (Theta theory)***

Theta theory is concerned with who is doing what to whom (theta roles or thematic roles).These form a crucial part of the syntactic meaning of the sentence. A sentence such as:

Alfa gave Al a compact-disc

has three thematic roles: Alfa refers to the person who is carrying out the action (the Agent role), the compact-disc to the object affected by it(the Patient), Al to the person who receives it (the Goal).

Government Binding

Government binding has been continuously developed and refined, through repeatedly applying it to a wide range of natural languages. The languages have ranged from languages traditionally associated with natural language processing, such as English and French, to less traditional languages, like German, Portuguese, Japanese, and even less well known languages such as Warlpiri (a free-word-order language of Australia). Through applying government binding to such a wide range of languages, it has been possible to move from the view of grammar as a set of rules towards the view that grammar is a set of interacting well-formedness principles, and parameters. This view came from the realisation that languages are very different at surface level, and require very different sets of rules for their respective grammars. These different grammars, are generated from common principles, and a few language dependant parameters. Government binding concentrates on these common principles. The use of government binding enables the capture of the attributes of all languages, in an elegant, and concise manner. It also provides a greater explanatory depth than is possible with rule based phrase structure grammar.

Deep Structure

Later versions of Chomsky's theory (1965) state the idea that each sentence has a surface structure and a deep structure. The surface structure is represented by the actual order of the words in a sentence. The deep structure represents the basic grammatical relationships from which the surface structure is derived. For example, the deep structure *Jane hit the boy* can be used to derive the surface structure *The boy was hit by Jane*.

Chomsky's theory also suggests that deep structures contain all the syntactic information necessary for interpreting the meanings of sentences; however, surface structures are necessary for representing the words of a sentence in the correct order.

A complete Chomskian grammar must specify transformational rules for mapping surface structures onto deep structures and deep structures onto surfaces structures.

The figure below shows the relationship between deep structures and surface structures in Chomsky's theory.

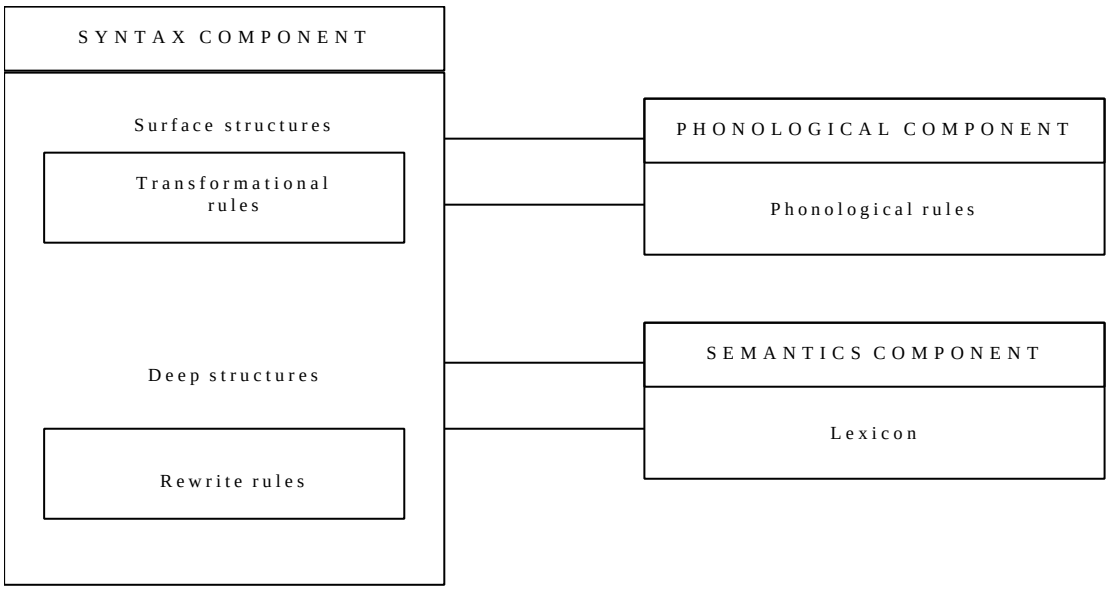

Figure 11 --Chomsky's (1965) theory of language

Lexicon

A lexicon is essential for the understanding of a language. A lexicon will list all the words in a language, each with its form, its meaning, and its lexical category (i.e. article, noun, verb, preposition, etc.). According to Miller and Gildea (1987, 94) Humans will have built a lexicon of approximately 80,000 words by the time they are 17. It is natural to think of individual words as the building blocks of sentences. In the standard theory (as summarised in Chomsky 1971) sentence generation begins from a context-free grammar generating a sentence structure and is followed by a selection of words for the structure from the lexicon. The context-free grammar and lexicon are

said to be the base of grammar.

Meaning properties

At the very minimum a lexicon must specify the meaningful words of the language, and must represent the meaning of these words (both simple and complex) in some way. For example an English lexicon must tell us that *procrastinate* means to defer action, *bachelor* means unmarried adult male, *mother* means female parent, and so on for all the words in the English language.

Lexical ambiguity

The following are examples of lexical ambiguity:

a) He found a *bat*.

(bat: bat and ball; flying rodent)

b) She couldn't *bear* children.

(bear: give birth to; put up with)

The italicised word in each of the above sentences is ambiguous because it has more than one meaning. The ability to detect ambiguity is crucial to language understanding and communicating the correct meaning depends on the speaker and the listener recognising the same meaning for a potentially ambiguous word.

The major difficulty in modelling the human lexicon is how to define the meaning of words. There are three main methods, of modelling meaning in a lexicon, in use today.

The Encyclopaedic view

In this method the meaning would contain a complete encyclopaedic entry. This means a complete list of all possible meanings would be attached to each word. For example the word cat might have its meaning defined as :

cat: cute, fluffy, animal, pet, ... ,etc.

A lexicon in a specialised domain of zoo keeping might define cat as:

cat : big, deadly, stealthy, swift, man-eater,...,etc.

The meaning of words obviously depends on the domain.

The Componential view

The componential view would only contain a few key words related to the word being described. For example the word student might be described with the key words

student: human, in education.

The advantage of this method is that the meaning should not change depending on the domain; however, there are difficulties with this approach. It is hard to find words that will exactly define the meaning of a word, another difficulty arises because each word used to describe a word must itself be described and this will use a lot of resources.

The Nominal view

The nominal view of a lexicon contains no meaning , just the lexical category and similar information of each word.

Parsing Issues and techniques

Parsing is the term used to talk about syntactic analysis. The word parsing is derived from the Latin phrase *pars orationis* (part of speech) and refers to the process of assigning a part of speech (Noun. Pronoun, Adjective, and so on) to each word in a sentence, and grouping words into phrases. A parser will output an analysis of the structure of the text it is given.

There are many theories on parsing, one way of displaying the results of a syntactic analysis is with a parse tree. A simple example is shown below.

Sentence to parse : The man likes pizza.

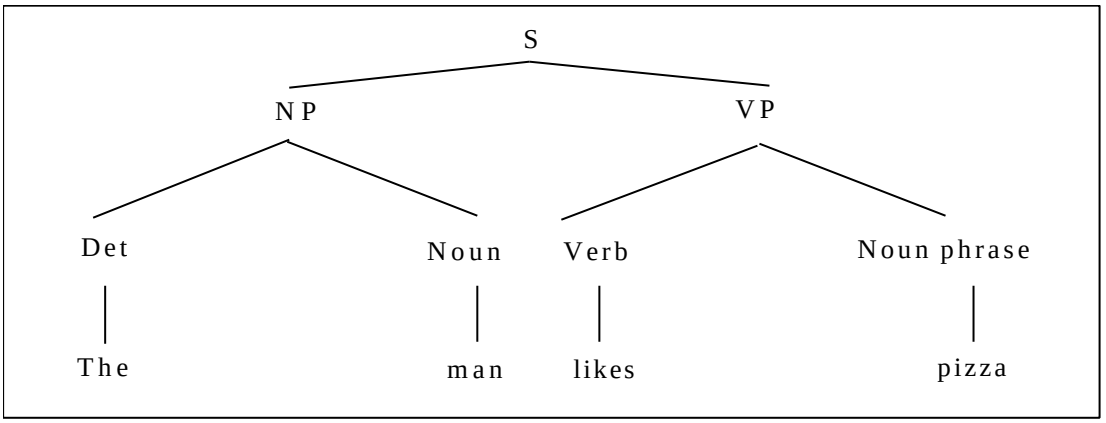

Figure 12 -- A parse tree

A Parse tree is a tree in which interior nodes represent phrases, links represent applications of grammatical rules, and leaf nodes represent words.

The best way to understand how a parser works is to trace the parsing of a sentence generated by a simple grammar (see the section of Generative Grammar).

S → NP VP	A sentence can consist of a noun phrase and a verb phrase.
NP → (det) N (PP)	A noun phrase can consist of an optional determiner, a noun, and an option prepositional phrase.
VP → V NP (PP)	A verb phrase can consist of a verb, a noun phrase, and an optional prepositional phrase.
PP → P NP	A prepositional phrase can consist of a preposition and a noun phrase.
N → man, boy, dog, pizza...	The nouns in the dictionary include man, boy...
V → eats, likes, bites...	The verbs in the dictionary include eats, likes...
P → with, in, near	The prepositions in the dictionary include with, in , near.
Det → a, the, one	The determiners include a, the, one.

We will use the same example sentence as before, and will show how the parse tree can be generated.

"*The man likes pizza*"

Moving from left to right, the first word we come across is *the*. The parser checks the dictionary, and discovers what category of word it belongs to, that is determiners. This could be seen, as checking the right-hand side of the rules, and discovering the category in

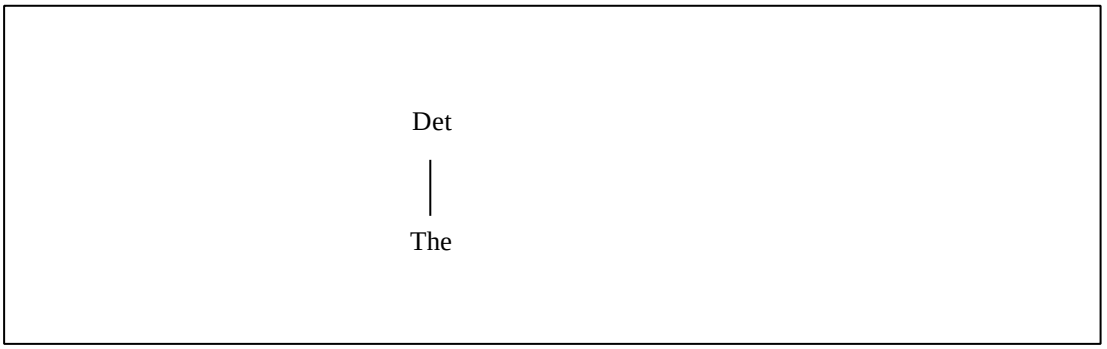

Figure 13 -- A parse tree (determiner)

the left-hand side. Now we can start to generate the parse tree.

Determiners, as all words, have to be part of some larger phrase. The parser can ascertain which phrase, by checking which rule has Det on it's right-hand side. The parser will discover that the only rule with Det on it's right-hand side is that for a noun phrase; therefore, more of the parse tree can be generated. This sub-tree must be held in memory.

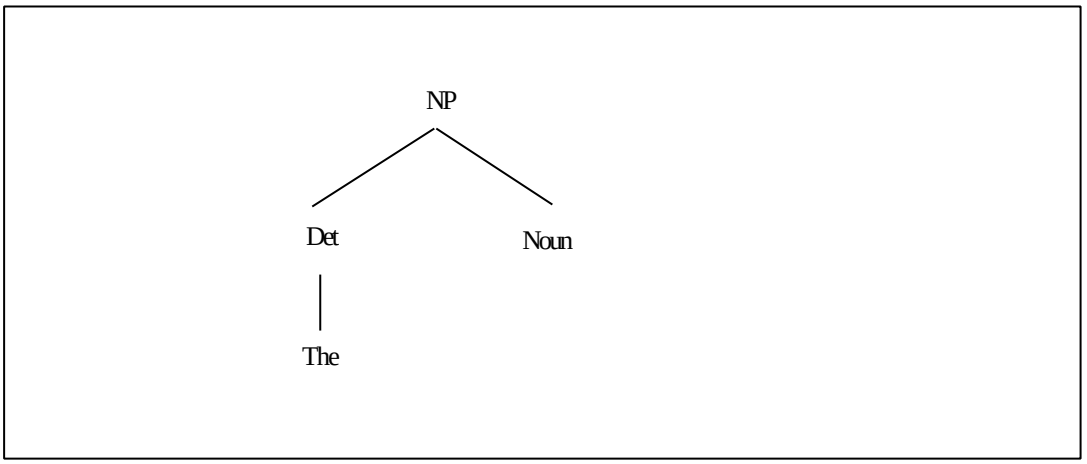

Figure 14 -- A parse tree (noun phrase)

The parser must also remember the current word, *the*, is part of a noun phrase. The noun phrase must be completed by filling in the rest of the words in the noun phrase sub-tree. Following the simple grammar specified earlier means this noun phrase must contain at least a noun, and could possibly have a prepositional phrase. While we are waiting for an opportunity to fill in the rest of the noun phrase the parse tree can continue to grow. Just as every word must belong to a larger phrase each noun phrase must also belong to some other structure. After the parser has checked the right hand side of the grammar rules, we find that a noun phrase has a few structures that it could belong to: a sentence, a verb phrase, or a prepositional phrase. In this case, we can decide, by using a root down approach. As all words, and phrases must eventually be part of a sentence, and a sentence must begin with a noun phrase.

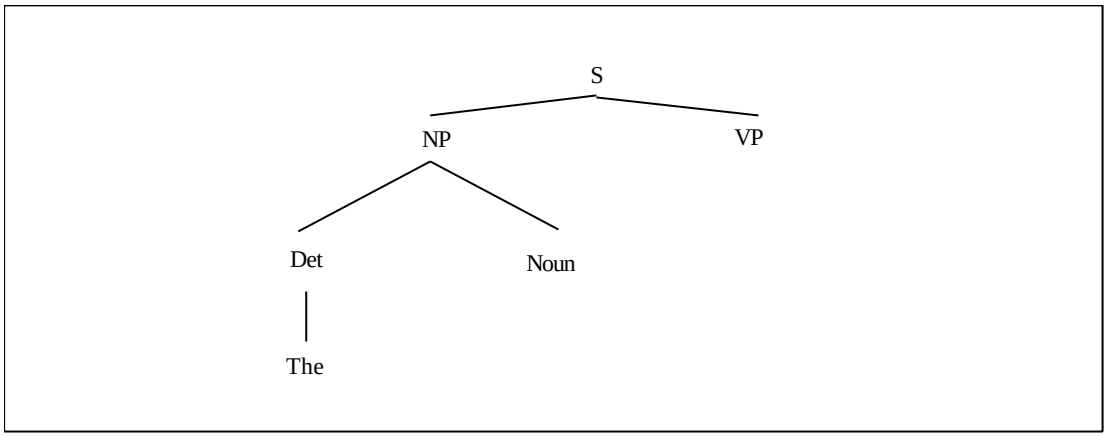

Figure 15 -- A parse tree (pre-empted verb phrase)

It is logical to expand the parse tree using the sentence rule. The tree expands like this. It is worth noting that the parser is now holding two incomplete branches of the parse tree in memory. The noun phrase, which is waiting for a noun, in order to be completed.. The other branch held in memory is the sentence its self, which needs a verb phrase to complete it. We can now go back to the sentence, "*The man likes pizza*." and check the second word "*man*" against the grammar rules. *Man* is part of the N rule, an so we can integrate *man* into the parse tree, completing the noun phrase.

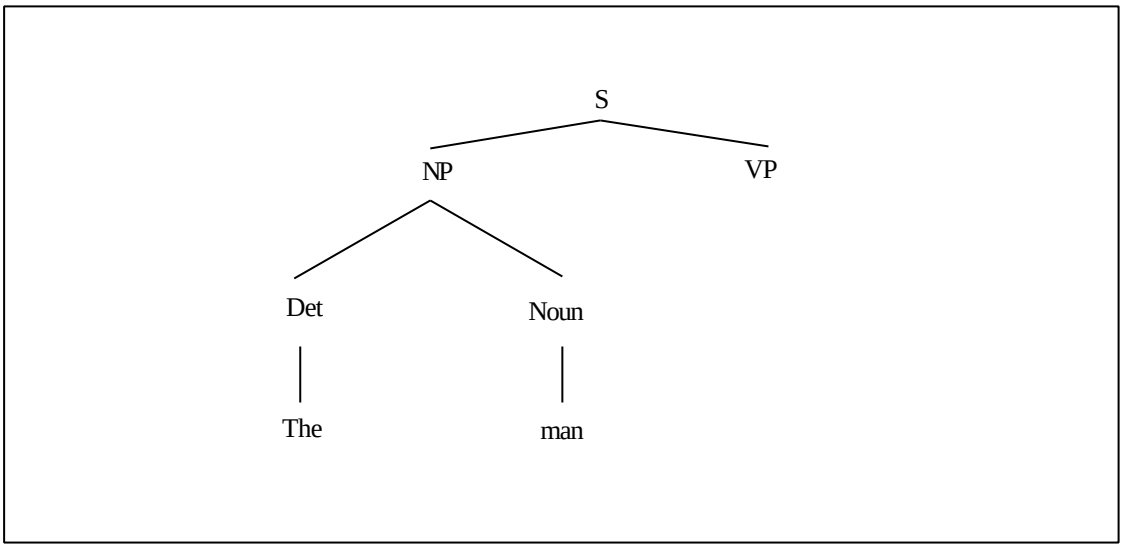

Figure 16 -- A parse tree (sentence)

The parser now no longer needs to remember that there is a noun phrase to be completed; all it need deal with is the incomplete sentence. The next word we get is *likes*, which we find to be a verb by looking at the grammar rules. A verb, as defined in our simple grammar is a part of a verb phrase, and cannot originate from anywhere else. The branch for the verb phrase has already been generated, so the verb, and the verb phrase nodes can simply be linked together. The verb phrase contains more than

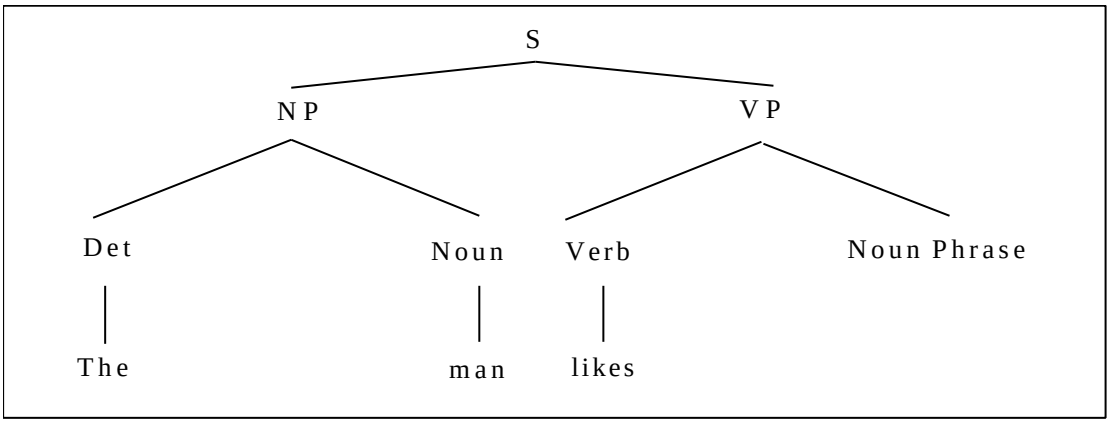

Figure 17 -- A parse tree (incomplete noun phrase)

just a verb it also has an object, in the form of a noun phrase. The parser pre-empts the need for the noun phrase, and so expands the parse tree automatically.

We are now expecting a noun phrase, in order to complete the verb phrase. The next word we get is *pizza*, which is a noun. The word *pizza* has completed the noun phrase, the noun phrase completed the verb phrase, which in turn completed the sentence. So now we have the parse-tree that we originally started with in Figure 12.

The parsing method we have just stepped through is using a bottom-up method. In a bottom-up parse, the parsing starts with the individual words, and attempts to build upwards. As we have seen, it takes a word, and finds the rule that fits. Another method of parsing uses the top-down method. In the top-down method, the parser starts at the abstract level of the sentence, and attempts to generate the structure of the tree, by working down to the words themselves. A top-down parse would start with node S, which by referencing the grammar rules, the parser knows must consist of a noun phrase, and a verb phrase. Again through referencing the grammar the parser knows a noun phrase must consist of a possible determinant, a noun, and a possible prepositional phrase. The parser will work its way down like this until a syntactically correct parsing has been achieved, or alternatively the sentence has been found not to be syntactically correct.

The choice of method, either top-down or bottom-up becomes important in complex grammars where the is a lot of scope for recursion. If we add more rules to the simple grammar we started with, we will be able to see how it affects the effectiveness of each of the two methods.

# S → NP VP PP	
S → NP VP	A sentence can consist of a noun phrase and a verb phrase.
# NP → det n	A noun phrase may consist of a determinant, and a noun.
# NP → det adj n	A noun phrase may consist of a determinant, an adjective, and a noun
# NP → det n PP	A noun phrase may consist of a determinant, a noun, and a prepositional phrase.
# NP → adj n PP	A noun phrase may consist of an adjective, a noun, and a prepositional phrase.
# NP → Pron	A noun phrase may consist of a single pronoun.
# VP → V NP	A verb phrase can consist of a verb and a noun phrase.
PP → P NP	A prepositional phrase can consist of a preposition and a noun phrase.
N → man, boy, dog, pizza...	The nouns in the dictionary include man, boy...
V → eats, likes, bites...	The verbs in the dictionary include eats, likes...
# adj → slimy, nasty, tasty..	The adjectives in the dictionary include slimy, nasty, tasty....
# Pron → I, he, she, it....	The Pronouns in the dictionary include I, he, she, it....
P → with, in, near	The prepositions in the dictionary include with, in , near.
Det → a, the, one	The determiners include a, the, one.

# Indicates a new rule, or a modified rule.

Note that where in the previous grammar there was one definition of a sentence, there is now two, and where there was only one definition of a noun phrase there are now five.

It is easy to see that when parsing with the top down method, there is immediately a choice between the two types of sentence. You have a 50/50 chance of getting it wrong before you even start processing! Each of the sentence types start with a noun phrase, of which there are now five types. You now have ten ways the sentence can be analysed, before you have even looked up anything in the word lists.

Backtracking

Backtracking is the term used when a parser has to revise decisions that it has previously made. It is easy to see the need for backtracking. When using a top-down method, as stated before, even with fairly simple grammars you will have a chance of getting it wrong before you even start. Backtracking is equally important when using the bottom-up method, to show this, consider a sentence that is structurally ambiguous. Structural ambiguity means that according to the rules of grammar, the sentence can be interpreted two or more different ways. Consider the following sentence:

He pointed at the girl with the stick.

In one case the NP consists of just *the girl,*

i.e. NP → det n

and the VP is *pointed at the girl.*

i.e. VP → v NP

The parser concludes that the PP is part of a sentence of the type NP VP PP

i.e. S → NP VP PP

In the other case the PP is taken to be part of the NP (*the girl with the stick*).

i.e. VP → v NP

NP → det n PP

The parser concludes it is a sentence of type NP VP

i.e. S → NP VP

If the rules are taken in order the first sentence identified will be the NP VP PP type. To get both possible sentence structures the parser will have to backtrack to the point where the rule,

NP → det n

was applied, and test if any other rules might be applicable.

Depth-first

There are two ways of making sure you have all the possible sentence structures. Namely to perform either a depth-first search or a breadth-first search using the grammar rules as the search space. Depth-first, where each sentence structure is followed until it is found to be a syntactically correct sentence or is proved to be a false trail. The parser will then return to the last choice it made, and consider all the alternatives, until every possible sentence

structure has been tried. Obviously parse paths that result in syntactically correct sentences must be remembered and retrieved at the end. The problem here is that it could prove slow to find all the possible sentence structures.

Breadth-first

Breadth-first parsing explores all possible sentence structures in parallel, so that both true and false trails are kept alive until the last moment. Obviously if this is done, there is no need to 'backtrack'. This method is more memory hungry, as each path through the grammar must be held in memory at the same time, until either the path has been proved false, or all possible sentence structures have been found.

Charts

Chart parsers are one way in which to minimise the amount of re-analysing that must be done when backtracking. Using the example sentences below, we are able to see that they have very different parses.

> Have the students in year four of Computer Science take the exam.
>
> Have the students in year four of Computer Science taken the exam?

If the parser is to be efficient, it must avoid reanalysing
"the students in year four of computer science" , as a noun phrase each time it backtracks. Once we discover that "the students in year four of computer science" is a noun phrase, it is possible to store and record that data in a chart. Because we are dealing with context free

grammars, any phrase that was found in the context of one branch of the search space, is equally valid in another branch. A deeper discussion of the use of charts in natural language processing is given in “An Introduction to Machine Translation” (Hutchins & Somers)

Implementation

Choosing a Programming Language

There are two basic types of programming language.

Imperative.

Tell the computer how to achieve a goal, by giving a sequence of instructions.

Examples: FORTRAN, Pascal, BASIC, COBOL.

Although NLP is possible using Imperative languages it lends itself more to declarative languages designed with symbol manipulation in mind.

Declarative.

Tell the computer what goal to achieve, by giving a specification.

Examples: Prolog, OBJ3, Lisp.

Prolog and Lisp are not entirely declarative - they have imperative features too, e.g. for

input and output.

The first and most fundamental Idea in AI programming languages was the use of the computer to manipulate arbitrary symbols, symbols that could stand for anything, not just numbers. List processing techniques came as a product of symbol manipulation and were first introduced in the IPL language.

IPL was created by Newell, Shaw and Simon (1957). Its design was guided by ideas from psychology, especially the intuitive notion of association. The primary elements of this language were symbols as opposed to numbers. To enable association to be made between these symbols list processing was introduced, which allowed programs to build data structures of unpredictable size and shape (when parsing a sentence it is impossible to know ahead of time what form the data structure will take.). Another feature introduced in IPL is the generator, a procedure for computing a series of values. It produces one value each time it is called and then is suspended so that it starts from where it left off last time it was called. The SAD-SAM program (see background section) was written in IPL.

The language used in the implementation section of this project should, ideally have the following properties:

- Be available in the University (and preferably available in some form for my home computer)
- Be well documented
- Have list processing techniques
- Contain simple pattern matching facilities
- Contain facilities to build complex knowledge structures
- Allow the programmer to split the problem solution into sub-sections.

LISP

LISP is the Second oldest programming language that is currently in widespread use (FORTRAN in the oldest). LISP, released by John McCarthy in 1958 incorporates the idea of list processing and some novel ideas about programming.

LISP is the primary AI programming language: it is used by the vast majority of AI researchers in all sub-fields, and has this position partly for historical reasons. It was established early in AI history, several large systems have been developed to support programming in the language, and all students in AI laboratories learn it. As a consequence LISP has become a shared language with which most AI researchers are familiar.

Besides LISP's use of lists structures as its only data type, it is different from other programming languages. Instead of describing computations as sequences of steps (instructions) LISP programs consist of functions defined in a very mathematical way..

LISP has developed a lot since its early days. Current LISP programming environments are themselves very large LISP programs. Two of the most highly developed LISP systems are MACLISP from M.I.T. and INTERLISP from B.B.N and Xerox.

INTERLISP has the following features

1. A uniform error handling system, which allows some kinds of automatic error correction such as spelling correction, entry to a special flexible debugging facility and handling of particular error conditions by user functions.
2. CLISP, which is an alternative, extensible, expression syntax.
3. Programmer's Assistant, which keeps track of the user's commands and also allows selected commands to be undone, re-tried or changed and re-tried.
4. Masterscope, which is a cross referencing facility to create a model of a user's program, that can be used to help manage a large system.
5. File package, which offers assistance such as storing functions that have been altered during a debugging session on a permanent file.

LISP can be used to produce some complex systems, Winograd's SHRDLU (see background section) was programmed in LISP.

There is a lot of information available about LISP, and implementations of it can be found both at the University and at the FTP sites of other Universities and commercial companies.

Prolog

Prolog is the most widely used logic programming language. It is used primarily in producing rapid-prototypes and for symbol manipulation tasks, such as writing compilers and parsing natural language It has also been used to produce expert systems.

Prolog is an expressive language for stating algorithms in computational linguistics. In NLP, we are frequently interested in manipulating symbols (words, partial phrases and other parts of speech) and structured objects (strings, sequences, trees, graphs) made from them. Prolog is a high level language that can directly express operations on symbols (represented by atoms, strings, numbers for instance) and structures (represented by lists or terms, for instance) without having to worry about how these high-level concepts are actually represented in the machine. Prolog allows us to talk about information at a very abstract level in terms of facts, and to express arbitrarily complex retrieval operations ('inferences') involving it. The concept of recursion plays a fundamental role in NLP. Linguistic objects are described by recursive data structures and operations on these data structures are naturally expressed as recursive algorithms. In common with other high-level programming languages Prolog places no restrictions on predicate definitions calling themselves (directly or indirectly), and so can express such algorithms directly.

I chose to program in Prolog, because it was designed for exactly for the type of processing of symbolic structures that NLP requires. It has also been proven, and is widely used in parsing natural language.

How Prolog works

Prolog is not a general-purpose theorem prover, and its powers of inference are limited. For example, it cannot answer these questions:

```
X > 1, X =< 2, integer(X).
25 is X^2 + Y^2, integer(X), integer(Y).
```

It cannot solve these questions because it can only prove something if it has some rule to compare with.

Prolog works by backward-chaining, using Modus Ponens:

```
If I am asked to prove Q
and I have the rule (P implies Q)
then if I can prove P, I can prove Q
    otherwise I shall assume Q  is false.
```

Note that if Prolog can't prove a goal, it assumes it to be false. This is called the closed world assumption.

Clauses

Prolog programs are written as a sequence of clauses. Each has the form

```
predicate( argument, argument, ... ).

or

predicate( argument, argument, ... ) :-
   predicate( argument, argument, ... ),
   predicate( argument, argument, ... ).
```

, means ``and".
:- means ``if".

All clauses are Horn clauses: one goal, implied by a conjunction of other goals (possibly none).

Prolog always executes the tail of a clause from left to right, as in conventional programming languages.

The effect of clause ordering

Because Prolog searches for clauses from top to bottom, the order in which they are written makes a difference. If a predicate has several solutions, their order of appearance is determined by the order of the clauses.

If the conditions in alternative clauses are not mutually exclusive, then backtracking will produce them both

Program components

Predicate names start with a lower-case letter. Example: is_a, diff.

The arguments to predicates can be any term:

Atoms.

Arbitrary names, used as logical constants. They must start with a lower case letter.

Example: x, identity_1.

Numbers.

Integers or floating-point numbers.

Example: 2, -25.39.

Variables.

Arbitrary names used as logical variables. They must start with a capital letter.

Example: X, N_plus_1.

Structures.

A functor.

Example: vec3(0,1,0), mat2(vec2(1,0), vec2(0,1)).

Lists.

Sequences. Lists are actually a type of structure, but have a special syntax.

Example: [the, dog, bit, the, cat], [], [[1,0,0], [0,1,0], [0,0,1]]

List processing

```
member( X, [X|Tail] ).

member( X, [Head|Tail] ) :-
   member( X, Tail ).
```

Read this as:

X is a member of any list whose

first element is X.

X is a member of any list if

X is a member of the tail of

the list.

The head of a list is its first element.

The tail of a list is the sub-list formed by its second, third, nth elements.

Appending to a list

```
append( [], L, L ).

append( [H|T], L, [H|T1] ) :-
   append( T, L, T1 ).
```

Read this as:

The result of joining the empty list

to L is L.

The result of joining the list whose

head is H and whose tail is T to

L is a list whose head is H and whose

tail is T1 if

   the result of joining T to L is T1.

Uses of lists

Lists can be used to represent sets. They can also be used to represent trees.

Lists can also be used effectively to represent sentences.

Operators

In the structure complex(a,b), complex is called the functor. Normally, structures are written in prefix notation, with the functor before its arguments.

Prolog allows us to declare that a given functor can be written between two arguments (infix), after an argument (postfix), or before an argument (prefix). Doing this avoids the need for brackets. Such functors are called operators. Operators can be given different precedence and associations.

Some functors, such as +, *, is, and \=, are already defined as operators by the system. This is why we can write C is A+B instead of is(C,+(A,B)). However, the bracketed form is also allowed.

Note: These operators are atoms. As well as sequences of letters, atoms can be sequences of "symbol characters". Examples: +, ++, -->.

The Cut(!)

To avoid writing the negation of a condition, Prolog uses the cut, written as !

The ! can be used to mean ``if you get this far, then discard (cut out) any alternative solutions".

The Cut(!) can be used to prevent backtracking. If it is placed as a sub-goal. If Prolog backtracks to the Cut it will fail the rule.

The top-level interpreter

All Prolog systems come with a top-level interpreter. This is a program which reads questions from the user, and calls Prolog's inference mechanism to determine their truth.

```
Loop:
   Display a '?-' prompt.
   Read the next question.
   If it is true,
      display the variable bindings
   otherwise
      say 'no'
   endif.
END Loop.
```

Efficiency Techniques

I found a good guide to writing efficient PROLOG code, Efficient Prolog: a practical guide, written by Michael Covington. In this report Covington (1989) Identifies a number of points for efficient Prolog

(It can be found at ftp://ai.uga.edu/pub/ai.reports/ai198908.ps.Z.):

- Think procedurally as well as declaratively.
- Narrow the search if possible.
- Let unification do the work.
- Avoid assert and retract.
- Understand tokenization.
- Avoid string processing.
- Recognise tail recursion.
- Let indexing help.
- Work at the beginning of lists.
- Avoid building data structures unless necessary.

Finding a foot hold

Before attempting to implement anything that I had learned through my project, I read a lot of material concerning Prolog, and how to write computer programs in it. A lot of the material I read was helpful, unfortunately some was less helpful. A problem I frequently encountered was the quality/abstraction of code examples, there are a lot of very simple Prolog examples, equally there are quite a few complex programs available from various sources; however, I was only able to find a few examples between this range. It would have

been a great benefit and sped my progress up if there were more examples of intermediate code available to examine.

Another problem I faced was the availability of other people to examine and comment on my code. My project supervisor was not a Prolog programmer, and so was only able to give limited advice and guidance. Very few of my colleagues know anything of Prolog, those that do, were at the time in the process of learning the language. A few had the advantage of having studied logic programming before, unfortunately I was not able to take this course. Even given these limitations it was often interesting and helpful to receive feedback from other people in a similar situation. I also found it helpful to give feedback on other peoples code.

First attempts

Having read around the Prolog programming language and experimenting with small chunks (typically 3 or 4 lines) of code. I decided to implement something related to my project. I decided to produce a small parser/phrase-tagger based on Chomsky's theories of generative grammar. It seemed to me, that the grammar rules would map easily to the logical definitions in Prolog. This program consisted of a small lexicon, defined at the top of the program, followed by the definitions of the grammar rules.

I based my program on a set of grammar rules, or re-write rules similar to those you have seen in the theory section of this report. This first program worked quite well, parsing and tagging short, simple sentences.

I used this program not only to parse sentences, but I also experimented, directing my experiments to the problem of detecting missing words from sentences. For example:

```
?- s([the,X,remembered,the,girl]).
```

Prolog will look for values of X that will, when inserted in the place of *X* will complete the sentence, agreeing with all the grammar rules that the program recognises. My program would first look at the rule defining sentences:

```
/*
** (X) is a sentence if (A) and (B) follow each other AND
**      ( (A) is a noun phrase AND (B) is a verb phrase )
*/
s(X) :- append(A,B,X),np(A),vp(B),
        nl,nl,write('np = '),write(A),write(' vp = '),write(B),
        nl,write('s = '),write(X).
```

This rule sates that a structure is a sentence if it can be described in terms of a noun phrase followed by a verb phrase. Because np is the next structure mentioned as part of a sentence, Prolog will look at the noun phrase rule:

```
/*
** (X) is a noun phrase if (A) and (B) follow each other AND
**                      ( (A) is a determinant AND (B) is a noun )
*/
np(X) :- append(A,B,X),det(A),n(B),
         nl,nl,write('det = '),write(A),
         write(' n = '),write(B),
         nl,write('np = '), write(X).
```

In this example X can be seen as a noun

Prolog would then find that two parts of the test sentence match this rule:

a) [the,X] (where X can be any value)

b) [the,girl]

Prolog finds the first noun defined in the lexicon and substitutes it for X in the above noun phrase.

a) [the,boy]

Prolog has now found all the noun phrases in this pass, so it will now look at the verb phrase rule:

```
/*
** (X) is a verb phrase if (A) and (B) follow each other AND
**                       ( (A) is a verb AND (B) is a noun phrase )
*/
vp(X) :- append(A,B,X),v(A),np(B),
         nl,nl,write('v = '),write(A),write(' np = '),write(B),
         nl,write('vp = '),write(X).
```

This rule states that a structure is a verb phrase if it can be described as a verb followed by a noun phrase.

Prolog would recognise [remembered,the,girl] as a valid verb phrase. Because the noun phrase [the,girl] has already been recognised as such, Prolog would see that the verb *remembered* is followed by a noun phrase, and recognise the verb phrase structure [remembered,the,girl].

This procedure can be seen in the output over the page.

```
Prolog-2 V2.35 PDE
Press F1 for help
?- ['c:\prolog\top-down\grammar.one'].
c:\prolog\top-down\grammar.one consulted
?- s([the,X,remembered,the,girl]).


det = [the] n = [boy]
np = [the,boy]

det = [the] n = [girl]
np = [the,girl]

v = [remembered] np = [the,girl]
vp = [remembered,the,girl]

np = [the,boy] vp = [remembered,the,girl]
s = [the,boy,remembered,the,girl]
X = boy
More (y/n)?
```

Selecting *y* will force Prolog to repeat this procedure, to find another value of X, until, eventually all the possible values of X (in this case nouns in the lexicon) have been exhausted.

Moving on

In the next version of my program I decided add an extra grammar rule, making the sentences that it could recognise less restricted and rigid. I decided that I would add the grammar rule that would allow a noun phrase to be built from a single pronoun. This would mean that more naturally sounding sentences like “she killed the boy” or “she killed him” could be recognised. This version was mainly used as a stepping stone, while I was still learning how to extend the range of the parser/phrase-tagger. A few more words were added, to existing lexical categories in the lexicon, and an additional lexical category was created, namely pronouns (he, his, it, its, her, she, hers, etc.).

The model of the grammar was built up incrementally, testing the new grammar rules as I added them.

In the next extension to my program I had planned to add grammar rules to allow the use of adjectives. Adjectives are interesting grammatical elements, because of the way they can be used to created infinitely long structures describing some arbitrary object or concept. The use of adjectives in this grammar is the first time I dealt with recursion. In this extension to the basic program three new grammar rules were added, and an extra lexical category.

I handled the recursive properties of adjectival phrases using the two rules shown below:

```
/*adjectival phrase*/

ap(X) :- append(A,B,X),adj(A),n(B),
         write('This adjectival phrase is made up of an adjective and
a noun'),
         nl,write('adj = '),write(X),
         write(' ap = '),write(X),nl.

ap(X) :- append(A,B,X),adj(A),ap(B),
         nl,write('This adjectival phrase is made up two or more
adjectives and a
noun'),
         nl,write('adj = '),write(A),write(' ap = '),write(B),
         nl,write('compound ap ='),write(X),nl.
```

The noun phrase that uses the adjectival phrase is shown here:

```
/*adjectival noun phrase eg 'the sly quick brown fox' */

np(X) :- append(A,B,X),det(A),ap(B),
nl,write('This noun phrase has a determinant and an adjectival
phrase'),
         nl,write('det = '),write(A),
         write(' ap = '),write(B),
         nl,write('np = '),write(X),nl.
```

These new rules, along with the extra lexical category allow sentences like "she fed the sly gorilla" and more complex sentences like "she fed the sly brown gorilla" and "she fed the sly quick brown gorilla" to be parsed and tagged correctly.

The next extension I made to my program was the addition of grammar rules to handle prepositional phrases. It was when testing this that I ran into a problem. The Prolog environment warned me that it had run out of workspace.

```
Prolog-2 V2.35 PDE                                                  Insert
Press F1 for help
?- ['c:\prolog\top-down\grammar.fou'].                  + Error menu ---+
c:\prolog\top-down\grammar.fou consulted                ¦ Abort         ¦
?- s([the,boy,with,the,girl]).                          ¦ (Break)       ¦
                                                        ¦ Exit          ¦
This noun phrase is made up from a determinant and a noun¦ (Fail)        ¦
det = [the] n = [boy]                                   ¦ Help          ¦
np = [the,boy]                                          ¦ (coRrect)     ¦
Evaluation aborted                                      ¦ (Trace)       ¦
                                                        ¦               ¦
?- s([the,boy,killed,the,boy,near,the,elephant]).       ¦               ¦
                                                        +---------------+
This noun phrase is made up from a determinant and a noun
det +-------------------------------------------------------------------+
np =¦Error no.173 (no_workseg)                                          ¦
    ¦Out of workspace                                                   ¦
    ¦Global: 200     Local: 41998                                       ¦
    ¦                                                                   ¦
    ¦                                                                   ¦
    ¦                                                                   ¦
    ¦                                                                   ¦
    ¦                                                                   ¦
    ¦                                                                   ¦
    +-------------------------------------------------------------------+
```

Obviously this shocked me a little and I was at a loss of how to remedy the situation. I had a break through, after reading a few books, some on parsing and artificial intelligence, and

some on Prolog. At this point I knew the advantages and disadvantages of both top-down and bottom up parses (see section on parsing issues and techniques), but it wasn't until I realised how these two techniques could be applied in Prolog, that I understood why my program was having problems with the amount resources available to it. I had built a top down parser, without actually realising it! A top-down parser will keep open all the branches of possible sentences it could find, until it finds one that is grammatically correct or alternatively all are ungrammatical. The way a top down parser might be implemented is to do with the position of the append statement in the grammar rules.

The query:

```
np(X) :- det(A),n(B),append(A,B,X)
```

might be called bottom up because it first selects lexical items and then verifies the combination. The query :

```
np(X) :- append(A,B,X),det(A),n(B)
```

could be called top down because it first selects the combination, and then, using a lexical search process, verifies that A is a determiner and B is a noun in the lexicon.

Bottom-Up Parser

After seeing first hand how Prolog interprets the grammar rules, I decided to continue implementing the rules as a top-down parser, but also to implement the rules as a bottom-up parser, and compare the differences between the two methods. From this point forward I decided define the lexicon separately from the grammar rules. This not only means I can use one lexicon for each of the two methods of parsing, it also means I need not re-enter the

lexicon code each time I start a new revision. Perhaps more importantly it means a more sophisticated lexicon could be implemented and used with these parsers. Very rarely do I get errors about running out of workspace when using the bottom-up parser.

Although I have noticed some other problems concerning this code. Due to the recursive properties of the grammar, the parser seems to be moving through the search space of the grammar and producing some interesting, if annoying results. The parser produces sentences, that whilst syntactically correct are difficult to comprehend, and can seemingly go on forever! For example, consider the following extract from some of the parser output:

```
This prepositional phrase is made up of a preposition and a noun phrase
prep = [with] np = [the,boy,with,her]
pp = [with,the,boy,with,her]
This noun phrase is made up of a noun and a prepositional phrase
np = [the,boy] pp = [with,the,boy,with,her]
This prepositional phrase is made up of a preposition and a noun phrase
prep = [with] np = [the,boy,with,the,boy,with,her]
pp = [with,the,boy,with,the,boy,with,her]
This noun phrase is made up of a noun and a prepositional phrase
np = [the,boy] pp = [with,the,boy,with,the,boy,with,her]
This prepositional phrase is made up of a preposition and a noun phrase
prep = [with] np = [the,boy,with,the,boy,with,the,boy,with,her]
pp = [with,the,boy,with,the,boy,with,the,boy,with,her]
This noun phrase is made up of a noun and a prepositional phrase
np = [the,boy] pp = [with,the,boy,with,the,boy,with,the,boy,with,her]
This prepositional phrase is made up of a preposition and a noun phrase
prep = [with] np = [the,boy,with,the,boy,with,the,boy,with,the,boy,with,her]
pp = [with,the,boy,with,the,boy,with,the,boy,with,the,boy,with,her]
```

This peculiarity seems to agree with Chomsky (see Chomskian Linguistics):

> It is impossible to list all the sentences of a particular language.

This peculiarity can in one sense be seen as proof of this; although whilst the sentences are syntactically correct it is very unlikely that such a sentence would be uttered.

Case Filter

The next idea I had planned to implement was a simple case filter. A case filter aims to model the way the brain relates certain words to other words in a sentence. For example:

He thought about himself.

He thought about him.

We are able to understand differences in the two sentences because of the use of case. In English there are three cases:

- Nominative (subject)
- Accusative (object)
- Genitive (possession)

I decided to implement a very simple filter to begin with. My aim was to program a filter that, when given a sentence could determine what case should be used and to give advice, or even change the case of appropriate words. First I defined a simple lexicon so that I would not need to worry about the complexities of a larger lexicon. The lexicon contained :

- A verb (kicked)
- A Subject (she)
- An Object (her)

Sentences would be in the basic form of subject-verb-object (SVO). I programmed the filter by using rules that would define a sentence written with correct use of case. The rules are stated here:

```
correct_case(X) :- subject_verb(A),object(B),append(A,B,X),
                   nl,write(B),write(' In Correct Case'),nl.
```

This rule states that if a sentence has the structure of subject verb object it should write out that it is in the correct case.

```
subject_verb(X) :- subject(A),v(B),append(A,B,X),
                   nl,write('Subject - Verb in right order'),nl.
```

Similarly this rule says if a subject-verb definition has the structure of subject-verb then it should write out that the subject and verb are in the right order.

The rules showing incorrect case where coded using the same pattern. I originally coded the case filter with two subject_verb rules: one for structures where the subject and verb follow each other, and one where an object was in the place of the subject. This in retrospect was a bad idea. When the program ran it would sometimes print out duplicate information. It also often gave misleading results, because it would get so far in a rule, only to fail and have to backtrack.

I changed the program so that in the case of an object-verb clause being in place of a subject-verb a separately named rule applies namely object_verb(X). The new version now handles simple subject-verb-object sentences and outputs appropriate advice on the case of words in the sentences.

I hope to extend this simple case filter to handle more complex case issues such as valency and possession.

Other experiments

There are a range of other ideas that I have tried to, or would like to implement. For example I have tried to implement an input routine, that would make it easier to enter a sentence. This program takes a set of words separated by spaces and terminated with a full-stop (.). The program then converts this input into a structure of words separated by commas (,). This is a much more natural way of entering the sentence data into programs: sentences are not words separated by commas but are words separated by spaces and ended with a full-stop..

I particularly find the idea of transforming a declarative sentence to an interrogative sentence very interesting. I hoped to be able to write a program that followed Chomsky's transformational rules, making modifications to an entered phrase structure tree and to output the modified phrase structure tree.

It would be interesting to experiment with trying to find the subject of a sentence. Subjects are very important grammatical structures. The ability to identify the subject of a sentence would have implications for a range of possible programs. A program might be able to identify the subject of a sentence, and then would be able to ask questions about the subject.

I did some work towards the goal of arranging sentence components in the right order, when they are passed in. This program models what might occur when translating deep structures to surface structures. Words are passed in and the program will sort them into the correct order for an English sentence.

I found a Government and Binding parser on the Internet and have studied the code, in the hope of extending and/or modifying it.

Conclusions

Learning

Through the work I have done during the lifetime of this project, I have learned a great deal. Not only have I taught myself the PROLOG programming language, but I now also have a good knowledge of current linguistic theories. Because I came fresh to linguistics and had to work at learning and understanding linguistic theories , I was unable to fully implement all of what I had learned due to time constraints.

Research Methods

The main methods of research I used in the process of writing this report were the library and the use of the Internet. The Internet proved useful in my goal of learning PROLOG: there are a lot of tutorials and FAQs available. There is information available about language and linguistics available on the Internet; although I gained most of my knowledge in this area by the use of the library. It would be useful for anyone considering continuing this project to join and read news groups and mailing lists, something which I did not do, and regret.

To join the association of logic programming send E-Mail to csa@doc.ic.ac.uk

For information about the Computation and Language Electronic pre-print Server send E-Mail to cmp-lg@xxx.lanl.gov with the subject “help”

Critical Appraisal

Throughout the writing of this report a balance had to be struck between learning linguistic theories and learning PROLOG so that I could implement them as PROLOG programs. I found this difficult, it is very hard to balance the two: knowing enough about a linguistic theory to be able to implement it, and knowing enough PROLOG to understand how it might be implemented. It would have been preferable to have known one area before coming to this project; however I believe I have made a good attempt at implementing a few key areas of a NLP program, and my knowledge is such that I would be able to continue developing a reasonable set of NLP procedures, that might be used in a larger program.

My knowledge of PROLOG has been my major stumbling block. I believe if I had spent more time learning PROLOG at the beginning of this project I would not have had as many problems as I did, and my code would probably be more efficient and easier to extend. I was particularly dis-heartened by the problems I faced in the way my parser handles prepositional phrases, with a better knowledge of PROLOG I am positive that these problems could be eradicated.

Appendices

Glossary

Accusative. The case of the object of a verb.

Adjective. One of the major syntactic categories, comprising words that typically refer to property or state.

Article. One of the minor syntactic categories, including the words a and the. Usually subsumed in the category determiner in contemporary theories of grammar.

Auxiliary. A special kind of verb used to express concepts related to the truth of a sentence, such as tense, negation, question/statement, necessary/possible.

Backtracking. A term used when a parser has to revise decisions that it has previously made.

Bottom-up Parse. A parsing method in which starts with the individual words of a sentence and tries to build upwards, through phrases to the complete sentence.

Behaviourism. A school of psychology, influential from the 1920s to the 1960s, that rejected the study of the mind as unscientific, and sought to explain the behaviour of organisms with laws of stimulus-response conditioning.

Case. A set of affixes, positions, or word forms that a language uses to distinguish the different roles of the participants in some event or state. Case typically correspond to the subject, object, indirect object, and the objects of various kinds of prepositions. In English case is what distinguishes between I, he, she, we, they, which are used as subjects, and me, him, her, us, them, which are used as objects of verbs, objects of verbs, and everywhere else.

Determiner. One of the minor syntactic categories, comprising the articles and similar words: a, the, some, more, much, many.

Generative grammar. See grammar

Grammar. A generative grammar is a set of rules that determines the form and meaning of words and sentences in a particular language as it is spoken in some community. A mental grammar is the hypothetical generative grammar stored unconsciously in a person's brain neither should be confused with perscriptive grammar, which is taught in schools and explained in style guides.

Head. The single word in a phrase, that determines the meaning and properties of the whole phrase.

Lexical entry. The information about a particular word (its meaning, syntactic category, etc.) stored in a dictionary.

Lexicon. A dictionary of words and meaning in a language.

Linguist. A scholar or scientist that studies how languages work. It does not refer to a person that speaks multiple languages.

Main verb. A verb that is not the auxiliary.

Modal. A kind of Auxiliary.

Movement. The principle kind of transformational rule in Chomsky's theory, it moves a phrase from its customary position in deep structure to some other unfilled position, leaving behind a "trace".

NLP. Natural Language Processing

Natural Language. A human language like English or Japanese, as opposed to a computer language or other formal representation.

Natural Language Processing. A school of computer science, in which Natural languages are used as data for computer programs. Often used for parsing or machine translation.

Nominative. The case of the subject of a sentence.

Noun. One of the major syntactic categories, comprising words that typically refer to an item, or person.

Object. The argument next to the verb, typically refers to the entity affected by the verb

Parsing. A process involved in sentence comprehension in which the syntactic categories of words are determined.

Phrase structure tree. A tree structure that shows how the words of a sentence have been broken into their syntactic categories.

Preposition. One of the major syntactic categories , comprising words that typically refer to spatial or temporal relationship. In, on , near

Pronoun. A word that stands for a whole noun phrase. I , me, my, you, your

Top-down parse. A parsing method in which the parser starts with a sentence rule, and tries to build its way down, through phrases to the individual words of a sentence.

Universal Grammar. The basic design underlying the grammars of all human languages

Verb. One of the major syntactic categories, comprising words that typically refer to some action.

X-bar theory. The particular kind of phrase structure rules thought to be used in human language, according to which all the phrases in all languages conform to a single plan. In the plan, the properties of the whole phrase are determined by the properties of a single element, the head of the phrase.

Code

Top-down parser/phrase-tagger

Program 1

```
/* A small grammar */


/* determinants */
det([the]).
det([a]).


/*nouns*/
n([boy]).
n([girl]).
n([gorilla]).
n([elephant]).
n([school]).

/*verbs*/
v([remembered]).
v([forgot]).
v([stroked]).


/* Grammar Definition */


/*
** (X) is a sentence if (A) and (B) follow each other AND
**                      ( (A) is a noun phrase AND (B) is a verb
phrase )
*/
s(X) :- append(A,B,X),np(A),vp(B),
      nl,nl,write('np = '),write(A),write(' vp = '),write(B),
      nl,write('s = '),write(X).


/*
** (X) is a verb phrase if (A) and (B) follow each other AND
**                      ( (A) is a verb AND (B) is a noun phrase )
*/
vp(X) :- append(A,B,X),v(A),np(B),
       nl,nl,write('v = '),write(A),write(' np = '),write(B),
       nl,write('vp = '),write(X).
```

```
/*
** (X) is a noun phrase if (A) and (B) follow each other AND
**                       ( (A) is a determinant AND (B) is a noun )
*/
np(X) :- append(A,B,X),det(A),n(B),
       nl,nl,write('det = '),write(A),
       write(' n = '),write(B),
       nl,write('np = '), write(X).

append([],L,L).
append([X|L1],L2,[X|L3]) :- append(L1,L2,L3).
```

Program 2

```
/* A small grammar */


/* determinants */
det([the]).
det([a]).


/*nouns*/
n([boy]).
n([girl]).
n([gorilla]).
n([elephant]).
n([school]).

/*pronouns*/

p([he]).
p([it]).
p([she]).
p([him]).
p([her]).

/*verbs*/
v([remembered]).
v([forgot]).
v([stroked]).
v([killed]).
v([fed]).
```

```
/* Grammar Definition */


/*
** (X) is a sentence if (A) and (B) follow each other AND
**                      ( (A) is a noun phrase AND (B) is a verb
phrase )
*/
s(X) :- append(A,B,X),np(A),vp(B),
      nl,write('This sentence is made of a noun phrase and a verb
phrase'),nl,
      nl,nl,write('np = '),write(A),write(' vp = '),write(B),
      nl,write('s = '),write(X).


/*
** (X) is a verb phrase if (A) and (B) follow each other AND
**                      ( (A) is a verb AND (B) is a noun phrase )
*/
vp(X) :- append(A,B,X),v(A),np(B),
       nl,write('This verb phrase is made up of a verb and a noun
phrase'),
       nl,write('v = '),write(A),write(' np = '),write(B),
       nl,write('vp = '),write(X),nl.


/*
** (X) is a noun phrase if (A) and (B) follow each other AND
**                      ( (A) is a determinant AND (B) is a noun )
*/
np(X) :- append(A,B,X),det(A),n(B),
       nl,write('This noun phrase is made up from a determinant and a
noun'),
       nl,write('det = '),write(A),
       write(' n = '),write(B),
       nl,write('np = '),write(X).

/* a second example of a noun phrase */

np(X) :- p(X),write('This noun phrase is made up of a single
pronoun'),
                 nl,write('p = '),write(X),
                 write(' np = '),write(X),nl.

append([],L,L).
append([X|L1],L2,[X|L3]) :- append(L1,L2,L3).
```

Program 3

```
/* A small grammar */


/* determinants */
det([the]).
det([a]).


/*nouns*/
n([boy]).
n([girl]).
n([gorilla]).
n([elephant]).
n([school]).

/*pronouns*/

p([he]).
p([it]).
p([she]).
p([him]).
p([her]).

/*verbs*/
v([remembered]).
v([forgot]).
v([stroked]).
v([killed]).
v([fed]).

/*adjectives*/
adj([quick]).
adj([brown]).
adj([sly]).

/* Grammar Definition */


/*
** (X) is a sentence if (A) and (B) follow each other AND
**                      ( (A) is a noun phrase AND (B) is a verb
phrase )
*/
s(X) :- append(A,B,X),np(A),vp(B),
      nl,write('This sentence is made of a noun phrase and a verb
phrase'),nl,
      nl,nl,write('np = '),write(A),write(' vp = '),write(B),
      nl,write('s = '),write(X).


/*
** (X) is a verb phrase if (A) and (B) follow each other AND
**                      ( (A) is a verb AND (B) is a noun phrase )
*/
```

```
vp(X) :- append(A,B,X),v(A),np(B),
       nl,write('This verb phrase is made up of a verb and a noun
phrase'),
       nl,write('v = '),write(A),write(' np = '),write(B),
       nl,write('vp = '),write(X),nl.


/*
** (X) is a noun phrase if (A) and (B) follow each other AND
**                       ( (A) is a determinant AND (B) is a noun )
*/
np(X) :- append(A,B,X),det(A),n(B),
       nl,write('This noun phrase is made up from a determinant and a
noun'),
       nl,write('det = '),write(A),
       write(' n = '),write(B),
       nl,write('np = '),write(X).

/*adjectival noun phrase eg 'the sly quick brown fox' */

np(X) :- append(A,B,X),det(A),ap(B),
       nl,write('This noun phrase has a determinant and an adjectival
phrase '),
       nl,write('det = '),write(A),
       write(' ap = '),write(B),
       nl,write('np = '),write(X),nl.

/* a second example of a noun phrase */

np(X) :- p(X),write('This noun phrase is made up of a single
pronoun'),
                  nl,write('p = '),write(X),
                  write(' np = '),write(X),nl.

/*adjectival phrase*/

ap(X) :- append(A,B,X),adj(A),n(B),
       write('This adjectival phrase is made up of an adjective and a
noun'),
       nl,write('adj = '),write(X),
       write(' ap = '),write(X),nl.

ap(X) :- append(A,B,X),adj(A),ap(B),
       nl,write('This adjectival phrase is made up two or more
adjectives and a noun'),
       nl,write('adj = '),write(A),write(' ap = '),write(B),
       nl,write('compound ap ='),write(X),nl.




append([],L,L).
append([X|L1],L2,[X|L3]) :- append(L1,L2,L3).
```

Program 4

```
/* A small grammar */


/* determinants */
det([the]).
det([a]).


/*nouns*/
n([boy]).
n([girl]).
n([gorilla]).
n([elephant]).
n([school]).

/*pronouns*/

p([he]).
p([it]).
p([she]).
p([him]).
p([her]).

/*verbs*/
v([remembered]).
v([forgot]).
v([stroked]).
v([killed]).
v([fed]).

/*adjectives*/
adj([quick]).
adj([brown]).
adj([sly]).


/*prepositions*/

prep([with]).
prep([in]).
prep([near]).




/* Grammar Definition */


/*
** (X) is a sentence if (A) and (B) follow each other AND
**                      ( (A) is a noun phrase AND (B) is a verb
phrase )
*/
s(X) :- append(A,B,X),np(A),vp(B),
```

```
      nl,write('This sentence is made of a noun phrase and a verb
phrase'),nl,
      nl,nl,write('np = '),write(A),write(' vp = '),write(B),
      nl,write('s = '),write(X).


/*
** (X) is a verb phrase if (A) and (B) follow each other AND
**                      ( (A) is a verb AND (B) is a noun phrase )
*/
vp(X) :- append(A,B,X),v(A),np(B),
       nl,write('This verb phrase is made up of a verb and a noun
phrase'),
       nl,write('v = '),write(A),write(' np = '),write(B),
       nl,write('vp = '),write(X),nl.

/*
** (X) is a verb phrase if (A) and (B) follow each other AND
**                      ( (A) is a verb phrase AND (B) is
prepositional phrase )
*/
vp(X) :- append(A,B,X),vp(A),pp(B),
       nl,write('This verb phrase is made up of a verb phrase and a
prepositional phrase'),
       nl,write('vp = '),write(A),write(' pp = '),write(B),
       nl,write('vp = '),write(X),nl.




/*
** (X) is a noun phrase if (A) and (B) follow each other AND
**                      ( (A) is a determinant AND (B) is a noun )
*/
np(X) :- append(A,B,X),det(A),n(B),
       nl,write('This noun phrase is made up from a determinant and a
noun'),
       nl,write('det = '),write(A),
       write(' n = '),write(B),
       nl,write('np = '),write(X).

/*adjectival noun phrase eg 'the sly quick brown fox' */

np(X) :- append(A,B,X),det(A),ap(B),
       nl,write('This noun phrase has a determinant and an adjectival
phrase '),
       nl,write('det = '),write(A),
       write(' ap = '),write(B),
       nl,write('np = '),write(X),nl.

/* another noun phrase */

np(X) :- p(X),write('This noun phrase is made up of a single
pronoun'),
                  nl,write('p = '),write(X),
                  write(' np = '),write(X),nl.
```

```
/*prepositional phrase*/

pp(X) :- append(A,B,X),prep(A),np(B),
       write('This prepositional phrase is made up of an preposition
and a noun phrase'),
       nl,write('prep = '),write(A),
       write(' np = '),write(B),nl,
       write('pp = '),write(X),nl.



/*adjectival phrase*/

ap(X) :- append(A,B,X),adj(A),n(B),
       write('This adjectival phrase is made up of an adjective and a
noun'),
       nl,write('adj = '),write(X),
       write(' ap = '),write(X),nl.

ap(X) :- append(A,B,X),adj(A),ap(B),
       nl,write('This adjectival phrase is made up two or more
adjectives and a noun'),
       nl,write('adj = '),write(A),write(' ap = '),write(B),
       nl,write('compound ap ='),write(X),nl.




append([],L,L).
append([X|L1],L2,[X|L3]) :- append(L1,L2,L3).
```

Program 5

```
/* A small grammar */


/* determinants */
det([the]).
det([a]).


/*nouns*/
n([boy]).
n([girl]).
n([gorilla]).
n([elephant]).
n([school]).
n([saw]). /* a woodwork saw cf the verb */
n([book]).

/*pronouns*/

p([he]).
p([it]).
p([she]).
p([him]).
p([her]).

/*verbs*/
v([remembered]).
v([forgot]).
v([stroked]).
v([killed]).
v([fed]).
v([saw]).

/*adjectives*/
adj([quick]).
adj([brown]).
adj([sly]).
adj([clever]).
adj([old]).
adj([dusty]).

/* Grammar Definition */


/*
** (X) is a sentence if (A) and (B) follow each other AND
**                      ( (A) is a noun phrase AND (B) is a verb
phrase )
*/
s(X) :- append(A,B,X),np(A),vp(B),
      nl,write('This sentence is made of a noun phrase and a verb
phrase'),nl,
      nl,nl,write('np = '),write(A),write(' vp = '),write(B),
      nl,write('s = '),write(X).
```

```
/*
** (X) is a verb phrase if (A) and (B) follow each other AND
**                      ( (A) is a verb AND (B) is a noun phrase )
*/
vp(X) :- append(A,B,X),v(A),np(B),
       nl,write('This verb phrase is made up of a verb and a noun
phrase'),
       nl,write('v = '),write(A),write(' np = '),write(B),
       nl,write('vp = '),write(X),nl.


/*
** (X) is a noun phrase if (A) and (B) follow each other AND
**                      ( (A) is a determinant AND (B) is a noun )
*/
np(X) :- append(A,B,X),det(A),n(B),
       nl,write('This noun phrase is made up from a determinant and a
noun'),
       nl,write('det = '),write(A),
       write(' n = '),write(B),
       nl,write('np = '),write(X).

/*adjectival noun phrase eg 'the sly quick brown fox' */

np(X) :- append(A,B,X),det(A),ap(B),
       nl,write('This noun phrase has a determinant and an adjectival
phrase '),
       nl,write('det = '),write(A),
       write(' ap = '),write(B),
       nl,write('np = '),write(X),nl.

/* a second example of a noun phrase */

np(X) :- p(X),write('This noun phrase is made up of a single
pronoun'),
                 nl,write('p = '),write(X),
                 write(' np = '),write(X),nl.

/*adjectival phrase*/

ap(X) :- append(A,B,X),adj(A),n(B),
       write('This adjectival phrase is made up of an adjective and a
noun'),
       nl,write('adj = '),write(X),
       write(' ap = '),write(X),nl.

ap(X) :- append(A,B,X),adj(A),ap(B),
       nl,write('This adjectival phrase is made up two or more
adjectives and a noun'),
       nl,write('adj = '),write(A),write(' ap = '),write(B),
       nl,write('compound ap ='),write(X),nl.
```

```
append([],L,L).
append([X|L1],L2,[X|L3]) :- append(L1,L2,L3).
```

Program 6

```
/* A small grammar */


/* determinants */
det([the]).
det([a]).


/*nouns*/
n([boy]).
n([girl]).
n([gorilla]).
n([elephant]).
n([school]).
n([saw]). /* a woodwork saw cf the verb */
n([book]).
n([red]). /*communist as opposed to the colour*/

/*pronouns*/

p([he]).
p([it]).
p([she]).
p([him]).
p([her]).

/*verbs*/
v([remembered]).
v([forgot]).
v([stroked]).
v([killed]).
v([fed]).
v([saw]).

/*adjectives*/
adj([quick]).
adj([brown]).
adj([sly]).
adj([clever]).
adj([old]).
adj([dusty]).
adj([red]).


/* Grammar Definition */


/*
** (X) is a sentence if (A) and (B) follow each other AND
**                      ( (A) is a noun phrase AND (B) is a verb
phrase )
```

```
*/
s(X) :- append(A,B,X),np(A),vp(B),
      nl,write('This sentence is made of a noun phrase and a verb
phrase'),nl,
      nl,nl,write('np = '),write(A),write(' vp = '),write(B),
      nl,write('s = '),write(X).


/*
** (X) is a verb phrase if (A) and (B) follow each other AND
**                      ( (A) is a verb AND (B) is a noun phrase )
*/
vp(X) :- append(A,B,X),v(A),np(B),
       nl,write('This verb phrase is made up of a verb and a noun
phrase'),
       nl,write('v = '),write(A),write(' np = '),write(B),
       nl,write('vp = '),write(X),nl.


/*
** (X) is a noun phrase if (A) and (B) follow each other AND
**                      ( (A) is a determinant AND (B) is a noun )
*/
np(X) :- append(A,B,X),det(A),n(B),
       nl,write('This noun phrase is made up from a determinant and a
noun'),
       nl,write('det = '),write(A),
       write(' n = '),write(B),
       nl,write('np = '),write(X).

/*adjectival noun phrase eg 'the sly quick brown fox' */

np(X) :- append(A,B,X),det(A),ap(B),
       nl,write('This noun phrase has a determinant and an adjectival
phrase '),
       nl,write('det = '),write(A),
       write(' ap = '),write(B),
       nl,write('np = '),write(X),nl.

/* a second example of a noun phrase */

np(X) :- p(X),write('This noun phrase is made up of a single
pronoun'),
                 nl,write('p = '),write(X),
                 write(' np = '),write(X),nl.

/*adjectival phrase*/

ap(X) :- append(A,B,X),adj(A),n(B),
       write('This adjectival phrase is made up of an adjective and a
noun'),
       nl,write('adj = '),write(X),
       write(' ap = '),write(X),nl.

ap(X) :- append(A,B,X),adj(A),ap(B),
       nl,write('This adjectival phrase is made up two or more
adjectives and a noun'),
       nl,write('adj = '),write(A),write(' ap = '),write(B),
```

```
       nl,write('compound ap ='),write(X),nl.
```

```
append([],L,L).
append([X|L1],L2,[X|L3]) :- append(L1,L2,L3).
```

Lexicon

```
/* A small lexicon */


/* determinants */
det([the]).
det([a]).


/*nouns*/
n([boy]).
n([girl]).
n([gorilla]).
n([elephant]).
n([school]).

/*pronouns*/

p([he]).
p([it]).
p([she]).
p([him]).
p([her]).

/*verbs*/
v([remembered]).
v([forgot]).
v([stroked]).
v([killed]).
v([fed]).

/*adjectives*/
adj([quick]).
adj([brown]).
adj([sly]).


/*prepositions*/

prep([with]).
prep([in]).
prep([near]).
```

Bottom-Up

Program 1

```
/* A small grammar */

/* Grammar Definition */


/*
** (X) is a sentence if (A) and (B) follow each other AND
**                      ( (A) is a noun phrase AND (B) is a verb
phrase )
*/
s(X) :- np(A),vp(B),append(A,B,X),
      nl,write('This sentence is made of a noun phrase and a verb
phrase'),nl,
      nl,nl,write('np = '),write(A),write(' vp = '),write(B),
      nl,write('s = '),write(X).


/*
** (X) is a verb phrase if (A) and (B) follow each other AND
**                      ( (A) is a verb AND (B) is a noun phrase )
*/
vp(X) :- v(A),np(B),append(A,B,X),
       nl,write('This verb phrase is made up of a verb and a noun
phrase'),
       nl,write('v = '),write(A),write(' np = '),write(B),
       nl,write('vp = '),write(X),nl.

/*
** (X) is a verb phrase if (A) and (B) follow each other AND
**                      ( (A) is a verb phrase AND (B) is
prepositional phrase )
*/
vp(X) :- vp(A),pp(B),append(A,B,X),
       nl,write('This verb phrase is made up of a verb phrase and a
prepositional phrase'),
       nl,write('vp = '),write(A),write(' pp = '),write(B),
       nl,write('vp = '),write(X),nl.




/*
** (X) is a noun phrase if (A) and (B) follow each other AND
**                      ( (A) is a determinant AND (B) is a noun )
*/
np(X) :- det(A),n(B),append(A,B,X),
       nl,write('This noun phrase is made up from a determinant and a
noun'),
       nl,write('det = '),write(A),
       write(' n = '),write(B),
       nl,write('np = '),write(X).

/*adjectival noun phrase eg 'the sly quick brown fox' */
```

```
np(X) :- det(A),ap(B),append(A,B,X),
       nl,write('This noun phrase has a determinant and an adjectival
phrase '),
       nl,write('det = '),write(A),
       write(' ap = '),write(B),
       nl,write('np = '),write(X),nl.

/* another noun phrase */

np(X) :- p(X),write('This noun phrase is made up of a single
pronoun'),
                 nl,write('p = '),write(X),
                 write(' np = '),write(X),nl.




/*prepositional phrase*/

pp(X) :- prep(A),np(B),append(A,B,X),
       write('This prepositional phrase is made up of an preposition
and a noun phrase'),
       nl,write('prep = '),write(A),
       write(' np = '),write(B),nl,
       write('pp = '),write(X),nl.



/*adjectival phrase*/

ap(X) :- adj(A),n(B),append(A,B,X),
       write('This adjectival phrase is made up of an adjective and a
noun'),
       nl,write('adj = '),write(X),
       write(' ap = '),write(X),nl.

ap(X) :- adj(A),ap(B),append(A,B,X),
       nl,write('This adjectival phrase is made up two or more
adjectives and a noun'),
       nl,write('adj = '),write(A),write(' ap = '),write(B),
       nl,write('compound ap ='),write(X),nl.




append([],L,L).
append([X|L1],L2,[X|L3]) :- append(L1,L2,L3).
```

Program 2

```
/* A small grammar */

/* Grammar Definition */
```

```
/*
** (X) is a sentence if (A) and (B) follow each other AND
**                      ( (A) is a noun phrase AND (B) is a verb
phrase )
*/
s(X) :- np(A),vp(B),append(A,B,X),
      nl,write('This sentence is made of a noun phrase and a verb
phrase'),nl,
      nl,nl,write('np = '),write(A),write(' vp = '),write(B),
      nl,write('s = '),write(X).


/*
** (X) is a verb phrase if (A) and (B) follow each other AND
**                      ( (A) is a verb AND (B) is a noun phrase )
*/
vp(X) :- v(A),np(B),append(A,B,X),
       nl,write('This verb phrase is made up of a verb and a noun
phrase'),
       nl,write('v = '),write(A),write(' np = '),write(B),
       nl,write('vp = '),write(X),nl.

/*
** (X) is a verb phrase if (A) and (B) follow each other AND
**                      ( (A) is a verb phrase AND (B) is
prepositional phrase )

vp(X) :- vp(A),pp(B),append(A,B,X),
       nl,write('This verb phrase is made up of a verb phrase and a
prepositional phrase'),
       nl,write('vp = '),write(A),write(' pp = '),write(B),
       nl,write('vp = '),write(X),nl.

*/



/*
** (X) is a noun phrase if (A) and (B) follow each other AND
**                      ( (A) is a determinant AND (B) is a noun )
*/
np(X) :- det(A),n(B),append(A,B,X),
       nl,write('This noun phrase is made up from a determinant and a
noun'),
       nl,write('det = '),write(A),
       write(' n = '),write(B),
       nl,write('np = '),write(X).

/*adjectival noun phrase eg 'the sly quick brown fox' */

np(X) :- det(A),ap(B),append(A,B,X),
       nl,write('This noun phrase has a determinant and an adjectival
phrase '),
       nl,write('det = '),write(A),
       write(' ap = '),write(B),
       nl,write('np = '),write(X),nl.

/* another noun phrase */
```

```
np(X) :- p(X),write('This noun phrase is made up of a single
pronoun'),
                  nl,write('p = '),write(X),
                  write(' np = '),write(X),nl.

np(X) :- np(A),pp(B),append(A,B,X),
                  write('This noun phrase is made up of a noun and a
prepositional phrase'),
                  nl,write('np = '),write(A),
                  write(' pp = '),write(B),nl.


/*prepositional phrase*/

pp(X) :- prep(A),np(B),append(A,B,X),
       write('This prepositional phrase is made up of a preposition
and a noun phrase'),
       nl,write('prep = '),write(A),
       write(' np = '),write(B),nl,
       write('pp = '),write(X),nl.



/*adjectival phrase*/

/* ap(X) :- adj(A),n(B),append(A,B,X),
       write('This adjectival phrase is made up of an adjective and a
noun'),
       nl,write('adj = '),write(X),
       write(' ap = '),write(X),nl.
*/
/*ap(X) :- adj(A),ap(B),append(A,B,X),
       nl,write('This adjectival phrase is made up two or more
adjectives and a noun'),
       nl,write('adj = '),write(A),write(' ap = '),write(B),
       nl,write('compound ap ='),write(X),nl.

*/



append([],L,L).
append([X|L1],L2,[X|L3]) :- append(L1,L2,L3).
```

Program 3

```
/* A small grammar */

/* Grammar Definition */


/*
** (X) is a sentence if (A) and (B) follow each other AND
**                      ( (A) is a noun phrase AND (B) is a verb
phrase )
```

```
*/
s(X) :- np(A),vp(B),append(A,B,X),
      nl,write('This sentence is made of a noun phrase and a verb
phrase'),nl,
      nl,nl,write('np = '),write(A),write(' vp = '),write(B),
      nl,write('s = '),write(X).


/*
** (X) is a verb phrase if (A) and (B) follow each other AND
**                      ( (A) is a verb AND (B) is a noun phrase )
*/
vp(X) :- v(A),np(B),append(A,B,X),
       nl,write('This verb phrase is made up of a verb and a noun
phrase'),
       nl,write('v = '),write(A),write(' np = '),write(B),
       nl,write('vp = '),write(X),nl.

/*
** (X) is a verb phrase if (A) and (B) follow each other AND
**                      ( (A) is a verb phrase AND (B) is
prepositional phrase )
*/
vp(X) :- vp(A),pp(B),append(A,B,X),
       nl,write('This verb phrase is made up of a verb phrase and a
prepositional phrase'),
       nl,write('vp = '),write(A),write(' pp = '),write(B),
       nl,write('vp = '),write(X),nl.




/*
** (X) is a noun phrase if (A) and (B) follow each other AND
**                      ( (A) is a determinant AND (B) is a noun )
*/
np(X) :- det(A),n(B),append(A,B,X),
       nl,write('This noun phrase is made up from a determinant and a
noun'),
       nl,write('det = '),write(A),
       write(' n = '),write(B),
       nl,write('np = '),write(X).

/*adjectival noun phrase eg 'the sly quick brown fox' */

np(X) :- det(A),ap(B),append(A,B,X),
       nl,write('This noun phrase has a determinant and an adjectival
phrase '),
       nl,write('det = '),write(A),
       write(' ap = '),write(B),
       nl,write('np = '),write(X),nl.

/* another noun phrase */

np(X) :- p(X),write('This noun phrase is made up of a single
pronoun'),
                 nl,write('p = '),write(X),
```

```
                write(' np = '),write(X),nl.



/*prepositional phrase*/

pp(X) :- prep(A),np(B),append(A,B,X),
       write('This prepositional phrase is made up of an preposition
and a noun phrase'),
       nl,write('prep = '),write(A),
       write(' np = '),write(B),nl,
       write('pp = '),write(X),nl.



/*adjectival phrase*/

/* ap(X) :- adj(A),n(B),append(A,B,X),
       write('This adjectival phrase is made up of an adjective and a
noun'),
       nl,write('adj = '),write(X),
       write(' ap = '),write(X),nl.
*/
/*ap(X) :- adj(A),ap(B),append(A,B,X),
       nl,write('This adjectival phrase is made up two or more
adjectives and a noun'),
       nl,write('adj = '),write(A),write(' ap = '),write(B),
       nl,write('compound ap ='),write(X),nl.

*/


/*%%%%%%%%%
%   'read_sentence' provides the ability to get input
%   in a natural fashion by typing in words separated
%   by spaces and terminated with a period.  Adapted
%   from _Prolog and Natural Language Analysis_ by
%   Pereira and Schieber.
%
%%%%%%%%%*/

read_sentence(Input) :- get0(Char), read_sentence(Char,Input).
read_sentence(Char,[]) :- period(Char),!.
read_sentence(Char,Input) :- space(Char),!,get0(Char1),
       read_sentence(Char1,Input).
read_sentence(Char,[Word|Words]) :- read_word(Char,Chars,Next),
       name(Word,Chars),
       read_sentence(Next,Words).

read_word(C,[],C) :- space(C),!.
read_word(C,[],C) :- period(C),!.
read_word(Char,[Char|Chars],Last) :- get0(Next),
read_word(Next,Chars,Last).

space(32).
period(46).
```

```
parse_sentence :- read_sentence(Sentence),write('This here'),nl,
                                    write(Sentence).


append([],L,L).
append([X|L1],L2,[X|L3]) :- append(L1,L2,L3).
```

Case Filter

Program 1 - Original attempt

```
v([kicked]).

object([her]).
subject([she]).

correct_case(X) :- subject_verb(A),object(B),append(A,B,X),
               nl,write(B),write(' In Correct Case'),nl.


correct_case(X) :- subject_verb(A),subject(B),append(A,B,X),
               nl,write(B),write(' in wrong case, use
accusative'),nl.


subject_verb(X) :- object(A),v(B),append(A,B,X),
              nl,write('Object - Verb In the wrong order'),nl,
              write('Use the nominative case of '),write(A).

subject_verb(X) :- subject(A),v(B),append(A,B,X),
              nl,write('Subject - Verb in right order'),nl.




append([],L,L).
append([X|L1],L2,[X|L3]) :- append(L1,L2,L3).
```

Program 2 -- amended

```
v([kicked]).

object([her]).
subject([she]).

correct_case(X) :- append(A,B,X),subject_verb(A),object(B),
                   nl,write(B),write(' In Correct Case'),nl.


correct_case(X) :- append(A,B,X),object_verb(A),subject(B),
                   nl,write(B),write(' in wrong case, use
accusative'),nl.


object_verb(X) :- append(A,B,X),object(A),v(B),
              nl,write('Object - Verb In the wrong order'),nl,
              write('Use the nominative case of '),write(A).

subject_verb(X) :- append(A,B,X),subject(A),v(B),
              nl,write('Subject - Verb in right order'),nl.




append([],L,L).
append([X|L1],L2,[X|L3]) :- append(L1,L2,L3).
```

Government Binding Parser

```
/*  GB.PL  */


%%%%%%%%
%       "Gibberish" -- A GB'ish parser!
%
%   A first attempt at a Government-Binding (GB) type
%   parser.  This parser is intended as an introductory
%   'toy' parser for NLU courses, so complexity will be
%   kept to a minimum while still being general enough
%   to be easily extended.  All constraints have been
%   been implemented as explicit prolog goals for
%   perspicuity.  Many optimizations are possible!  The
%   gapping mechanism actually performs a transformation
%   of the sentence into a 'normal' form and so has been
%   made general enough to move arbitrary structures
%   through the parse tree.
%
%   Author: Cameron Shelley
%   Address: cpshelley@violet.waterloo.edu
%        University of Waterloo
%
/*  GB.PL  */
```

```
%   Comments are welcome!
%
%   This software is released into the public domain on the
%   condition that the author is cited as such, and all
%   modifications remain in the public domain; and this
%   condition is imposed on all subsequent users.
%
%   Modification History:
%   ---------------------
%   Jan 17/91 - creation
%   Feb 4/91  - fixed modal(nil) matching bug in "modal-2".
%   Feb 4/91  - added general conjunction rule "conj".
%            (idea from Steve Green -- thanks Steve!)
%
%%%%%%%%%

parse :- read_sentence(Sentence),
     sentence(Struc,Sentence,[]),
     print_struct(Struc).

%%%%%%%%%
%   'sentence' will parse the basic np,vp structure
%   at the top level.  Different sentence types will
%   be added (ie. questions).
%
%   sentence(
%       Struc  : return structure from sentence call
%       )
%
%%%%%%%%%

%
%   normal sentence
%

sentence(Struc) -->
    noun_phrase(Np,Pers,Nnum,nogap,nogap),
    {Nnum = Vnum},
    verb_phrase(Vp,Pers,Vnum,nogap),
    {Struc = s(Np,Vp)}.

%
%   question with modal transformed to initial position
%

sentence(Struc) -->
    modal(M,_,_,_,_,nogap,_),
    noun_phrase(Np,Pers,Nnum,nogap,nogap),
    {Nnum = Vnum},
    verb_phrase(Vp,Pers,Vnum,M),
    {Struc = q(Np,Vp)}.

%%%%%%%%%
%   'noun_phrase' will parse the various types of np's and
%   should subcategorize between np's and sbars at some point.
%   Also, proper nouns and pronouns can be treated as special
```

```
%   np's in this system.
%
%   noun_phrase(
%       Struc : return structure,
%       Pers  : np 'person' = first | second | third,
%       Nnum  : np 'number' = sing | plur,
%       Gap   : transformed np (if any),
%       Gapout: output gap if Gap not resolved
%   )
%
%%%%%%%%%

noun_phrase(Struc,Pers,Nnum,Gap,Gapout) -->
    [],
    {Gap =.. [np|_]},
    {Gapout = nogap},
    {Struc = Gap}.

noun_phrase(Struc,Pers,Num,Gap,Gapout) -->
    determiner(Det,Dnum),
    {Nnum = Dnum},
    noun_bar(Nbar,Nnum),
    {Pers = third},
    {Gapout = Gap},
    {InStruc = np(Det,Nbar)},
    conj(Struc,InStruc,Nnum,Num,np).

%
%   determiner is the noun phrase specifier
%
%   determiner(
%       Struc : return structure,
%       Dnum  : det 'number' = sing | plur
%   )
%
%   No determiner is considered to pluralize the np, ie:
%   "cats go" but not *"cat goes".  The default could be
%   changed to "all" or "some" if desired.
%

determiner(Struc,Dnum) --> [Word], {lexdet(Word,Dnum)}, {Struc =
det(Word)}.
determiner(Struc,Dnum) --> [], {Dnum = plur}, {Struc = det(nil)}.

lexdet(the,_).
lexdet(a,sing).
lexdet(an,sing).

%
%   noun_bar is here just a noun with arguments.  A treatment
%   of adjectives should be added.
%
%   noun_bar(
%       Struc : return structure,
%       Nnum  : noun 'number' parsed = sing | plur
%       )
%
```

```
noun_bar(Struc,Nnum) -->
    noun(N,Nnum),
    noun_args(Nmod),
    {Struc = nbar(N,Nmod)}.

%
%   mass nouns should be considered as noun_bars in this system!
%

noun(Struc,Nnum) --> [Word], {Nnum = sing}, {lexnoun(Word,_)},
    {Struc = noun(Word)}.
noun(Struc,Nnum) --> [Word], {Nnum = plur}, {lexnoun(_,Word)},
    {Struc = noun(Word)}.

lexnoun(cat,cats).
lexnoun(dog,dogs).
lexnoun(stamp,stamps).
lexnoun(office,offices).
lexnoun(car,cars).
lexnoun(man,men).
lexnoun(chance,chances).
lexnoun(house,houses).
lexnoun(bar,bars).

%
%   noun_args here allows only pp's or nil's.  Handling of
%   embedded sentences can be added as suggested.
%
%   noun_args(
%       Struc : return structure
%       )
%

noun_args(Struc) -->
    prep_phrase(Pp),
    {Struc = n_args(Pp)}.

%noun_args(Struc) -->
%   sentence_bar(Sb),
%   {Struc = n_args(Sb)}.

noun_args(Struc) -->
    [],
    {Struc = n_args(nil)}.

%%%%%%%%%
%   'verb_phrase' will parse off the predicate of a sentence.
%   Auxiliaries could be added as suggested.  Sensitivity to
%   tense would also be handy.
%
%   verb_phrase(
%       Struc : return structure,
%       Mpers : 'person' of subject input to modal,
%       Mnum  : 'number' of subject input to modal,
%       Mgap  : gap (if any) input to modal
%   )
%
%   'Xpers' and 'Xnum' represent constraints passed to the
```

```
%   verb phrase which may be altered by the components and
%   passed to the next component as 'Ypers' or 'Ynum', ie.
%   Mpers ==> Vpers.
%
%%%%%%%%%

verb_phrase(Struc,Mpers,Mnum,Mgap) -->
    modal(M,Mpers,Vpers,Mnum,Vbnum,Mgap,Vbgap),
    verb_bar(Vb,Vpers,Vbnum,Vbgap),
    {InStruc = vp(M,Vb)},
    conj(Struc,InStruc,Mpers,Mnum,vp).

%
%   'verb_bar' parses a verb followed by arguments, if any.
%   Auxiliaries can be handled as specifiers before the actual
%   verb is read.  Subcategorization (Scat) could also be made
%   more detailed.
%
%   verb_bar(
%       Struc : return structure,
%       Pers  : 'person' of subject (check for agreement),
%       Vnum  : 'number' of subject (check for agreement again),
%       Pgap  : transformed np from predicate (if any)
%   )
%
%   {Gapout = nogap} ensures that the parse doesn't end with
%   an unresolved structure being gapped.
%

verb_bar(Struc,Pers,Vnum,Pgap) -->
    verb(V,Pers,Vnum,Scat),
    predicate(P,Pgap,Gapout,Scat),
    {Gapout = nogap},
    {Struc = vbar(V,P)}.

%
%   'modal' accepts the specifier of a vp.  It should be
%   expanded to help compute the mood and tense of the
%   sentence.
%
%   modal(
%       Struc : return structure,
%       Mpers : 'person' of subject np,
%       Vpers : 'person' resulting from 'modal' ("nil" if found),
%       Mnum  : 'number' of subject np,
%       Vbnum : 'number' resulting from 'modal' ("inf" if found),
%       Mgap  : transformed modal (if any),
%       Vbgap : gap resulting from 'modal' (unchanged if found)
%   )
%

modal(Struc,Mpers,Vpers,Mnum,Vbnum,Mgap,Vbgap) --> [Word],
    {lexmodal(Word)}, {Vbgap = Mgap}, {Vbnum = inf},
    {Vpers = nil}, {Struc = modal(Word)}.
modal(Struc,Mpers,Vpers,Mnum,Vbnum,Mgap,Vbgap) --> [],
    {Mgap =.. [modal|[X]]}, {X \== nil}, {Vbgap = nogap}, {Vbnum =
inf},
    {Vpers = nil}, {Struc = Mgap}.
```

```
modal(Struc,Mpers,Vpers,Mnum,Vbnum,Mgap,Vbgap) --> [],
    {Mgap = nogap}, {Vbgap = nogap}, {Vbnum = Mnum},
    {Vpers = Mpers}, {Struc = modal(nil)}.

lexmodal(can).
lexmodal(could).
lexmodal(will).
lexmodal(would).
lexmodal(shall).
lexmodal(should).

%
%   'verb' parses the verb from the input if it is found in
%   the lexicon.  "lexverb" could contain more info on the
%   verb.
%
%   verb(
%       Struc : return structure,
%       Pers  : 'person' of the subject (for agreement check),
%       Vnum  : 'number' of the subject (for agreement check again!),
%       Scat  : SubCATegory of the verb =
%               dt (ditransitive : two objects) |
%               tv (transitive : one object) |
%               iv (intransitive : no objects)
%   )
%

verb(Struc,Pers,Vnum,Scat) --> [Word],
    {Pers \== third; Vnum = plur}, {lexverb(Scat,Word,_,_)},
    {Struc = verb(Word)}.
verb(Struc,Pers,Vnum,Scat) --> [Word],
    {Pers = third}, {Vnum = sing}, {lexverb(Scat,_,Word,_)},
    {Struc = verb(Word)}.
verb(Struc,Pers,Vnum,Scat) --> [Word],
    {Vnum \== inf}, {lexverb(Scat,_,_,Word)},
    {Struc = verb(Word)}.

lexverb(dt,give,gives,gave).
lexverb(tv,have,has,had).
lexverb(tv,see,sees,saw).
lexverb(iv,go,goes,went).
lexverb(tv,want,wants,wanted).
lexverb(tv,drive,drives,drove).

%
%   'predicate' parses the subcategorized dt, tv, or iv arguments
%   of the verb.
%
%   predicate(
%       Struc : return structure,
%       Pgap  : transformed np gap (if any),
%       Gapout: output any unresolved gap,
%       Scat  : SubCATegory to be returned
%   )
%

predicate(Struc,Pgap,Gapout,Scat) -->
    {Scat = dt},
```

```
    noun_phrase(Np1,_,_,nogap,_),
    noun_phrase(Np2,_,_,Pgap,Gapout),
    {Struc = pred(Np1,Np2)}.

predicate(Struc,Pgap,Gapout,Scat) -->
    {Scat = tv},
    noun_phrase(Np,_,_,Pgap,Gapout),
    {Struc = pred(Np)}.

predicate(Struc,Pgap,Gapout,Scat) -->
    {Scat = iv},
    [],
    {Gapout = Pgap},
    {Struc = pred(nil)}.

%%%%%%%%%
%   'prep_phrase' does the obvious.  Gapping could be introduced
%   to handle transformed pp's (but I doubt it :).
%
%   prep_phrase(
%       Struc : return structure
%   )
%
%%%%%%%%%

prep_phrase(Struc) -->
    preposition(P),
    noun_phrase(Np,_,_,nogap,_),
    {InStruc = pp(P,Np)},
    conj(Struc,InStruc,_,_,pp).

preposition(Struc) --> [Word], {lexprep(Word)}, {Struc = prep(Word)}.

lexprep(to).
lexprep(from).
lexprep(by).
lexprep(of).
lexprep(for).
lexprep(with).

%%%%%%%%%
%   'conj' will parse off a conjuction followed by a constituent
%   of category 'Cat'.  The result will be the right sister of
%   the previously parsed structure passed in.
%
%   conj(
%       OutStruc : result structure from conj,
%       InStruc  : previous structure parsed,
%       Arg1     : first constraint on constituent,
%       Arg2     : second constraint on constituent,
%       Cat      : category of new structure to be parsed
%   )
%
%   By McCawley's usage (McCawley 1988, Vol 1 & 2), constituents
%   should only be conjoined to others of the same category; ie.
%   np "and" np, vp "or" vp, etc.  If no conjunction is found
%   (conj-2,3), then the result structure is unchanged.
%
```

```
%%%%%%%%%

conj(OutStruc,InStruc,Arg1,Arg2,Cat) -->
    conjunction(C,Num),
    construct(Constr,Cat,Arg1,Arg2,Num),
    {OutStruc =.. [Cat,InStruc,C,Constr]}.

conj(Struc,Struc,_,_,vp) --> [].
conj(Struc,Struc,Arg,Arg,_) --> [].

conjunction(conj(Word),Num) --> [Word], {lexconj(Word,Num)}.

lexconj(and,plur).
lexconj(or,sing).

%
%   the meaning of the last three args for 'construct' depend
%   on which constituent is being parsed.  For np, the number
%   of the conjoined np is the 'number' of the first conjunction.
%   This is just a convenient heuristic.  For vp, the person
%   and number must still agree across conjunction.  For pp,
%   no such constraints are necessary.
%

construct(Struct,np,_,Num,Num) -->
    noun_phrase(Struct,_,_,nogap,_).
construct(Struct,vp,Pers,Vnum,_) -->
    verb_phrase(Struct,Pers,Vnum,nogap).
construct(Struct,pp,_,_,_) -->
    prep_phrase(Struct).

%%%%%%%%%
%   'read_sentence' provides the ability to get input
%   in a natural fashion by typing in words separated
%   by spaces and terminated with a period.  Adapted
%   from _Prolog and Natural Language Analysis_ by
%   Pereira and Schieber.
%
%%%%%%%%%

read_sentence(Input) :- get0(Char), read_sentence(Char,Input).
read_sentence(Char,[]) :- period(Char),!.
read_sentence(Char,Input) :- space(Char),!,get0(Char1),
    read_sentence(Char1,Input).
read_sentence(Char,[Word|Words]) :- read_word(Char,Chars,Next),
    name(Word,Chars),
    read_sentence(Next,Words).

read_word(C,[],C) :- space(C),!.
read_word(C,[],C) :- period(C),!.
read_word(Char,[Char|Chars],Last) :- get0(Next),
read_word(Next,Chars,Last).

space(32).
period(46).
```

Phrase Structure Diagram Printer

```
/*  PRETTY.PL  */


%
% From: psmielke@lotus.waterloo.edu (Peter Mielke)
% Thanks, Pete!
%
% print a structure tree
%
% eg.  print_struct(a(b,c,d(e,f))) gives:
%
% a(
%     b,
%     c,
%     d(
%         e,
%         f
%     )
% )
%
print_struct(Tree) :-
    Tree =.. List,
    p_struct(0,List), nl.

%
% the first element of the list is the name of the "functor"
% from the =.. command
%
p_struct(_,[]).
p_struct(Num,[Head|List]) :-
    is_sarg(List), !,
    nl, tab(Num), write( Head), put(40), % left bracket
    put(32), get_head(Element,List),write(Element), put(32), put(41).
p_struct(Num,[Head|List]) :-
    nl, tab(Num), write( Head ), put(40), % left bracket
    Num2 is Num + 4,
    p_list(Num2,List).

%
% a lousy way to see if the function only has a single arguement
%
is_sarg([]).
is_sarg([Head|Tail]) :-
    Tail = [],
    atom( Head ).

%
% an even lousier way of getting an element out of a single list
(i.e. [a] )
%
get_head( Head, [Head|_]).

%
% prints out the list of terms for a "functor"
```

```
%
p_list(_,[]) :-
    put(41). % finishing right bracket
p_list(Num,[Head|List]) :-
    atom(Head), !,
    nl, tab(Num), write( Head ), p_comma(List),
    p_list(Num,List).
p_list(Num,[Head|List]) :-
    Head =.. Sublist,
    p_struct(Num,Sublist),
    p_comma(List),
    p_list(Num,List).

%
% print a comma only if we are NOT at the end of the list
%
p_comma([]).
p_comma(_) :-
    put(44).
```

URLS of interest

below you will find a selection of Internet sites with interesting material relating to this project.

Http://www.uni-verse.com/	Uni-Verse Real-time Translation
http://www.systransoft.com/	SYSTRAN
http://www.nyu.edu/pages/linguistics/	Linguistics at New York University
http://almond.srv.cs.cmu.edu/afs/cs/project/ai-repository/	AI repository
http://bobo.link.cs.cmu.edu/dougb/playground.html	The Natural Language Playground
http://linguistlist.org/	The Linguist list
http://ciips.ee.uwa.edu.au/~hutch/research/seminars/talk2/sld001.htm	How to pass the Turing test
http://www.dcs.ex.ac.uk/~masoud/Yazdani/course/prolog/	A Prolog course

Bibliography

The word crunchers. The Guardian On-line supplement November 27 1997 (page 3)

Akmajin, Demers et al. Linguistics: An Introduction to Language and Communication, Fourth Edition, M.I.T. Press, 1995

Barr & Feigenbaum. The Handbook of Artificial Intelligence: Volume 1, Pitman Books Limited 1983.

Barr & Feigenbaum. The Handbook of Artificial Intelligence: Volume 2, William Kaufmann Inc. 1982.

Blake. Case. Cambridge University Press 1994.

Boden. Artificial Intelligence and natural Man: second edition, expanded. The M.I.T. Press 1987.

Bratko. PROLOG: Programming For Artificial Intelligence, second edition. Addison-Wesley 1991.

Cook & Newson. Chomsky's Universal Grammar: An Introduction, second edition. Blackwell 1996.

Dougherty. Natural Language Computing: An English Generative Grammar in Prolog. Lawrence Erlbaum Associates, 1994.

Gazdar & Mellish. Natural Language Processing in PROLOG: An Introduction to Computational Linguistics. Addison-Wesley, 1989.

Green & Coulson. Language Understanding: Current Issues, Second Edition. Open University Press, 1995.

Hutchins & Somers. An Introduction To Machine Translation. Academic Press, 1992.

McHale & Myaeng. Integration of conceptual graphs and government-binding theory. Butterworth-Heinmann. 1992

Pinker. The Language Instinct. Penguin 1995.

Russell & Norvig. Artificial Intelligence: A Modern Approach. Prentice-Hall. 1995.