M.Phil. Project
Computer Speech and Language Processing

Simplistic Question Answering

Matthew Purver
Sidney Sussex College
University of Cambridge

October 9, 2000

Abstract

The aim of this project was to develop a system that could identify text passages which answer a question. The approach taken used ideas from various participants in the recent TREC-8 conference, and added the use of notions of sentence structure - particularly structural transformation and matching. The system was successfully tested on training data incorporating a wide range of sentence structure phenomena. Performance was evaluated in blind tests on two sets of data and results were encouraging, with good levels of both recall and precision being achieved.

Contents

I Thesis **6**

1 Introduction **7**
- 1.1 Overview . . . 7
- 1.2 Question-Answering . . . 7
- 1.3 The State of the Art . . . 9
  - 1.3.1 Early Work in QA . . . 9
  - 1.3.2 The TREC-8 QA Track . . . 9
  - 1.3.3 Possible Improvements . . . 11
- 1.4 Description of Project . . . 12
  - 1.4.1 Objectives and Approach . . . 12
  - 1.4.2 System Overview . . . 12

2 Shallow Text Processing **14**
- 2.1 Overview . . . 14
- 2.2 The Front End . . . 15
  - 2.2.1 Part-of-Speech Tagging . . . 15
  - 2.2.2 Simple Noun Phrase Bracketing . . . 16
  - 2.2.3 Rewriting . . . 16
  - 2.2.4 Stemming . . . 17
  - 2.2.5 Simple Noun Phrase Adjustment . . . 17
- 2.3 Syntactic Processing . . . 17
  - 2.3.1 Verb Groups . . . 17
  - 2.3.2 Noun Groups . . . 19
  - 2.3.3 Prepositional Phrases . . . 19
  - 2.3.4 Output . . . 20
- 2.4 Semantic Feature Attachment . . . 20
  - 2.4.1 Verb Group Semantics . . . 20
  - 2.4.2 Noun Group Semantics . . . 21
  - 2.4.3 Prepositional Phrase Semantics . . . 22

3 Relational Structures **24**
- 3.1 Coreference . . . 24
  - 3.1.1 Entity Indexing . . . 25

3.1.2 Coreference Resolution 25
3.2 Extraction of Sub-sentential Units 26
3.2.1 Conjunctions 27
3.2.2 Subordinate/Relative Clauses 28
3.2.3 Punctuation 28
3.3 Structure Extraction 28
3.3.1 Predicate-Argument Structures 28
3.3.2 State Structures 29
3.4 Coindexing 29

4 Structure Matching 31
4.1 Direct Structure Matching 32
4.1.1 Word Matching 32
4.1.2 Verb Group Matching 33
4.1.3 Noun Group Matching 34
4.1.4 Modifier Phrase Matching 35
4.2 Structure Transformation 37
4.2.1 General Structural Transformations 37
4.2.2 Specific Structural Equivalences 39
4.3 Question-word Matching 40
4.4 Summary of Capabilities 42
4.4.1 Answer Passage Phenomena 42
4.4.2 Query Types 44
4.4.3 Problematic Phenomena 44

5 Evaluation and Results 48
5.1 System Output 48
5.2 Training Data 49
5.3 Evaluation Method 49
5.3.1 Manual Annotation 49
5.3.2 Performance Measures 50
5.3.3 Blind Testing 50
5.4 Results 50
5.4.1 Training Data 50
5.4.2 First Blind Test 51
5.4.3 Second Blind Test 52
5.4.4 TREC-8 Comparison 53
5.5 Conclusions 55

II Appendices 60

A Query/Answer Corpora 61

A.1 Training Set 61
A.2 First Blind Test Set 68
A.3 Second Blind Test Set 70

B Code Listing 72

B.1 Perl/Shell Scripts 72
B.1.1 Perl Code for OALD Pre-processing 72
B.1.2 Perl Scripts for Test Data Pre-processing 74
B.1.3 Shell Scripts for Shallow Text Processing 74
B.2 Prolog Lexical Resources 76
B.2.1 OALD Resources 76
B.2.2 Hand-Coded Lexicon 77
B.3 Prolog Code 81
B.3.1 Top Level 81
B.3.2 Structural Matching/Transformation 90
B.3.3 Structural Extraction 101
B.3.4 Shallow Text Processing 125
B.3.5 Miscellaneous 168

List of Figures

1.1 System Overview 13

2.1 Shallow Text Processor: Front End 15
2.2 Example A1 after PoS tagging & NP bracketing 16
2.3 Example A1 after pre-processing 17
2.4 Shallow Text Processor: Main Functions 18
2.5 Example A1 after syntactic processing 20
2.6 Example A1 after shallow text processing 23

3.1 Example Q after entity indexing 25

List of Tables

2.1 NG semantic classes 22
2.2 PP semantic classes 22

4.1 Wh-query word semantic classes and phrase types 41
4.2 Lexical/Structural Phenomena 43
4.3 Query Types 44
4.4 Problematic Phenomena 45
4.5 Problematic Query Phenomena 45

5.1 Performance on Training Data 51
5.2 Performance in First Blind Test 52
5.3 Performance in Second Blind Test 53

Part I

Thesis

Chapter 1

Introduction

1.1 Overview

The aim of this project was to develop a question-answering (QA) system that makes use of sentence structure. Conventional information retrieval (IR) techniques, which do not take structure into account, are not ideally suited to QA - in terms both of determining whether a passage really answers a query, and of delivering a satisfactory answer.

A system was developed which used sentence structure to extract simple relations between words and phrases. These relations could then be matched between query and answer passages, thus both ensuring that a true answer is present, and identifying the essential matching words for use as an answer. Chapters 2 and 3 describe the text processing and relation extraction, and chapter 4 describes the matching process.

The system was evaluated over a wide range of query types and answer sentence phenomena, and had its final performance measured by blind testing. This process, together with the results, is presented in chapter 5.

The complete sets of training and test data, together with the program code, are given in a separate volume (appendices A and B).

In this chapter, the problems of QA for IR systems are described, and a summary of the state of the art is given. I then explain the approach taken in this project.

1.2 Question-Answering

Conventionally, work in the field of IR has concentrated on ad-hoc document retrieval (DR), to the extent that the terms are often used synonymously. In DR, a set of documents is retrieved in response to a user's query, usually ranked in order of relevance. This process will be familiar to anyone who has used a WWW search engine.

Conventional DR systems use statistical methods based on the relative frequencies of keywords in the query and in candidate documents. While attempts have been made to apply natural language processing (NLP) techniques to DR (see e.g. [21], [38]), they have not found their way into widespread use due to the high effectiveness and speed of simple statistical methods.

In more recent years, however, attention has begun to be focussed on the related task of question-answering (QA) from document collections. In contrast to DR, where complete *documents* or *passages* are retrieved and ranked by relevance, QA requires that the system give an *answer* to the user's query. As pointed out in the specifications of the Eighth Text Retrieval Conference (TREC-8) QA Track [32], this is highly desirable:

> Current information retrieval systems allow us to locate documents that might contain the pertinent information, but most of them leave it to the user to extract the useful information from a ranked list. This leaves the (often unwilling) user with a relatively large amount of text to consume. There is an urgent need for tools that would reduce the amount of text one might have to read in order to obtain the desired information. [...]
>
> People have questions and they need answers, not documents. Automatic question answering will definitely be a significant advance in the state-of-art information retrieval technology.

The QA task poses two major problems for standard DR techniques, which we can understand by considering the following example query:

Q *Who is the president of the USA?*

together with the following sentences which might be selected by a DR system searching for an answer (as all contain the keywords *is the president of the USA*):

A1 *Bill Clinton is the president of the USA.*
A2 *Even if one dislikes Bill Clinton, it is important to remember that although many people do not agree with his policies, he is one of the most powerful men on earth: he is the leader of the free world and the president of the USA.*
A3 *Hillary Clinton is the wife of the president of the USA.*

Example A1 does not seem to pose a problem: the sentence answers the question and could even be returned whole as an acceptable answer.

Example A2 shows the first problem: what do we return as the answer? Returning the whole sentence does not seem to be acceptable, and if we apply a window to reduce the amount of text returned, there is no guarantee that we will capture the essential words *Bill Clinton* due to the distance from the keywords.

Example A3 shows the second problem: although it matches the keywords, it does not contain an answer at all.

1.3 The State of the Art

1.3.1 Early Work in QA

There has been interest in natural language QA systems for many years, but until recently the emphasis of the majority of the systems developed was on answering questions from a structured knowledge-base (see e.g. Lehnert [18], [19]). As a result, work tended to centre around full syntactic and semantic analysis of both queries and answers, together with AI techniques such as theorem-proving to extract answers from the knowledge-base (Levine & Fedder [20] give a succinct overview of this type of approach).

While this may still be a sensible approach for situations where the body of information is structured, time and processing constraints (together with a large degree of unpredictability of format and content) prevent these methods from being used in the case of QA from large unstructured collections of documents (such as the WWW).

1.3.2 The TREC-8 QA Track

In 1999, the DARPA-sponsored Text Retrieval Conference (TREC) set up a QA track for its eighth conference (TREC-8). A description of the track is available at [31], with the detailed specifications at [32]. The overall aim was to investigate the possible approaches that could be used to produce a fully automatic QA system which could provide answers to questions, given an unstructured corpus of documents which contained those answers.

The participating systems were tested by their performance on 200 test questions, with each question guaranteed to be answered by at least one document in the corpus. For each question, systems could return a ranked list of 5 possible answer strings. These strings were portions of the surface text, of limited length (two categories of answers were permitted, with the answer text length limited to either 50 or 250 bytes – so approximately 6 or 30 words). The answers were then assessed and scored by human judges. A more detailed description of this process is available in [41].

A variety of approaches were used by the participants, with varying degrees of success. Most of the systems can be classified as follows:

Pure DR-based Systems

The simplest systems used DR techniques to retrieve the most relevant text passages based on the keywords in the query. Examples include the systems used by the University of Massachusetts [3], the University of Waterloo [10], and National Taiwan University [22], as well as the passage-based runs of AT&T [2] & [33]. Various methods were used to determine which query words should be used for

retrieval, and various DR techniques were employed, but no true NLP techniques were used for either query or potential answer passage processing.

These systems performed relatively well when retrieving 250-byte passages, but less well when restricted to 50 bytes as there was no guarantee that the passage selected would contain an answer as well as the query keywords: this is the problem we observed with our example A2 above.

Query Processing and Named Entity Extraction

The majority of systems used the same approach as an initial stage to retrieve documents or shorter passages that contained the answer, but also used some NLP techniques to process the query and to identify candidate answer entities. These included the entity-based runs of AT&T [2] & [33] and the systems of Xerox [15], Southern Methodist University [25], GE/University of Pennsylvania [26], NRC/University of Ottawa [24], the University of Iowa [12], MITRE [5], LIMSI-CNRS [13] and IBM [29] & [30].

While the exact techniques used differed, most had the following approaches in common:

Shallow Query Parsing The query text is tagged and parsed using a shallow text processor, resulting in a parse tree structure containing (at least) noun phrases.

Query Type Identification The result of parsing is examined to determine the type of answer that the query is expecting. This type is defined as being a member of a set of semantic classes such as, amongst others, *OBJECT*, *PERSON* and *TIME*. This identification is generally given by a particular question word (e.g. *Who?* shows the question expects an answer of type *PERSON*) or series of words (e.g. *How long?* might expect *DISTANCE* or *DURATION*), but may also require identification of a head noun phrase (e.g. *"Which is the largest city in Germany?"* might be processed to a resulting type of *CITY*).

Entity Extraction The candidate answer passages are similarly processed to identify noun phrases with their corresponding semantic types. Those that match the class expected by the query are chosen as possible answers. Various methods, generally based around frequency and relative position in sentences, are then used to rank the possible answer entities.

This approach provided some of the best-performing systems in TREC-8, but is still subject to our second problem above – non-answers can be mistakenly selected. There is also a reduced version of the first problem: if two entities of the correct type are present in the answer, there is no guarantee that relative distance from keywords will select the correct one.

IE-based Systems

A few of the systems (NTT Data [39], Cymfony [35] & [36], the University of Sheffield [16], New Mexico State University [28]) viewed the problem as one of Information Extraction (IE). In IE, the central aim is to extract from the text certain known types of information about entities, and this is generally achieved by attempting to fill entries in a pre-generated template.

In practice, these systems operated in a similar manner to those described in the previous section, as they are limited to what Srihari & Li [35] call Named Entity IE (named entity extraction). However, as they point out, this approach does reveal a possible next step: the use of what they call General Event IE. This would involve the extraction of general predicate-argument relations to fill templates with pre-defined slots (but no set of pre-defined values for the predicate).

Other Approaches

Two systems attempted to parse the answer passages and use the parse tree structure in some way. The system developed by the University of Maryland [27] used a shallow parser to produce a dependency tree for both query and answer. Entities in the answer then had to be nodes in the tree connected by a path in the same order as the corresponding entities in the query. However, as neither direction nor path length were required to match, sentences like our example A3 would not be prevented from being chosen.

The CL Research system [23] used the notion of *semantic triples*. This approach involved the identification of entities together with their roles in the sentence (subject, object, location etc.). However, performance was degraded by the necessary use of a deep (and non-robust) parser, as well as by the lack of coreference resolution. It was also hindered by its method of reporting: by reporting full sentences instead of attempting to identify an answer entity or summarise the structure, it suffered from the same problem as DR systems in that answers were often lost when truncating to the TREC length limit.

1.3.3 Possible Improvements

Many of the TREC participants cite the following possibilities for improvement: better DR, coreference resolution, use of structural relations and treatment of synonyms. In this project I hope to show that some of these improvements are possible.

In particular, most of the TREC systems make no use of answer sentence structure, and so have no way of distinguishing between answers (say, examples A1 and A2 above) and non-answers (like A3), or to choose the correct answer entity from a number of possibilities. Those that did attempt to use parse structure were either not robust or did not enforce structural relations fully.

1.4 Description of Project

1.4.1 Objectives and Approach

The aim of my project was therefore to investigate the use of sentence structure in answer identification, by attempting to match simple structures extracted from both query and candidate answer passages. The intention was to combine the common query type/semantic class approach with some shallow structure information and then devise techniques for matching these structures.

Due to the restricted time available, certain necessary but existing-technology elements of a real-world system were simulated rather than developed. The initial identification of candidate answer passages containing question keywords through standard DR techniques was assumed, by only considering such passages (hereafter referred to as *answers*). The identification and classification of unknown named entities (e.g. proper names) was also assumed, by adding all entities to a lexicon.

For the same reason, it was also decided not to attempt to deal with certain classes of query/answer passage phenomenon: those that deal with tense and thus require knowledge of document creation time; and those that require inference and real-world knowledge.

1.4.2 System Overview

The system has the following overall structure (see figure 1.1):

Shallow Text Processing The query and answers are PoS-tagged and parsed to give a shallow tree structure, and semantic information is attached.

Structural Relation Extraction The resulting tree is subjected to coreference resolution and sub-sentential unit identification before simple structures are extracted.

Structural Matching The system attempts to match the query and answer structures, using a combination of plain matching rules and structural transformations. Matching portions are then selected and given as output.

These modules are described in chapters 2, 3 and 4 respectively. Where possible, their operation is illustrated by use of the example query and answers given above - although other examples are used to illustrate certain phenomena.

The system was developed and tested using a wide variety of query types and answer phenomena. Final performance was evaluated by blind testing: the evaluation process and results are detailed in chapter 5.

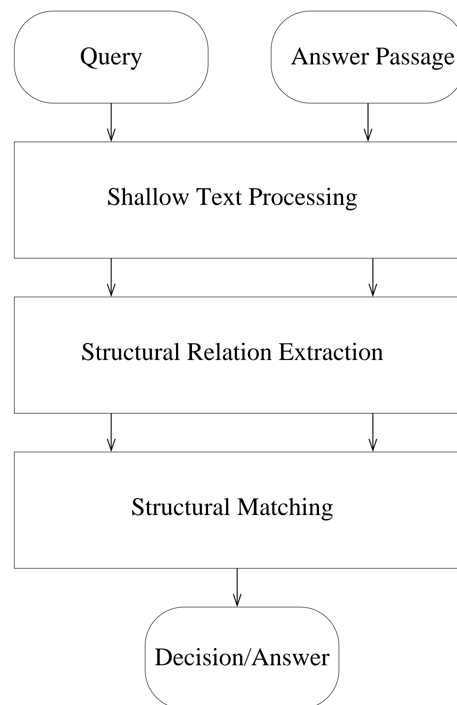

Figure 1.1: System Overview

Chapter 2

Shallow Text Processing

2.1 Overview

A suitable text processor was required to extract information about both syntactic structure and lexical semantics from the query and answers. A shallow parser was preferred, due to its inherent robustness and to the fact that full parse tree information was not required.

Various shallow parsers are available, and both FASTUS (see [4]) and TTP (see [37], [38]) were investigated. However, as the requirements in terms both of parse tree complexity and semantic features were not precisely known at the start of the project, a processor was independently developed specifically for the task.

A finite-state transducer design was used, coded in Prolog (and incorporating an external part-of-speech (PoS) tagger). It is non-deterministic: several possible parse trees can be extracted in the case of ambiguity. The output consists of noun groups (NGs) and verb groups (VGs) under a single sentence node, with modifiers (prepositional phrases (PPs) and adverb groups) optionally attached to these groups.

In addition to syntactic structure, the processor also attaches certain lexical semantic features to the groups (including semantic classes as described in the previosu section). These are required for later matching with question words and structures, and are also used in coreference resolution.

This chapter describes of the operation of the processor: section 2.2 describes the front end (tagging, stemming and NP grouping); section 2.3 describes the syntactic processing; and section 2.4 describes the semantic processing. The process is illustrated by use of example answer A1 from the previous section: the processing of queries is very similar. Code is given in full in the second volume, appendix B.

2.2 The Front End

Before full syntactic processing, text is PoS tagged and stemmed. Simple rewriting rules are used to expand abbreviations, and simple noun phrases (NPs) are formed. This stage is described in detail in this section (and illustrated in figure 2.1). The output is a list of tagged words and NPs as shown in figure 2.3.

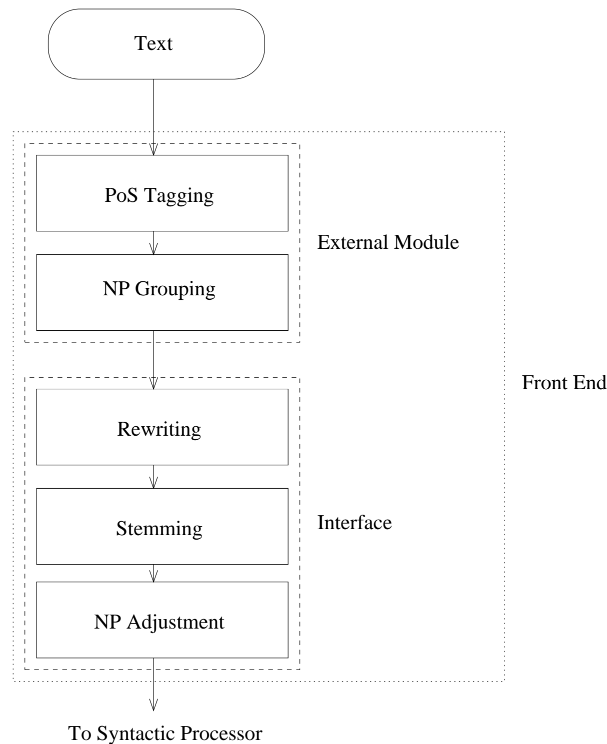

Figure 2.1: Shallow Text Processor: Front End

2.2.1 Part-of-Speech Tagging

The tagger used is `shallowproc`, developed at Cambridge University by Sylvia Knight and described in [17]: it uses its own tagset developed from the Wall Street Journal corpus which contains useful extensions from more standard tagsets, such as new tags for prepositions and conjunctions.

As the tagger is an external module called from Prolog, any PoS tagger could be used as long as it could be trained on the same tagset: for example, a version of the Brill tagger is available.

2.2.2 Simple Noun Phrase Bracketing

An additional feature of `shallowproc` is its ability to group words to form simple NPs according to a list of rules (essentially groups of determiners, adjectives, nouns, proper nouns, numbers and pronouns). While the results of this grouping process were not ideal for the purposes of this project, it was considered quicker to use this feature and modify the results (see below) than build a more suitable NP grouper from scratch.

More features were available (more complex NP identification, verb subject identification) but were not used as they were often found to be inaccurate.

The output of `shallowproc`, a list of tagged words together with brackets to indicate NPs (see figure 2.2), is then piped directly into Prolog.

```
(NP
Bill NN
)NP
(NP
Clinton NP
)NP
is VBZ
(NP
the DET
president NN
)NP
of PREP
(NP
the DET
Usa NP
)NP
. .
```

Figure 2.2: Example A1 after PoS tagging & NP bracketing

2.2.3 Rewriting

After reading in this output, converting to lower case and attaching PoS tags to words to form compound Prolog terms, some rewriting rules are applied.

Known abbreviations are expanded using information from the Oxford Advanced Learner's Dictionary (OALD) (e.g. *it's* → *it is*). Hand-coded rules are also used to build multi-word units (e.g. *such as*) and to correct some common mistaggings.

2.2.4 Stemming

Nouns, verbs, adjectives and adverbs are stemmed to their root form. This is required in order to enable later stages to match queries and answers containing words with different surface forms (e.g. matching different verb formations with each other, singular nouns with their plural equivalents, comparative adjectives with their root forms).

A dictionary-based stemmer was written. The OALD was used as it provides inflectional information, and a set of rules was composed for each of its inflectional types. Irregular words were extracted from the dictionary using Perl scripts and then simply listed in all forms. An exception was made for many noun plurals described as irregular by the dictionary, but which could be captured by a number of rules, avoiding a full listing.

2.2.5 Simple Noun Phrase Adjustment

After stemming, the NPs produced by `shallowproc` are adjusted to better suit the purposes of the system. As adverbs are not included in NPs, they have to be adjusted to allow for cases where an adverb modifies an adjective (*"an extremely fat man"* would be given the form *"[an] extremely [fat man]"*, which is then adjusted to *"[an extremely fat man]"*). The existential *there* is removed from NPs, as it is treated as part of verb groups by later stages. Question-words are also removed from NPs as they are treated differently during later stages.

Once these pre-processing stages are complete, the resulting Prolog list of tagged, stemmed words (see figure 2.3) is passed to the main parsing functions.

```
[np:[bill/nn],np:[clinton/np],be/vbz,np:[the/det,president/nn],
                              of/prep,np:[the/det,usa/np],'.'/'.']
```

Figure 2.3: Example A1 after pre-processing

2.3 Syntactic Processing

This stage of the processor builds NGs, VGs and PPs under a single S node. These groups consist of Prolog lists with a phrase type marker prefix. The output is a shallow tree as shown in figure 2.5.

2.3.1 Verb Groups

The first step is to build VGs: these consist of verbs with associated function words, together with modifiers such as adverbial groups.

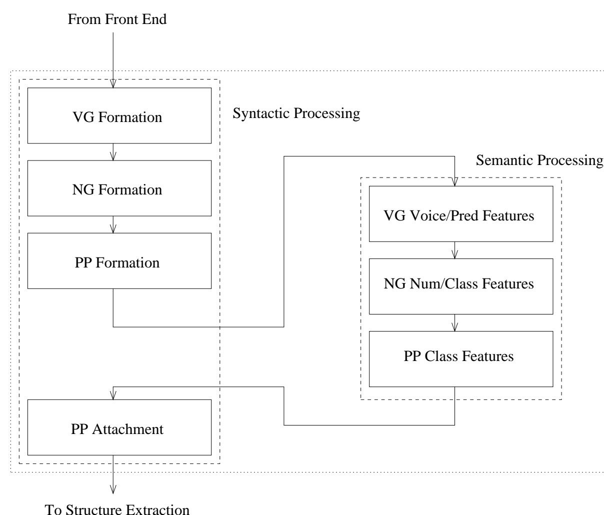

Figure 2.4: Shallow Text Processor: Main Functions

Basic VGs

Strings of consecutive VG-type words are grouped together. This includes: all verbs (including modals); all adverbs not already assigned to NPs; complementisers (such as *that*, *whether*); the infinitive *to*; and the existential *there* (although in answers, only when followed by the verb *be* - it is otherwise assumed to be a mistagged adverbial *there*).

Adverb Attachment

Any VGs resulting from the previous stage that contain only adverbs are attached to the nearest VG as modifiers: *"he [drove] the car [quickly]"* → *"he [drove quickly] the car"*. Adverbs are given a separate `adv:` sub-group within a VG: the actual form of the VG in this example would be
`vg:[v:[drive/vbd],adv:[quickly/rb]]`.

Adverb Grouping

Any adverbs left within the main `v:` sub-group are moved to the `adv:` sub-group.

Question-verb Combination

Queries often have an inverted surface syntactic form. The inversion is removed by combining the auxiliary VG with the main VG: *"[does] he [drive] the car"* → *"he [does drive] the car"*). In the case of conjoined VPs (e.g. *"does he [drive] the car and [walk] the dog"*), the auxiliary is combined with both following VGs.

Movement of wh-query words due to inversion is also removed at this stage: *"what [does] he [drive]"* → *"he [does drive] what"*.

2.3.2 Noun Groups

The next stage groups simple NPs together, either as conjunctions or compounds, to form NGs. VGs acting as noun modifiers can also be included.

Compound NPs

Consecutive NPs containing only a single noun-type word are combined - these may be either compound nouns (*banana production*) or proper names (*Bill Clinton*). Possessives are also combined here (the PoS tagger returns the possessive adjunct *'s* as a separate word), as are some adjectival forms such as the common (in QA) *how long/old/etc.*.

The formation of these compounds is currently not compulsory to avoid forcing errors, but it might be acceptable to enforce it to prevent unnecesary time being spent in later backtracking.

Conjoined NPs

Lists of conjoined NPs (separated by conjunctions or commas, and possibly including phrases such as *etc.*) are combined. This combination is not compulsory, as mistakes could be made: we want to use it in cases like *"Bill likes [Jane and Mary]"*, but not in *"Bill likes [Jane and Mary] likes John"*.

An additional constraint imposed to prevent erroneous combination is that all NPs in the list must belong to the same semantic class (see below): this worked for all examples used, but might not be desirable in all cases (*"cabbages and kings"*, *"arms and the man"*).

VG-NP Combination

VGs consisting of single-word present participles or of *to*+infinitives were combined with the NP following them (*singing policemen*).

2.3.3 Prepositional Phrases

PPs are formed by combining a preposition with the following NP/NG: `pp:[of np:[the USA]]`.

Once formed, PPs are non-deterministically attached to NGs and VGs. This stage is actually delayed until after semantic feature attachment (see next section) so as to allow some degree of determinism: only PPs of certain classes (e.g. time) can be attached to VGs.

2.3.4 Output

The output of this stage is therefore a sentence with words assigned as far as possible to NGs, VGs and PPs. Function words (such as conjunctions) remain outside groups, as do most wh-question words and punctuation.

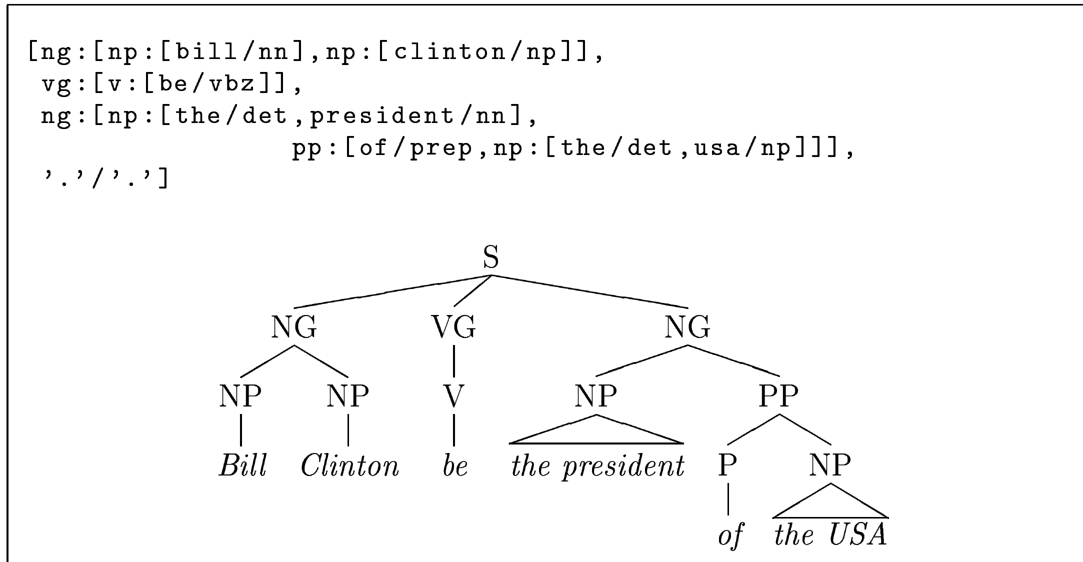

Figure 2.5: Example A1 after syntactic processing

2.4 Semantic Feature Attachment

Once grouping is complete, the heads of groups are identified and various semantic features are attached. For NGs, these features are number (singular/plural) and semantic class; for PPs, semantic class only. In both cases, the features are required for coreference resolution and for matching against query words. For VGs, the features are voice (active/passive) and predicate name: these are required for structural matching.

2.4.1 Verb Group Semantics

VG features are attached as a compound term between phrase type marker and phrase list: `vg:VOICE#PRED:[...]`

Voice

Voice is represented as a Prolog atom: `act` or `pas`. A VG is taken to be passive if it includes a verb such as *be*, *become* followed by a past participle (*"John [is liked] by Mary"*), or includes only a past participle (*"John, [liked] by Mary, ..."*).

Predicate

Predicate name is also atomic, and is the stem of the head verb in the VG. The head verb is taken to be the last verb in the VG, except in the case where it is preceded by a verb like *want*, where this preceding verb is taken instead. This prevents a VG such as *"want to X"*, *"intend to X"* being taken to have the meaning *X*. The list of such verbs is specified manually in a lexicon.

2.4.2 Noun Group Semantics

Again, NG features are attached as a compound term between phrase type marker and phrase list: `np:NUMBER/CLASSLIST:[...]` Both features are taken from the head of the NG: this is taken to be the last noun (including numbers) in the phrase, or just the last word if no nouns are found.

Number

Number is represented as an atom: either `s` or `p`. Any NG whose head has a plural PoS tag is marked as plural, as are NGs that are lists of conjunctions; otherwise this feature defaults to singular.

Semantic Class

This feature is represented as a list of possible class names (each NG might have more than one possible class). If the head of a NG is not a noun, a dummy value is assigned.

Most TREC-8 participants identified a number of classes which corresponded to various question types. A similar approach was taken here, although less fine-grained than most of the TREC systems: the number of classes was kept to the minimum required to satisfy the demands of coreference resolution and the query types of the examples used. The possible NG classes are shown in table 2.1. The "person" class also had an attached gender feature, to aid in coreference resolution.

A real-world system might require a wider range of classes: for example, many subdivisions of the "number" class might be required such as "date", "length" and "money".

The OALD provides names of many common towns, cities and countries, so was used to look up classes for these words. It also provides many common personal names, but does not provide gender information, so all other classes were simply listed in a hand-built lexicon. This would become cumbersome for a large system: a neater solution for dictionary words might be to use a hierarchical dictionary such as the Cambridge University Press CIDE+. However, this would still leave the problem of being unable to deal with unseen words (extremely

Class	Description	Examples
`per/m`	Person (male)	Mr., Mike
`per/f`	Person (female)	queen, Mary
`per/_`	Person (either)	student, candidate
`obj`	Concrete object	banana, kitchen
`abs`	Abstract object	election, answer
`loc`	Location	Wales, Cambridge
`org`	Organisation	university, Microsoft
`num`	Number	1991, length
`xxx`	Non-noun	(e.g. adjectives)

Table 2.1: NG semantic classes

common in news text, where new names occur frequently), so a named entity extractor (of the type now state-of-the-art) would have to be used.

2.4.3 Prepositional Phrase Semantics

For PPs, class must depend both on the preposition and the class of the subsequent NG: while *in Wales* is a location, *about Wales* is not, and neither is *in moderation*. Based on the analysis in [34], a set of classes for prepositions was created, and some simple rules devised to determine whether these could be carried over to form the overall class of the PP:

Class	Description	Examples	Required NG Class
`loc`	Location	in, outside	`loc`, `obj`
`tim`	Time	in, before	`num`
`man`	Manner	by	(anything)
`pos`	Possession	in, of	(anything)
`sop`	"Inverse" possession	with, without	(anything)
`xxx`	Unknown	(anything)	(anything)

Table 2.2: PP semantic classes

Any combination that does not fit a rule is given the "unknown" class: at present, due to the narrow coverage of the small set of rules, this applies to the majority of PPs. The rules have been chosen only to cover PPs that can be answers to "where" and "when" questions (together with aspects of possession, which were also determined to be useful). Classes for prepositions are listed in a hand-created lexicon: as there is a finite (and small) number of them (prepositions are closed-class words), this seems reasonable.

```
[ng:s/[per/m]:[np:s/[per/m]:[bill/nn],
               np:s/[per/m]:[clinton/np]],
 vg:act#be:[v:[be/vbz]],
 ng:s/[per/_]:[np:s/[per/_]:[the/det,president/nn],
               pp:[loc,pos]:[of/prep,
                             np:s/[loc,pos]:[the/det,usa/np]],
 '.'/'.']
```


Figure 2.6: Example A1 after shallow text processing

Chapter 3

Relational Structures

After the processing described in chapter 2, we have a shallow parse tree: a sentence consisting of verb and noun groups. In order to make a decision as to whether an answer truly matches a query, we need some information about how these groups are related (i.e. about their roles in the sentence). This is important if we want to exclude examples like A3 from being identified as correct answers.

In the TREC-8 conference, two systems made clear attempts to do this. The University of Maryland system [27] used a parser that produced a dependency tree encapsulating relational information; while the CL Research system [23] extracted entities and stored them together with direct information about their role (e.g. subject or object and the verb governing them).

A similar (if more simplistic) approach to the CL system was taken here, with simple relational structures being extracted. These structures can either convey a general predicate-argument relation (for example, the subject-verb-object relation *"X likes Y"*) or an existential relation (*"X is Y"*). The form of these structures is described in section 3.3.

In order that these structures carry all the information required, we first need to go through a process of coreference resolution, described in section 3.1. Complex sentences are then syntactically simplified (see section 3.2), to allow the structures to be easily extracted (section 3.3).

Finally, I describe a method of coindexing which allows information to be shared between simple structures extracted from a complex sentence (section 3.4).

3.1 Coreference

Before attempting to identify structures, we attempt to resolve anaphoric expressions, including pronouns, proper and definite NGs, and expressions of quantities, time and location. This is vital in cases such as our example A2, where we need to establish that *he* refers to *Bill Clinton*: extraction of the structure corresponding

to *he is the President of the USA* will not help us unless this has been done.

3.1.1 Entity Indexing

All NGs are asserted in the Prolog database as possible referent entities, and replaced in the sentence by index markers pointing to these entities. An exception is made for NGs which contain only pronouns, as we do not wish to use them as referents.

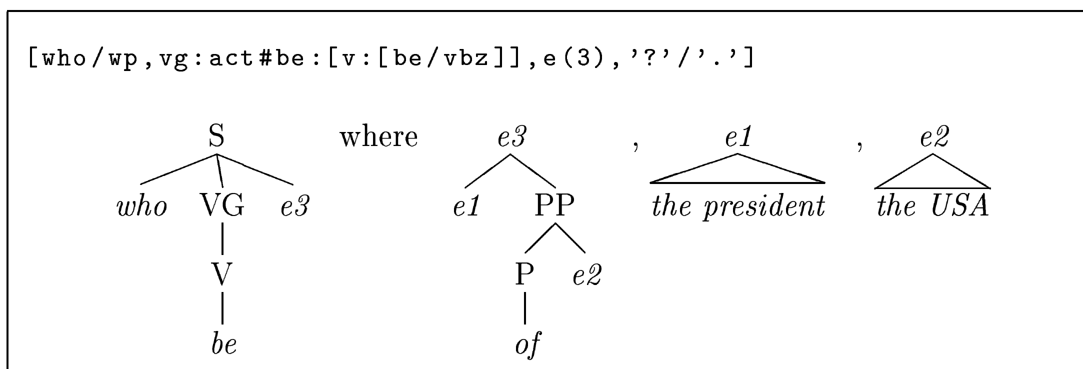

Figure 3.1: Example Q after entity indexing

3.1.2 Coreference Resolution

Anaphoric expressions can now be replaced with suitable referents, if these can be found.

Pronouns

Pronouns are listed in a lexicon with details of the allowable semantic class, gender and number of their possible referents. Some pronouns need than one entry: *they* can refer to a singular organisation (*"Microsoft announced today that they have bought China"*) but must refer to a plural entity of any other class. Each indexed entity is examined in turn: if features match, the pronoun is replaced by the index marker pointing to this entity. In example A2, this results in *he* (both occurrences) being replaced by the index marker pointing to the entity *Bill Clinton*.

Prolog backtracking ensures that all possible referents can be considered in turn. Certain personal pronouns (such as *I*, *you*) are not resolved as they are both unlikely to have referents in the sentence and unlikely to be helpful in terms of QA.

Proper names

Proper NGs can be resolved to other proper NGs which contain the same or more information (e.g. also including an attached PP): the index marker for the anaphoric NG is changed to point to the referent NG.

This is required in cases such as the matching of *"Today Ken Livingstone opened London's new assembly; Livingstone is London's mayor"* to the query *"Is Ken Livingstone the mayor of London?"* - we need to resolve *Livingstone* to *Ken Livingstone*.

The referent NG is not allowed to be a list of conjunctions: we would not want *Livingstone* to be resolved to *Ken Livingstone and Frank Dobson*.

Definite NPs

Similarly, definite NPs (containing definite articles) can be resolved to NPs that contain more information. Again, referents must not be lists of conjunctions.

In this way we could deal with *"Bush is running for president of the USA: currently Clinton is the president"* by resolving *the president* to *the president of the USA*.

Numerical Expressions

Numerical expressions such as *one*, *some*, *a lot* can be resolved to plural NPs (consider a case such as *"there are many mountains in Wales, and one is Snowdon"* as an answer to *"Is Snowdon in Wales?"*). In most cases, the numerical expression is replaced directly by the plural referent: in the case of *one*, a new singular NP equivalent must be created (consider *"there are many high mountains in Wales, but only one is worth climbing, and it is Snowdon"* - we need *it* → *one* → *mountain(s) in Wales*, and this will only be possible if *one* is taken to be singular).

Location and Time

These expressions (*there*, *then*) can be resolved to suitable PPs (*"I love climbing in Wales: after all, Snowdon is there"* requires *there* → *in Wales*). At present, as PPs are not indexed, the anaphoric expression is replaced directly by a copy of the referent PP, rather than an index marker. This is acceptable with the system in its current state, but might become undesirable if later operations are added which affect the PPs.

3.2 Extraction of Sub-sentential Units

Most sentences consist of more than one clause. As we are attempting to extract simple structures, we need to split such sentences into smaller units, each of

which correspond as closely as possible to individual structures. For our example A2, this would produce *"Although one may dislike Bill Clinton"*, *"it is important to remember that Bill Clinton is one of the most powerful men in the world"*, and *"Bill Clinton is president of the USA"*. This section describes the method of splitting into these small units.

For answers, a single possible sub-unit is extracted, and Prolog backtracking chooses each in turn until one matches the query in hand. For queries, the `findall/3` predicate is used to build a list of sub-sentences, all of which must be matched by an answer.

3.2.1 Conjunctions

The most obvious method of combining two or more simple concepts into a complex sentence is by the use of conjunctions. Three methods of conjunction have been considered here: sentence, verb phrase and noun phrase conjunction.

The descriptions below are expressed in terms of a single conjunction joining two phrases: in all cases the same rules are applied to multiple conjunctions (comma/conjunction-separated lists of phrases).

S Conjunction

If a conjunction separates two parts of a sentence, both of which contain VGs, it is considered to be a conjunction of two separate sentences and these are extracted as individual sub-sentences. *"Pat is in the kitchen and Mike is in the garden"* will be split into the sub-units *"Pat is in the kitchen"* and *"Mike is in the garden"*.

VP Conjunction

If a conjunction separates two parts of the sentence containing VGs, and is immediately followed by a VG, it is considered to be a conjunction of two VPs. In this case, the sentence is split into two sub-units at the conjunction, and the first NP argument from the first unit copied to the second. *"Livingstone defeated all the other candidates and is London's mayor"* will be split into *"Livingstone defeated all the other candidates"* and *"Livingstone is London's mayor"*.

NP Conjunction

As described above, the parser groups NP conjunctions as NGs. For answers, this treatment is sufficient (they are left as NGs, and later matching rules allow for this). For queries, it is not sufficient, as we need to extract two logical sub-queries which can both be checked against an answer - Q10 *"Where are Pat and Mike?"* must be converted to *"Where is Pat?"* and *"Where is Mike?"*. This allows not only obvious matches such as A10.1, but more complex examples such as A10.2.

To achieve this, the sentence is copied, once for each of the NPs in the conjunction, with each individual NP in turn replacing the conjoined NG.

3.2.2 Subordinate/Relative Clauses

Relative clauses introduce a similar situation, as illustrated by example A10.6 *"Pat, who is reading, is in the lounge ..."*. We can extract two smaller units, each corresponding to a single simple structure: the "internal" sentence *"Pat is reading"* and the "external" sentence *"Pat is in the lounge"*. Punctuation is used to determine the extent of the relative clause, after Briscoe ([6], [7]).

Other subordinate clauses are dealt with in the same way: *"Pat, although he is reading, is in the lounge"* would produce two very similar sub-sentences.

3.2.3 Punctuation

Further use of punctuation is made to split into sub-sentences at colons and semi-colons, or even at e.g. full stops if more than one sentence is present in the passage.

3.3 Structure Extraction

Once simple sentential units are available, relational structures are extracted. This section describes both the form of these structures and their method of extraction.

3.3.1 Predicate-Argument Structures

Any relation involving a predicate (a non-existential verb) is expressed as a *predicate-argument* (PA) structure. A PA-structure takes the form

```
s:[Predicate, Argument1, Argument2]
```

This is based on the first-order logic interpretation of a standard transitive verb, more usually expressed as $Predicate(Argument1, Argument2)$: it can be considered as a more general version of a verb-subject-object relationship.

The extraction of these structures is based entirely on word order. The predicate is the VG of the simple sentence. Any NGs and other words before the predicate are compounded to form the first argument. Similarly, all NGs after the predicate form the second argument.

This compounding process allows ditransitive verbs (*"A gives B C"*) to fit the same structure - in this case, of the form `s:[Pred, A1, [A2 A3]]`. This was shown to deal successfully with examples including the ditransitive verb *"make"* (see example 9).

While verbs of many other sub-categories can be similarly accommodated, some give problems: for instance complementising verbs. In answer passages, we circumvent this problem by choosing a subset of the sentence with which we can deal successfully: a form such as *"it has been confirmed that Hawaii produces bananas"* will still allow us to produce the expected simple structure from *Hawaii produces bananas*. In queries there is currently no method for doing this: it is assumed that query sentences remain simple.

Intransitive verbs can form PA-structures with only one argument: this allows intransitive queries to be matched by transitive answers (as the shorter structure can still be extracted). However, we considered no intransitive verbs in our training or test sets, and examination of the TREC data shows that intransitive verbs are very uncommon in queries.

It may be that further study of examples including verbs of different sub-category show that a more advanced treatment is required: if so, it would be possible to allow PA-structures to have variable length (number of arguments).

3.3.2 State Structures

Although it is possible to represent existential verbs (e.g. *be*) as predicates in a PA-structure as above, this was found to have disadvantages. Existential relations are not only expressed by verbs, but by forms such as compound NGs or PP attachment: we would like our example Q to be matched by a sentence containing the NG *Bill Clinton, the president of the USA*.

In order to facilitate the matching that this requires, existential verbs and NGs (but not conjunctions) were used to produce structures with no predicate value (*state*-structures), of the form

```
s:[Argument1, Argument2]
```

Our example A1 will produce the structure

```
s:[[bill clinton],[the president of the usa]]
```

as will A2 (along with other less relevant structures).

3.4 Coindexing

This method of producing simple structures from sub-units of a complex sentence has an associated problem: what do we do if we require some information from more than one structure? We would like a query such as *"Does the President of the USA smoke cigars?"* to be answered successfully by *"Bill Clinton is the President of the USA and smokes cigars"*: but this produces structures which express *"Bill Clinton is the President of the USA"* and *"Bill Clinton smokes cigars"*. Neither of these contain the answer to the query.

A possible approach might be to enter these two structures in a knowledge base, and attempt to answer the query using some sort of logical theorem prover. A simpler approach has been taken here: as both structures contain the same entity *Bill Clinton*, we can modify this entity with the state information from the first structure, and it will automatically be carried over to the second.

We do this by setting up an equivalence between state-structures and compound NGs: if a state-structure corresponds to a compound NG, this NG is created: the entity `[bill clinton]` becomes `[[bill clinton],[the president of the USA]]`. Now, our second (PA) structure refers to this NG and so becomes `[smoke,[[bill clinton],[the president of the USA]],cigars]`: and this is able to match the original query. Similar rules are defined for NG formation by PP attachment.

Chapter 4

Structure Matching

Once the possible state- and/or PA-structures can be extracted from an answer, they can be examined and matched against the current query structure. This chapter describes this matching process.

In cases where the structures are sufficiently similar, this can be achieved by direct matching of predicates and arguments, using a set of lexical matching rules. There are 15 of these rules (3 at the word level and 12 for phrases). They are applied simplest first (from individual word matching to complex NG matching), but can be used in any combination via Prolog backtracking. These rules are described in section 4.1.

However, it is often the case that query and answer structures are significantly different. In this case, direct matching would only be possible if rules were very loose – in other words, tending towards DR-style keyword matching with its associated problems. In order to keep matching tight, I transform the structures to make them sufficiently similar for the matching rules to then be applied. There are 6 transformation rules (4 rules dealing with general phenomena and 2 rules only applicable to certain classes of predicate/argument) which are described in section 4.2. The transformations are generally applied to the answer structures (as query structures are usually already in a simple form) but some can be applied to either - see below. In testing, it was found that the majority of answers required at least one transformation before matching was successful.

This combination of structural transformation and matching is sufficient to deal with yes/no queries. For other query types involving wh-words, a further set of rules is required to specify the matching of these words against answer phrases. Some complex query types observed in TREC-8 also require structural transformation. These query-specific rules (6 word/phrase matching rules and 4 structural transformations) are described in section 4.3.

I then give a summary of the structural phenomena that can be dealt with using this philosophy in section 4.4, together with a discussion of some that are problematic. As a number of different structural phenomena must be illustrated, the examples throughout this chapter are taken from the training data given in

appendix A rather than the running examples used until now.

4.1 Direct Structure Matching

If the structures are sufficiently similar (lists of the same length), then each element of the query structure is matched individually against its corresponding element in the answer structure:

$$s : [Q1, Q2, \ldots] \Leftrightarrow s : [A1, A2, \ldots] \quad \text{if} \quad Q1 \Leftrightarrow A1,\, Q2 \Leftrightarrow A2,\, \ldots \tag{4.1}$$

(The symbol $\Leftrightarrow$ denotes matching).

In the simplest cases, where the corresponding elements of the structures are identical, this rule is sufficient (I use the symbol $\leadsto$ here to denote structure extraction, and only show a simplified version of the structure to aid legibility):
Q1 *"Is Snowdon in Wales?"* $\leadsto$ `s:[snowdon,[in wales]]`
A1.1 *"Snowdon is in Wales"* $\leadsto$ `s:[snowdon,[in wales]]`

4.1.1 Word Matching

It is unusual to find such a high degree of similarity, so a series of rules are required to allow the individual elements of the structures to match each other. The simplest form of matching is tried first: similar but non-identical words can be allowed to match under the following conditions.

Noun Matching

Nouns can match other nouns with the same stem, even if the PoS tag is not identical. This allows plurals (with a NNS tag) to match singular nouns (with NN tag) or nouns that have been mistagged as NP.

Adjective/Adverb Matching

Adjectives match other adjectives with the same stem, as long as the answer word has at least the same comparative degree as the query word: this allows a query about *tall mountains* to be matched by a passage about the *tallest mountain*, but not vice versa.

Pronoun Matching

Non-resolvable personal pronouns can match each other (*you* matches *one*).

4.1.2 Verb Group Matching

The word matching outlined above will only be sufficient in cases where the structures are identical except for individual word variations. More usually, there will be differences at the phrase level, so phrase matching rules are applied next. In the case of PA-structures, we first try to match the predicate (a VG).

Unmodified VGs

VGs without attached modifiers are allowed to match as long as both the voice and predicate features are identical, no matter what words are within the V sub-group.

$$\begin{array}{c} VG_Q : Voice\#Pred \\ | \\ V_Q \end{array} \Leftrightarrow \begin{array}{c} VG_A : Voice\#Pred \\ | \\ V_A \end{array} \tag{4.2}$$

This deals with cases such as:

Q5 *"Does Hawaii produce bananas?"*

⇝ `s:[vg:act#produce:[v:[does produce]],hawaii,bananas]`

A5.1 *"Hawaii produces bananas"*

⇝ `s:[vg:act#produce:[v:[produces]],hawaii,bananas]`

VG Modifiers

If a VG from an answer structure contains modifiers (adverbial phrases or PPs), it is allowed to match an unmodified query VG as above:

$$\begin{array}{c} VG_Q : V\#P \\ | \\ V_Q \end{array} \Leftrightarrow \begin{array}{c} VG_A : V\#P \\ V_A \quad Mod_1 \quad Mod_2 \ldots \end{array} \tag{4.3}$$

However, if the query VG contains a modifier, the answer VG must contain a matching modifier (see below for the rules used to match PPs):

$$\begin{array}{c} VG_Q : V\#P \\ V_Q \quad Mod_{Q1} \quad Mod_{Q2} \ldots \end{array} \Leftrightarrow \begin{array}{c} VG_A : V\#P \\ V_A \quad Mod_{A1} \quad Mod_{A2} \ldots \end{array} \tag{4.4}$$

$$\text{if} \quad \forall i \Big[\exists j [Mod_{Qi} \Leftrightarrow Mod_{Aj}] \Big]$$

This allows A6.1 to match Q5 shown above:

A6.1 *"Hawaii produced bananas in 1991"*

⇝ `s:[[produce,[in 1991]],hawaii,bananas]`

⇔ `s:[produce,hawaii,bananas]`

but would not allow the previous A5.1 to match Q6:

Q6 *"Did Hawaii produce bananas in 1991?"*

⇝ `s:[[produce,[in 1991]],hawaii,bananas]`

4.1.3 Noun Group Matching

If predicate matching is successful, we proceed to try to match the arguments (which are usually NGs, although they can be PPs - see next section). In the case of state-structures, where no predicate needs to be matched, these rules are tried immediately.

Simple NPs

NPs match if they share a semantic class, and all words from the query NP except for determiners are present in the answer NP. This will allow a query like Q2 about *a racecourse* to be matched by an answer containing *a very long racecourse*, but not vice versa.

$$\underbrace{NP_Q : Sem_Q}_{\dots Det_{Qx} \dots X_{Qi} \dots} \Leftrightarrow \underbrace{NP_A : Sem_A}_{\dots Det_{Ay} \dots X_{Aj} \dots} \tag{4.5}$$

$$\text{if} \quad \exists S[(S \in Sem_Q) \cap (S \in Sem_A)]$$

$$\text{and} \quad \forall i \Big[\exists j [X_{Qi} \Leftrightarrow X_{Aj}] \Big]$$

An exception is made for certain determiners (such as *every*, *all*) which must be present in the answer if present in the query - this allows answers containing *a racecourse*, *the racecourse*, *racecourse* to match a query concerning *a racecourse*, but not a query about *all racecourses*.

Number does not have to be matched (*"there are racecourses in Newmarket"* is an acceptable answer to *"is there a racecourse in Newmarket?"*, and conversely *"there is a racecourse ..."* answers *"are there racecourses ...?"*).

Complex NGs

Any NG matches any other NG if they share a semantic class, and all words contained in sub-NPs or the query NG are present in sub-NPs of the answer NG. NPs contained within PPs are not considered as sub-NPs for this purpose. Any PPs present in the query NG must also be matched (see below for PP rules). In this way, *large racecourse [with a water jump]* matches a query about *racecourses*, but not vice versa.

$$\underbrace{NG_Q}_{\dots X_{Qi} \dots} \Leftrightarrow \underbrace{NG_A}_{\dots X_{Aj} \dots} \quad \text{if} \quad \forall i \Big[\exists j [X_{Qi} \Leftrightarrow X_{Aj}] \Big] \tag{4.6}$$

Members of Complex NGs

Any query phrase can be matched by a NG in the answer if that NG contains a matching phrase. This allows an answer containing *a mountain in Wales* to match queries about *mountains* and queries about *in Wales*.

$$X_Q \Leftrightarrow \underset{\ldots X_A \ldots}{NG_A} \quad \text{if} \quad X_Q \Leftrightarrow X_A \tag{4.7}$$

Inclusions

An NG can match a phrase (usually a PP) beginning with an "inclusive" word (*like*, *such as*) if it matches the rest of the phrase. This allows, say, *Hawaii* to match *islands such as the beautiful Hawaii.*

$$NG_1 \Leftrightarrow \underset{Inc \quad NG_2}{X} \quad \text{if} \quad NG_1 \Leftrightarrow NG_2 \tag{4.8}$$

Expansions

A NG can match the form "NG1 of NG2" if it matches NG2, as long as NG1 and NG2 share semantic class, This allows expanded descriptions to be matched (*Newmarket* matches *the town of Newmarket*) but not general possessives (*the mayor of Newmarket*).

$$NG_A \Leftrightarrow \underset{NG_1 : Sem_1 \quad \underset{of \quad NG_2 : Sem_2}{PP}}{NG_B} \quad \text{if} \quad NG_A \Leftrightarrow NG_2 \tag{4.9}$$

$$\text{and} \quad \exists S[(S \in Sem_1) \cap (S \in Sem_2)]$$

Relaxed NG-PP Matching

An additional rule allows NGs to match PPs containing them, but only in the context of a verb nominalisation construction (see below).

4.1.4 Modifier Phrase Matching

We have already mentioned the requirement to match PPs and adverbial phrases as part of VG and NG matching. We may also require PPs to be matched if they are arguments themselves.

Simple Prepositional Phrases

PPs match each other if they share semantic class, and their constituent NGs match.

$$\underset{Prep_1 \quad NG_1}{PP_1 : Sem_1} \Leftrightarrow \underset{Prep_2 \quad NG_2}{PP_2 : Sem_2} \quad \text{if} \quad NG_1 \Leftrightarrow NG_2 \tag{4.10}$$

$$\text{and} \quad \exists S[(S \in Sem_1) \cap (S \in Sem_2)]$$

This allows PPs with any "location"-class preposition to match each other (*in Newmarket* matches *at Newmarket*). Of course, this therefore allows *outside Newmarket* to match as well: this was considered reasonable, as it conveys information likely to provide an answer to the related query. This behaviour could be changed if prepositions are given a finer-grained semantic classification (e.g. "internal location", "external location", "general location").

Complex PPs

PPs can also match the inner levels of "stacked" PPs if all levels share semantic class.

$$PP_X \Leftrightarrow \underset{Prep_1 \quad \underset{NG_2 \quad PP_2 : Sem_2}{NG_1}}{PP_1 : Sem_1} \quad \text{if} \quad PP_X \Leftrightarrow PP_2 \tag{4.11}$$

$$\text{and} \quad \exists S[(S \in Sem_1) \cap (S \in Sem_2)]$$

This allows the simple PP *in Suffolk* to match constructions such as *in a town in Suffolk*.

Possessive PPs

"Possession"-class PPs are allowed to match NGs that contain the possessive suffix *'s*, as long as they contain a NG that matches the other contents of the possessive NG. Thus *of London* matches *London's*.

$$\underset{Prep_1 \quad NG_1}{PP : \texttt{pos}} \Leftrightarrow \underset{NG_2 \quad 's}{NG : \texttt{pos}} \quad \text{if} \quad NG_1 \Leftrightarrow NG_2 \tag{4.12}$$

Adverbial Phrases

Adverbial phrases match if all words present in the query phrase are present in the answer phrase:

$$\underset{\ldots X_{Qi} \ldots}{AdvP_Q} \Leftrightarrow \underset{\ldots X_{Aj} \ldots}{AdvP_A} \quad \text{if} \quad \forall i \Big[\exists j [X_{Qi} \Leftrightarrow X_{Aj}] \Big] \tag{4.13}$$

4.2 Structure Transformation

For cases where the structures do not match directly, a set of structural transformation rules is defined. Application of these rules is achieved by applying a transformation to one structure and then testing for a direct match using the matching rules from the previous section.

$$Q \Leftrightarrow A \quad \text{if} \quad Q \Leftrightarrow A' \quad \text{where} \quad A' = T_A(A) \tag{4.14}$$

or

$$Q \Leftrightarrow A \quad \text{if} \quad Q' \Leftrightarrow A \quad \text{where} \quad Q' = T_Q(Q) \tag{4.15}$$

4.2.1 General Structural Transformations

The first set of transformation rules capture general concepts about sentence structure equivalence: their use is independent of the predicate or argument semantics.

Active-Passive

A PA-structure (query or answer) with a passive VG at its head can be converted into the equivalent active structure, as long as its second argument is a *by*-PP. This argument is converted into a NP and the argument order is reversed.

$$s : [VG_{pas}, Arg1, pp : [by, Arg2]] \rightarrow s : [VG_{act}, Arg2, Arg1] \tag{4.16}$$

This rule will enable the following example to match Q9:
A9.6 *"One is made very fat by bananas"*

$\leadsto$ `s:[vg:pas#:make:[...],[one very fat],[by bananas]]`

$\rightarrow$ `s:[vg:act#make:[...],bananas,[one very fat]]`

Existential Ordering

A state-structure (query or answer) can have the order of its arguments reversed (*"A is B"* and *"B is A"* are logically equivalent). Care needed to be taken here to prevent infinite recursion when applying the transformations.

$$s : [Arg1, Arg2] \rightarrow s : [Arg2, Arg1] \tag{4.17}$$

This rule will enable the following examples to match Q2:
A2.2 *"In Newmarket there is a racecourse"*
$\leadsto$ `s:[[in newmarket],racecourse]`
$\rightarrow$ `s:[racecourse,[in newmarket]]`

Existential Arguments

The contents of a single NG argument of any structure can be treated as a separate state-structure, unless that argument is a list of conjunctions.

$$s : [\ldots, ng : [Arg1, Arg2, \ldots], \ldots] \rightarrow s : [Arg1, Arg2, \ldots] \tag{4.18}$$

This allows the following example to match Q1:
A1.4 *"Snowdon, in Wales, is a serious mountain"*
$\leadsto$ `s:[ng:[snowdon,[in wales]],[a serious mountain]]`
$\rightarrow$ `s:[snowdon,[in wales]]`

This rule is only applicable in answers: queries of the form *"Snowdon, in Wales?"* do not seem to be common!

Verb Nominalisation

A PA-structure whose predicate is a generic utility verb (*do*, *have*, *make*) and one of whose arguments contains a NP with a head noun whose lemma corresponds to a verb, can be converted into the equivalent PA-structure with the new lemma verb as its predicate. The lemmatised NP is removed from its argument. This lemma information is currently entered manually in the lexicon as a series of verb/noun pairs (e.g. *produce/production*, *produce/product*) - a lemmatising dictionary such as CIDE+ could be used instead.

$$s : [VG_{util}, Arg1, Arg2, \ldots] \rightarrow s : [VG_{lemma}, Arg1', Arg2, \ldots] \tag{4.19}$$

$$\text{where} \quad Arg1' = Arg1 \setminus NP_{lemma}$$

Similarly, a state-structure containing such an argument can be converted into the corresponding PA-structure.

$$s : [Arg1, Arg2, \ldots] \rightarrow s : [VG_{lemma}, Arg1', Arg2, \ldots] \tag{4.20}$$

$$\text{where} \quad Arg1' = Arg1 \setminus NP_{lemma}$$

This rule is only applied to answer structures as it is assumed that queries will not exhibit verb nominalisation. However, it would be trivial to create a parallel rule for query structure conversion. It is of course only applied if we are attempting to match to a query PA-structure with the lemma verb as its predicate.

In order for this transformation to allow successful matching, subsequent rules are relaxed: in particular, PPs are allowed to match NGs once this transformation has been applied. This is due to the large variation in forms that can be encountered within this general class of phenomenon: *"production [of bananas] [in Hawaii]"*, *"[banana] production [by Hawaii]"*, *"[bananas], products [of Hawaii]"*, *"[Hawaii], producer [of bananas]"* would all be expected to match Q5. This relaxation makes this possible:

A5.12 *"In Hawaii, [...], there is [...] production of bananas"*
⇝ `s:[[in hawaii],[production of bananas]]`
→ `s:[produce,[in hawaii],[of bananas]]`
⇔ `s:[produce,[hawaii],[bananas]]`

The disadvantage of this relaxation is that forms such as *all products of Hawaii except bananas* will match successfully, although they do not logically provide an answer to the query. This might be avoided if the lemma-containing nouns can be further classified (as, say, "agents", "entities", "concepts") and individual rules created for each class, but time did not permit investigation of this.

4.2.2 Specific Structural Equivalences

Some answer phenomena required structural rules that should only apply in certain contexts (for example, with arguments of certain semantic classes).

Verbs of Possession

A PA-structure whose predicate corresponds to the notion of "possession" (*have*, *contain*) can be transformed into a state-structure containing a "possession"-class PP.

$$s : [VG_{pos}, Arg1, Arg2] \rightarrow s : [Arg2, pp : [pos] : [of, Arg1]] \tag{4.21}$$

This will allow the following example to match Q2:

A2.4 *"Newmarket has a racecourse"*
⇝ `s:[have,[newmarket],[a racecourse]]`
→ `s:[[of newmarket],[a racecourse]]`
⇔ `s:[[in newmarket],[a racecourse]]`

An analogous rule allows the same kind of structure to be transformed into a

state-structure containing an "inverse-possession"-class PP:

$$s : [VG_{sop}, Arg1, Arg2] \rightarrow s : [Arg1, pp : [sop] : [with, Arg2]] \quad (4.22)$$

A2.4 *"Newmarket has a racecourse"*
⇝ `s:[have,[newmarket],[a racecourse]]`
→ `s:[[newmarket],[with a racecourse]]`
(although this isn't useful in this example).

Numerical Quantities

This rule is currently only implemented for "number"-class arguments. A PA-structure whose predicate corresponds to the notion of "possession" (*have*) can be transformed into a state-structure.

$$s : [VG_{pos}, Arg1, Arg2] \rightarrow s : [Arg1, Arg2] \quad (4.23)$$

This will allow the following example to match Q7:
A7.2 *"Everest has a height of 28,000"*
⇝ `s:[have,[everest],[a height of 28000]]`
→ `s:[[everest],[28000]]`

4.3 Question-word Matching

So far, all rules have been query-type-independent. The rules and transformations outlined above are sufficient to allow answers to be matched to yes/no queries, but there many other important types (as can be seen in the list of TREC-8 queries given in [41]).

A further set of rules is required to allow wh-queries to be dealt with - these rules specify the types of phrase that wh-query words (and phrases) can match.

Single Wh-Words

Wh-query words are specified in the lexicon with lists of semantic class and of phrase type. When on their own, they are allowed to match any phrase of the correct type which shares a semantic class with the wh-word. When combined with other words in an NP, other rules apply (see below).

Quantities

A common query form involves *how* with adjectives of quantity: many of the TREC queries begin with *how many*, *how much*, etc. With our semantic class system, this construction is allowed to match any NG in the "number" class: *how high* can match *28,000 ft.*, or just *28,000*. A finer-grained classification

where	`loc`	PP
when	`tim`	PP, NG, NP
who, whom	`per`, `org`	NG, NP
what	`abs`, `obj`, `org`	NG, NP
which	`abs`, `obj`, `org`	NG, NP
whose	`pos`	PP
how	`man`	PP, AdvP
why	`rea`	PP

Table 4.1: Wh-query word semantic classes and phrase types

would allow a distinction between, say "height" and "money", to prevent this query from being matched by *$28,000*.

Queries of the form *what* with nouns of quantity are matched in the same way: *what height* will match the same phrases as *how high*.

A structural transformation has also been created to allow the additional form of such queries *what is the height of X* to match the same phrases as *what height is X* (and *how high is X*).

How+Adjective

This general structure, wherein the adjective is **not** an adjective of quantity, can match any phrase that contains a form of the same adjective. This allows, say, *how useful* in a query to be matched by *extremely useful* in an answer passage.

What/Which+NG

This is a very common query form in the TREC exercise: *what city*, *which costume designer*. In this case, we allow this query NP to match any NG of the same semantic class (*what costume designer* will match a "person"-type NG). Once again, a more detailed semantic classification system would be useful here to prevent *what city* from accidentally matching non-city "locations", such as countries.

"Name" Queries

Another common query form observed in TREC is *What is the name of ...*, both for people (e.g. *...the president of the USA*) and other objects (e.g. *...the largest city in Germany*). As answers to this kind of query are extremely unlikely to contain any phrase literally corresponding to *the name*, these queries are transformed structurally into their logical equivalent (*Who is ...* for people, *What is ...* for others) and then matched as before.

In the same way, the form *Name ...* → *Who/What is ...*, and *Name the X who ...* → *Which X ...*.

4.4 Summary of Capabilities

This section gives a summary of the capabilities of the system. It shows that a wide range of answer phenomena can be dealt with, along with all the most common query types. I conclude by describing some of the phenomena that are currently beyond its capabilities and suggesting possible methods for tackling them.

4.4.1 Answer Passage Phenomena

Table 4.2 gives a list of the wide range of phenomena (lexical and structural) which can be matched by the system, together with examples taken from the training and test data shown in full in appendix A. The requirements posed by the phenomena range from simple non-identical word matching to complex structural transformation.

Direct Match	**A1.1** *"Snowdon is in Wales"*
Singular/Plural	**A2.6** *"There are several racecourse in Newmarket"*
Unnecessary Modifiers	**A2.14** *"There is a fine racecourse for all lengths in Newmarket"* **A1.5** *"Snowdon, the highest mountain in the UK, is in Wales"*
Stem Matching	**A4.1** *"The biggest IT company is Microsoft"*
Inclusions	**L5.1** *"The Pacific islands like Hawaii produce tropical fruit like bananas"*
Expansions	**A2.12** *"In the Suffolk town of Newmarket there is a racecourse"*
Simple PPs	**A3.2** *"Livingstone is mayor of London"*
Possessive PPs	**A3.5** *"Ken Livingstone defeated all the other candidates and is London's mayor"*
Verb Form Variations	**A5.6** *"Hawaii is producing bananas"*
VG Modifiers	**A6.1** *"Hawaii produced bananas in 1991"*
Verbs of Possession	**A2.4** *"Newmarket has a racecourse"*
Passives	**A9.6** *"One is made very fat by bananas"*
Existential Ordering	**A2.2** *"In Newmarket there is a racecourse"*
Existential Compounds	**A1.4** *"Snowdon, in Wales, is a serious mountain"*
Coreference	**A10.5** *"Pat is in the kitchen and Mike is there too"*
Conjunctions	**A10.2** *"Pat is in the lounge and Mike is in the garden"*
Relative Clauses (internal)	**A5.16** *"Hawaii specializes in the production of tropical fruit, which includes bananas and pineapples"*
Relative Clauses (external)	**A10.6** *"Pat, who is reading, is in the lounge"*
Relative Clauses (coindexed)	**A5.16** *"Tropical fruit production, which includes banana production, is the mainstay [...] of Hawaii"*
Subordinate Clauses	**A12.6** *"There is a library, with all the texts the student needs, in every college in Cambridge"*
Verb Nominalisation	**A5.12** *"In Hawaii, which is not far from California, there is large scale production of bananas"*

Table 4.2: Lexical/Structural Phenomena

4.4.2 Query Types

Table 4.3 shows the range of query types that can be dealt with. Where possible, these are illustrated with examples from appendix A: where no example was encountered, an example from TREC-8 is given.

The "simple wh-word" type includes *why* and *how* questions. These were not tested: although they will form legitimate query structures, they may cause problems as no effort has been spent in developing the necessary rules for answer entity matching. It should be noted that they also caused problems for many TREC-8 participants.

Yes/No	**Q1** *"Is Snowdon in Wales?"*
Simple Wh-Word	**Q3** *"Who is Mayor of London?"*
Quantities	**Q7** *"What is the height of Everest?"* (*"What height is Everest?"*) (*"How high is Everest?"*)
How+Adjective	**TREC Q161** *"How rich is Bill Gates?"*
What/Which+NG	**TREC Q47** *"What company is the largest Japanese ship builder?"*
What Name	**TREC Q5** *"What is the name of the managing director of Apricot Computer?"*
Name	**TREC Q66** *"Name the first private citizen to fly in space."*
Name+NG	**TREC Q65** *"Name a country that is developing a magnetic levitation system."*

Table 4.3: Query Types

4.4.3 Problematic Phenomena

Table 4.4 outlines structural phenomena which are currently beyond the capabilities of the system: while all are illustrated with answer examples, they could also occur in queries. The first two are thought to be achievable: the rest pose larger problems.

Table 4.5 gives a similar list of problematic phenomena which only appear in queries. All could be dealt with given more time.

Tense, Synonyms

While these phenomena might be important in a real-world QA system, they were not dealt with in this project due to the limited time available. However, both have been shown to be treatable.

Tense	*"There used to be a racecourse in Newmarket"*
Synonyms	*"There is a racetrack in Newmarket"*
Negatives	*"Snowdon is not in England"*
Inference	*"Students in Cambridge can use the many libraries"*
Reasoning	*"K2 is 27,500 ft. high, and Everest is 500 feet higher"*
Comparatives	*"Snowdon is higher than the mountains in England"*
Ellipsis	*"Wales is a fine country and Snowdon [is] a fine mountain"*
Adjuncts	*"Microsoft is a big IT company and so is IBM"*
Hypothetical	*"If Snowdon were in Nepal, it would seem tiny"*
Sub-category Variations	*"Did Pat give Mike a present?"* *"Pat gave a present to Mike"*

Table 4.4: Problematic Phenomena

PPs split from their NGs	*"Of which country is Clinton the president?"*
Split PPs	*"Which country is Clinton the president of?"*

Table 4.5: Problematic Query Phenomena

A treatment of tense could be achieved in a similar manner to the current treatment of passives: identification of verb forms together with a "tense" feature on VGs or structures. However, this would need to be combined with time-stamp information associated with the answer passages: a passage in the present tense, but written in 1985, might be considered as a legal answer to a query in the past tense, but not to a query in the present tense. This issue was addressed successfully by some TREC participants: see [26].

Synonymous (or hypo/hypernymous) expressions will require the addition of a suitable lexical resource such as WordNet, together with some matching rules specifying acceptable degrees of synonymity (perhaps with a score?). Again, this was tackled successfully in TREC: see e.g. [13], [22].

Negatives, Inference, Reasoning

These phenomena were also not attempted due to the limited time available: however, they seem to pose real problems.

Negative sentences are currently treated exactly as their affirmative counterparts. To a certain extent, this approach seems reasonable, as our aim is to identify a passage that answers a query: *"Hawaii does not produce bananas"*

would be a perfectly acceptable and informative answer to Q5. It is less clear for wh-questions, however: *"Macrosoft, despite the name, is not a large IT company"* is probably not a useful answer to Q4. While these negative answers would be easily spotted by a user if presented in full (and might even be considered useful), they could be positively misleading if the system attempted to answer the question directly (so produced *"Macrosoft"* from the previous example). If this were the case, a treatment of negatives would be required: this might be achievable by using negative words as adverbial modifiers and thus creating a "negative" feature for VGs or structures, although more complex cases will require more thought: *"Hawaii produces bananas but not guavas"* is a good answer to Q5, but should be rejected for the similar *"Does Hawaii produce guavas?"*. This might be treatable by extracting two conjoined structures (one negative, one positive) - in other words, by treating NP conjunctions in answers in a similar manner to their current treatment in queries.

The phenomena we have described as inference and reasoning are in themselves large areas. As a minimum, we will require: some real-world knowledge; a logical capability such as theorem-proving; calculation capability; pragmatic theory.

Comparatives

Sentences of the type shown above are difficult to exclude due to our lack of ability to determine the attachment point of PPs. While it might be possible to spot and constructions such as *higher than X* and prevent them from matching *X*, it is difficult to determine the scope of the *than*-phrase, especially when it contains PPs that might attach anywhere (even to the VG).

Ellipsis, Adjuncts

It appears to be in the nature of our predicate-argument approach that sentences with ellipsis of a predicate will cause problems. Resolution of ellipsis is non-trivial and is the subject of much research. Similarly, it is not easy to determine the referent of adjunct words like *so* and *too*.

Hypothetical

Some attempt has already been made to address a particular instance of hypothetical sentences: the parser prevents VGs such as *want to be* from being assigned the meaning *be*. While it might be possible to similarly detect phrases introduced by hypothetical conjunctions (*if*, *unless*), it will be difficult to determine their scope (and thus determine which apparently-matching phrases should be excluded). Matters are made worse if subordinate phrases are introduced within the hypothetical phrase: *"If Snowdon, which is in Wales, were in Nepal, ..."* does provide a legal answer.

Query Phenomena

All of these phenomena could be dealt with by structural transformations. The first two are versions of the inverted query form and could be transformed to their non-inverted equivalent. The "name who" form could be dealt with either by a structural transformation or a phrase matching rule: in either case, matching *"Name the X who/which/that"* with the same rules currently used for *"Which X"*.

Chapter 5

Evaluation and Results

One of the advantages of the structural matching approach is that it identifies the matching portions of answer text, providing us with sensible material for use as the system output.

This output (described in section 5.1) also provides a basis for a method of evaluation: the system output could be compared with an equivalent manual annotation of the answer text. Achieving consisteny in manual annotation is notoriously difficult (as noted in the TREC-8 evaluation [41]), and two different methods were investigated before settling on a reasonably consistent minimal, "narrow-scope" technique (see section 5.3).

Using this method, the performance of the system was measured in two blind tests, as well as over all the training data used in development, and the results are given in section 5.4. Performance on the training data and the first blind test was good, despite a wide range of sentence phenomena being present. The second test caused problems as it introduced several new phenomena which had not been encountered in development.

Finally, I set out the conclusions that can be drawn from this project in section refsec:conc.

5.1 System Output

Answer generation is a large and problematic area in itself (see e.g. [18]). The TREC-8 QA track avoided the need for full natural language answer generation by returning suitable portions of surface text. This seems a sensible approach for QA from a document corpus, and was adopted here.

Structural matching allows us to avoid the problems of text windowing experienced by DR systems (see example A2 in chapter 1), as we can return only the portions of text which are matched. In example A2, this would mean returning *Bill Clinton* and *the president of the USA*. This suggests a possible user interface: the answer passage could be displayed, with matching text highlighted. In this

way, the user would be able to reject any falsely identified answers. Given the short timescale of this project, a highlighting interface was not developed, and the matching portions are returned as plain text.

The text returned is therefore unlikely to make up a well-formed linguistic entity. Due to the structural transformation approach, it may not directly mirror the structure of the query (consider cases involving verb nominalisation). It was therefore also though useful to identify the answer entity corresponding to the wh-word in wh-questions.

5.2 Training Data

In order to test the system on a wide range of answer structure phenomena, while keeping the demands on the manually specified lexicon to a minimum, it was necessary to use specially created examples rather than real-world (e.g. newspaper) text.

The full set of training sentences is given in appendix A. For each query, there is a set of potential answer passages, all of which contain the keywords of the query. Some do provide the answer (marked A) and some do not (marked N). Sentences which were regarded as logically providing an answer, but one which would probably be rejected by a human judge with world knowledge, were marked L. Sentences outside the scope of this system due to use of synonyms or inference were marked P.

5.3 Evaluation Method

5.3.1 Manual Annotation

In order to evaluate performance, each set of answer sentences was annotated manually by an independent "notional user" who had no detailed knowledge of how the system worked. The idea was to manually mark the portions of text which were felt to match the query – this could then be used as a basis for judging whether the output of the system was acceptable.

It is not easy to develop a strategy for manually selecting phrases which can be successfully and consistently applied in all situations. Two strategies were considered: a "narrow-scope" approach whereby the minimum possible amount of text required to show a match was selected (not including any unnecessary adjectives or other modifiers), and a "broad-scope" approach whereby all text which could be considered to match was selected (allowing whole phrases to be selected). Although neither were easy to apply consistently, the narrow-scope selections were considerably more consistent and so were used in the evaluation.

Even this strategy was difficult for some of the more complex sentences, for example when deciding whether to include prepositions in sentences that included

verb nominalisation. Because of this, an "overlap" category was included in the results as well as an "exact match" category - see below.

All manually judged selections are given in full in appendix A.

5.3.2 Performance Measures

The system output for a particular answer could now be judged to be correct if it matches the manual annotations. The manually and automatically generated sets of output were considered to *match exactly* if all words matched apart from determiners, conjunctions and existential words (e.g. *"there is"*).

Due to the difficulty of consistent marking, they were also considered to *overlap* if the only non-matching words were parts of a verb form (e.g. modals, complementisers), prepositions, or anaphors whose referent had already been selected. Otherwise they were considered not to match.

The conventional measures of recall and precision can now be calculated:

$$Recall = n(correct \,\&\, identified)/n(correct)$$

$$Precision = n(correct \,\&\, identified)/n(identified)$$

5.3.3 Blind Testing

For performance evaluation, a set of blind tests was performed. Query and answer sets were prepared and manually annotated by the same independent user.

Due to the requirement for manual lexicon entries, a special procedure had to be established. Before the unseen sentences were submitted to the system, the vocabulary was presented and added to the lexicon. Once sentences had been received, a preliminary run of the shallow text processor was made to check for any missed words and spelling mistakes. Once any final additions had been made, the test was run. Once testing was complete, the manual annotations were provided and results assessed.

5.4 Results

5.4.1 Training Data

The final system was tested retrospectively over all training data, and the following results were obtained:

Errors were due to the following causes:

False Identifications Only two N-class answer passages were falsely selected: one is a comparative (A1.8) and one a hypothetical sentence (A2.3).

Query	Exact Match	Overlap	No Match	Not Identified	Falsely ID'd
Q1	4	0	1	0	1
Q2	12	2	0	0	1
Q3	5	0	0	0	0
Q4	3	0	1	0	0
Q5	9	8	0	0	0
Q6	2	1	0	0	0
Q7	3	2	2	0	0
Q8	3	0	0	0	0
Q9	4	5	0	1	0
Q10	6	0	1	1	0
Q11	2	0	1	4	0

Table 5.1: Performance on Training Data

Missing Identifications 6 A-class answers were not identified. One is a verb nominalisation form that has not yet been dealt with, and the rest would now, with hindsight, be classified as P-class: they either involve real-world knowledge (*the secretary's job* → *something the secretary *does**) or the acceptance that *with PERSON* is a location.

No Match Six answers have been classed as not matching. In two cases, alternative answers that seem acceptable have been generated from the sentence: *in the UK* instead of *in Wales* from A1.5, *"in the extension building"* instead of *at a meeting* from A10.4. In the other four cases, information has been indeed been missed or added (A4.2,A11.1).

Considering exact and overlapping matches as successes, the performance on the training set is therefore 71 successes, 12 failures and 2 false identifications, which gives figures of 86% recall and 97% precision. If we accept our new classification and alternative answers, the measured recall improves to 94%.

5.4.2 First Blind Test

In the first blind test (query set 12), one yes/no-query and 24 possible answer sentences were presented to the system. Manual judgement after the test classified three sentences as N-class (no answer) and three as P-class (pragmatic inference, world knowledge). This left 18 sentences that should have been identified.

The system identified 14 of the 18, and falsely identified one of the N-class sentences. Without examining the output, this appears reasonable: recall of 78% and precision of 93%. The N-class sentence 12.3 was mistakenly chosen as it contained a pseudo-answer in a hypothetical clause. The four correct sentences not identified (12.6, 12.7, 12.8, 12.9) were missed due to, respectively: inability to

Query	Exact Match	Overlap	No Match	Not ID'd	Falsely ID'd
Q12 (broad-scope)	6	3	5	4	1
Q12 (narrow-scope)	9	3	2	4	1
Q12 (final system)	11	4	2	1	1

Table 5.2: Performance in First Blind Test

handle subordinate clauses; no resolution of possessive pronouns; no resolution of numerical anaphora; no notion of "inverse possession". Most of these deficiencies have since been remedied (as described in the previous chapters).

However, once the output had been examined, five of the correctly identified sentences were judged to have given non-matching output, which reduces the measured performance to 50% recall and 90% precision. In four cases, this was due to poor reporting (at this stage, broad-scope answers were being used, and it was difficult to keep the scope of the answer consistent with the manual set). In one case, a false structure had been generated.

In preparation for the second blind test, the reporting was greatly improved (mainly by the choice of the more consistent narrow-class answers). This increased measured performance to 67% recall, 93% precision. The addition of features (subordinate clauses, anaphora, possession), also increased the performance, to 83% recall and 94% precision. Those remaining incorrect are due to either the lack of possessive pronoun resolution, or the returning of (probably acceptable) alternative output which does not match the manual annotations.

Of the 17 correct answers given by the final system, 12 (or 70%) required at least one structural transformation, showing how important this concept is.

5.4.3 Second Blind Test

With the system in its final state, a second blind test was performed, this time using eight separate wh-queries. While performance was reasonable on the first two, a number of hitherto unseen query and answer phenomena together with many requirements for world knowledge made overall performance poor: only 4 of a possible 21 correct answers were identified.

Some of the failures were caused by deficiencies in the system as it stood – for example, anaphora resolution needs to be improved (*the most important* → *town in Norway*). Some were caused by incorrect PoS tagging (*subject* as an adjective, *cost* as a noun). Some were caused by phenomena that were known to be problematic, e.g. hypothetical clauses and inference.

However, most were caused by the introduction of new phenomena: multiple query-words in Q15; category words (in Q17 we don't need to match *newspaper* in

Query	Exact Match	Overlap	No Match	Not Identified	Falsely ID'd
Q13	1	0	0	1	0
Q14	2	0	0	1	1
Q15	0	0	0	2	0
Q16	0	0	0	3	0
Q17	0	0	0	3	0
Q18	0	0	0	1	1
Q19	1	0	0	2	0
Q20	0	0	0	4	0

Table 5.3: Performance in Second Blind Test

the compound *the Guardian newspaper*); adverbial modifiers of quantity (*how fast* - only adjectival versions had been considered previously); and some properties of words which come from world knowledge (*a X company* → *company which produces X*, *their beer* → *the beer they produce*, *X is married to Y* → *X and Y are married*).

It seems that while the structural transformation approach can be successful at handling known phenomena, it is (unsurprisingly) not robust to new phenomena which require new transformations. A real-world system based on this approach would require a large amount of training data in order to allow these transformations to be determined and tested.

5.4.4 TREC-8 Comparison

Although the TREC-8 questions are available in [41], the answer corpora were not, so a direct comparison with the TREC-8 performance is not possible. Some portions of answer text were available (from the output of AT&T's 250-byte DR-based system [33]) and some of these showed the immediate context of the answer. Of these, slightly over half would have been answerable (with the appropriate lexicon additions), but half would not. Many of these required inference or calculation:

Q199 *"How tall is the Matterhorn?"*

required the answer *14,776 feet 9 inches* from *"about 7 inches higher than 14,776 feet 2 inches"*. The rest introduced new phenomena:

Q1 *"Who is the author of the book [...]?"*

had in the answer sentence *"[...] by Hugo Young"*. In order to cope with this kind of example, we need the knowledge that *"by"* indicates authorship. While this could be expressed as a structural transformation, we would also need one to cope with *"Who wrote [...]"* and other equivalents. As stated in the previous

section, some effort would be required to determine and test a wide range of possible transformations.

5.5 Conclusions

- An operational QA system based on structural matching was developed and tested.
- A structural matching approach requires the use of structural transformations. The majority of answers from both training and test data required at least one transformation before successful matching was possible.
- A wide range of query and answer phenomena can be dealt with (see chapter 4).
- This approach gave good performance on the training data and first blind test set.
- The approach does not appear to be robust to unseen phenomena, so requires a large amount of training data.

Bibliography

[1] AARTS, B. *English syntax and argumentation.* Macmillan, 1997.

[2] ABNEY, S., COLLINS, M., AND SINGHAL, A. Answer extraction. In *Proceedings of the Sixth Applied Natural Language Processing Conference* (Apr.-May 2000), Association for Computational Linguistics, Morgan Kaufmann, pp. 296–301.

[3] ALLAN, J., CALLAN, J., FENG, F.-F., AND MALIN, D. INQUERY and TREC-8. In *The Eighth Text REtrieval Conference (TREC 8)* (1999). NIST Special Publication: in press.[1]

[4] APPELT, D. E., HOBBS, J. R., BEAR, J., ISRAEL, D., AND TYSON, M. FASTUS: a finite state processor for information extraction from real world text. In *13th International Joint Conference on Artificial Intelligence (IJCAI-93)* (1993), vol. 2, pp. 1172–1178.

[5] BRECK, E., BURGER, J., FERRO, L., HOUSE, D., LIGHT, M., AND MANI, I. A Sys called Qanda. In *The Eighth Text REtrieval Conference (TREC 8)* (1999). NIST Special Publication: in press.[1]

[6] BRISCOE, E. J. Parsing (with) punctuation etc. Rank Xerox Research Laboratory, Grenoble, MLTT-TR-002, 1994.

[7] BRISCOE, E. J. The syntax and semantics of punctuation and its use in interpretation. In *Punctuation in Computational Linguistics* (1996), B. Jones, Ed., pp. 1–8. Proceedings of ACL SIGPARSE Workshop.

[8] BURTON-ROBERTS, N. *An introduction to English syntax*, second ed. Addison Wesley Longman, 1997.

[9] CARDIE, C., NG, V., PIERCE, D., AND BUCKLEY, C. Examining the role of statistical and linguistic knowledge sources in a general-knowledge question-answering system. In *Proceedings of the Sixth Applied Natural Language Processing Conference* (Apr.-May 2000), Association for Computational Linguistics, Morgan Kaufmann, pp. 180–187.

[1] Available at http://trec.nist.gov/pubs/trec8/t8_proceedings.html

[10] Cormack, G. V., Clarke, C. L. A., Palmer, C. R., and Kisman, D. I. E. Fast automatic passage ranking (multitext experiments for TREC-8. In *The Eighth Text REtrieval Conference (TREC 8)* (1999). NIST Special Publication: in press.[1]

[11] Cowie, J., and Lehnert, W. Information extraction. *Communications of the ACM 39*, 1 (Jan. 1996), 80–91.

[12] Eichmann, D., and Srinivasan, P. Filters, webs and answers: The University of Iowa TREC-8 results. In *The Eighth Text REtrieval Conference (TREC 8)* (1999). NIST Special Publication: in press.[1]

[13] Ferret, O., Grau, B., Illouz, G., Jacquemin, C., and Masson, N. QALC - the question-answering program of the language and cognition group at LIMSI-CNRS. In *The Eighth Text REtrieval Conference (TREC 8)* (1999). NIST Special Publication: in press.[1]

[14] Gazdar, G., and Mellish, C. *Natural language processing in PROLOG: an introduction to computational linguistics.* Addison-Wesley, 1989.

[15] Hull, D. A. Xerox TREC-8 question answering track report. In *The Eighth Text REtrieval Conference (TREC 8)* (1999). NIST Special Publication: in press.[1]

[16] Humphreys, K., Gaizauskas, R., Hepple, M., and Sanderson, M. University of Sheffield TREC-8 Q & A system. In *The Eighth Text REtrieval Conference (TREC 8)* (1999). NIST Special Publication: in press.[1]

[17] Knight, S. *First Year Report.* PhD thesis, Computer Laboratory, University of Cambridge, 2000. In publication.[2]

[18] Lehnert, W. G. *The Process of Question Answering: A Computer Simulation of Cognition.* Lawrence Erlbaum Associates, 1978.

[19] Lehnert, W. G. Question answering in natural language processing. In *Natural Language Question Answering Systems*, L. Bolc, Ed. Carl Hanser Verlag, 1980, pp. 9–71.

[20] Levine, J. M., and Fedder, L. The theory and implementation of a bidirectional question answering system. Tech. Rep. 182, Computer Laboratory, University of Cambridge, Oct. 1989.

[21] Lewis, D. D., and Sparck Jones, K. Natural language processing for information retrieval. *Communications of the ACM 39*, 1 (Jan. 1996), 92–101.

[2] Available at http://www.cl.cam.ac.uk/~sfk1000

[22] Lin, C.-J., and Chen, H.-H. Description of preliminary results to TREC-8 QA task. In *The Eighth Text REtrieval Conference (TREC 8)* (1999). NIST Special Publication: in press.[1]

[23] Litkowski, K. C. Question-answering using semantic relation triples. In *The Eighth Text REtrieval Conference (TREC 8)* (1999). NIST Special Publication: in press.[1]

[24] Martin, J., and Lankester, C. Ask Me Tomorrow: the NRC and University of Ottawa question answering system. In *The Eighth Text REtrieval Conference (TREC 8)* (1999). NIST Special Publication: in press.[1]

[25] Moldovan, D., Harabagiu, S., Paşca, M., Mihalcea, R., Goodrum, R., Gîrju, R., and Rus, V. LASSO: A tool for surfing the answer net. In *The Eighth Text REtrieval Conference (TREC 8)* (1999). NIST Special Publication: in press.[1]

[26] Morton, T. S. Using coreference in question answering. In *The Eighth Text REtrieval Conference (TREC 8)* (1999). NIST Special Publication: in press.[1]

[27] Oard, D. W., Wang, J., Lin, D., and Soboroff, I. TREC-8 experiments at Maryland, CLIR, QA and routing. In *The Eighth Text REtrieval Conference (TREC 8)* (1999). NIST Special Publication: in press.[1]

[28] Ogden, B., Cowie, J., Ludovik, E., Molina-Salgado, H., Nirenburg, S., Sharples, N., and Sheremtyeva, S. CRL's TREC-8 systems. In *The Eighth Text REtrieval Conference (TREC 8)* (1999). NIST Special Publication: in press.[1]

[29] Prader, J., Radev, D., Brown, E., Coden, A., and Samn, V. The use of predictive annotation for question answering in TREC8. In *The Eighth Text REtrieval Conference (TREC 8)* (1999). NIST Special Publication: in press.[1]

[30] Radev, D. R., Prager, J., and Samn, V. Ranking suspected answers to natural language questions using predictive annotation. In *Proceedings of the Sixth Applied Natural Language Processing Conference* (Apr.-May 2000), Association for Computational Linguistics, Morgan Kaufmann, pp. 150–157.

[31] Singhal, A. 1999 TREC-8 question answering track. WWW document, July 1999.[3]

[3]http://www.research.att.com/~singhal/qa-track-spec.txt

[32] Singhal, A. Question answering track at TREC-8. WWW document, 1999.[4]

[33] Singhal, A., Abney, S., Bacchiani, M., Collins, M., Hindle, D., and Pereira, F. AT&T at TREC-8. In *The Eighth Text REtrieval Conference (TREC 8)* (1999), National Institute of Standards and Technology. NIST Special Publication: in press.[1]

[34] Sparck Jones, K., and Boguraev, B. A note on a study of cases. *Computational Linguistics 13*, 1-2 (Jan.-June 1987), 65–68.

[35] Srihari, R., and Li, W. Question answering supported by information extraction. In *The Eighth Text REtrieval Conference (TREC 8)* (1999). NIST Special Publication: in press.[1]

[36] Srihari, R., and Li, W. A question answering system supported by information extraction. In *Proceedings of the Sixth Applied Natural Language Processing Conference* (Apr.-May 2000), Association for Computational Linguistics, Morgan Kaufmann, pp. 166–172.

[37] Strzalkowski, T. TTP: a fast and robust parser for natural language. In *Proceedings of the 15th International Conference on Computational Linguistics (COLING-92)* (1992), GETA(IMAG), pp. 198–204.

[38] Strzalkowski, T. Robust text processing in automated information retrieval. In *Readings in information retrieval*, K. Sparck Jones and P. Willett, Eds. Morgan Kaufmann, 1997, pp. 317–322.

[39] Takaki, T. NTT DATA: Overview of system approach at TREC-8 ad hoc and question answering. In *The Eighth Text REtrieval Conference (TREC 8)* (1999). NIST Special Publication: in press.[1]

[40] Thompson, P., and Dozier, C. C. Name recognition and retrieval performance. In *Natural Language Processing Information Retrieval*, T. Strzalkowski, Ed. Kluwer Academic Publishers, 1999, pp. 261–272.

[41] Vorhees, E. M., and Tice, D. M. The TREC-8 question answering track evaluation. In *The Eighth Text REtrieval Conference (TREC 8)* (1999). NIST Special Publication: in press.[1]

[4] http://www.research.att.com/~singhal/qa-track.html

Part II

Appendices

Appendix A

Query/Answer Corpora

This appendix contains all query and answer passages used, both for training and blind testing.
Q denotes a query.
A denotes sentences containing a correct answer.
L denotes sentences containing a legitimate answer, though one that might not be picked out by a human.
N denotes non-answer sentences that would have been selected as candidates by an initial keyword search.
P denotes sentences containing a correct answer that would require capabilities beyond the scope of the system as originally envisaged.

Text segments marked *(text)* are those chosen by the system for highlighting. If a sentences contains no such marked segment, it was rejected by the system. Text segments marked *[text]* are the equivalent manual selections.

A.1 Training Set

Q 1 *Where is Snowdon ?*

A 1.1 *[(Snowdon) is (in Wales)].*

A 1.2 *[(Snowdon) is] a mountain [(in Wales)].*

A 1.3 *Mount [(Snowdon) is (in Wales)].*

A 1.4 *[(Snowdon, in Wales)], is a serious mountain.*

A 1.5 *[(Snowdon)], the highest mountain (in the UK), [is in Wales].*

L 1.1 *(Snowdon) is (in the UK).*

P 1.1 *(Snowdon) is located (in Wales).*

P 1.2 *(Snowdon) is to be found (in Wales).*

N 1.1 *Snowdon is easy to climb.*

N 1.2 *Snowdon is a beautiful mountain.*

N 1.3 *Snowdon is in every guide book.*

N 1.4 *Snowdon, among UK mountains, is not very impressive.*

N 1.5 *When travelling in England and Wales, you should see Snowdon.*

N 1.6 *Wales is a fine country and Snowdon a fine mountain.*

N 1.7 *Wales is a fine country and Snowdon is a fine mountain.*

N 1.8 *(Snowdon) is taller than any mountain (in England).*

Q 2 *Is there a racecourse in Newmarket ?*

A 2.1 *[There is (a racecourse in Newmarket)].*

A 2.2 *[(In Newmarket) there is (a racecourse)].*

A 2.3 *[(In Newmarket)], like many country towns, [there is (a racecourse)].*

A 2.4 *[(Newmarket has a racecourse)].*

A 2.5 *[There is (a racecourse at Newmarket)].*

A 2.6 *[There are several (racecourses in Newmarket)].*

A 2.7 *[The (racecourse)] at Ascot is finer than that [(at Newmarket)].*

A 2.8 *[As (in Newmarket), there is (a racecourse)] in Wetherby.*

A 2.9 *[There is (a racecourse)] in Wetherby, [as (in Newmarket)].*

A 2.10 *[There is (a)] beautiful [(racecourse in Newmarket)].*

A 2.11 *[(In Newmarket) there is (a)] very popular [(racecourse)].*

A 2.12 *[(In)] the Suffolk town of [(Newmarket) there is (a racecourse)].*

A 2.13 *[(In Newmarket)], a small country town, [there is (a)] magnificent [(racecourse)].*

A 2.14 *[There is (a)] fine [(racecourse)] for all lengths [(in Newmarket)].*

L 2.1 *There is a racecourse on Newmarket heath.*

N 2.1 *Newmarket immediately suggests a racecourse to me.*

N 2.2 *There is a racecourse in many small towns but I am not sure about Newmarket.*

N 2.3 *There are a lot of jockeys to be seen (in Newmarket) so there must be (a racecourse) nearby.*

Q 3 *Who is Mayor of London ?*

A 3.1 *The ballot result is that [(Ken Livingstone) is (Mayor of London)].*

A 3.2 *[(Livingstone) is (Mayor of London)].*

A 3.3 *[The (Mayor of London) is (Ken Livingstone)].*

A 3.4 *Many voters are very glad that [(Livingstone) is (Mayor of London)].*

A 3.5 *[(Ken Livingstone)] defeated all the other candidates and [is (London's Mayor)].*

L 3.1 *(Ken) is (Mayor of London)!*

N 3.1 *There were many early candidates, including Ken Livingstone, for the Mayor of London.*

N 3.2 *Frank Dobson and Ken Livingstone both wanted to be Mayor of London.*

P 3.1 *Ken Livingstone has won the race for Mayor of London.*

Q 4 *Which are the big IT companies ?*

A 4.1 *[(Microsoft, Intel and Cisco) are] all [(big IT companies)].*

A 4.2 *[(Microsoft) is a (big IT company) and so is IBM].*

A 4.3 *[(Microsoft) is a (big IT company)] and ShinyNew.com is a tiny IT company.*

A 4.4 *[(Cisco) is a] recent [(big IT company)].*

L 4.1 *(The biggest IT company) is (Microsoft).*

L 4.2 *All (the big IT companies), like (IBM), get fossilised.*

N 4.1 *The big IT companies have innovation problems.*

N 4.2 *There are many big IT companies.*

P 4.1 *(The big IT companies) include (IBM).*

Q 5 *Does Hawaii produce bananas ?*

A 5.1 *[(Hawaii produces bananas)], coconuts and dates.*

A 5.2 *[(Hawaii produces bananas)] as well as coconuts.*

A 5.3 *[(Hawaii)] does not produce apples, but [(does produce bananas)].*

A 5.4 *[(Bananas are produced by Hawaii)].*

A 5.5 *Most [(bananas are produced by Hawaii)].*

A 5.6 *[(Hawaii is producing bananas)].*

A 5.7 *[The (production of)] tropical fruit, pineapples, [(bananas)], etc, is a major operation [(in Hawaii)].*

A 5.8 *[The (production) of] tropical fruit, that is pineapples and [(bananas)], is important [(in Hawaii)].*

A 5.9 *[(Hawaii)] specializes in [the (production) of] tropical fruit, which includes [(bananas)] and pineapples.*

A 5.10 *[(Hawaii)] specializes in [the (production) of] tropical fruit, which includes pineapples and lychees, and most importantly [(bananas)].*

A 5.11 *[(Hawaii)] specializes in [the (production of)] tropical fruit, pineapples, lychees, and especially [(bananas)].*

A 5.12 *[In (Hawaii)], which is not far from California, [there is] large scale [(production of bananas)].*

A 5.13 *[In (Hawaii)], which is not far from California, [there is] large scale [(banana production)].*

A 5.14 *[(Hawaii)], the biggest island in its group, is conspicuous for [the (production of)] tropical fruit, pineapples, [(bananas)] and the like.*

A 5.15 *[(Hawaii)], which is the biggest island in the group, is conspicuous for [its (production of bananas)].*

A 5.16 *Tropical fruit production, which includes [(banana production)], is the mainstay of the economy [in] the US Pacific island (of [Hawaii)].*

A 5.17 *Tropical fruit production, including [(banana production)], is visible everywhere [(in Hawaii)].*

A 5.18 *(All A answers to Q6)*

L 5.1 *The Pacific islands like (Hawaii produce) tropical fruit like (bananas).*

N 5.1 *The US imports bananas from Jamaica and pineapples from Hawaii.*

P 5.1 *The (products of Hawaii) include (bananas).*

Q 6 *Did Hawaii produce bananas in 1991 ?*

A 6.1 *[(Hawaii produced bananas in 1991)].*

A 6.2 *[(Hawaii] used to [produce bananas in 1991)].*

A 6.3 *[(In 1991, Hawaii produced bananas)].*

N 6.1 *(All A answers to Q5)*

Q 7 *What is the height of Everest ?*

A 7.1 *[(The height of Everest) is (28000)].*

A 7.2 *[(Everest has a height of 28000)].*

A 7.3 *There are many mountains in the Himalayas, including [(Everest), which (has a height of 28000)].*

A 7.4 *[(Everest)] is the highest mountain in the world, [with (a height of 28000)].*

A 7.5 *K2 in the Himalayas has a height of 27000, K4 is 27500 high, but [(Everest)] overtops them all [with (a] magnificent [height of 28000)].*

A 7.6 *If we list [the heights of] the biggest mountains in the Himalayas we have K2 at 27000, K4 at 27500, [Everest at 28000].*

A 7.7 *If we list [the heights of] the biggest mountains in the Himalayas we have K2 27000, K4 27500, [Everest 28000].*

L 7.1 *The height of Everest is staggering.*

L 7.2 *(Everest has a staggering height).*

N 7.1 *One can ask the height of Everest, but the answer isn't really important.*

P 7.1 *(Everest) is (28000), but height isn't everything: Kayfour is more beautiful.*

P 7.2 *(Everest) is (28000), it's an impressive height.*

P 7.3 *(Everest) is (28000 high).*

Q 8 *Does the Queen live in London ?*

A 8.1 *[(The Queen lives in)] several different places which include Windsor, [(London)] and Edinburgh.*

A 8.2 *[(The Queen)] of England [(lives in London)].*

A 8.3 *People object to [(the Queen)] having so many palaces, and certainly she has them all over the place, so sometimes she lives at Windsor and sometimes she lives at Sandringham and sometimes she [(lives in London)].*

N 8.1 *The Queen often goes to London from where she lives in Windsor.*

Q 9 *Do bananas make you fat ?*

A 9.1 *There is a lot of argument about whether it is possible for fruit to make you fat, but the general opinion is that some fruit, especially if not eaten in moderation, for example [(bananas], do [make you)] very [(fat)] indeed.*

A 9.2 *Many studies have been done on the effects of eating different sorts of fruit, and the conclusion is that while citrus fruits like oranges do not make you fat, other fruits can make you fat, and [(bananas)] especially (do [make you fat)].*

A 9.3 *[(Bananas)] and papayas, but not guavas, (do [make you fat)].*

A 9.4 *Many people say they like eating fruit and they don't worry about whether it makes you fat, but eating a lot of [(bananas)] sure (does [make you fat)].*

A 9.5 *There's a lot of argument about [bananas making you fat].*

A 9.6 *[(One is made)] very [(fat by bananas)].*

A 9.7 *Eating [(bananas makes you fat)].*

A 9.8 *[(You are)] sure (going to be [made fat by)] eating [(bananas)].*

A 9.9 *The doctor said that eating [(bananas makes you fat)].*

A 9.10 *A lot of things make you fat, and [a (banana makes you fat)].*

L 9.1 *Things like chocolates and (bananas do make you fat).*

L 9.2 *(Bananas can make you fat).*

N 9.1 *Fat is a problem for many teenagers because they do so much eating, but it is not clear whether bananas are pernicious, though it is certainly the case that a lot of chocolate makes teenagers fat.*

P 9.1 *Bananas contribute to obesity.*

Q 10 *Where are Pat and Mike ?*

A 10.1 *[(Pat) and (Mike) are (in the lounge)].*

A 10.2 *[(Pat) is (in the lounge) and (Mike) is (in the garden)].*

A 10.3 *[Pat and Mike are with Sue and Mary].*

A 10.4 *[(Pat) and (Mike) are at a meeting] on new projects somewhere (in the extension building).*

A 10.5 *[(Pat) is (in the kitchen) and (Mike) is there too].*

A 10.6 *[(Pat)], who is reading, [is (in the lounge) and (Mike)], who is singing, [is (in the bathroom)].*

A 10.7 *[(Pat) is] reading [(in the lounge) and (Mike) is] singing [(in the bathroom)].*

A 10.8 *[(Pat) and (Mike)], who loathe one another, [are] unfortunately both [(in the lounge)].*

L 10.1 *Pat is in the lounge and Mike is somewhere.*

N 10.1 *I do not know where Pat and Mike are.*

N 10.2 *I know where Pat is but not Mike.*

N 10.3 *Pat is thinking and Mike is thinking too.*

P 10.1 *I think (Pat) and (Mike) are still (in the lounge).*

Q 11 *Who decides on student admissions ?*

A 11.1 *[(The secretary)] is (the person) who [(decides on student admissions)].*

A 11.2 *[(Decisions on student admissions are made by the secretary)].*

A 11.3 *[Deciding on student admissions is the secretary's] job.*

A 11.4 *[Deciding on student admissions is] the job [of the secretary].*

A 11.5 *[Deciding on student admissions is done by the secretary].*

A 11.6 *[Its deciding on student admissions that] makes [the secretary's] job hard.*

A 11.7 *[(The secretary and his advisors decide on student admissions)].*

A.2 First Blind Test Set

The sentences in this section (and the next) are listed in the order in which they were received (and submitted to the system), rather than being separated into groups of A, N, P and L.

Q 12 *Is there a library in Cambridge?*

A 12.1 *All the colleges [in (Cambridge have libraries)].*

A 12.2 *All the colleges [in (Cambridge have a library)].*

A 12.3 *Each college [in (Cambridge has a library)].*

A 12.4 *[There is (a library)], a chapel, and a hall in each of the colleges [(in Cambridge)].*

A 12.5 *[There is (a library)] in each of the colleges and each of the departments [(in Cambridge)].*

A 12.6 *[There is (a library)], with all the texts the student needs, in every college [(in Cambridge)].*

A 12.7 *Students [in Cambridge] who need books can find them in their college [library].*

P 12.1 *Students [in Cambridge] needing books can find them in the many [libraries].*

A 12.8 *Those needing books while [(in Cambridge)] can find them in one way or another, for instance in one of the many [(libraries)] that there are there.*

A 12.9 *[(Cambridge)] is a town [(with)] many [(libraries)].*

A 12.10 *[(Cambridge)] is a town that [has] many [(libraries] in) it.*

A 12.11 *[(Cambridge)] is a town which [(has libraries)] in all of its colleges and departments.*

A 12.12 *[There is (a)] university [(library in Cambridge)].*

A 12.13 *[(Cambridge)], a large university town, [has (a) very large university (library] in) it.*

A 12.14 *Oxford and [(Cambridge)] both [(have libraries)].*

A 12.15 *For those wanting [(libraries)] and laboratories when students [(in Cambridge)], there are many there.*

A 12.16 *We are pleased to announce a large benefaction to the most important [(library in Cambridge)], the university library.*

A 12.17 *What about [(libraries in Cambridge)], one may ask, and get the answer, [there are many].*

P 12.2 *All of the central facilities [in Cambridge] are proving very expensive, especially the university [library].*

A 12.18 *[(Libraries)] and laboratories are everywhere [(in Cambridge)].*

P 12.3 *One can visit many towns, [Cambridge] for instance, and not be able to find the town [library].*

N 12.1 *They announced funds for Cambridge university and for libraries.*

N 12.2 *Students, in Cambridge as elsewhere, want libraries.*

N 12.3 *If they want (a library) for disabled children (in Cambridge) they will have to raise the rates.*

A.3 Second Blind Test Set

Q 13 *Where is Trondheim ?*

A 13.1 *[(Trondheim) is (in northern Norway)].*

A 13.2 *There are a number of old towns [in Norway], of which [Trondheim] is historically the most important.*

N 13.1 *The cathedral at Trondheim, then an important town, is a fine example of the French gothic style.*

Q 14 *Who is Sylvia ?*

N 14.1 *Who is (Sylvia), what indeed is (she), as the poet asks of her ?*

A 14.1 *[Sylvia is the subject of a well-known song].*

A 14.2 *[(Sylvia) is (John's oldest daughter)].*

A 14.3 *There is noone but [(Sylvia)] for me, [(the queen of my heart)].*

Q 15 *What and where are the Dardanelles ?*

A 15.1 *[The Dardanelles are the narrow straits south of Istanbul].*

N 15.1 *The Gallipoli landings in the Dardanelles were a military disaster.*

N 15.2 *Many modern politicians have no idea where the Dardanelles are.*

A 15.2 *[The Dardanelles] are where the Gallipoli landings were, so they [are the narrow straits near Istanbul].*

Q 16 *Is Baltimore the capital of Maryland ?*

A 16.1 *Many of the states have impressive buildings in their [capital] cities, like [Baltimore in Maryland] has.*

A 16.2 *Most state [capitals] have fine official buildings, I know that [Baltimore in Maryland] has.*

A 16.3 *When in [Baltimore] you can see that this state [capital of Maryland] has fine buildings.*

N 16.1 *Baltimore in Maryland is very near the national capital.*

Q 17 *What does the Guardian newspaper cost ?*

A 17.1 *The prices of all the [newspapers] have gone up so [the Guardian], which used to be pretty reasonable, now [costs one pound].*

A 17.2 *All [newspapers have large costs] of production, but [the Guardian's] are lower than most.*

A 17.3 *[The Guardian costs a lot as a newspaper], both to produce and to buy.*

N 17.1 *The Guardian newspaper has costs.*

Q 18 *Are Mary and Joe married ?*

N 18.1 *(Mary) and (Joe) live together but I cannot say whether they are (married).*

P 18.1 *Being married is something Mary has talked about a lot but I don't know how John feels about it.*

P 18.2 *Mary has been married to John for quite a long time now.*

A 18.1 *[Mary] is very happy and busy with everything, house, children, the lot, [in being married to John].*

Q 19 *Does Baxters produce both beer and cider ?*

A 19.1 *[Baxters are the producers of] the finest [cider] in the west country but [their beer] is lousy.*

A 19.2 *[Baxters is a cider and beer] company just like all the other big [producers].*

A 19.3 *The market has decided to dump shares in companies that produce cider, even though [(Baxters produces beer as well as cider)].*

N 19.1 *There are beer companies and cider producers, and there is Baxters.*

Q 20 *How fast does Concorde fly ?*

L 20.1 *[Concorde flies as fast as it can].*

A 20.1 *[Concorde flies at 750 mph].*

A 20.2 *[Concorde flies very fast].*

A 20.3 *Among the civilian planes that fly fast, there is one that far exceeds the others, namely [Concorde], which [goes at 750 mph].*

A 20.4 *[Concorde flies at supersonic speeds].*

N 20.1 *Flying fast on Concorde is one way of getting to Washington.*

N 20.2 *Everyone likes to fly fast, and that includes the people who use Concorde.*

Appendix B

Code Listing

This appendix contains all code developed as part of the QA system. Section B.1 contains scripts for pre-processing of query and answer passages, and of lexical resources. Section B.2 contains illustrative excerpts from the lexical resources developed. Section B.3 contains the main program Prolog code.

B.1 Perl/Shell Scripts

B.1.1 Perl Code for OALD Pre-processing

`process.perl`

This script processes the OALD dictionary in its original text form into a Prolog-readable format.

```
#! /usr/bin/perl
#
# Perl script to process OTA machine-readable text710.dat file
# into a Prolog-readable format
#
# Usage: ./process.perl < text710.dat [> OUTPUT.PL]
#
# Matthew Purver, 20/6/2000

# read a line from stdin
while ($line = <STDIN>) {

    # split at white-space into words
    # remembering some "words" have a single space in
    @words = split( /\s{2,}/, $line );

    # loop through words array
    for ($i = 0; $i < 3; $i++) {
        $words[$i] =~ s/\'/\\\'/g;   # ' -> \'
        $words[$i] =~ s/,/\',\'/g;   # , -> ','
        $words[$i] =~ tr/A-Z/a-z/;   # lower case
        if ( $i == 2 ) {
            $words[$i] =~ s/\$//g;   # remove $
            $words[$i] =~ s/\%//g;   # remove %
            $words[$i] =~ s/\*//g;   # remove *
        }
```

```
    }

    # and print out the result
    print "word( '", $words[0], "', ['", $words[2], "'] ).\n";

}
```

irreg_nouns.perl

This script pulls out nouns with irregular plurals from the Prolog-readable version of the OALD. The result must be manually edited to insert the correct plurals.

```
#! /usr/bin/perl
#
# Perl script to extract nouns with irregular plurals
# from processed OTA dictionary
# the actual irregular plural forms must be put in manually!
#
# Usage: irreg_nouns.perl < ota.pl [> OUTPUT.PL]
#
# Matthew Purver, 28/6/2000

# read a line from stdin
while ($line = <STDIN>) {

    # find 'ki' marker
    # but miss out those with rule-based forms
    if ( ( $line =~ /.*\'ki\'.*/ ) &&
         ( $line !~ /.*man\'.*/ ) &&
         ( $line !~ /.*um\'.*/ ) &&
         ( $line !~ /.*us\'.*/ ) &&
         ( $line !~ /.*a\'.*/ ) &&
         ( $line !~ /.*on\'.*/ ) &&
         ( $line !~ /.*is\'.*/ ) &&
         ( $line !~ /.*o\'.*/ ) &&
         ( $line !~ /.*child\'.*/ ) &&
         ( $line !~ /.*foot\'.*/ ) &&
         ( $line !~ /.*tooth\'.*/ ) &&
         ( $line !~ /.*ouse\'.*/ ) &&
         ( $line !~ /.*fe\'.*/ ) &&
         ( $line !~ /.*f\'.*/ ) &&
         ( $line !~ /.*ex\'.*/ ) &&
         ( $line !~ /.*ix\'.*/ ) &&
         ( $line !~ /.*eau\'.*/ ) ) {

        # write out word and (poor!) guess at plural
        $line =~ s/word\( (.*), \[.*/irreg_noun\( $1, $1 \)\./ ;
        print $line;

    }

}
```

irreg_ads.perl

This script pulls out adjectives and adverbs with irregular comparative and superlative forms from the Prolog-readable version of the OALD. The result must be manually edited to insert the correct forms.

```
#! /usr/bin/perl
#
```

```
# Perl script to extract adjectives and adverbs
# with irregular comparative / superlative forms
# from processed OTA dictionary
# the actual irregular forms must be put in manually!
#
# Usage: irreg_ads.perl < ota.pl [> OUTPUT.PL]
#
# Matthew Purver, 4/7/2000

# read a line from stdin
while ($line = <STDIN>) {

    # find 'oe' marker
    if ( $line =~ /.*\'oe\'.*/ ) {

        # write out word with (poor!) guesses at comp/sup
        $line =~ s/word\( (.*)(\w)\', \[.*/
                   irreg_ad\( $1$2\', $1$2$2er\', $1$2$2est\' \)\./ ;
        print $line;

    }

}
```

B.1.2 Perl Scripts for Test Data Pre-processing

`separate.perl`

This script separates a text file containing a set of blind test data in the format received, into individual text files for each query and answer.

```
#! /usr/bin/perl
#
# Script to convert supplied blind test query/sentence text file
# into individual Q/S text files for each passage

# read line
while ($line = <STDIN>) {

    # if query: expect e.g. "Q100  Is this a query?"
    if ( $line =~ s/(Q\d+) (.+$)/$1 $2/ ) {
        `echo \"$2\" > ./$1.txt\n`;
    }
    # if answer: expect e.g. "S100.1  Yes it is."
    if ( $line =~ s/(S\d+\.\d+) (.+$)/$1 $2/ ) {
        `echo \"$2\" > ./$1.txt\n`;
    }

}
```

B.1.3 Shell Scripts for Shallow Text Processing

`shallowproc.sh`

This script is an interface to `shallowproc`.

```
#! /bin/sh

# shell script to call shallowproc text processing tools
# expects one command-line argument: the sentence to be processed (in quotes)
```

```
# e.g. shallowproc "the cat sat on the mat."

SHALLOW_DIR="../shallowproc/"
#OUTPUT_DIR="../pl/"

PROG1="./tokenise"
PROG2="./tagtext"
PROG3="./bracket"
PROG4="./cnp"
PROG5="./verbs"

# change to shallowproc dir (as tools assume we have)
cd $SHALLOW_DIR

# pass the sentence to all relevant tools
#echo $1 | $PROG1
echo $1 | $PROG1 | $PROG2 | $PROG3 | $PROG5

# no need to change back as we've called a new shell
#cd $OUTPUT_DIR
```

B.2 Prolog Lexical Resources

B.2.1 OALD Resources

Excerpt from ota.pl

This is an excerpt from the OALD once converted into a Prolog-readable form.

```
word( 'abandon', ['h0','l@'] ).
word( 'abandoned', ['hc','hd','oa'] ).
word( 'abandoning', ['hb'] ).
word( 'abandonment', ['l@'] ).
word( 'abandons', ['ha'] ).
word( 'abase', ['h2'] ).
word( 'abased', ['hc','hd'] ).
word( 'abasement', ['l@'] ).
word( 'abases', ['ha'] ).
word( 'abash', ['h1'] ).
word( 'abashed', ['hc','hd'] ).
word( 'abashes', ['ha'] ).
...
```

Excerpt from irreg_verbs.pl

This is an excerpt from the lexicon of irregular verbs.

```
% Irregular Verb List v1.0
% Sylvia Knight, April 1998

% Modified June 2000 - Matthew Purver
% now ( VB, VBP, VBZ, VBG, VBD, VBN )

irreg_verb( agree, agree, agrees, agreeing, agreed, agreed ).
irreg_verb( arise, arise, arises, arising, arose, arisen ).
irreg_verb( awake, awake, awakes, awaking, awoke, awoken ).
irreg_verb( babysit, babysit, babysits, babysitting, babysat, born ).
irreg_verb( backbite, backbite, backbites, backbiting, backbit, backbitten ).
irreg_verb( backslide, backslide, backslides, backsliding, backslid, backslid ).
irreg_verb( be, are, is, being, was, been ).
...
```

Excerpt from irreg_nouns.pl

This is an excerpt from the lexicon of irregular nouns.

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% irreg_nouns.pl
%
% List of nouns with irregular plurals
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

irreg_noun( 'frau', 'frauen' ).
irreg_noun( 'grand prix', 'grands prix' ).
irreg_noun( 'herr', 'herren' ).
irreg_noun( 'monsieur', 'messieurs' ).
...
```

Excerpt from irreg_ads.pl

This is an excerpt from the lexicon of irregular adjectives and adverbs.

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% irreg_ads.pl
%
% List of adjectives with irregular comparative
% / superlative forms
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

irreg_ad( 'bad', 'worse', 'worst' ).
irreg_ad( 'big', 'bigger', 'biggest' ).
irreg_ad( 'cruel', 'crueller', 'cruellest' ).
...
```

B.2.2 Hand-Coded Lexicon

Excerpt from lexical_semantics.pl

This is an excerpt from the semantic lexicon. It contains information about semantic class for nouns, pronouns, wh-query words and prepositions, together with lemma information for nominaliseable verbs.

```
%%----------------------------------------------------
%% lexical_semantics.pl
%%
%% Lexicon of semantic data for nouns, pronouns,
%% prepositions, wh-words
%%
%%----------------------------------------------------

%%----------------------------------------------------
%% Nouns
%% noun( Word, SemanticClass )
%%
%% Semantic classes: per(son), obj(ect), loc(ation), tim(e),
%%   abs(tract), org(anisation)
%%
%% per also has m/f gender attached
%%----------------------------------------------------

% look up locations in OTA dictionary
% (have code NM (countries) or NN (towns))
noun( A, [loc] ) :-
        word( A, Code ),
        (
          memberchk( 'nm', Code );
          memberchk( 'nn', Code )
        ).

% locations not listed in the OTA dictionary (like Birmingham!)
noun( dardanelles, [loc] ).
noun( gallipoli, [loc] ).
...

% all other classes
noun( admission, [abs] ).
noun( advisor, [per/_] ).
noun( answer, [abs] ).
```

```
noun( apple, [obj] ).
...

%%-----------------------------------------------------
%% Noun/Verb Lemma Pairs
%% lemma( Noun, Verb )
%%-----------------------------------------------------
lemma( admission, admit ).
lemma( announcement, announce ).
lemma( answer, answer ).
lemma( answer, reply ).
...

%%-----------------------------------------------------
%% Wh-query words
%% whq( Word, SemanticClass, PhraseType )
%%
%% Semantic classes: per(son), obj(ect), loc(ation), tim(e),
%%   abs(tract), org(anisation), man(ner), rea(son)
%% Phrase types: pp, np, ng, adv
%%-----------------------------------------------------
whq( where/wrb, [loc], [pp] ).
whq( when/wrb, [tim], [pp, ng, np] ).
whq( how/wrb, [man], [pp, adv] ).
whq( why/wrb, [rea], [pp] ).
whq( who/wp, [per/_, org], [ng, np] ).
whq( whom/wp, [per/_, org], [ng, np] ).
whq( whose/det, [pos], [pp] ).
whq( what/wp, [abs, obj, num], [ng, np] ).
whq( which/det, [abs, obj, org, loc], [ng, np] ).

%%-----------------------------------------------------
%% Prepositions
%% prep( Word, SemanticClass )
%%
%% Semantic classes: loc(ation), tim(e), pos(session),
%%   sop (inverse pos), man(ner)
%%-----------------------------------------------------
prep( aka, [] ).
prep( aboard, [loc] ).
prep( about, [] ).
prep( above, [loc] ).
prep( across, [] ).
prep( after, [tim] ).
...

%%-----------------------------------------------------
%% Pronouns
%% pro( Word, SemanticClass, Gender, Number, Resolvable )
%%-----------------------------------------------------
pro( he, [per/m], s, y ).
pro( her, [per/f], s, y ).
pro( it, [obj, abs, org, loc], s, y ).
...

%%-----------------------------------------------------
%% Numerical NP Anaphors
%% np_anaphor( Word, Number )
%%-----------------------------------------------------
np_anaphor( some/det, p ).
```

```
np_anaphor( one/cd, s ).
np_anaphor( lot/_, p ). % could be stemmed "lots/nns", or "a lot"
np_anaphor( few/jj, p ).


%%-------------------------------------------------------
%% PP Anaphors
%% pp_anaphor( Word, SemanticClass )
%%-------------------------------------------------------
pp_anaphor( there/ex, loc ).
pp_anaphor( there/rb, loc ).
pp_anaphor( therein/rb, loc ).
pp_anaphor( then/rb, tim ).


%%-------------------------------------------------------
%% Possessive Pronouns
%% pos_anaphor( Word, SemanticClass, Number )
%%-------------------------------------------------------
pos_anaphor( their/det, [obj, abs, org, loc, per/_], p ).
pos_anaphor( his/det, [per/m], s ).
pos_anaphor( her/det, [per/f], s ).
pos_anaphor( its/det, [obj, abs, org, loc], s ).


%%-------------------------------------------------------
%% Ditransitive Verbs
%% verb( Word, SubCat )
%%-------------------------------------------------------
verb( make, ditrans ).
verb( give, ditrans ).
verb( ask, ditrans ).


%%-------------------------------------------------------
%% Hypothetical Verbs
%% hypo_verb( Word )
%%-------------------------------------------------------
hypo_verb( want ).
hypo_verb( like ).
hypo_verb( wish ).
hypo_verb( pretend ).


%%-------------------------------------------------------
%% Generic Utility Verbs
%% generic_verb( Word )
%%-------------------------------------------------------
generic_verb( do ).
generic_verb( have ).
generic_verb( make ).
...


%%-------------------------------------------------------
%% Verbs of Possession
%% pos_verb( Word )
%%-------------------------------------------------------
pos_verb( have ).
pos_verb( contain ).
pos_verb( own ).


%%-------------------------------------------------------
%% Existential Verbs
```

```
%% be_verb( Word )
%%------------------------------------------------------
be_verb( be, _ ).
be_verb( include, _ ).
be_verb( locate, pas ).
be_verb( find, pas ).
be_verb( call, pas ).


%%------------------------------------------------------
%% Inclusive words
%% include_word( Word )
%%------------------------------------------------------
include_word( as/prep ).
include_word( like/prep ).
include_word( with/prep ).
include_word( vg:[include/vbg] ).
include_word( vg:[be/vbg] ).
include_word( 'such as'/prep ).
include_word( 'for example'/prep ).
include_word( 'for instance'/prep ).
include_word( 'that is'/prep ).
include_word( especially/rb ).
include_word( say/_ ).


%%------------------------------------------------------
%% Nouns & Adjectives of Quantity
%% quantity( Adjective, Noun )
%%------------------------------------------------------
quantity( old, age ).
quantity( far, distance ).
quantity( high, height ).
quantity( tall, height ).
quantity( wide, width ).
...


%%------------------------------------------------------
%% Definite Determiners
%% definite_det( Word )
%%-------------------------------------------------------
definite_det( both ).
definite_det( each ).
definite_det( either ).
definite_det( every ).
,,,
```

B.3 Prolog Code

B.3.1 Top Level

main.pl

```
%%----------------------------------------------------
%% main.pl
%%
%% Loads all modules.
%%
%% Once loaded, use predicates in q_a.pl to run system:
%%   query/0, query/1, query_file/1 etc. to set query
%%   answer/0, answer/1, answer_file/1,
%%   answer_all_files/2 etc. to test candidate answers
%%----------------------------------------------------

% built-in Prolog libraries
:- use_module( library( lists ) ).
:- use_module( library( system ) ).

% if old SICStus version, load replacements for missing
% built-in predicates
:- prolog_flag( version, VString ),
        name( VString, VCodes ),
        (
          (
            name( 'SICStus 3.7', OldCodes ),
            prefix( OldCodes, VCodes ),
            compile( prolog )
          );
          (
            name( 'SICStus 3.8', NewCodes ),
            prefix( NewCodes, VCodes )
          );
          (
            nl, print( '*** Unknown Prolog ***' ), nl, nl,
            abort
          )
        ).

% lexical data
% only get loaded once as they're slow & don't change often
:- current_predicate( word, _ );
        compile( '../ota/ota' ).
:- current_predicate( irreg_verb, _ );
        compile( '../ota/irreg_verbs' ).
:- current_predicate( irreg_noun, _ );
        compile( '../ota/irreg_nouns' ).
:- current_predicate( irreg_ad, _ );
        compile( '../ota/irreg_ads' ).
:- current_predicate( noun, _ );
        compile( 'lexical_semantics' ).

% low level modules
:- compile( [lists, display, debug] ).
:- compile( [io, stem, preparse] ).

% high level modules
:- compile( [syn_defs, ng_syn, vg_syn, pp, sem] ).
:- compile( [anaphora, simplify] ).
:- compile( [parse, struct] ).
:- compile( [q_a, match] ).
```

q_a.pl

```
%%-----------------------------------------------------
%% q_a.pl
%%
%% Contains: top-level Q/A predicates
%%-----------------------------------------------------

:- dynamic current_query/1, current_answer/1,
        current_answer_list/1, process_type/1.

%%-----------------------------------------------------
%% query/0
%%
%% Description:
%%   Prompts user for query and calls query/1
%% Succeeds: 0-1
%% Side Effects: may cause current_query/1 to be asserted
%%------------------------------------------------------

query :-
        write( 'Query? :>' ),
        read( Text ),
        query( Text ).


%%-----------------------------------------------------
%% query/1
%% query( +String )
%%
%% Description:
%%   Calls tagger and reads result, passes to query_main/1
%% Succeeds: 0-1
%% Side Effects: may cause current_query/1 to be asserted
%%------------------------------------------------------

query( Text ) :-
        call_tagger( Text, Stream ),
        read_sentence( Stream, Query ),
        !,
        query_main( Query ).


%%-----------------------------------------------------
%% query_file/0
%%
%% Description:
%%   Prompts for query file name, calls query_file/1
%% Succeeds: 0-1
%% Side Effects: may cause current_query/1 to be asserted
%%------------------------------------------------------

query_file :-
        write( 'Query file name? :>' ),
        read( QueryFile ),
        query_file( QueryFile ).


%%-----------------------------------------------------
%% query_file/1
%% query_file( +FileNameString )
%%
%% Description:
%%   Passes query file to tagger, calls query_main/1
%% Succeeds: 0-1
```

```
%% Side Effects: may cause current_query/1 to be asserted
%%--------------------------------------------------------

query_file( QueryFile ) :-
        call_tagger( QueryFile, Stream ),
        read_sentence( Stream, Query ),
        !,
        query_main( Query ).


%%-------------------------------------------------------
%% query_tagged_file/0
%%
%% Description:
%%   Prompts for *tagged* query file name, calls
%%   query_tagged_file/1
%% Succeeds: 0-1
%% Side Effects: may cause current_query/1 to be asserted
%%--------------------------------------------------------

query_tagged_file :-
        write( 'Query␣file␣name?␣:>' ),
        read( QueryFile ),
        query_file( QueryFile ).


%%-------------------------------------------------------
%% query_tagged_file/1
%% query_tagged_file( +FileNameString )
%%
%% Description:
%%   Reads *tagged* query from file, calls query_main/1
%% Succeeds: 0-1
%% Side Effects: may cause current_query/1 to be asserted
%%--------------------------------------------------------

query_tagged_file( QueryFile ) :-
        get_sentence( QueryFile, Query ),
        !,
        query_main( Query ).


%%-------------------------------------------------------
%% query_main/1
%% query_main( +SentList )
%%
%% Description:
%%   Gets structure from query sentence and asserts as
%%   current_query/1
%% Succeeds: 0-1
%% Side Effects: may cause current_query/1 to be asserted
%%--------------------------------------------------------

query_main( Query ) :-
        retractall( current_query( _ ) ),
        retractall( process_type( _ ) ),
        assert( process_type( query ) ),
        pre_process( Query, Q1 ),
        process_query( Q1, Q ),
        display_struct( Q ),
        assert( current_query( Q ) ).


%%-------------------------------------------------------
%% answer/0
```

```
%%
%% Description:
%%   Succeeds if text contains an answer to current query.
%%   Prompts user for potential answer and calls answer/1
%% Succeeds: 0-1
%% Side Effects: may cause current_answer/1 to be asserted
%%-------------------------------------------------------

answer :-
        write( 'Answer? :>' ),
        read( Text ),
        answer( Text ).


%%------------------------------------------------------
%% answer/1
%% answer( +String )
%%
%% Description:
%%   Succeeds if String contains an answer to current query.
%%   Calls tagger and reads result, passes to answer_main/1
%% Succeeds: 0-1
%% Side Effects: may cause current_answer/1 to be asserted
%%-------------------------------------------------------

answer( Text ) :-
        call_tagger( Text, Stream ),
        read_sentence( Stream, Answer ),
        !,
        answer_main( Answer, _, _ ).


%%------------------------------------------------------
%% answer_file/0
%%
%% Description:
%%   Succeeds if file contains an answer to current query.
%%   Prompts for potential answer file name,
%%   calls answer_file/1
%% Succeeds: 0-1
%% Side Effects: may cause current_answer/1 to be asserted
%%-------------------------------------------------------

answer_file :-
        write( 'Answer file name? :>' ),
        read( AnswerFile ),
        answer_file( AnswerFile ).


%%------------------------------------------------------
%% answer_file/1
%% answer_file( +FileNameString )
%%
%% Description:
%%   Succeeds if file contains an answer to current query.
%%   Reads potential answer from file,
%%   calls answer_main/1.
%%   Just an interface to answer_file/3 discarding the
%%   answer info
%% Succeeds: 0-1
%% Side Effects: may cause current_answer/1 to be asserted
%%-------------------------------------------------------

answer_file( File ) :-
        answer_file( File, _, _ ).
```

```
%%-----------------------------------------------------
%% answer_file/3
%% answer_file( +FileNameString, -Answer, -Highlight )
%%
%% Description:
%%   Succeeds if file contains an answer to current query.
%%   Reads potential answer from file, calls
%%   answer_main/1 and passes back answer and output text
%% Succeeds: 0-1
%% Side Effects: may cause current_answer/1 to be asserted
%%------------------------------------------------------

answer_file( AnswerFile, Ans, Highlight ) :-
        call_tagger( AnswerFile, Stream ),
        read_sentence( Stream, Answer ),
        !,
        answer_main( Answer, Ans, Highlight ).


%%-----------------------------------------------------
%% answer_tagged_file/0
%%
%% Description:
%%   Succeeds if file contains an answer to current query.
%%   Prompts for *tagged* potential answer file name,
%%   calls answer_tagged_file/1
%% Succeeds: 0-1
%% Side Effects: may cause current_answer/1 to be asserted
%%------------------------------------------------------

answer_tagged_file :-
        write( 'Answer file name? :>' ),
        read( AnswerFile ),
        answer_tagged_file( AnswerFile ).


%%-----------------------------------------------------
%% answer_tagged_file/1
%% answer_tagged_file( +FileNameString )
%%
%% Description:
%%   Succeeds if file contains an answer to current query.
%%   Reads *tagged* potential answer from file,
%%   calls answer_main/1.
%%   Just an interface to answer_tagged_file/3 discarding the
%%   answer info
%% Succeeds: 0-1
%% Side Effects: may cause current_answer/1 to be asserted
%%------------------------------------------------------

answer_tagged_file( File ) :-
        answer_tagged_file( File, _, _ ).


%%-----------------------------------------------------
%% answer_tagged_file/3
%% answer_tagged_file( +FileNameString, -Answer, -Highlight )
%%
%% Description:
%%   Succeeds if file contains an answer to current query.
%%   Reads *tagged* potential answer from file, calls
%%   answer_main/1 and passes back answer and output text
%% Succeeds: 0-1
```

```
%% Side Effects: may cause current_answer/1 to be asserted
%%-----------------------------------------------------

answer_tagged_file( AnswerFile, Ans, Highlight ) :-
        get_sentence( AnswerFile, Answer ),
        !,
        answer_main( Answer, Ans, Highlight ).


%%----------------------------------------------------
%% answer_main/3
%% answer_main( +SentList, -Answer )
%%
%% Description:
%%   Succeeds if SentList contains an answer to the
%%   current query.
%%   Calls answer_list/3 and passes answer back
%% Succeeds: 0*
%% Side Effects: may cause current_answer/1 to be asserted
%%-----------------------------------------------------

answer_main( Answer, Ans, Highlight ) :-
        retractall( process_type( _ ) ),
        assert( process_type( answer ) ),
        current_query( QList ),
        answer_list( QList, Answer, Ans, Highlight ).


%%----------------------------------------------------
%% answer_list/4
%% answer_list( +QueryStructList, +AnswerSentList, -AnswerList )
%%
%% Description:
%%   Succeeds if AnswerSentList contains an answer to each
%%   element in the current query list.
%%   AnswerList will contain the answer for each element
%% Succeeds: 0*
%% Side Effects: may cause current_answer/1 to be asserted
%%-----------------------------------------------------

% base case
answer_list( [], _Ans, [], [] ).

% for each query, process answer and call match/2,
% then get current answer if it succeeds
answer_list( [Q | Tail], Answer, [Ans | ATail], [High | HTail] ) :-
        retractall( current_answer( _ ) ),
        retractall( current_answer_list( _ ) ),
        display_simple( 'Query', Q ),
        pre_process( Answer, A1 ),
        process_answer( A1, A, Q ),
        display_simple( 'Found', A ),
        match( Q, A ),
        display_struct( A ),
        display_answer,
        current_answer( Ans ),
        current_answer_list( High ),
        !,
        answer_list( Tail, Answer, ATail, HTail ).


%%----------------------------------------------------
%% answer_all_files/2
%% answer_all_files( +DirNameString, ?FileNamePatternString )
%%
```

```
%% Description:
%%   Tests each potential answer file matching
%%   FileNamePatternString in the directory DirNameString
%%   against the current query, and displays results.
%%   Gets list of file names & calls answer_all_files/3
%% Succeeds: 1
%% Side Effects: may cause current_answer/1 to be asserted
%%-----------------------------------------------------

answer_all_files( Dir, Pattern ) :-
        directory_files( Dir, FileList ),
        answer_all_files( Dir, Pattern, FileList ).


%%-----------------------------------------------------
%% answer_all_files/3
%% answer_all_files( +DirNameString, ?FileNamePatternString,
%%                   +FileNameList )
%%
%% Description:
%%   Tests each potential answer file in FileNameList matching
%%   FileNamePatternString in the directory DirNameString
%%   against the current query, and displays results.
%%   Tests name patter match and calls answer_file/3
%% Succeeds: 1
%% Side Effects: may cause current_answer/1 to be asserted
%%-----------------------------------------------------

% base case
answer_all_files( _Dir, _Pattern, [] ).

% check name matches pattern, call answer_file/2
% then display successful answer
answer_all_files( Dir, Pattern, [Head | Tail] ) :-
        check_slash( Dir, SDir ),
        atom_concat( Pattern, _, Head ),
        atom_concat( SDir, Head, File ),
        answer_file( File, Ans, Highlight ),
        !,
        display_result( 'Succeeded', Head, Ans, Highlight ),
        answer_all_files( Dir, Pattern, Tail ).

% if name matches pattern but answer_file/2 failed,
% display failure message
answer_all_files( Dir, Pattern, [Head | Tail] ) :-
        atom_concat( Pattern, _, Head ),
        !,
        display_result( 'Failed', Head, [], [] ),
        answer_all_files( Dir, Pattern, Tail ).

% if name doesn't match pattern, recurse doing nothing
answer_all_files( Dir, Pattern, [_Head | Tail] ) :-
        answer_all_files( Dir, Pattern, Tail ).


%%-----------------------------------------------------
%% answer_all_tagged_files/2
%% answer_all_tagged_files( +DirNameString, ?FileNamePatternString )
%%
%% Description:
%%   Tests each potential answer file matching
%%   FileNamePatternString in the directory DirNameString
%%   against the current query, and displays results.
%%   Gets list of file names & calls answer_all_tagged_files/3
%% Succeeds: 1
```

```
%% Side Effects: may cause current_answer/1 to be asserted
%%-------------------------------------------------------

answer_all_tagged_files( Dir, Pattern ) :-
        directory_files( Dir, FileList ),
        answer_all_tagged_files( Dir, Pattern, FileList ).


%%------------------------------------------------------
%% answer_all_tagged_files/3
%% answer_all_tagged_files( +DirNameString, ?FileNamePatternString,
%%                   +FileNameList )
%%
%% Description:
%%   Tests each potential answer file in FileNameList matching
%%   FileNamePatternString in the directory DirNameString
%%   against the current query, and displays results.
%%   Tests name patter match and calls answer_tagged_file/3
%% Succeeds: 1
%% Side Effects: may cause current_answer/1 to be asserted
%%-------------------------------------------------------

% base case
answer_all_tagged_files( _Dir, _Pattern, [] ).

% check name matches pattern, call answer_tagged_file/2
% then display successful answer
answer_all_tagged_files( Dir, Pattern, [Head | Tail] ) :-
        check_slash( Dir, SDir ),
        atom_concat( Pattern, _, Head ),
        atom_concat( SDir, Head, File ),
        answer_tagged_file( File, Ans, Highlight ),
        !,
        display_result( 'Succeeded', Head, Ans, Highlight ),
        answer_all_tagged_files( Dir, Pattern, Tail ).

% if name matches pattern but answer_tagged_file/2 failed,
% display failure message
answer_all_tagged_files( Dir, Pattern, [Head | Tail] ) :-
        atom_concat( Pattern, _, Head ),
        !,
        display_result( 'Failed', Head, [], [] ),
        answer_all_tagged_files( Dir, Pattern, Tail ).

% if name doesn't match pattern, recurse doing nothing
answer_all_tagged_files( Dir, Pattern, [_Head | Tail] ) :-
        answer_all_tagged_files( Dir, Pattern, Tail ).


%%------------------------------------------------------
%% process_query/2
%% process_query( +RawSentList, ?StructList )
%%
%% Description:
%%   StructList is a list of pred-arg/state structures
%%   corresponding to the sentence RawSentList.
%%   StructList has more than one member if the query
%%   contains conjunctions, with one member for each
%%   logical sub-query
%%   Calls parse/2 to parse, simplify/2 to simplify syntax,
%%   prep_struct/2 to create existential relations,
%%   q_sub_sent/2 to find all possible simple sub-sentence,
%%   then q_struct_list/2 to create list of structures
%% Succeeds: 1*
%% Side Effects: none
```

```
%%-------------------------------------------------------

% parse, make list of conjunctions, get struct for each
process_query( RawSent, StructList ) :-
        parse( RawSent, ParseSent ),
        simplify( ParseSent, SimpleSent ),
        prep_struct( SimpleSent, PrepSent ),
        findall( C, q_sub_sent( PrepSent, C ), CList ),
        q_struct_list( CList, StructList ).

% if failed, report
process_query( RawSent, [] ) :-
        process_fail( RawSent ).


%%------------------------------------------------------
%% process_answer/3
%% process_answer( +RawSentList, ?AStruct, +QStruct )
%%
%% Description:
%%   AStruct is a pred-arg/state structure extracted from
%%   the sentence RawSentList which might match the query
%%   structure QStruct.
%%   Calls parse/2 to parse, simplify/2 to simplify syntax,
%%   prep_struct/2 to create existential relations,
%%   sub_sent/2 to pull out one simple sub-sentence, then
%%   a_struct/3 to create structure
%% Succeeds: 1*
%% Side Effects: none
%%-------------------------------------------------------

% parse, get sub-sentence, get struct
process_answer( RawSent, Struct, Query ) :-
        parse( RawSent, ParseSent ),
        simplify( ParseSent, SimpleSent ),
        prep_struct( SimpleSent, PrepSent ),
        sub_sent( PrepSent, SubSent ),
        a_struct( SubSent, Struct, Query ).

% if failed, report
process_answer( RawSent, [], _ ) :-
        process_fail( RawSent ).


%%------------------------------------------------------
%% process_fail/1
%% process_fail( +RawSentList )
%%
%% Description:
%%   Displays the parse tree and fails, to assist in
%%   debugging / understanding failures
%% Succeeds: 0
%% Side Effects: none
%%-------------------------------------------------------

% parse, remove e() entities, display tree, fail
process_fail( RawSent ) :-
        parse( RawSent, ParseSent ),
        subst_e( ParseSent, CookedSent ),
        display_simple( 'Parsed', CookedSent ),
        display_struct( CookedSent ),
        !,
        fail.
```

B.3.2 Structural Matching/Transformation

match.pl

```
%%----------------------------------------------------
%% match.pl
%%
%% Contains: predicates for structure / element matching
%%----------------------------------------------------

%%----------------------------------------------------
%% match/2
%% match( +Q, +A )
%%
%% Description:
%%   Succeeds if Q and A match. Interface to match/5.
%%   Q, A may be structures, phrases or words
%% Succeeds: 0*
%% Side Effects: may assert instances of current_answer/1
%%-----------------------------------------------------

% call match/5, assert answer text and  display debug trace
% assert answer structure as the last current_answer/1 so it
% will only be picked up in the case of yn-queries where no
% other answer has been asserted
match( Q, A ) :-
        match( Q, A, AnswerList, y/y, Trace ),
        !,
        stem( SurfaceAns, AnswerList ),
        assert( current_answer_list( SurfaceAns ) ),
        assertz( current_answer( yes ) ),
        display_trace( 'Succeeded:', Trace ).

% if above fails, get rid of false answers
match( _, _ ) :-
        retractall( current_answer( _ ) ),
        fail.


%%----------------------------------------------------
%% match/3
%% match( +Q, +A, +ReverseArgsAtom )
%%
%% Description:
%%   Succeeds if Q and A match. Interface to match/4
%%   without debugging info.
%%   Q, A may be structures, phrases or words
%% Succeeds: 0*
%% Side Effects: may assert instances of current_answer/1
%%-----------------------------------------------------

% call match/4 and throw away debugging info
match( Q, A, YN ) :-
        match( Q, A, YN, _ ),
        !.


%%----------------------------------------------------
%% match/5
%% match( +Q, +A, -AnswerList,
%%        +ReverseArgsAtom, -DebugTrace )
%%
%% Description:
%%   Succeeds if Q and A match.
```

```
%%   Q, A may be structures, phrases or words.
%%   AnswerList contains the narrow-class matched phrase
%%   for display.
%%   ReverseArgsAtom prevents infinite recursion when
%%   matching reversed arguments in state-structures:
%%   should be initially set to 'y'.
%%   DebugTrace will contain a list of rules applied with
%%   details of what matched
%% Succeeds: 0*
%% Side Effects: may assert instances of current_answer/1
%%-----------------------------------------------------

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% The obvious case
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% anything matches itself
% except whole structures & question words
% (to prevent "where is snowdon" matching "where is snowdon")
match( A, A, A,
       _/_, [[A, A, 0]] ) :-
        \+ A = s:_,
        \+ (
             A = WhWord,
             whq( WhWord, _, _ )
           ),
        !.


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% Word rules
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% nouns match the same word with a N-type tag
match( Noun/Type1, Noun/Type2, Noun/Type2,
       _/_, [[Noun/Type1, Noun/Type2, 1]] ) :-
        member( Type1, [nn, np, nns] ),
        member( Type2, [nn, np, nns] ),
        !.

% adjectives match the same word with a Adj-type tag
% as long as the tag in the answer is at least as "strong" as in the query
match( Adj/Type1, Adj/Type2, Adj/Type2,
       _/_, [[Adj/Type1, Adj/Type2, 2]] ) :-
        nth( Q, [jj, jjr, jjs], Type1 ),
        nth( A, [jj, jjr, jjs], Type2 ),
        A >= Q,
        !.

% unresolvable pronouns match (e.g. "you" matches "one")
% remembering that "one" may be tagged CD
match( P1/Tag1, P2/Tag2, P2/Tag2,
       _/_, [[P1/Tag1, P2/Tag2, 3]] ) :-
        memberchk( Tag1, [cd, pp] ),
        memberchk( Tag2, [cd, pp] ),
        pro( P1, _, _, n ),
        pro( P2, _, _, n ),
        !.


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% Verb group rules
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% VGs match if voice & head match and any modifiers of
```

```
% query VG can be matched
match( vg:Voice#Pred:[v:V1 | T1], vg:Voice#Pred:[v:V2 | T2], [V2 | Ans],
       _/_, [[[v:V1 | T1], [v:V2 | T2], 11] | Trace] ) :-
        sublist( T2A, T2 ),
        match_list( T1, T2A, Ans,
                    y/y, Trace ),
        !.

% adverbial modifiers match if all query constituents can be matched
match( adv:List1, adv:List2, Ans,
       _/_, [[adv:List1, adv:List2, 12] | Trace] ) :-
        sublist( List2A, List2 ),
        length( List1, L ),
        length( List2A, L ),
        permutation( List2A, List2B ),
        match_list( List1, List2B, Ans,
                    y/y, Trace ),
        !.


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% Query word rules
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% wh-question words match any phrase if semantics match
% (which gets asserted as answer)
match( WhWord, Type:Feat:Phrase, Phrase,
       _/_, [[WhWord, Type:Feat:Phrase, 21]] ) :-
        whq( WhWord, WhSemList, WhTypeList ),
        member( Type, WhTypeList ),
        member( Sem, WhSemList ),
        sem( Type:Feat:Phrase, Sem ),
        asserta( current_answer( Phrase ) ),
        !.

% wh-question words match themselves, pre-qualified
% (e.g. "Wales is where (Snowdon is)")
% (first phrase gets asserted as answer)
match( WhWord, T1:F1:[T2:F2:P2, WhWord], P2,
       _/_, [[WhWord, T1:F1:[T2:F2:P2, WhWord], 22]] ) :-
        whq( WhWord, WhSemList, _WhTypeList ),
        member( Sem, WhSemList ),
        sem( T2:F2:P2, Sem ),
        asserta( current_answer( T2:F2:P2 ) ),
        !.

% how + adj-word (e.g. how green? (is my valley))
% (not allowed for quantities (e.g. how high? <-/-> very high)
match( ng:F1:[how/wrb, np:F1:[Adj/AdjType]], Ans,
       Type:Feat:Phrase,
       _/_, [[ng:F1:[how/wrb, np:F1:[Adj/AdjType]],
             Type:Feat:Phrase, 23] | Trace] ) :-
        ad_type( AdjType ),
        \+ quantity( Adj, _ ),
        flatten_nps( Phrase, P2 ),
        member( Adj2/Type, P2 ),
        ad_type( Type ),
        match( Adj/AdjType, Adj2/Type, Ans,
               y/y, Trace ),
        !.

% same for quantities (e.g. how long? (is a piece of string))
% -> we want to allow them to match a quantity phrase, so allow
% them to match whatever "what length?" matches
match( ng:F1:[how/wrb, np:F1:[Adj/AdjType]], A, Ans,
```

```
        _/_, [[ng:F1:[how/wrb, np:F1:[Adj/AdjType]],
               ng:[num]:[what/wp, np:[num]:[Noun/nn]], 24] | Trace]) :-
         ad_type( AdjType ),
         quantity( Adj, Noun ),
         match( ng:[num]:[what/wp, np:[num]:[Noun/nn]], A, Ans,
                y/y, Trace ),
         !.

% what + quantity noun (e.g. what height? (is Everest))
% can match any quantity NG
% (which gets asserted as answer)
match( ng:F1:[what/wp, np:F1:[Noun/NType]], A, A,
        _/_, [[ng:F1:[what/wp, np:F1:[Noun/NType]], A, 25]]) :-
         noun_type( NType ),
         quantity( _, Noun ),
         sem( A, num ),
         asserta( current_answer( A ) ),
         !.

% what + quantity noun (e.g. what height? (is Everest))
% can match any inv-possessive PP containging a suitable NG
% (everest, with its 28000 height),(everest, a mountain of 28000)
match( ng:F1:[what/wp, np:F1:[Noun/NType]], pp:Feat:[Prep, NP], Ans,
        _/_, [[pp:Feat:[Prep, NP], NP, 26] | Trace]) :-
         noun_type( NType ),
         quantity( _, Noun ),
%        member( sop, Feat ),
         np( NP ),
         match( ng:F1:[what/wp, np:F1:[Noun/NType]], NP, Ans,
                y/y, Trace ),
         !.

% for quantity nouns,
% (e.g. "what is the height of" -> "what height is")
% this is actually a whole structure rule, but easier here
match( s:[what/wp, ng:F1:[NP1, pp:F2:[of/prep, NP2]]], A, Ans,
        _/_, [[s:[what/wp, ng:F1:[NP1, pp:F2:[of/prep, NP2]]],
               s:[ng:F1:[what/wp, Noun], NP2], 27] | Trace]) :-
         np_head( NP1, Noun/_ ),
         quantity( _Adj, Noun ),
         match( s:[ng:F1:[what/wp, np:s/[num]:[Noun/nn]], NP2], A, Ans,
               y/y, Trace ),
         !.

% what/which + any NP (e.g. what city? (is the capital of France))
% can match any noun phrase that matches NP sem
% (which gets asserted as answer)
match( ng:F1:[WhWord, NP], A, A,
        _/_, [[ng:F1:[WhWord, NP], A, 28]]) :-
         member( WhWord, [what/wp, which/det] ),
         np( NP ),
         np( A ),
         sem( NP, Sem ),
         sem( A, Sem ),
         asserta( current_answer( A ) ),
         !.


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% Noun group rules
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% any phrase matches a compound NP if it matches one part of it
match( Q, Type:Feat:List, Ans,
        _/_, [[Q, Type:Feat:List, 31] | Trace] ) :-
```

```
        np( Type:Feat:List ),
        member( A, List ),
        match( Q, A, Ans,
               y/y, Trace ),
        !.

% a compound NP matches another if all query constituents match
% and semantics match
match( ng:_N1/Sem1:List1, ng:_N2/Sem2:List2, Ans,
       _/_, [[List1, List2, 32] | Trace] ) :-
        member( Sem, Sem1 ),
        member( Sem, Sem2 ),
        sublist( List2A, List2 ),
        length( List1, L ),
        length( List2A, L ),
        permutation( List2A, List2B ),
        match_list( List1, List2B, Ans,
                    y/y, Trace ),
        !.

% any NP matches another if all query constituents match after
% flattening sub-NPs, and semantics match
match( T1:_N1/Sem1:List1, T2:_N2/Sem2:List2, Ans,
       _/_, [[List1, List2, 33] | Trace] ) :-
        np_type( T1 ),
        np_type( T2 ),
        member( Sem, Sem1 ),
        member( Sem, Sem2 ),
        flatten_nps( List1, FlatList1 ),
        flatten_nps( List2, FlatList2 ),
        sublist( List2A, FlatList2 ),
        length( FlatList1, L ),
        length( List2A, L ),
        permutation( List2A, List2B ),
        match_list( FlatList1, List2B, Ans,
                    y/y, Trace ),
        !.

% a simple NP matches another simple NP if all its words (except
% non-required determiners) are there and semantics match
% (but not necessarily number)
match( np:N1/Sem1:List1, np:N2/Sem2:List2, Ans,
       _/_, [[np:N1/Sem:List1, np:N2/Sem:List2, 34] | Trace] ) :-
        member( Sem, Sem1 ),
        member( Sem, Sem2 ),
        remove_dets( List1, List1A ),
        sublist( List2A, List2 ),
        length( List1A, L ),
        length( List2A, L ),
        permutation( List2A, List2B ),
        match_list( List1A, List2B, Ans,
                    y/y, Trace ),
        !.

% NP matches "NP2 of NP" if sem the same
% i.e. "the town of Newmarket" OK, "the mayor of London" not OK
match( A, ng:F1:[B, pp:F2:[of/prep, C]], Ans,
       _/_, [[A, ng:F1:[B, pp:F2:[of/prep, C]], 35] | Trace] ) :-
        np( A ),
        sem( B, Sem ),
        sem( C, Sem ),
        match( A, C, Ans,
               y/y, Trace ),
        !.
```

```
% NP1 matches "like NP2" if NP1 matches NP2
match( Q, Type:Feat:[Word, A], Ans,
       _/_, [[Q, Type:Feat:[Word, A], 36] | Trace] ) :-
        include_word( Word ),
        match( Q, A, Ans,
               y/y, Trace ),
        !.


% after verb nominalisation from state structures,
% NPs can match PPs containing them (or NGs containing those PPs)
match( Q, A, [Prep, Ans],
       lemma/_, [[A, NP, 37] | Trace] ) :-
        np( Q ),
        (
            tree_member( pp:_Feat:[Prep/prep, NP], A );
            A = pp:_Feat:[Prep/prep, NP]
        ),
        np( NP ),
        match( Q, NP, Ans,
               y/y, Trace ),
        !.


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% Prepositional phrase rules
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% PPs match if their consituent NGs match and they share semantics
match( pp:Sem1:[Prep1, NP1], pp:Sem2:[Prep2, NP2], [Prep2, Ans],
       _/_, [[pp:Sem1:[Prep1, NP1], pp:Sem2:[Prep2, NP2], 41] | Trace] ) :-
        member( Sem, Sem1 ),
        member( Sem, Sem2 ),
        match( NP1, NP2, Ans,
                     y/y, Trace ),
        !.

% something matches a stacked PP if it matches the inner PP and
% all levels share semantics
% "in Suffolk" -> "in a town in Suffolk" (but not the other way around)
match( Q, pp:Sem1:[Prep1, ng:_Feat:[NP1, pp:Sem2:[Prep2, NP2]]], Ans,
       _/_, [[pp:Sem1:[Prep1, ng:_Feat:[NP1, pp:Sem2:[Prep2, NP2]]],
              pp:Sem2:[Prep2, NP2], 42] | Trace] ) :-
        member( Sem, Sem1 ),
        member( Sem, Sem2 ),
        match( Q, pp:Sem2:[Prep2, NP2], Ans,
               y/y, Trace ),
        !.

% PP "of X" matches possessive form "X's"
match( pp:_Sem:[of/prep, NP1], ng:F1:[NP2, np:F2:[Pos/pos]], Ans2,
       _/_, [[pp:_Sem:[of/prep, NP1],
              ng:F1:[NP2, np:F2:[Pos/pos]], 43] | Trace] ) :-
        match( NP1, NP2, Ans,
               y/y, Trace ),
        append( [Ans], [Pos/pos], Ans2 ),
        !.

% and the same the other way around
match( ng:F1:[NP1, np:F2:[Pos/pos]], pp:_Sem:[of/prep, NP2], [of/prep | Ans],
       _/_, [[ng:F1:[NP1, np:F2:[Pos/pos]],
              pp:_Sem:[of/prep, NP2], 43] | Trace] ) :-
        match( NP1, NP2, Ans,
               y/y, Trace ),
```

```
        !.


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% General structure rules
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% first try to match pred & args individually (no transformation)
match( s:Q, s:A, AnswerList,
       YN, [[s:Q, s:A, 100] | Trace] ) :-
       match_list( Q, A, AnswerList,
                   YN, Trace ),
       !.

% existential cases match two phrases within a single arg
% as long as that arg isn't a list of conjoined NPs
match( Q, s:Args, Ans,
       _/_, [[Q, Type:Feat:List, 101] | Trace] ) :-
        tree_member( Type:Feat:List, Args ),
        \+ list_of_nps_no_punct( List ),
        sublist( [A, B], List ),
        match( Q, s:[A, B], Ans,
               y/y, Trace ),
        !.

% a query matches a passive answer if it matches the active version
match( Q, s:[vg:pas#Pred:VG2, A, pp:_:[by/prep, B]],
       [AnsP, AnsA, [by/prep, AnsB]],
       _/_, [[Q, [VG2, A, [by/prep, B]], 102] | Trace] ) :-
        match( Q, s:[vg:act#Pred:VG2, B, A], [AnsP, AnsB, AnsA],
               y/y, Trace ),
        !.

% and the same the other way around:
% a passive query matches an answer if its active equivalent does
match( s:[vg:pas#Pred:VG1, A, pp:_:[by/prep, B]], S, Ans,
       _/_, [[[VG1, A, [by/prep, B]], S, 102] | Trace] ) :-
        match( s:[vg:act#Pred:VG1, B, A], S, Ans,
               y/y, Trace ),
        !.

% possessive pred-arg structures match possessive PP state-structures
match( s:[vg:act#Pred:_, NP1, NP2], A, Ans,
       _/_, [[s:[Pred, NP1, NP2], s:[pp:[of/prep, NP1], NP2], 103] |
            Trace] ) :-
        pos_verb( Pred ),
        np( NP1 ),
        np( NP2 ),
        match( s:[pp:[loc,pos]:[of/prep, NP1], NP2], A, Ans,
               y/y, Trace ),
        !.

% and the same the other way around
match( Q, s:[vg:act#Pred:[v:VG | _], NP1, NP2], [VG, Ans1, Ans2],
       _/_, [[s:[Pred, NP1, NP2], s:[pp:[of/prep, NP1], NP2], 103] |
            Trace] ) :-
        pos_verb( Pred ),
        np( NP1 ),
        np( NP2 ),
        match( Q, s:[pp:[loc,pos]:[of/prep, NP1], NP2], [[_Prep, Ans1], Ans2],
               y/y, Trace ),
        !.

% and the equivalent rule for inverse possession
match( Q, s:[NP1, pp:Feat:[Prep, NP2]], [Ans1, [Prep, Ans2]],
```

```
       _/_, [[s:[NP1, pp:[Prep, NP2]], s:[pp:[of/prep, NP1], NP2], 104] |
            Trace] ) :-
        np( NP1 ),
        np( NP2 ),
        member( sop, Feat ),
        match( Q, s:[pp:[loc,pos]:[of/prep, NP1], NP2], [[_APrep, Ans1], Ans2],
               y/y, Trace ),
        !.

% query matches "NP1 has NP2" if it matches "NP1 is NP2" for quantities
match( Q, s:[vg:act#Pred:[v:VG | _], NP1, NP2], [VG | Ans],
       _/_, [[[have, NP1, NP2], [NP1, NP2], 105] | Trace] ) :-
        pos_verb( Pred ),
        sem( NP2, num ),
        match( Q, s:[NP1, NP2], Ans,
               y/y, Trace ),
        !.

% "what is the name of (person)" translates to "who is (person)"
% "what is the name of (non-person)" translates to "what is (non-person)"
match( s:[what/wp, ng:F1:[np:_F2:[_/det, name/_],
                          pp:_F3:[of/prep, NP]]], A, Ans,
       _/_, [[ng:F1:[WhWord, NP], A, 106] | Trace]) :-
        np( NP ),
        sem( NP, Sem ),
        whq( who/wp, WhoSem, _ ),
        ( memberchk( Sem, WhoSem ) ->
            WhWord = who/wp
        ;
            WhWord = what/wp
        ),
        match( s:[WhWord, NP], A, Ans,
               y/y, Trace ),
        !.

% "name the (person)" translates to "who is (person)"
% "name the (non-person)" translates to "what is (non-person)"
match( s:[np:Feat:[name/_], NP], A, Ans,
       _/_, [[np:Feat:[WhWord, NP], A, 107] | Trace]) :-
        np( NP ),
        sem( NP, Sem ),
        whq( who/wp, WhoSem, _ ),
        ( memberchk( Sem, WhoSem ) ->
            WhWord = who/wp
        ;
            WhWord = what/wp
        ),
        match( s:[WhWord, NP], A, Ans,
               y/y, Trace ),
        !.

% "name the (person) who" translates to "which (person)"
% "name the (non-person) which" translates to "which (non-person)"
match( s:[ng:_:[np:_:[name/_], NP, WhWord], X], A, Ans,
       _/_, [[[ng:F:[which/det, NP], X], A, 108] | Trace]) :-
        np( NP ),
        (
          whq( WhWord, _, _ );
          (
            WhWord = np:_:[Wh2],
            whq( Wh2, _, _ )
          );
          WhWord = that/comp
        ),
        feat( NP, F ),
```

```
        match( s:[ng:F:[which/det, NP], X], A, Ans,
               y/y, Trace ),
        !.

% query matches answer if it can match via verb nominalisation
% of the answer (assume for now that queries don't need this)
% when the answer contains a generic utility verb
match( s:[vg:Vo1#Pred:VG1 | Q], s:[vg:Vo2#Util:[v:V | T] | A], Ans,
       _/_, [[s:[vg:Vo2#Util:[v:V | T] | A],
              s:[vg:Vo2#Pred:[v:V2 | T] | A2], 109] | Trace] ) :-
        generic_verb( Util ),
        lemma( Noun, Pred ),
        check_and_remove( Noun, A, A2 ),
        append( V, [Noun/nn], V2 ),
        match( s:[vg:Vo1#Pred:VG1 | Q],
               s:[vg:Vo2#Pred:[v:V2 | T] | A2], Ans,
               lemma/y, Trace ),
        !.

% query matches answer if it can match via verb nominalisation
% of the answer (assume for now that queries don't need this)
% when the answer is a state structure
match( s:[vg:Vo1#Pred:VG1 | Q], s:A, Ans,
       _/_, [[s:A, s:[vg:act#Pred:[v:[Noun/nn]] | A2], 109] | Trace] ) :-
        lemma( Noun, Pred ),
        check_and_remove( Noun, A, A2 ),
        match( s:[vg:Vo1#Pred:VG1 | Q],
               s:[vg:act#Pred:[v:[Noun/nn]] | A2], Ans,
               lemma/y, Trace ),
        !.

% same for a state structure with reversed args
match( s:[vg:Vo1#Pred:VG1 | Q], s:[J,K], Ans,
       _/_, [[s:A, s:[vg:act#Pred:[v:[Noun/nn]] | A2], 109] | Trace] ) :-
        A = [K,J],
        lemma( Noun, Pred ),
        check_and_remove( Noun, A, A2 ),
        match( s:[vg:Vo1#Pred:VG1 | Q],
               s:[vg:act#Pred:[v:[Noun/nn]] | A2], Ans,
               lemma/y, Trace ),
        !.


% for existential case, try args the other way around, but only once!
match( s:[A, B], C, Ans,
       X/y, [[[A, B], [B, A], 110] | Trace] ) :-
        match( s:[B, A], C, Ans,
               X/n, Trace ),
        !.

% (or do the same on the answer side)
match( Q, s:[A, B], [AnsA, AnsB],
       X/y, [[[A, B], [B, A], 110] | Trace] ) :-
        match( Q, s:[B, A], [AnsB, AnsA],
               X/n, Trace ),
        !.


%%-----------------------------------------------------
%% match_list/5
%% match_list( +QList, +AList, -AnswerList,
%%             +ReverseArgsAtom, -DebugTrace )
%%
%% Description:
%%   Succeeds if all elements in QList and AList match (calls
```

```
%%   match/5 on each one).
%% Succeeds: 0*
%% Side Effects: may assert instances of current_answer/1
%%-----------------------------------------------------------

% base case
match_list( [], [], [],
            _/_, [] ).

% recurse, calling match/5 and appending debug trace info
% save some time by checking length and only matching head once
match_list( [Head1 | Tail1], [Head2 | Tail2], [AHead | ATail],
            YN, Trace ) :-
        length( Tail1, L ),
        length( Tail2, L ),
        match( Head1, Head2, AHead, YN, Trace1 ),
        !,
        match_list( Tail1, Tail2, ATail, YN, Trace2 ),
        append( Trace1, Trace2, Trace ).


%%----------------------------------------------------------
%% remove_dets/2
%% remove_dets( +RawWordList, ?CookedWordList )
%%
%% Description:
%%   Removes unnecessary determiners (before matching of
%%   NPs)
%% Succeeds: 1
%% Side Effects: none
%%-----------------------------------------------------------

% base case
remove_dets( [], [] ).

% remove determiners except "every"
remove_dets( [Det/det | Tail1], Tail2 ) :-
        \+ required_det( Det ),
        !,
        remove_dets( Tail1, Tail2 ).

% otherwise recurse
remove_dets( [Head | Tail1], [Head | Tail2] ) :-
        remove_dets( Tail1, Tail2 ).


%%----------------------------------------------------------
%% required_det/1
%% required_det( +Word )
%%
%% Description:
%%   Succeeds if Word is a determiner that must be matched
%%   (e.g. "every")
%% Succeeds: 0-1
%% Side Effects: none
%%-----------------------------------------------------------

% determiners that must be matched
required_det( every ).
required_det( all ).


%%----------------------------------------------------------
%% matchchk/2
%% match( +Q, +A )
```

```
%%
%% Description:
%%   Succeeds if Q and A match, but can only succeed once.
%%   Interface to match/5 without debugging info or answer.
%%   Q, A may be structures, phrases or words
%% Succeeds: 0*
%% Side Effects: may assert instances of current_answer/1,
%%   (which is a shame, I'd like it not to)
%%-------------------------------------------------------

% call match/5 and throw away debugging info
matchchk( Q, A ) :-
        match( Q, A, _, n/n, _ ),
        !.


check_and_remove( Noun, A, A2 ) :-
        member( NP, A ),
        np_head( NP, Noun/Type ),
        noun_type( Type ),
        phrase_select( Noun/Type, NP, np:_Num/_Sem:List ),
        np_semantics( np:List, Sem2 ),
        np_number( np:List, Num2 ),
        substitute( NP, A, np:Num2/Sem2:List, A2 ).

check_and_remove( Noun, A, A2 ) :-
        member( Arg, A ),
        tree_member( NP, Arg ),
        np_head( NP, Noun/Type ),
        noun_type( Type ),
        phrase_select( NP, Arg, Arg2 ),
        substitute( Arg, A, Arg2, A2 ).
```

B.3.3 Structural Extraction

struct.pl

```
%%-----------------------------------------------------
%% struct.pl
%%
%% Contains: predicates for building pred-arg
%%           or state structures
%%-----------------------------------------------------

:- dynamic co_replace/2.

%%-----------------------------------------------------
%% prep_struct/2
%% prep_struct( +RawSentList, ?CookedSentList )
%%
%% Description:
%%   CookedSentList has existential verbs removed and
%%   resulting existential relation information
%%   propagated throughout by co-indexing. VG-PPs are
%%   then attached (we've waited until now to prevent
%%   them being attached to existential verbs)
%% Succeeds: 1+
%% Side Effects: none
%%------------------------------------------------------

prep_struct( RawSent, CookedSent ) :-
        remove_be( RawSent, BeSent ),
        co_index( BeSent, CoSent ),
        attach_vg_pps( CoSent, CookedSent ).


%%-----------------------------------------------------
%% remove_be/2
%% remove_be( +RawSentList, ?CookedSentList )
%%
%% Description:
%%   CookedSentList is the result of converting "X is Y"
%%   relations into NG "ng:[X, Y]" relations
%% Succeeds: 1+
%% Side Effects: may assert instances of co_replace/2
%%------------------------------------------------------

% base case
remove_be( [], [] ).

% convert "X vg:be Y" -> "ng:[X, Y]" if possible
% replace in tail to prevent nasty circular problems with
% co_index/2 later
remove_be( [X, vg:Voice#Pred:_VG, Y | Tail1], [Z | Tail2] ) :-
        be_verb( Pred, Voice ),
        be_relation( X, Y, Z ),
        !,
        tail_co_index( X, Z, Tail1, Tail3 ),
        tail_co_index( Y, Z, Tail3, Tail4 ),
        remove_be( Tail4, Tail2 ).

% or just remove the "be" group
remove_be( [vg:Voice#Pred:_VG | Tail1], Tail2 ) :-
        be_verb( Pred, Voice ),
        !,
        remove_be( Tail1, Tail2 ).
```

```
% can do the same with any present continous verb followed by PP
% if sem is loc,tim "X is Ying in Z" -> "X is in Z"
% first, doing coindex conversion
remove_be( [X, vg:_Feat:[v:V | _T], pp:PP | Tail1], [Z | Tail2] ) :-
        member( be/_, V ),
        member( _/vbg, V ),
        (
          sem( pp:PP, loc );
          sem( pp:PP, tim )
        ),
        be_relation( X, pp:PP, Z ),
        tail_co_index( X, Z, Tail1, Tail3 ),
        remove_be( Tail3, Tail2 ).

% can do the same with any present continous verb followed by PP
% if sem is loc,tim "X is Ying in Z" -> "X is in Z"
% or just removing VG
remove_be( [vg:_Feat:[v:V | _T], pp:PP | Tail1], Tail2 ) :-
        member( be/_, V ),
        member( _/vbg, V ),
        (
          sem( pp:PP, loc );
          sem( pp:PP, tim )
        ),
        remove_be( [pp:PP | Tail1], Tail2 ).

% otherwise recurse doing nothing
remove_be( [Head | Tail1], [Head | Tail2] ) :-
        remove_be( Tail1, Tail2 ).


%%-----------------------------------------------------
%% be_relation/3
%% be_relation( +X, +Y, -Z )
%%
%% Description:
%%   Makes a new indexed entity Z from an existential
%%   relation between X and Y and asserts co_replace/2
%%   for later co-indexing
%% Succeeds: 1
%% Side Effects: may assert instances of co_replace/2,
%%   e_number/1, e/2
%%-----------------------------------------------------

% PP NP - attach and remember to replace the NP later
be_relation( pp:X, Y, e( N ) ) :-
        np( Y ),
        !,
        feat( Y, Feat ),
        make_e( N, ng:Feat:[Y, pp:X] ),
        assertz( co_replace( Y, e( N ) ) ).

% NP PP - as above
be_relation( X, pp:Y, e( N ) ) :-
        np( X ),
        !,
        feat( X, Feat ),
        make_e( N, ng:Feat:[X, pp:Y] ),
        assertz( co_replace( X, e( N ) ) ).

% PP PP - attach inside and remember to replace the NP
be_relation( pp:F:[Prep, X], pp:Y, pp:F:[Prep, e( N )] ) :-
        np( X ),
        !,
        feat( X, Feat ),
```

```
        make_e( N, ng:Feat:[X, pp:Y] ),
        assertz( co_replace( X, e( N ) ) ).

% NP NP - compound and remember to replace both
be_relation( X, Y, e( N ) ) :-
        np( X ),
        np( Y ),
        !,
        member( Z, [X, Y] ),
        feat( Z, Feat ),
        make_e( N, ng:Feat:[X, Y] ),
        assertz( co_replace( X, e( N ) ) ),
        assertz( co_replace( Y, e( N ) ) ).


%%-----------------------------------------------------
%% co_index/2
%% co_index( +RawSentList, ?CookedSentList )
%%
%% Description:
%%   CookedSentList is the result of replacing all
%%   entities in RawSentList that require co-indexing
%%   with their complex equivalents. Similar replacements
%%   are made within e/2 entities
%% Succeeds: 1
%% Side Effects: retracts co_replace/2 as used, changes
%%   e/2 definitions
%%-----------------------------------------------------

% for each co_replace/2 instance, subst in sentence
% and within any e() entities
co_index( Sent1, Sent3 ) :-
        co_replace( e( X ), e( Y ) ),
        retract( co_replace( e( X ), e( Y ) ) ),
        tree_subst( e( X ), Sent1, e( Y ), Sent2 ),
        subst_e_e( X, e( Y ), Y ),
        co_index( Sent2, Sent3 ),
        !.

% and stop when all done
co_index( Sent, Sent ) :-
        \+ co_replace( _, _ ).


%%-----------------------------------------------------
%% tail_co_index/4
%% tail_co_index( X, Y, +RawSentList, ?CookedSentList )
%%
%% Description:
%%   CookedSentList is the result of replacing X with
%%   Y in RawSentList, if this has been asserted as
%%   something that needs doing (co_replace/2).
%%   Does not affect e/2 entitiese or retract co_replace/2
%% Succeeds: 0-1
%% Side Effects: none
%%-----------------------------------------------------

% if this one needs replacing, subst
tail_co_index( X, Z, Tail1, Tail2 ) :-
        co_replace( X, Z ),
        tree_subst( X, Tail1, Z, Tail2 ),
        !.

% if not
tail_co_index( X, Z, Tail1, Tail1 ) :-
```

```
        \+ co_replace( X, Z ).


%%-----------------------------------------------------
%% q_struct_list/2
%% q_struct_list( +SentList, ?StructList )
%%
%% Description:
%%   StructList is a list of structures, one extracted from
%%   each member of the list of sentences SentList. For
%%   use processing queries, where a list of logical
%%   sub-queries is being considered
%% Succeeds: 0-1
%% Side Effects: none
%%------------------------------------------------------

% base case
q_struct_list( [], [] ).

% tidy, get structure & recurse
q_struct_list( [Sent | Tail1], [Struct | Tail2] ) :-
        tidy( Sent, TidySent ),
        make_struct( TidySent, Struct ),
        q_struct_list( Tail1, Tail2 ).


%%-----------------------------------------------------
%% a_struct/2
%% a_struct( +SentList, ?Struct )
%%
%% Description:
%%   StructList is one possible structure extracted from
%%   the sentence SentList. For use with answers
%% Succeeds: 1*
%% Side Effects: none
%%------------------------------------------------------

% tidy, get sublist, then get structure
a_struct( Sent, Struct, Query ) :-
        tidy( Sent, TidySent ),
        get_sub_list( TidySent, SubList, Query ),
        check_num_args( SubList, ArgList, Query ),
        make_struct( ArgList, Struct ).


%%-----------------------------------------------------
%% get_sub_list/3
%% get_sub_list( +Sent, ?SubSent, +QStruct )
%%
%% Description:
%%   SubSent is a possible sub-list of the sentence Sent
%%   that will be useful in matching the query structure
%%   QStruct (i.e. contains the right number of VGs).
%% Succeeds: 1*
%% Side Effects: none
%%------------------------------------------------------

% if query contains a predicate, answer must contain it
get_sub_list( Sentence, SubList, s:Query ) :-
        Query = [vg:_#Pred:_ | _Tail],
        append( _PreSubList, SubList, _PostSubList, Sentence ),
        member( vg:_#Pred:_, SubList ),
        count_verbs( Query, N ),
        count_verbs( SubList, N ),
        length( Query, Q ),
```

```
        length( SubList, S ),
        S >= Q.

% if query contains a nominaliseable predicate, answer can
% contain a utility predicate
get_sub_list( Sentence, SubList, s:Query ) :-
        Query = [vg:_#Pred:_ | _Tail],
        lemma( _, Pred ),
        append( _PreSubList, SubList, _PostSubList, Sentence ),
        member( vg:_#Util:_, SubList ),
        generic_verb( Util ),
        count_verbs( Query, N ),
        count_verbs( SubList, N ),
        length( Query, Q ),
        length( SubList, S ),
        S >= Q.

% if query contains a pos-predicate or nominaliseable predicate,
% answer can contain no verbs
get_sub_list( Sentence, SubList, s:[vg:_#Pred:_ | _Tail] ) :-
        (
            pos_verb( Pred );
            lemma( _, Pred )
        ),
        append( _PreSubList, SubList, _PostSubList, Sentence ),
        count_verbs( SubList, 0 ),
        length( SubList, S ),
        S >= 2.

% if no query predicate, just needs the same number of VGs (i.e. 0)
get_sub_list( Sentence, SubList, s:Query ) :-
        count_verbs( Query, 0 ),
        append( _PreSubList, SubList, _PostSubList, Sentence ),
        count_verbs( SubList, 0 ),
        length( Query, Q ),
        length( SubList, S ),
        S >= Q.

% unless we allow a pos-predicate in the answer
get_sub_list( Sentence, SubList, s:Query ) :-
        count_verbs( Query, 0 ),
        append( _PreSubList, SubList, _PostSubList, Sentence ),
        member( vg:_#Pred:_, SubList ),
        pos_verb( Pred ),
        count_verbs( SubList, 1 ),
        length( Query, Q ),
        length( SubList, S ),
        S >= Q.


%%----------------------------------------------------
%% check_num_args/3
%% check_num_args( +Sent1, ?Sent2, +QStruct )
%%
%% Description:
%%   Sent2 is an adjustment of Sent1 with NGs/PPs compounded
%%   where necessary to give the correct number of
%%   arguments to match the query structure QStruct
%% Succeeds: 1+
%% Side Effects: none
%%----------------------------------------------------

% with ditransitive verbs, passive form may need some massaging
check_num_args( [SNP, vg:pas#Sem:VG, ONP, ByPP], Sent, Query ) :-
        verb( Sem, ditrans ),
```

```
        check_num_args( [SNP, ONP, vg:pas#Sem:VG, ByPP], Sent, Query ).

% create pred-arg relation (list of same length as query with one VG)
% if query is a pred-arg structure
check_num_args( Sent1, Sent2, s:Query ) :-
        member( vg:_, Query ),
        append( Prefix, [vg:List | Suffix], Sent1 ),
        squish_to_length( 1, Prefix, Pre2 ),
        length( Pre2, P ),
        length( Query, Q ),
        squish_to_length( (Q - 1) - P, Suffix, Suf2 ),
        append( Pre2, [vg:List | Suf2], Sent2 ),
        length( Sent2, Q ).

% create pred-arg relation (list of (same length as query)+1 with one VG)
% if query is a state structure and the answer VG is a pos_verb
check_num_args( Sent1, Sent2, s:Query ) :-
        \+ member( vg:_, Query ),
        append( Prefix, [vg:Voice#Sem:List | Suffix], Sent1 ),
        pos_verb( Sem ),
        squish_to_length( 1, Prefix, Pre2 ),
        length( Pre2, P ),
        length( Query, Q ),
        squish_to_length( Q - P, Suffix, Suf2 ),
        append( Pre2, [vg:Voice#Sem:List | Suf2], Sent2 ),
        S is Q + 1,
        length( Sent2, S ).

% create state relation (list of length 2 with no VGs)
% if query is state structure
check_num_args( Sent1, Sent2, s:Query ) :-
        \+ member( vg:_, Sent1 ),
        \+ member( vg:_, Query ),
        append( Prefix, Suffix, Sent1 ),
        squish_to_length( 1, Prefix, Pre2 ),
        length( Pre2, 1 ),
        squish_to_length( 1, Suffix, Suf2 ),
        length( Suf2, 1 ),
        append( Pre2, Suf2, Sent2 ).

% create state relation (list of length 2 with no VGs)
% if query is a pred_arg structure with a pos-predicate or lemma-predicate
check_num_args( Sent1, Sent2, s:Query ) :-
        \+ member( vg:_, Sent1 ),
        member( vg:_#Sem:_, Query ),
        (
            pos_verb( Sem );
            lemma( _, Sem )
        ),
        append( Prefix, Suffix, Sent1 ),
        squish_to_length( 1, Prefix, Pre2 ),
        length( Pre2, 1 ),
        squish_to_length( 1, Suffix, Suf2 ),
        length( Suf2, 1 ),
        append( Pre2, Suf2, Sent2 ).


%%----------------------------------------------------
%% squish_to_length/3
%% squish_to_length( +L, +List1, ?List2 )
%%
%% Description:
%%   Sent2 is an adjustment of Sent1 with NGs/PPs compounded
%%   where necessary to give the correct number of
%%   arguments to match the query structure QStruct
```

```
%% Succeeds: 1+
%% Side Effects: none
%%-----------------------------------------------------

% if already short enough, do nothing
squish_to_length( L, In, In ) :-
        length( In, I ),
        L >= I,
        !.

% otherwise squish the last lot together as a NG, finding a
% suitable set of features
squish_to_length( L, In, Out ) :-
        L2 is L-1,
        append( Prefix, Suffix, In ),
        length( Prefix, L2 ),
        get_feats( Suffix, Feat ),
        append( Prefix, [ng:Feat:Suffix], Out ).


%%----------------------------------------------------
%% get_feats/2
%% get_feats( +List, ?Feats )
%%
%% Description:
%%   Feats are possible semantic features chosen from a
%%   member of List - NG if possible, PP if not
%% Succeeds: 1*
%% Side Effects: none
%%-----------------------------------------------------

% if we can find NGs with defined semantics, use any of them
get_feats( Suffix, Num/Sem ) :-
        member( _:Num/Sem:_, Suffix ),
        \+ Sem = [xxx].

% if only NGs with undefined semantics available, use that
% but no point doing this more than once
get_feats( Suffix, Num/[xxx] ) :-
        member( _:Num/[xxx]:_, Suffix ),
        \+ (
             member( _:_/Sem:_, Suffix ),
             \+ Sem = [xxx]
           ),
        !.

% if no NGs, use a PP class
get_feats( Suffix, s/Sem ) :-
        member( pp:Sem:_, Suffix ).


%%----------------------------------------------------
%% make_struct/2
%% make_struct( +SentList, ?Struct )
%%
%% Description:
%%   Checks that SentList contains suitable material
%%   and converts it into a pred-arg or state structure
%% Succeeds: 0-1
%% Side Effects: none
%%----------------------------------------------------

% pred-arg structure (S-V-O -> P-A-A)
make_struct( [SNP, vg:VG, ONP], s:[vg:VG, SNP, ONP] ).
```

```
% state structure
make_struct( [NP1, NP2], s:[NP1, NP2] ).


%%-----------------------------------------------------
%% tidy/2
%% tidy( +RawSentList, ?CookedSentList )
%%
%% Description:
%%   CookedSentList is the result of replacing e() indexed
%%   entities with their referents, removing punctuation
%%   (we don't need it any more) and checking that we
%%   haven't been over-zealous in compounding everything
%% Succeeds: 1
%% Side Effects: none
%%-----------------------------------------------------

tidy( MessySent, TidySent ) :-
        subst_e( MessySent, ESent ),
        strip_punct( ESent, PunctSent ),
        expand( PunctSent, TidySent ).


%%-----------------------------------------------------
%% expand/2
%% expand( +RawSentList, ?CookedSentList )
%%
%% Description:
%%   If RawSentList contains only one member (because we've
%%   overcompressed an existential sentence into one big
%%   NP), CookedSentList becomes the contents of that
%%   single phrase
%% Succeeds: 1
%% Side Effects: none
%%-----------------------------------------------------

% if we've over-compressed (just one phrase in sentence), split
expand( [_Type:_Feat:List], List ) :-
        !.

% otherwise do nothing
expand( A, A ).
```

anaphora.pl

```
%%-----------------------------------------------------
%% anaphora.pl
%%
%% Contains: predicates for resolving anaphoric references
%%-----------------------------------------------------

:- dynamic e/2, e_number/1, pp_referent/1.

%%-----------------------------------------------------
%% anaphora/2
%% anaphora( +RawSentList, ?SemSentList )
%%
%% Description:
%%   SemSentList is the result of resolving anaphoric pronouns.
%%   Just an interface to anaphora/5
%% Succeeds: 1*
%% Side Effects: none
%%-----------------------------------------------------

% just call anaphora/4
```

```
anaphora( A, B ) :-
        anaphora( A, B, [], _ ).


%%----------------------------------------------------
%% anaphora/4
%% anaphora( +RawSentList, ?SemSentList,
%%           +ResolvedInList, -ResolvedOutList )
%%
%% Description:
%%   SemSentList is the result of resolving (where possible)
%%   anaphoric pronouns in sentence RawSentList.
%%   ResolvedInList & ResolvedOutList maintain lists of resolved
%%   pronoun-referent pairs at each stage.
%% Succeeds: 1*
%% Side Effects: none
%%----------------------------------------------------

% base case
anaphora( [], [],
          Resolved, Resolved ).

% if a pronoun already resolved, keep the same referent
anaphora( [Head | Tail1], [Referent | Tail2],
          ResolvedIn, ResolvedOut ) :-
        pronoun( Head, Pronoun ),
        member( [Pronoun, Referent], ResolvedIn ),
        !,
        anaphora( Tail1, Tail2,
                  ResolvedIn, ResolvedOut ).

% if any entity already resolved, keep the same referent
anaphora( [Head | Tail1], [Referent | Tail2],
          ResolvedIn, ResolvedOut ) :-
        member( [Head, Referent], ResolvedIn ),
        !,
        anaphora( Tail1, Tail2,
                  ResolvedIn, ResolvedOut ).

% a resolvable pronoun can be resolved to a suitable NP referent
% and added to the resolved list
anaphora( [Head | Tail1], [e( N ) | Tail2],
          ResolvedIn, ResolvedOut ) :-
        pronoun( Head, Pronoun ),
        pro( Pronoun, ProSem, Num, y ),
        np( e( N ) ),
        sem( e( N ), Sem ),
        num( e( N ), Num ),
        member( Sem, ProSem ),
%       e( N, Type:Num/NPSem:List ),
%       np( Type:Num/NPSem:List ),
%       member( Sem, ProSem ),
%       member( Sem, NPSem ),
        anaphora( Tail1, Tail2,
                  [[Pronoun, e( N )] | ResolvedIn], ResolvedOut ).

% a definite NP or proper NP can be resolved
% to a suitable NP/NG referent and added to the resolved list
% must match so that referent contains same or more info, not less
% (i.e. "ken" -> "the lovely ken livingstone", but not vice versa)
% and referent musn't be a conjoined list
% (i.e. "ken" -/-> "ken and frank")
anaphora( [e( M ) | Tail1], [e( N ) | Tail2],
          ResolvedIn, ResolvedOut ) :-
        (
```

```
          definite_np( e( M ) );
          proper_np( e( M ) )
        ),
        e( M, NP1 ),
        e( N, Type:Feat:NP2 ),
        np_type( Type ),
        \+ N = M,
        \+ list_of_nps( NP2 ),
        \+ list_of_nps_no_punct( NP2 ),
        matchchk( NP1, Type:Feat:NP2 ),
        anaphora( Tail1, Tail2,
                  [[e( M ), e( N )] | ResolvedIn], ResolvedOut ).

% PP anaphors can be resolved to a suitable PP
anaphora( [Head | Tail1], [PP | Tail2],
          ResolvedIn, ResolvedOut ) :-
%       np_head( Head, H ),
        pp_anaphor( Head, Sem ),
        pp_referent( PP ),
        sem( PP, Sem ),
        anaphora( Tail1, Tail2,
                  [[Head, PP] | ResolvedIn], ResolvedOut ).

% a NP anaphor can be resolved to a plural NP referent
% which should be forced to be the next one in "many of the Xs" cases
anaphora( [Head | Tail1], [e( N ) | Tail2],
          ResolvedIn, ResolvedOut ) :-
        np_head( Head, H ),
        np_anaphor( H, p ),
        np( e( N ) ),
        num( e( N ), p ),
        anaphora( Tail1, Tail2,
                  [[Head, e( N )] | ResolvedIn], ResolvedOut ).

% as above, but in the case of "one", we need to create a new singular
% equivalent (which might be referred to by e.g. a later "it"!)
anaphora( [Head | Tail1], [e( M ) | Tail2],
          ResolvedIn, ResolvedOut ) :-
        np_head( Head, H ),
        np_anaphor( H, s ),
        np( e( N ) ),
        num( e( N ), p ),
        e( N, Type:p/Sem:List ),
        make_e( M, Type:s/Sem:List ),
        anaphora( Tail1, Tail2,
                  [[Head, e( M )] | ResolvedIn], ResolvedOut ).

% recurse into phrases (not bothering with VGs)
anaphora( [Type:Feat:List1 | Tail1], [Type:Feat:List2 | Tail2],
          ResolvedIn, ResolvedOut2 ) :-
        \+ Type = vg,
        \+ pronoun( Type:Feat:List1, _ ),
        !,
        anaphora( List1, List2,
                  ResolvedIn, ResolvedOut1 ),
        anaphora( Tail1, Tail2,
                  ResolvedOut1, ResolvedOut2 ).

% recurse into e()s that are NGs
% (don't bother with NPs - we''ve dealt with ones with PP heads)
% retracting first so things inside don't refer to this!
anaphora( [e( N ) | Tail1], [e( N ) | Tail2],
          ResolvedIn, ResolvedOut2 ) :-
        e( N, ng:Feat:List1 ),
        retract( e( N, ng:Feat:List1 ) ),
```

```
        !,
        anaphora( List1, List2,
                  ResolvedIn, ResolvedOut1 ),
        retractall( e( N, _ ) ),  % required to allow above line to backtrack
        assert( e( N, ng:Feat:List2 ) ),
        anaphora( Tail1, Tail2,
                  ResolvedOut1, ResolvedOut2 ).

% for anything else (including unresolved pronouns),
% recurse, not adding to seen list
anaphora( [Head | Tail1], [Head | Tail2],
          ResolvedIn, ResolvedOut ) :-
        anaphora( Tail1, Tail2,
                  ResolvedIn, ResolvedOut ).


%%-------------------------------------------------------
%% find_anaphora/1
%% find_anaphora( +SentList )
%%
%% Description:
%%   Finds possible NP referents for later pronoun resolution
%%   and asserts them in the Prolog database
%% Succeeds: 1
%% Side Effects: asserts referent/1 clauses
%%--------------------------------------------------------

% base case
find_anaphora( [] ).

find_anaphora( [Type:Feat:NP | Tail] ) :-
        np_type( Type ),
        np_head( Type:Feat:NP, _/pp );
        !,
        find_anaphora( NP ),
        find_anaphora( Tail ).

find_anaphora( [Type:Feat:NP | Tail] ) :-
        np_type( Type ),
        !,
        asserta( referent( Type:Feat:NP ) ),
        find_anaphora( NP ),
        find_anaphora( Tail ).

find_anaphora( [_Head | Tail] ) :-
        find_anaphora( Tail ).


%%-------------------------------------------------------
%% find_anaphora/2
%% find_anaphora( +SentList, ?RefList )
%%
%% Description:
%%   Finds possible NP/PP referents for later anaphor resolution
%%   and asserts them in the Prolog database, while replacing
%%   them with e/2 indexed entities
%% Succeeds: 1
%% Side Effects: causes e/2, e_number/1 clauses to be asserted
%%--------------------------------------------------------

% base case
find_anaphora( [], [] ).

% NPs that aren't themselves anaphoric:
% check inside NP, then index & recurse
```

```
find_anaphora( [Type:Feat:NP1 | Tail1], [e( N ) | Tail2] ) :-
        np_type( Type ),
        np_head( Type:Feat:NP1, H ),
        \+ H = _/pp,
        \+ np_anaphor( H, _ ),
        \+ pp_anaphor( H, _ ),
        !,
        find_anaphora( NP1, NP2 ),
        make_e( N, Type:Feat:NP2 ),
        find_anaphora( Tail1, Tail2 ).

% location or time PPs: check inside & assert without indexing
find_anaphora( [pp:Sem:PP1 | Tail1], [pp:Sem:PP2 | Tail2] ) :-
        (
          memberchk( loc, Sem );
          memberchk( tim, Sem )
        ),
        !,
        find_anaphora( PP1, PP2 ),
        asserta( pp_referent( pp:Sem:PP2 ) ),
        find_anaphora( Tail1, Tail2 ).

% any other phrase: check inside & recurse, but don't index this one
find_anaphora( [Type:Feat:P1 | Tail1], [Type:Feat:P2 | Tail2] ) :-
        !,
        find_anaphora( P1, P2 ),
        find_anaphora( Tail1, Tail2 ).

% otherwise recurse doing nothing
find_anaphora( [Head | Tail1], [Head | Tail2] ) :-
        find_anaphora( Tail1, Tail2 ).


%%-------------------------------------------------------
%% prep_anaphora/0
%%
%% Description:
%%   Retracts indexed entity predicates, and starts the
%%   entity number predicate e_number/1 at 1.
%% Succeeds: 1
%% Side Effects: retracts co_replace/2, e/2, e_number/1
%%   clauses, asserts e_number/1
%%-------------------------------------------------------

prep_anaphora :-
        retractall( co_replace( _, _ ) ),
        retractall( e( _, _ ) ),
        retractall( e_number( _ ) ),
        retractall( pp_referent( _ ) ),
        assert( e_number( 1 ) ).


%%-------------------------------------------------------
%% make_e/2
%% make_e( -Number, +NP )
%%
%% Description:
%%   Finds the next available number for an indexed entity
%%   and asserts it (referring to NP). Increments e_number/1
%%   counter
%% Succeeds: 1
%% Side Effects: asserts e/2, retracts & asserts e_number/1
%%-------------------------------------------------------

% find number, assert e/2, increment number, reset e_number/1
```

```
make_e( N, NP ) :-
        e_number( N ),
        assert( e( N, NP ) ),
        M is N + 1,
        retract( e_number( N ) ),
        assert( e_number( M ) ).


%%-----------------------------------------------------
%% subst_e/2
%% subst_e( +ESentList, ?NPSentList )
%%
%% Description:
%%   NPSentList is the result of substituting the indexed
%%   NPs back in for the e/2 entities. Interface to
%%   subst_e/3
%% Succeeds: 1
%% Side Effects: none
%%------------------------------------------------------

% call subst_e/3 beginning with the last to be asserted
% as later ones may refer to earlier ones
subst_e( Sent1, Sent2 ) :-
        e_number( N ),
        M is N - 1,
        subst_e( Sent1, Sent2, M ).


%%-----------------------------------------------------
%% subst_e/3
%% subst_e( +ESentList, ?NPSentList, +CounterNumber )
%%
%% Description:
%%   NPSentList is the result of substituting the indexed
%%   NPs back in for the e/2 entities.
%% Succeeds: 1
%% Side Effects: none
%%------------------------------------------------------

% finished
subst_e( Sent, Sent, 0 ).

% substitute all occurrences of this e() and recurse for next
subst_e( Sent1, Sent3, N ) :-
        e( N, NP ),
        tree_subst( e( N ), Sent1, NP, Sent2 ),
        subst_e_e( N, NP, N ),
        M is N - 1,
        subst_e( Sent2, Sent3, M ).


%%-----------------------------------------------------
%% subst_e_e/3
%% subst_e_e( +ENumber, ?NP, +CounterNumber )
%%
%% Description:
%%   Substitutes occurrences of e(ENumber) for NP in
%%   other e()s with numbers less than or equal to CounterNumber
%% Succeeds: 1
%% Side Effects: may change some definitions of e/2
%%------------------------------------------------------

subst_e_e( _N, _NP, 0 ).

% make sure we don't create recursive references
```

```
subst_e_e( N, e( X ), X ) :-
        Y is X - 1,
        !,
        subst_e_e( N, e( X ), Y ).

subst_e_e( N, NP, X ) :-
        e( X, NP1 ),
        tree_subst( e( N ), NP1, NP, NP2 ),
        retract( e( X, NP1 ) ),
        assert( e( X, NP2 ) ),
        Y is X - 1,
        !,
        subst_e_e( N, NP, Y ).


%%-----------------------------------------------------
%% subst_e_e_up/3
%% subst_e_e_up( +ENumber, ?NP, +CounterNumber )
%%
%% Description:
%%   Substitutes occurrences of e(ENumber) for NP in
%%   other e()s with numbers less than or equal to CounterNumber
%% Succeeds: 1
%% Side Effects: may change some definitions of e/2
%%-----------------------------------------------------

subst_e_e_up( _N, _NP, X ) :-
        e_number( X ).

% make sure we don't create recursive references
subst_e_e_up( N, e( X ), X ) :-
        Y is X + 1,
        !,
        subst_e_e_up( N, e( X ), Y ).

subst_e_e_up( N, NP, X ) :-
        e( X, NP1 ),
        tree_subst( e( N ), NP1, NP, NP2 ),
        retract( e( X, NP1 ) ),
        assert( e( X, NP2 ) ),
        Y is X + 1,
        !,
        subst_e_e_up( N, NP, Y ).



%%-----------------------------------------------------
%% pronoun/2
%% pronoun( +WordOrNP, ?Pronoun )
%%
%% Description:
%%   Succeeds if WordOrNP is a pronoun (either on its own
%%   or in its own NP, and instantiates Pronoun to the
%%   pronoun word
%% Succeeds: 0-1
%% Side Effects: none
%%-----------------------------------------------------

% X/pp is a pronoun
pronoun( Pronoun/pp, Pronoun ).

% np:_:[X/pp] is too
pronoun( np:_Feat:[Pronoun/pp], Pronoun ).
```

```
%%-----------------------------------------------------
%% proper_np/1
%% proper_np( +NP )
%%
%% Description:
%%   Succeeds if NP is a proper NP - must contain no
%%   determiners, and head must be of type 'np'
%% Succeeds: 0-1
%% Side Effects: none
%%------------------------------------------------------

% for simple NPs, head will be tagged as 'np'
proper_np( Type:Feat:List ) :-
        np_type( Type ),
        np_head( Type:Feat:List, _Word/np ),
        \+ memberchk( List, _Det/det ).

% for compound NPs, head must be a proper NP
proper_np( Type:Feat:List ) :-
        np_type( Type ),
        np_head( Type:Feat:List, NP ),
        \+ memberchk( List, _Det/det ),
        proper_np( NP ).

% for indexed entity, look up & recurse
proper_np( e( N ) ) :-
        e( N, NP ),
        proper_np( NP ).


%%-----------------------------------------------------
%% definite_np/1
%% definite_np( +NP )
%%
%% Description:
%%   Succeeds if NP is a definite NP - must contain a
%%   definite article and cannot be a NG (as this might
%%   contain PPs).
%% Succeeds: 0-1
%% Side Effects: none
%%------------------------------------------------------

% for simple NPs, must contain (begin with?) definite determiner
definite_np( np:_Feat:List ) :-
%       np_type( Type ),
        memberchk( Det/det, List ),
        definite_det( Det ).

% for compound NPs, head must be a definite NP
%definite_np( NG ) :-
%       np( NG ),
%       np_head( NG, NP ),
%       definite_np( NP ).

% for indexed entity, look up & recurse
definite_np( e( N ) ) :-
        e( N, NP ),
        definite_np( NP ).


%%-----------------------------------------------------
%% indefinite_np/1
%% indefinite_np( +NP )
%%
%% Description:
```

```
%%   Succeeds if NP is an indefinite NP - must contain an
%%   indefinite article, or be plural
%% Succeeds: 0-1
%% Side Effects: none
%%----------------------------------------------------

% if plural, just check it's not definite
indefinite_np( NP ) :-
        np( NP ),
        num( NP, p ),
        \+ definite_np( NP ).

% if singular, check not definite and contains article
indefinite_np( Type:Feat:List ) :-
        np_type( Type ),
        num( Type:Feat:List, s ),
        memberchk( List, Det/det ),
        \+ definite_det( Det ),
        \+ definite_np( Type:Feat:List ).

% for compound NPs, head must be an indefinite NP
indefinite_np( NG ) :-
        np( NG ),
        np_head( NG, NP ),
        indefinite_np( NP ).

% for indexed entity, look up & recurse
indefinite_np( e( N ) ) :-
        e( N, NP ),
        indefinite_np( NP ).
```

simplify.pl

```
%%---------------------------------------------------
%% simplify.pl
%%
%% Contains: predicates for simplifying complex sentences
%%           into lists of simple logical sub-sentences
%%           and for later extracting them
%%---------------------------------------------------

%%---------------------------------------------------
%% sub_sent/2
%% sub_sent( +RawSentList, ?SubSentList )
%%
%% Description:
%%   SubSentList is a possible sub-sentence of RawSentList.
%% Succeeds: 1*
%% Side Effects: none
%%----------------------------------------------------

sub_sent( Sentence, SubSent ) :-
        append( Prefix, SubSent, Suffix, Sentence ),
        \+ memberchk( &, SubSent ),
        sub_prefix( Prefix ),
        sub_suffix( Suffix ).


%%---------------------------------------------------
%% q_sub_sent/2
%% q_sub_sent( +RawSentList, ?SubSentList )
%%
%% Description:
%%   For use with queries only: SubSentList is a possible
%%   sub-sentence of RawSentList. Differs to sub_sent/2
```

```
%%   in that NP conjunctions are separated out, so that
%%   a separate query will be formed for each member
%% Succeeds: 1
%% Side Effects: none
%%-------------------------------------------------------

q_sub_sent( Sentence, Result ) :-
        n_conjunctions( Sentence, Sent1 ),
        sub_sent( Sent1, Result ).


%%------------------------------------------------------
%% n_conjunctions/2
%% n_conjunctions( +RawSentList, ?CookedSentList )
%%
%% Description:
%%   Substitutes any NP conjunctions with any
%%   single one of the individual NPs, for use in
%%   making individual logical queries. Must be used
%%   while we still have e() entities present
%% Succeeds: 1*
%% Side Effects: none
%%-------------------------------------------------------

% substitute any conj-list e() with one of its members
n_conjunctions( Sentence, Result ) :-
        e( N, ng:_Feat:List ),
        list_of_nps_no_punct( List ),
        tree_member( e( N ), Sentence ),
        member( NP, List ),
        np( NP ),
        \+ conj_word( NP ),
        tree_subst( e( N ), Sentence, NP, Result ).

% or do nothing if no such e() exists
% & is present in the sentence
n_conjunctions( Sentence, Sentence ) :-
        \+ (
             e( N, ng:_Feat:List ),
             list_of_nps_no_punct( List ),
             tree_member( e( N ), Sentence )
           ).


%%------------------------------------------------------
%% sub_prefix/1
%% sub_prefix( +SentList )
%%
%% Description:
%%   Succeeds if SentList is empty or ends with the
%%   sentence separator '&'
%% Succeeds: 0-1
%% Side Effects: none
%%-------------------------------------------------------

sub_prefix( P ) :-
        reverse( P, S ),
        sub_suffix( S ).


%%------------------------------------------------------
%% sub_suffix/1
%% sub_suffix( +SentList )
%%
%% Description:
```

```
%%   Succeeds if SentList is empty or begins with the
%%   sentence separator '&'
%% Succeeds: 0-1
%% Side Effects: none
%%-----------------------------------------------------

% empty list is OK
sub_suffix( [] ).

% anything beginning with & is OK
sub_suffix( [& | _S] ).


%%-----------------------------------------------------
%% simplify/2
%% simplify( +RawSentList, ?SubSentList )
%%
%% Description:
%%   Turns a syntactically complex sentence into a list
%%   (conjunction) of logically equivalent simple sentences,
%%   separated by the character '&'. Can't deal with
%%   disjunctions - treats "or" just like "and"
%% Succeeds: 1
%% Side Effects: none
%%-----------------------------------------------------

simplify( RawSent, CookedSent ) :-
        sub_clause( RawSent, SubSent ),
        vp_conjunctions( SubSent, VPSent ),
        s_conjunctions( VPSent, SCSent ),
        s_punct( SCSent, CookedSent ).


%%-----------------------------------------------------
%% sub_clause/2
%% sub_clause( +RawSentList, ?SubSentList )
%%
%% Description:
%%   Turns a sentence containing relative & subordinate
%%   clauses into a list (conjunction) of logically
%%   equivalent simple sentences separated by '&'
%% Succeeds: 1
%% Side Effects: none
%%-----------------------------------------------------

% inter relative clauses (subj)
% X, who likes ..., hates ... -> X likes ... & X hates ...
sub_clause( Sent1, Sent3 ) :-
        append( Pre, [Punct, SubWord, vg:VG1 | Tail],
                [Punct, vg:VG2 | Suffix], Sent1 ),
        sub_punct( Punct ),
        get_last( Pre, X, _ ),
        match_rel( SubWord, X ),
        !,
        append( Pre, [vg:VG1 | Tail], [&, X, vg:VG2 | Suffix],
                Sent2 ),
        sub_clause( Sent2, Sent3 ).

% post relative clauses (subj)
% ... X, who likes ... -> ... X & X likes ...
sub_clause( Sent1, Sent3 ) :-
        append( Pre, [Punct, SubWord, vg:VG | Tail], Sent1 ),
        sub_punct( Punct ),
        get_last( Pre, X, _ ),
        match_rel( SubWord, X ),
```

```
        !,
        append( Pre, [&, X, vg:VG | Tail], Sent2 ),
        sub_clause( Sent2, Sent3 ).

% as above, but don't need the comma
% ... X who likes ... -> ... X & X likes ...
sub_clause( Sent1, Sent3 ) :-
        append( Pre, [SubWord, vg:VG | Tail], Sent1 ),
        get_last( Pre, X, _ ),
        match_rel( SubWord, X ),
        !,
        append( Pre, [&, X, vg:VG | Tail], Sent2 ),
        sub_clause( Sent2, Sent3 ).

% inter relative clauses (obj)
% X, who Y likes ..., hates ... -> Y likes X ... & X hates ...
sub_clause( Sent1, Sent3 ) :-
        append( Pre, [Punct, SubWord, Y, vg:VG1 | Tail],
                [Punct, vg:VG2 | Suffix], Sent1 ),
        sub_punct( Punct ),
        np( Y ),
        get_last( Pre, X, Pre2 ),
        match_rel( SubWord, X ),
        !,
        append( Pre2, [Y, vg:VG1, X | Tail], [&, X, vg:VG2 | Suffix], Sent2 ),
        sub_clause( Sent2, Sent3 ).

% post relative clauses (obj)
% ... X, who Y likes ... -> ... X & Y likes X ...
sub_clause( Sent1, Sent3 ) :-
        append( Pre, [Punct, SubWord, Y, vg:VG | Tail], Sent1 ),
        sub_punct( Punct ),
        np( Y ),
        get_last( Pre, X, _ ),
        match_rel( SubWord, X ),
        !,
        append( Pre, [&, Y, vg:VG, X | Tail], Sent2 ),
        sub_clause( Sent2, Sent3 ).

% as above but don't need the comma
% ... X who Y likes ... -> ... X & Y likes X ...
sub_clause( Sent1, Sent3 ) :-
        append( Pre, [SubWord, Y, vg:VG | Tail], Sent1 ),
        np( Y ),
        get_last( Pre, X, _ ),
        match_rel( SubWord, X ),
        !,
        append( Pre, [&, Y, vg:VG, X | Tail], Sent2 ),
        sub_clause( Sent2, Sent3 ).

% subordinate clauses
% X, though he likes Y, hates Z -> X though he likes Y & X hates Z
% X, like the people he likes, hates Z ->
%                       X like the people he likes & X hates Z
sub_clause( Sent1, Sent3 ) :-
        append( Pre, [Punct, SubWord | Tail],
               [Punct | Suffix], Sent1 ),
        sub_punct( Punct ),
        sub_word( SubWord ),
        memberchk( vg:_VG1, Tail ),
        last( Pre, X ),
        !,
        append( Pre, [SubWord | Tail], [&, X | Suffix],
                Sent2 ),
        sub_clause( Sent2, Sent3 ).
```

```
% subordinate clauses
% Though he likes Y, X hates Z -> Though he likes Y & X hates Z
sub_clause( Sent1, Sent3 ) :-
        append( [SubWord | Tail], [Punct | Suffix], Sent1 ),
        sub_punct( Punct ),
        sub_word( SubWord ),
        memberchk( vg:_VG1, Tail ),
        !,
        append( [SubWord | Tail], [& | Suffix], Sent2 ),
        sub_clause( Sent2, Sent3 ).

% if none of the above apply, stop
sub_clause( Sent, Sent ).


%%-----------------------------------------------------
%% get_last/3
%% get_last( +SentList, ?Last, ?Rest )
%%
%% Description:
%%   Last is either the last member of SentList, or the
%%   NP inside it (if is is a PP). Rest is what's left of
%%   SentList after Last has been removed
%% Succeeds: 1+
%% Side Effects: none
%%-----------------------------------------------------

% if the last element is a PP, we might just want its NP
% "In Wales, which is a lovely country, is a mountain"
get_last( Pre, NP, Pre2 ) :-
        reverse( Pre, [pp:_F:[_Prep, NP] | Erp] ),
        reverse( Erp, Pre2 ).

% or just return last element - even if PP
% "In Wales, which is where I live, is a mountain"
% match_rel/2 should only allow the right choice to be made
get_last( Pre, X, Pre2 ) :-
        reverse( Pre, [X | Erp] ),
        reverse( Erp, Pre2 ).


%%-----------------------------------------------------
%% match_rel/2
%% match_rel( ?RelWord, +Phrase )
%%
%% Description:
%%   Succeeds if RelWord is a relative pronoun which suits
%%   Phrase in both expected phrase type and semantic class
%% Succeeds: 1*
%% Side Effects: none
%%-----------------------------------------------------

match_rel( RelWord, X ) :-
        rel( RelWord, SemList, TypeList ),
        type( X, Type ),
        member( Type, TypeList ),
        sem( X, Sem ),
        member( Sem, SemList ).


%%-----------------------------------------------------
%% rel/3
%% rel( ?RelWord, ?SemList, ?TypeList )
%%
```

```
%% Description:
%%    SemList and TypeList contain the features that are
%%    suitable for RelWord when introducing a relative
%%    clause
%% Succeeds: 0-1
%% Side Effects: none
%%-----------------------------------------------------

% any wh-word with its related features
rel( Word, Sem, Type ) :-
        whq( Word, Sem, Type ).

% "that" with any class, any NP
rel( that/comp, [loc, tim, per/_, obj, abs, num, org], [np, ng] ).


%%-----------------------------------------------------
%% sub_word/1
%% sub_word( ?SubWord )
%%
%% Description:
%%    SubWord is a word which can introduce a subordinate
%%    clause
%% Succeeds: 1*
%% Side Effects: none
%%-----------------------------------------------------

% e.g. Jack, though I dislike him, is worthy
sub_word( _/subconj ).

% e.g. Jack, when I insult him, is polite
sub_word( _/wrb ).

% e.g. Jack, among those I dislike, is the worst
sub_word( _/prep ).

% (as prep, but all preps are in PPs by this point)
sub_word( pp:_ ).


%%-----------------------------------------------------
%% sub_punct/1
%% sub_punct( ?SubWord )
%%
%% Description:
%%    SubPunct is a punctuation mark which can introduce
%%    and end a relative or subordinate clause
%% Succeeds: 1*
%% Side Effects: none
%%-----------------------------------------------------

% comma
sub_punct( ','/',' ).

% dash
sub_punct( '-'/':' ).


%%-----------------------------------------------------
%% vp_conjunctions/2
%% vp_conjunctions( +RawSentList, ?SubSentList )
%%
%% Description:
%%    Turns a sentence containing a list of conjoined VPs
%%    into the equivalent list of conjoined full sentences
```

```
%%   e.g. "X did Y and did Z" -> "X did Y and X did Z"
%% Succeeds: 1
%% Side Effects: none
%%--------------------------------------------------------

% if sentence contains conjoined VPs, find subject (last
% previous NP) and insert in appropriate places
vp_conjunctions( Sent1, Sent2 ) :-
        append( Subj, [vg:VG | List1], Sent1 ),
        list_of_vps( [vg:VG | List1] ),
        end_member( NP, Subj ),
        np( NP ),
        !,
        insert_subj( NP, List1, List2 ),
        append( Subj, [vg:VG | List2], Sent2 ).

% otherwise do nothing
vp_conjunctions( Sent, Sent ).


%%-------------------------------------------------------
%% s_conjunctions/2
%% s_conjunctions( +RawSentList, ?ConjSentList )
%%
%% Description:
%%   Replaces conjunctions separating sub-sentences (i.e.
%%   those that separate portions of sentences containing
%%   VGs) with sentence boundary marker '&'
%% Succeeds: 1
%% Side Effects: none
%%--------------------------------------------------------

% if sentence contains a conjunction prefixed by a whole
% number of sub-sentences, and there's a verb after it,
% turn it into '&' and check again
% I'd like to make V = S, but we're not picking up sub-sents
% separated by adverbs like "so"
s_conjunctions( Sentence, Sent2 ) :-
        append( Prefix, [_Conj/Type | Suffix], Sentence ),
        memberchk( Type, [cc, subconj] ),
        count_sents( Prefix, S ),
        count_syn_vgs( Prefix, V ),
        V >= S,
        memberchk( vg:_, Suffix ),
        % maybe this should be count_syn_vgs( Suffix, >0 ) ??
        !,
        append( Prefix, [& | Suffix], Sent1 ),
        s_conjunctions( Sent1, Sent2 ).

% otherwise stop
s_conjunctions( Sentence, Sentence ).


%%-------------------------------------------------------
%% s_punct/2
%% s_punct( +RawSentList, ?ConjSentList )
%%
%% Description:
%%   Replaces sentence-boundary (. ? !) and sub-sentence
%%   (; : - ...) punctuation marks at the top level with
%%   sentence boundary marker '&'
%% Succeeds: 1
%% Side Effects: none
%%--------------------------------------------------------
```

```
% base case
s_punct( [], [] ).

% replace _/. and recurse
s_punct( [_/'.' | T1], [& | T2] ) :-
        !,
        s_punct( T1, T2 ).

% replace _/: and recurse
s_punct( [_/':' | T1], [& | T2] ) :-
        !,
        s_punct( T1, T2 ).

% otherwise recurse doing nothing
s_punct( [H | T1], [H | T2] ) :-
        s_punct( T1, T2 ).


%%------------------------------------------------------
%% list_of_vps/1
%% list_of_vps( +SentList )
%%
%% Description:
%%   Succeeds if SentList is a list of conjoined VPs (of
%%   the form "... did X, did Y(,) and did Z"
%% Succeeds: 0-1
%% Side Effects: none
%%------------------------------------------------------

% "did X ... and did Y"
list_of_vps( List ) :-
        append( [vg:_ | Tail1], [_Conj/cc, vg:_ | Tail2], List ),
        \+ member( vg:_, Tail1 ),
        \+ member( vg:_, Tail2 ).

% "did X ... , and did Y"
list_of_vps( List ) :-
        append( [vg:_ | Tail1], [','/',', _Conj/cc, vg:_ | Tail2], List ),
        \+ member( vg:_, Tail1 ),
        \+ member( vg:_, Tail2 ).

% "did X ... , (did Y ... , ...)"
list_of_vps( List ) :-
        append( [vg:_ | Tail1], [','/',' | Tail2], List ),
        \+ member( vg:_, Tail1 ),
        list_of_vps( Tail2 ).


%%------------------------------------------------------
%% insert_subj/3
%% insert_subj( +SubjNP, +RawSentList, ?SubjSentList )
%%
%% Description:
%%   Inserts SubjNP before each VG in RawSentList, together
%%   with the sentence boundary marker '&'
%% Succeeds: 1
%% Side Effects: none
%%------------------------------------------------------

% base case
insert_subj( _Subj, [], [] ).

% insert Subj before a VG and recurse
insert_subj( Subj, [vg:VG | Tail1], [&, Subj, vg:VG | Tail2] ) :-
        !,
```

```
        insert_subj( Subj, Tail1, Tail2 ).

% otherwise recurse doing nothing
insert_subj( Subj, [Head | Tail1], [Head | Tail2] ) :-
        insert_subj( Subj, Tail1, Tail2 ).


%%--------------------------------------------------------
%% count_sents/2
%% count_sents( +SentList, ?N )
%%
%% Description:
%%   N is the number of sub-sentences (separateed by '&')
%%   in the complex sentence SentList
%% Succeeds: 1
%% Side Effects: none
%%---------------------------------------------------------

% base case
count_sents( [], 1 ).

% add 1 for each '&' and recurse
count_sents( [& | Tail], N ) :-
        !,
        count_sents( Tail, M ),
        N is M + 1.

% otherwise recurse doing nothing
count_sents( [_Head | Tail], N ) :-
        count_sents( Tail, N ).
```

B.3.4 Shallow Text Processing

parse.pl

```
%%---------------------------------------------------------
%% parse.pl
%%
%% Contains: top-level predicates for building parse trees
%%---------------------------------------------------------

%%---------------------------------------------------------
%% parse/2
%% parse( +RawSentList, ?CookedSentList )
%%
%% Description:
%%   CookedSentList is the shallow parse tree produced
%%   from RawSentList, except that VG-PPs have not yet
%%   been attached (as we want to wait until existential
%%   verbs have been removed so that we don't lose them).
%%   NGs have been replaced by e() index entities and
%%   anaphora has been resolved where possible.
%% Succeeds: 1*
%% Side Effects: none
%%----------------------------------------------------------

parse( RawSent, CookedSent ) :-
        syn_process( RawSent, SynSent ),
        sem_process( SynSent, SemSent ),
        attach_np_pps( SemSent, NPSent ),
        prep_anaphora,
        find_anaphora( NPSent, ESent ),
        anaphora( ESent, CookedSent ).


%%---------------------------------------------------------
%% syn_process/2
%% syn_process( +RawSentList, ?SynSentList )
%%
%% Description:
%%   SynSentList is the result of forming NPs, PPs and VGs
%%   within RawSentList
%% Succeeds: 1*
%% Side Effects: none
%%----------------------------------------------------------

syn_process( RawSent, SynSent ) :-
        make_vgs( RawSent, VGSent ),
        attach_advs( VGSent, AdvSent ),
        move_advs( AdvSent, MoveSent ),
        ( process_type( query ) ->
            combine_q_vgs( MoveSent, CombSent )
        ;
            CombSent = MoveSent
        ),
        np_compounds( CombSent, CompSent ),
        np_conjunctions( CompSent, NPSent ),
        make_ngs( NPSent, NGSent ),
        make_pps( NGSent, SynSent ).


%%---------------------------------------------------------
%% sem_process/2
%% sem_process( +RawSentList, ?SemSentList )
%%
```

```
%% Description:
%%   SemSentList is the result of applying semantic
%%   feature info to NP/PP/VGs. NPs get number and
%%   semantic class list, PPs get semantic class list,
%%   VGs get voice and predicate name
%% Succeeds: 1
%% Side Effects: none
%%-----------------------------------------------------

sem_process( RawSent, SemSent ) :-
        np_num_sem( RawSent, NSent ),
        vg_semantics( NSent, SemSent ).
```

sem.pl

```
%%-----------------------------------------------------
%% sem.pl
%%
%% Contains: predicates for attaching semantic info to
%%           parse tree structures
%%-----------------------------------------------------

%%-----------------------------------------------------
%% np_num_sem/2
%% np_num_sem( +RawSentList, ?SemSentList )
%%
%% Description:
%%   SemSentList is the result of applying NP/PP semantic
%%   class and number info to sentence RawSentList
%%   np:[...] -> np:Num/Sem:[...]
%%   pp:[...] -> pp:Sem:[...]
%% Succeeds: 1*
%% Side Effects: none
%%-----------------------------------------------------

% base case
np_num_sem( [], [] ).

% recurse, assigning NP the semantic class
% and sing/plural number of its contents
np_num_sem( [np:List | Tail1], [np:Num/Sem:List | Tail2] ) :-
        np_number( np:List, Num ),
        np_semantics( np:List, Sem ),
        !,
        np_num_sem( Tail1, Tail2 ).

% recurse, recursing within NG and taking new semantic class and
% number from last subNP, but overriding to plural if it contains
% conjoined NPs
np_num_sem( [ng:List1 | Tail1], [ng:Num/Sem:List2 | Tail2] ) :-
        !,
        np_num_sem( List1, List2 ),
        np_number( ng:List2, Num ),
        np_semantics( ng:List2, Sem ),
        np_num_sem( Tail1, Tail2 ).

% recurse, recursing within PP and finding new semantic class
np_num_sem( [pp:List1 | Tail1], [pp:Sem:List2 | Tail2] ) :-
        !,
        np_num_sem( List1, List2 ),
        pp_semantics( List2, Sem ),
%       List2 = [_Prep/prep, _Type:_Num/Sem:_List],
        np_num_sem( Tail1, Tail2 ).

% otherwise, recurse doing nothing
```

```
np_num_sem( [Head | Tail1], [Head | Tail2] ) :-
        np_num_sem( Tail1, Tail2 ).


%%--------------------------------------------------
%% np_semantics/2
%% np_semantics( +NP, ?SemList )
%%
%% Description:
%%   SemList is the list of possible semantic classes of
%%   the head of the noun phrase NP.
%% Succeeds: 1*
%% Side Effects: none
%%--------------------------------------------------

% NG: take class from the head NP
np_semantics( ng:List, Sem ) :-
        np_head( ng:List, _Type:_Num/Sem:_NPList ),
        !.

% NP: look up class for the head noun
np_semantics( np:List, Sem ) :-
        np_head( np:List, LastWord/Type ),
        noun_semantics( LastWord/Type, Sem ).


%%--------------------------------------------------
%% np_number/2
%% np_number( +NP, ?NumberAtom )
%%
%% Description:
%%   NumberAtom is either 's' or 'p'. A NP is singular unless
%%   it is a conjoined list or its head is plural.
%% Succeeds: 1
%% Side Effects: none
%%--------------------------------------------------

% NG: if it's a list of conjoined NPs, it's plural
np_number( ng:List, p ) :-
        list_of_nps_no_punct( List ),
        !.

% NG: otherwise take number from the head
np_number( ng:List, Num ) :-
        np_head( ng:List, _Type:Num/_Sem:_NPList ),
        !.

% NP: take number from the head
np_number( np:List, Num ) :-
        np_head( np:List, _Head/Type ),
        ( Type = nns ->
            Num = p
        ;
            Num = s
        ).


%%--------------------------------------------------
%% np_head/2
%% np_head( +NP, ?HeadNPOrNoun )
%%
%% Description:
%%   HeadNPOrNoun is the head of NP. For a complex NG,
%%   it will be the last NP. For a simple NP, it will be
%%   the last noun, or the last word if no nouns can be found.
```

```
%% Succeeds: 1
%% Side Effects: none
%%-------------------------------------------------------

% make it work after semantics attached
np_head( Type:_Feat:List, Head ) :-
        np_head( Type:List, Head ).

% head of a NG is the last NP
np_head( ng:List, Type:Head ) :-
        end_member( Type:Head, List ),
        np_type( Type ).

% head of a NP is the last noun
np_head( np:List, Head/Type ) :-
        end_member( Head/Type, List ),
        noun_type( Type ),
        !.

% or if no nouns, just the last word (e.g. adj)
np_head( np:List, Head ) :-
        last( List, Head ).

% look up indexed entity & recurse
np_head( e( N ), Head ) :-
        e( N, NP ),
        np_head( NP, Head ).


%%-------------------------------------------------------
%% noun_semantics/2
%% noun_semantics( +Noun, ?ClassList )
%%
%% Description:
%%   ClassList is a list of possible semantic classes for
%%   the tagged noun Noun
%% Succeeds: 1
%% Side Effects: none
%%-------------------------------------------------------

% possessives: pos
noun_semantics( _Pos/pos, [pos] ).

% numbers: allow tim, num
noun_semantics( _AnyNumber/cd, [tim, num] ).

% pronouns: look up class list
noun_semantics( Word/pp, Sem ) :-
        pro( Word, Sem, _Number, _YN ),
        !.

% noun: look up class list
noun_semantics( Word/Type, Sem ) :-
        noun_type( Type ),
        noun( Word, Sem ),
        !.

% if not listed in lexicon, assign 'xxx'
% this is important as we often get verb participles mistagged NN
noun_semantics( Word/Type, [xxx] ) :-
        noun_type( Type ),
        !,
        \+ noun( Word, _Sem ).

noun_semantics( _Word/Type, [xxx] ) :-
```

```
        \+ noun_type( Type ).


%%-----------------------------------------------------
%% pp_semantics/2
%% pp_semantics( +PP, ?ClassList )
%%
%% Description:
%%   ClassList is a list of possible semantic classes for
%%   PP, or just [xxx] if none defined
%% Succeeds: 1
%% Side Effects: none
%%------------------------------------------------------

% if we can find class, return all possibilities
pp_semantics( PP, SemList ) :-
        findall( Sem, pp_sem( PP, Sem ), SemList ),
        \+ SemList = [],
        !.

% otherwise undefined
pp_semantics( _PP, [xxx] ).


%%-----------------------------------------------------
%% pp_sem/2
%% pp_sem( [+Prep, +NP], ?ClassAtom )
%%
%% Description:
%%   ClassAtom is a possible semantic class for
%%   the PP consisting of tagged Prep and NP
%% Succeeds: 1
%% Side Effects: none
%%------------------------------------------------------

% location: must have (concrete) obj or location NP
pp_sem( [Prep/prep, NP], loc ) :-
        prep( Prep, SemList ),
        member( loc, SemList ),
        np( NP ),
        (
          sem( NP, loc );
          sem( NP, obj )
        ).

% time: must have number NP
pp_sem( [Prep/prep, NP], tim ) :-
        prep( Prep, SemList ),
        member( tim, SemList ),
        np( NP ),
        sem( NP, num ).

% possession: any NP
pp_sem( [Prep/prep, NP], pos ) :-
        prep( Prep, SemList ),
        member( pos, SemList ),
        np( NP ).

% "inverse" possession: any NP
pp_sem( [Prep/prep, NP], sop ) :-
        prep( Prep, SemList ),
        member( sop, SemList ),
        np( NP ).
```

```
%%-----------------------------------------------------
%% vg_semantics/2
%% vg_semantics( +RawSentList, ?SemSentList )
%%
%% Description:
%%   SemSentList is the result of applying voice & VG head
%%   (predicate) info to sentence RawSentList
%% Succeeds: 1*
%% Side Effects: none
%%------------------------------------------------------

% base case
vg_semantics( [], [] ).

% be+V/vbn -> passive V
vg_semantics( [vg:[v:List | T] | Tail1], [vg:pas#Sem:[v:List | T] | Tail2] ) :-
        member( PassVerb, [be, become, get] ),
        nth( N, List, PassVerb/_ ),
        nth( M, List, Sem/Type ),
        M > N,
        member( Type, [vbd, (vbd/_), vbn, (vbn/_)] ),
        !,
        vg_semantics( Tail1, Tail2 ).

% V/vbn at the end -> passive V
vg_semantics( [vg:[v:List | T] | Tail1], [vg:pas#Sem:[v:List | T] | Tail2] ) :-
        end_member( Sem/vbn, List ),
        vg_semantics( Tail1, Tail2 ).

% V1 to V2 -> active V1 if V1 is a want-type verb
% (prevents "want to be" -> be, allows "used to be" -> be)
vg_semantics( [vg:[v:List | T] | Tail1], [vg:act#V1:[v:List | T] | Tail2] ) :-
        end_member( V2/vb, List ),
        end_member( V1/Type, List ),
        vb_type( Type ),
        hypo_verb( V1 ),
        nth( N1, List, V1/Type ),
        nth( N2, List, V2/vb ),
        N2 > N1,
        N is N2 - 1,
        nth( N, List, _/to ),
        !,
        vg_semantics( Tail1, Tail2 ).

% otherwise any V at the end -> active V
vg_semantics( [vg:[v:List | T] | Tail1], [vg:act#Sem:[v:List | T] | Tail2] ) :-
        end_member( Sem/Type, List ),
        vb_type( Type ),
        !,
        vg_semantics( Tail1, Tail2 ).

% otherwise recurse doing nothing
vg_semantics( [Head | Tail1], [Head | Tail2] ) :-
        vg_semantics( Tail1, Tail2 ).


%%-----------------------------------------------------
%% sem/2
%% sem( +Phrase, ?SemAtom )
%%
%% Description:
%%   SemAtom is a possible semantic class for a NP/PP/whq,
%%   or predicate name for a VG
%% Succeeds: 1*
%% Side Effects: none
```

```
%%------------------------------------------------------

% for NP: member of semantic class list
sem( Type:_Num/SemList:_List, Sem ) :-
        np_type( Type ),
        member( Sem, SemList ).

% for PP: member of semantic class list
sem( pp:SemList:_List, Sem ) :-
        member( Sem, SemList ).

% for VG: predicate
sem( vg:_Voice#Sem:_List, Sem ).

% for whq-word: member of semantic class list
sem( WhWord, Sem ) :-
        whq( WhWord, SemList, _ ),
        member( Sem, SemList ).

% look up indexed entity & recurse
sem( e( N ), Sem ) :-
        e( N, NP ),
        sem( NP, Sem ).


%%-----------------------------------------------------
%% no_gender/2
%% no_gender( +SemClass, ?SemAtom )
%%
%% Description:
%%   SemAtom is the broad class of SemClass (gender removed)
%%   i.e. per for per/_
%%   (or no change for anything else)
%% Succeeds: 1
%% Side Effects: none
%%------------------------------------------------------

no_gender( Class/_, Class ) :-
        !.

no_gender( Class, Class ).


%%-----------------------------------------------------
%% num/2
%% num( +Phrase, ?NumAtom )
%%
%% Description:
%%   NumAtom is 's' or 'p' for a NP
%% Succeeds: 1
%% Side Effects: none
%%------------------------------------------------------

% just return number feature
num( Type:Num/_SemList:_List, Num ) :-
        np_type( Type ).

% look up indexed entity & recurse
num( e( N ), Num ) :-
        e( N, NP ),
        num( NP, Num ).


%%-----------------------------------------------------
%% feat/2
```

```
%% feat( +Phrase, ?FeatCompound )
%%
%% Description:
%%   FeatCompound is number & semantic class info for a NP,
%%   (in form Num/SemList), or voice & predicate info for
%%   a VG (in form Voice#Pred)
%% Succeeds: 1*
%% Side Effects: none
%%------------------------------------------------------

% for NP: Num/SemList
feat( Type:Feat:_List, Feat ) :-
        np_type( Type ).

% for VG: Voice#Pred
feat( vg:Feat:_List, Feat ).

% look up indexed entity & recurse
feat( e( N ), Feat ) :-
        e( N, NP ),
        feat( NP, Feat ).


%%-----------------------------------------------------
%% type/2
%% type( +Phrase, ?TypeAtom )
%%
%% Description:
%%   TypeAtom is phrase type marker for any phrase,
%%   e.g. "np" for a NP
%% Succeeds: 1*
%% Side Effects: none
%%------------------------------------------------------

% for any phrase: get type marker
type( Type:_, Type ).

% look up indexed entity & recurse
type( e( N ), Type ) :-
        e( N, NP ),
        type( NP, Type ).
```

vg_syn.pl

```
%%-----------------------------------------------------
%% vg_syn.pl
%%
%% Contains: predicates defining syntactic rules for
%%           building verb groups (VGs)
%%-----------------------------------------------------

%%-----------------------------------------------------
%% make_vgs/2
%% make_vgs( +RawSentList, ?CookedSentList )
%%
%% Description:
%%   CookedSentList is the result of forming VG-type
%%   words (forcibly) into VGs
%% Succeeds: 1*
%% Side Effects: none
%%------------------------------------------------------

% base case
make_vgs( [], [] ).
```

```
% if we find a VG-type word, form biggest legal VG
make_vgs( [Head | Tail1], [vg:[v:Result] | Tail2] ) :-
%       vg_word( Head ),
        largest_vg( [Head | Tail1], Result, Tail ),
        \+ illegal_vg( Result ),
        !,
        make_vgs( Tail, Tail2 ).

% otherwise, recurse doing nothing
make_vgs( [Head | Tail1], [Head | Tail2] ) :-
        make_vgs( Tail1, Tail2 ).


%%-----------------------------------------------------
%% largest_vg/3
%% largest_vg( +RawList, ?VGList, ?RemainderList )
%%
%% Description:
%%   VGList is the list of all VG-type words at the head
%%   of RawList, with RemainderList the tail left over.
%% Succeeds: 0-1
%% Side Effects: none
%%-----------------------------------------------------

% for VG-type words: add to VGList and recurse
largest_vg( [Head | Tail1], [Head | VGTail], Rest ) :-
        vg_word( Head ),
        !,
        largest_vg( Tail1, VGTail, Rest ).

% for "there": if in query, add to VGList and recurse
largest_vg( [there/ex | Tail1], [there/ex | VGTail], Rest ) :-
        process_type( query ),
        !,
        largest_vg( Tail1, VGTail, Rest ).

% for "there": if in answer, add to VGList and recurse if followed by
% "is": otherwise, might really be the location anaphor "there"
largest_vg( [there/ex | Tail1], [there/ex | VGTail], Rest ) :-
        process_type( answer ),
        Tail1 = [be/_ | _ ],
        !,
        largest_vg( Tail1, VGTail, Rest ).

% when we meet a non-VG-type word, initialise empty VGList
% and copy what's left to RemainderList
largest_vg( Rest, [], Rest ).


%%-----------------------------------------------------
%% attach_advs/2
%% attach_advs( +RawList, ?CookedList )
%%
%% Description:
%%   CookedList is the result of attaching (forcibly)
%%   all free adverbs (VGs consisting entirely of adverbs)
%%   to the nearest VG
%% Succeeds: 1
%% Side Effects: none
%%-----------------------------------------------------

% if no free advs and something to attach them to, do nothing
attach_advs( In, In ) :-
        \+ (
              member( vg:[v:List], In ),
```

```
                all_advs( List )
            ).

% if we find a free adv VG, attach to nearest VG and recurse
attach_advs( In, Out ) :-
        nth( N, In, vg:[v:List1], Rest ),
        all_advs( List1 ),
        nearest( N, vg:[v:List2], Rest, M ),
        !,
        subst_nth( M, vg:[v:List2,adv:List1], Rest, Mid ),
        attach_advs( Mid, Out ).

% if we can't find a nearest VG, attach to NP and recurse
attach_advs( In, Out ) :-
        nth( N, In, vg:[v:List1], Rest ),
        all_advs( List1 ),
        nearest( N, np:List2, Rest, M ),
        !,
        append( List1, List2, List ),
        subst_nth( M, np:List, Rest, Mid ),
        attach_advs( Mid, Out ).


%%-----------------------------------------------------
%% all_advs/1
%% all_advs( +List )
%%
%% Description:
%%   Succeeds if all members of List are of adverb type.
%%   This was done using all_members, but can't be if we
%%   need to accept /rbr, /rbs types
%% Succeeds: 0-1
%% Side Effects: none
%%------------------------------------------------------

% base case
all_advs( [] ).

% recurse, checking type
all_advs( [_/Type | T] ) :-
        adv_type( Type ),
        all_advs( T ).


%%-----------------------------------------------------
%% move_advs/2
%% move_advs( +RawList, ?CookedList )
%%
%% Description:
%%   CookedList is the result of separating (forcibly)
%%   all adverbs within VGs from the V sub-group to a new
%%   ADV sub-group
%% Succeeds: 1
%% Side Effects: none
%%------------------------------------------------------

% base case
move_advs( [], [] ).

% for a VG: call move_all_advs/2 and recurse
move_advs( [vg:VG1 | Tail1], [vg:VG2 | Tail2] ) :-
        move_all_advs( VG1, VG2 ),
        move_advs( Tail1, Tail2 ),
        !.
```

```
% otherwise recurse, doing nothing
move_advs( [Head | Tail1], [Head | Tail2] ) :-
        move_advs( Tail1, Tail2 ).


%%---------------------------------------------------
%% move_all_advs/2
%% move_all_advs( +RawVGList, ?CookedVGList )
%%
%% Description:
%%   CookedVGList is the result of moving (forcibly)
%%   all adverbs from within the V sub-group to a new
%%   ADV sub-group
%% Succeeds: 1
%% Side Effects: none
%%----------------------------------------------------

% if no advs within V group, do nothing
move_all_advs( [v:List | Tail], [v:List | Tail] ) :-
        \+ (
             member( _Adv/Type, List ),
             adv_type( Type )
           ).

% if adv found: if no ADV group, create and move
move_all_advs( [v:List1], Result ) :-
        nth( _N, List1, Adv/Type, List2 ),
        adv_type( Type ),
        move_all_advs( [v:List2, adv:[Adv/Type]], Result ).

% if adv found: if ADV group exists, append
move_all_advs( [v:List1, adv:Mod], Result ) :-
        nth( _N, List1, Adv/Type, List2 ),
        adv_type( Type ),
        append( Mod, [Adv/Type], Mod2 ),
        move_all_advs( [v:List2, adv:Mod2], Result ).


%%---------------------------------------------------
%% combine_q_vgs/2
%% combine_q_vgs( +RawSentList, ?CookedSentList )
%%
%% Description:
%%   CookedSentList is the result of combining (forcibly)
%%   a "modal-type" VG with the main VG in a query
%% Succeeds: 1
%% Side Effects: none
%%----------------------------------------------------

% inverted query:
% "Where does X live?" -> "X [does,live] where?"
% "In which city does X live?" -> "X [does,live] in which city?"
combine_q_vgs( Sent1, Sent2 ) :-
        append( Prefix, [vg:[v:[Verb/Type] | T] | Tail], Sent1 ),
        inverted_q_prefix( Prefix ),
        count_syn_vgs( Prefix, 0 ),
        q_verb( Verb/Type ),
        count_syn_vgs( Tail, N ),
        N >= 1,
        !,
        combine_vgs( [vg:[v:[Verb/Type] | T] | Tail], Result ),
        append( Result, Prefix, Sent2 ).

% normal modal-type query:
% "Does X live in London?" -> "X [does,live] in London?"
```

```
combine_q_vgs( Sent1, Sent2 ) :-
        append( Prefix, [vg:[v:[Verb/Type] | T] | Tail], Sent1 ),
        count_syn_vgs( Prefix, 0 ),
        q_verb( Verb/Type ),
        count_syn_vgs( Tail, N ),
        N >= 1,
        !,
        combine_vgs( [vg:[v:[Verb/Type] | T] | Tail], Result ),
        append( Prefix, Result, Sent2 ).

% otherwise do nothing:
% "Who lives in London?" -> "Who [live] in London?"
combine_q_vgs( Sent, Sent ).


%%-----------------------------------------------------
%% inverted_q_prefix/1
%% inverted_q_prefix( +List )
%%
%% Description:
%%   Succeeds if List is the beginning of an inverted query
%%   i.e. begins with a whq-word or preposition+whq-word
%% Succeeds: 0-1
%% Side Effects: none
%%-----------------------------------------------------

inverted_q_prefix( [WhWord | _T] ) :-
        whq( WhWord, _, _ ).

inverted_q_prefix( [np:[WhWord | _NP] | _T] ) :-
        whq( WhWord, _, _ ).

inverted_q_prefix( [_Prep/prep, WhWord | _T] ) :-
        whq( WhWord, _, _ ).

inverted_q_prefix( [_Prep/prep, np:[WhWord | _NP] | _T] ) :-
        whq( WhWord, _, _ ).


%%-----------------------------------------------------
%% q_verb/1
%% q_verb( +TaggedWord )
%%
%% Description:
%%   Succeeds if TaggedWord is a "modal-type" query verb
%% Succeeds: 0-1
%% Side Effects: none
%%-----------------------------------------------------

% modals: e.g. "should X marry Y?"
q_verb( _Verb/Type ) :-
        member( Type, [md, (md/_)] ).

% be, do, have: e.g. "is X living with Y?", "does X even like Y?",
% "has X already married Z?"
q_verb( Verb/_Type ) :-
        member( Verb, [be, do, have] ).


%%-----------------------------------------------------
%% combine_vgs/2
%% combine_vgs( +RawSentList, ?CookedSentList )
%%
%% Description:
%%   CookedSentList is the result of compounding (forcibly)
```

```
%%   all VGs in RawSentList
%% Succeeds: 1
%% Side Effects: none
%%------------------------------------------------------

% if no other VGs to combine with, do nothing: e.g. "is Snowdon in Wales?"
combine_vgs( [vg:VG | Tail], [vg:VG | Tail] ) :-
        \+ member( vg:_, Tail ).

% otherwise call combine_vgs/3
combine_vgs( [vg:VG | Tail1], Tail2 ) :-
        memberchk( vg:_, Tail1 ),
        combine_vgs( vg:VG, Tail1, Tail2 ).


%%------------------------------------------------------
%% combine_vgs/3
%% combine_vgs( +QueryVG, +RawSentList, ?CookedSentList )
%%
%% Description:
%%   CookedSentList is the result of adding (forcibly)
%%   QueryVG to the VGs in RawSentList
%% Succeeds: 1
%% Side Effects: none
%%------------------------------------------------------

% base case
combine_vgs( _, [], [] ).

% if a list of conjoined VPs, give this Q-verb to each one
combine_vgs( QVG, [vg:VG1 | Tail1], [vg:VG2 | Tail2] ) :-
        list_of_vps( [vg:VG1 | Tail1] ),
        !,
        append_vgs( QVG, vg:VG1, vg:VG2 ),
        combine_vgs( QVG, Tail1, Tail2 ).

% otherwise give QVG to the first (only?) verb
combine_vgs( QVG, [vg:VG1 | Tail], [vg:VG2 | Tail] ) :-
        !,
        append_vgs( QVG, vg:VG1, vg:VG2 ).

% otherwise recurse until we find a VG
combine_vgs( QVG, [Head | Tail1], [Head | Tail2] ) :-
        combine_vgs( QVG, Tail1, Tail2 ).


%%------------------------------------------------------
%% np_vg/1
%% np_vg( +VG )
%%
%% Description:
%%   Succeeds if VG is the type of VG that belongs in a
%   NG (e.g. single-word "ing" verb)
%% Succeeds: 0-1
%% Side Effects: none
%%------------------------------------------------------

% true for ing-VGs
np_vg( VG ) :-
        ing_vg( VG ).

% true for to-VGs
np_vg( VG ) :-
        to_vg( VG ).
```

```
%%---------------------------------------------------------
%% ing_vg/1
%% ing_vg( +VG )
%%
%% Description:
%%   Succeeds if VG is a single-word "ing" VG
%% Succeeds: 0-1
%% Side Effects: none
%%---------------------------------------------------------

% true if V sub-group contains one word tagged 'vbg'
ing_vg( vg:[v:[_/vbg] | _Tail] ).

% true if V sub-group contains one word tagged 'vbg' (complex)
ing_vg( vg:[v:[_/(vbg/_)] | _Tail] ).

% if VG has sem features attached, remove & recurse
ing_vg( vg:_:VG ) :-
        ing_vg( vg:VG ).


%%---------------------------------------------------------
%% to_vg/1
%% to_vg( +VG )
%%
%% Description:
%%   Succeeds if VG is "to" + a single-word infintive verb
%% Succeeds: 0-1
%% Side Effects: none
%%---------------------------------------------------------

% true if V sub-group is "to verb"
to_vg( vg:[v:[_/to, _/vb] | _Tail] ).

% true if V sub-group is "to verb" (complex tag)
to_vg( vg:[v:[_/to, _/(vb/_)] | _Tail] ).

% if VG has sem features attached, remove & recurse
to_vg( vg:_:VG ) :-
        to_vg( vg:VG ).


%%---------------------------------------------------------
%% append_vgs/3
%% append_vgs( +VG1, +VG2, ?VG3 )
%%
%% Description:
%%   VG3 is the result of appending VG1 and VG2
%% Succeeds: 0-1
%% Side Effects: none
%%---------------------------------------------------------

% if VGs just contain V sub-groups, just append
append_vgs( vg:[v:V1], vg:[v:V2], vg:[v:V] ) :-
        append( V1, V2, V ).

% if one contains a ADV group, copy it
append_vgs( vg:[v:V1, adv:M1], vg:[v:V2], vg:[v:V, adv:M1] ) :-
        append( V1, V2, V ).

% if one contains a ADV group, copy it
append_vgs( vg:[v:V1], vg:[v:V2, adv:M2], vg:[v:V, adv:M2] ) :-
        append( V1, V2, V ).
```

```
% if both contain ADV groups, append them
append_vgs( vg:[v:V1, adv:M1], vg:[v:V2, adv:M2], vg:[v:V, adv:M] ) :-
        append( V1, V2, V ),
        append( M1, M2, M ).


%%-----------------------------------------------------
%% count_syn_vgs/2
%% count_syn_vgs( +SentList, ?N )
%%
%% Description:
%%   N is the number of non-NG-type VGs in SentList
%% Succeeds: 0-1
%% Side Effects: none
%%------------------------------------------------------

count_syn_vgs( [], 0 ).

count_syn_vgs( [vg:VG | Tail], N ) :-
        np_vg( vg:VG ),
        !,
        count_syn_vgs( Tail, N ).

count_syn_vgs( [vg:_ | Tail], N ) :-
        !,
        count_syn_vgs( Tail, M ),
        N is M + 1.

count_syn_vgs( [_Head | Tail], N ) :-
        count_syn_vgs( Tail, N ).


%%-----------------------------------------------------
%% count_verbs/2
%% count_verbs( +SentList, ?N )
%%
%% Description:
%%   N is the number of VGs in SentList
%% Succeeds: 0-1
%% Side Effects: none
%%------------------------------------------------------

% base case
count_verbs( [], 0 ).

count_verbs( [vg:VG | Tail], N ) :-
        ing_vg( vg:VG ),
        count_verbs( Tail, N ).

count_verbs( [vg:_ | Tail], N ) :-
        count_verbs( Tail, M ),
        N is M + 1.

count_verbs( [Head | Tail], N ) :-
        \+ Head = vg:_,
        count_verbs( Tail, N ).
```

ng_syn.pl

```
%%-----------------------------------------------------
%% ng_syn.pl
%%
%% Contains: predicates defining syntactic rules for
%%           building noun groups (NGs)
%%-----------------------------------------------------
```

```
%%-----------------------------------------------------
%% np_compounds/2
%% np_compounds( +RawSentList, ?CookedSentList )
%%
%% Description:
%%   CookedSentList is the result of compounding
%%   (optionally) adjacent single-word NPs, and
%%   (forcibly) possessives and how+adj phrases
%% Succeeds: 1*
%% Side Effects: none
%%-----------------------------------------------------

% base case
np_compounds( [], [] ).

% recurse, compounding possessives into NP
np_compounds( [np:H1, np:[H2/pos] | Tail1],
              [ng:[np:H1, np:[H2/pos]] | Tail2] ) :-
        !,
        np_compounds( Tail1, Tail2 ).

% recurse, compounding how+adj/adv into NP
np_compounds( [how/wrb, NP | Tail1],
              [ng:[how/wrb, NP] | Tail2] ) :-
        np( NP ),
        np_head( NP, _Adj/Type ),
        ad_type( Type ),
        !,
        np_compounds( Tail1, Tail2 ).

% recurse, compounding NP+adj into NP
np_compounds( [NP1, NP2 | Tail1],
              [ng:[NP1, NP2] | Tail2] ) :-
        np( NP1 ),
        np( NP2 ),
        np_head( NP2, _Adj/Type ),
        adj_type( Type ),
%       !,
        np_compounds( Tail1, Tail2 ).

% recurse, compounding two adjacent simple (single-word) NPs
np_compounds( [H1, H2 | Tail1], [ng:[H1, H2] | Tail2] ) :-
        simple_np( H1 ),
        simple_np( H2 ),
%       !,
        np_compounds( Tail1, Tail2 ).

% or recurse, doing nothing
np_compounds( [Head | Tail1], [Head | Tail2] ) :-
        np_compounds( Tail1, Tail2 ).


%%-----------------------------------------------------
%% np_conjunctions/2
%% np_conjunctions( +RawSentList, ?CookedSentList )
%%
%% Description:
%%   CookedSentList is the result of compounding
%%   (optionally) lists of conjoined NPs
%% Succeeds: 1*
%% Side Effects: none
%%-----------------------------------------------------

% base case
```

```
np_conjunctions( [], [] ).

% recurse, conjoining comma/conj separated lists of NPs
np_conjunctions( Sentence, [ng:Prefix | Tail2] ) :-
        reverse_append( Prefix, Tail1, Sentence ),
        list_of_similar_nps( Prefix ),
%       !, not obligatory as may be e.g. conjoined VPs
        np_conjunctions( Tail1, Tail2 ).

% or recurse, doing nothing
np_conjunctions( [Head | Tail1], [Head | Tail2] ) :-
        np_conjunctions( Tail1, Tail2 ).


%%-------------------------------------------------------
%% make_ngs/2
%% make_ngs( +RawSentList, ?CookedSentList )
%%
%% Description:
%%   CookedSentList is the result of compounding
%%   (forcibly) NP-type VGs with NPs
%% Succeeds: 1*
%% Side Effects: none
%%--------------------------------------------------------

% base case
make_ngs( [], [] ).

% recurse, applying NG <- VG(ing) NP
make_ngs( [vg:VG, NP | Tail1], [ng:[vg:VG, NP] | Tail2] ) :-
        np_vg( vg:VG ),
        np( NP ),
        !,
        make_ngs( Tail1, Tail2 ).

% or recurse, doing nothing
make_ngs( [Head | Tail1], [Head | Tail2] ) :-
        make_ngs( Tail1, Tail2 ).


%%-------------------------------------------------------
%% list_of_nps/1
%% list_of_nps( +SentList )
%%
%% Description:
%%   Succeeds if SentList is a list of conjoined NPs
%% Succeeds: 0-1
%% Side Effects: none
%%--------------------------------------------------------

% true for "NP1 etc"
list_of_nps( [NP1, Word] ) :-
        np( NP1 ),
        conj_word( Word ).

% true for "NP1, etc"
list_of_nps( [NP1, ','/',', Word] ) :-
        np( NP1 ),
        conj_word( Word ).

% true for "NP1 and NP2"
list_of_nps( [NP1, _/cc, NP2] ) :-
        np( NP1 ),
        np( NP2 ).
```

```
% true for "NP1, and NP2"
list_of_nps( [NP1, ','/',', _/cc, NP2] ) :-
        np( NP1 ),
        np( NP2 ).

% true for "NP1 [,*and*] (L)" if true for L
list_of_nps( [NP1 | Tail] ) :-
        np( NP1 ),
        append( ConjList, [NP2 | Tail2], Tail ),
        np( NP2 ),
        all_conjs( ConjList ),
        list_of_nps( [NP2 | Tail2] ).


%%-----------------------------------------------------
%% conj_word/1
%% conj_word( ?Word )
%%
%% Description:
%%   Succeeds if Word is a NP/word often found in
%%   conjunctions
%% Succeeds: 1*
%% Side Effects: none
%%------------------------------------------------------

conj_word( np:[etc/fw] ).
conj_word( np:['etc.'/fw] ).
conj_word( np:[both/np] ).
conj_word( np:['so_on'/np] ).

% if features attached, remove & check
conj_word( np:_:X ) :-
        conj_word( np:X ).

% look up e()
conj_word( e( N ) ) :-
        e( N, X ),
        conj_word( X ).


%%-----------------------------------------------------
%% all_conjs/1
%% all_conjs( +List )
%%
%% Description:
%%   Succeeds if all members of List are conjunction
%%   tokens: i.e. commas or "and"s
%% Succeeds: 0-1
%% Side Effects: none
%%------------------------------------------------------

% base case
all_conjs( [] ).

% commas acceptable
all_conjs( [','/',' | T] ) :-
        all_conjs( T ).

% and so is "and"
all_conjs( [_/cc | T] ) :-
        all_conjs( T ).


%%-----------------------------------------------------
%% list_of_similar_nps/1
```

```
%% list_of_similar_nps( +SentList )
%%
%% Description:
%%   Succeeds if SentList is a list of conjoined NPs
%%   which all share semantic class. Just an interface
%%   to list_of_similar_nps/2
%% Succeeds: 0-1
%% Side Effects: none
%%-------------------------------------------------------

% call list_of_similar_nps/2
list_of_similar_nps( A ) :-
        list_of_nps( A ),
        similar_nps( A, _ ).


%%------------------------------------------------------
%% similar_nps/2
%% similar_nps( +SentList, ?SemAtom )
%%
%% Description:
%%   Succeeds if all NPs in SentList
%%   share the semantic class SemAtom
%% Succeeds: 0-1
%% Side Effects: none
%%-------------------------------------------------------

% base case
similar_nps( [], _ ).

% non-NPs or conj words don't need to share sem
similar_nps( [Head | Tail], Sem ) :-
        (
          \+ np( Head );
          conj_word( Head )
        ),
        similar_nps( Tail, Sem ).

% other NPs must, (but [per]s don't need to share gender
similar_nps( [NP | Tail], Sem ) :-
        np( NP ),
        np_semantics( NP, SemList ),
        member( Sem1, SemList ),
        no_gender( Sem1, Sem ),
        similar_nps( Tail, Sem ).


%%------------------------------------------------------
%% list_of_nps_no_punct/1
%% list_of_nps_no_punct( +SentList )
%%
%% Description:
%%   Succeeds if SentList is a list of conjoined NPs.
%%   Equivalent to list_of_nps/1 for use once punctation
%%   has been removed
%% Succeeds: 0-1
%% Side Effects: none
%%-------------------------------------------------------

% true for "NP1 etc."
list_of_nps_no_punct( [NP1, NP2] ) :-
        np( NP1 ),
        conj_word( NP2 ).

% true for "NP1 and NP2"
```

```
list_of_nps_no_punct( [NP1, _/cc, NP2] ) :-
        np( NP1 ),
        np( NP2 ).

% true for "NP1 (L)" if true for L
list_of_nps_no_punct( [NP1 | Tail] ) :-
        np( NP1 ),
        list_of_nps_no_punct( Tail ).

% true for "NP1 (L)" if true for L
list_of_nps_no_punct( [NP1, _/cc | Tail] ) :-
        np( NP1 ),
        list_of_nps_no_punct( Tail ).
```

syn_defs.pl

```
%%-----------------------------------------------------
%% syn_defs.pl
%%
%% Contains: definitions for syntactic rule predicates
%%-----------------------------------------------------

%%-----------------------------------------------------
%% simple_np/1
%% simple_np( +NP )
%%
%% Description:
%%   Succeeds if NP is a single-word NP
%% Succeeds: 0-1
%% Side Effects: none
%%------------------------------------------------------

% single word NP (could be mistagged as NN/NNS)
simple_np( np:[_Word/Type] ) :-
        noun_type( Type ).

% look up indexed entity & recurse
simple_np( e( N ) ) :-
        e( N, NP ),
        simple_np( NP ).


%%-----------------------------------------------------
%% np/1
%% np( +NP )
%%
%% Description:
%%   Succeeds if NP is a NP (i.e. has a NP-type marker)
%% Succeeds: 0-1
%% Side Effects: none
%%------------------------------------------------------

% np:_ is a NP
np( np:_List ).

% and so is ng:_
np( ng:_List ).

% look up indexed entity & recurse
% (actually, only NPs get indexed at the moment, so don't
% need to recurse, but this is safer in case of changes)
np( e( N ) ) :-
        e( N, NP ),
        np( NP ).
```

```
%%-----------------------------------------------------
%% np_type/1
%% np_type( +TypeMarkerAtom )
%%
%% Description:
%%   Succeeds if TypeMarkerAtom is of NP type
%% Succeeds: 0-1
%% Side Effects: none
%%------------------------------------------------------

np_type( np ).
np_type( ng ).


%%-----------------------------------------------------
%% noun_type/1
%% noun_type( +TypeMarkerAtom )
%%
%% Description:
%%   Succeeds if TypeMarkerAtom is of noun type
%% Succeeds: 0-1
%% Side Effects: none
%%------------------------------------------------------

noun_type( nn ).
noun_type( nns ).
noun_type( np ).
noun_type( cd ).


%%-----------------------------------------------------
%% ad_type/1
%% ad_type( +TypeMarkerAtom )
%%
%% Description:
%%   Succeeds if TypeMarkerAtom is of adj or adv type
%% Succeeds: 0-1
%% Side Effects: none
%%------------------------------------------------------

ad_type( A ) :-
        adj_type( A ).

ad_type( A ) :-
        adv_type( A ).


%%-----------------------------------------------------
%% adv_type/1
%% adv_type( +TypeMarkerAtom )
%%
%% Description:
%%   Succeeds if TypeMarkerAtom is of adv type
%% Succeeds: 0-1
%% Side Effects: none
%%------------------------------------------------------

adv_type( rb ).
adv_type( rbr ).
adv_type( rbs ).


%%-----------------------------------------------------
%% adj_type/1
```

```
%% adj_type( +TypeMarkerAtom )
%%
%% Description:
%%   Succeeds if TypeMarkerAtom is of adj type
%% Succeeds: 0-1
%% Side Effects: none
%%-----------------------------------------------------

adj_type( jj ).
adj_type( jjr ).
adj_type( jjs ).


%%----------------------------------------------------
%% vb_type/1
%% vb_type( +TypeMarkerAtom )
%%
%% Description:
%%   Succeeds if TypeMarkerAtom is of verb type
%% Succeeds: 0-1
%% Side Effects: none
%%-----------------------------------------------------

vb_type( vb ).
vb_type( vbz ).
vb_type( vbp ).
vb_type( vbd ).
vb_type( vbn ).
vb_type( vbg ).

% for compound types caused by shallowproc verb info,
% convert to simple type & recurse
vb_type( V/_ ) :-
        vb_type( V ).


%%----------------------------------------------------
%% illegal_vg/1
%% illegal_vg( +WordList )
%%
%% Description:
%%   Succeeds if WordList cannot form a legal VG on its
%%   own
%% Succeeds: 0-1
%% Side Effects: none
%%-----------------------------------------------------

% "to" on its own (probably mistagged prep)
illegal_vg( [_/to] ).

% "that" on its own (probably mistagged relative)
illegal_vg( [_/comp] ).

% empty list is not legal!
illegal_vg( [] ).


%%----------------------------------------------------
%% vg_word/1
%% vg_word( +TaggedWord )
%%
%% Description:
%%   Succeeds if TaggedWord can form part of a VG
%% Succeeds: 0-1
%% Side Effects: none
```

```
%%-------------------------------------------------------

% any verb type
vg_word( _Word/Type ) :-
        vb_type( Type ).

vg_word( _Word/Type ) :-
        adv_type( Type ).

% modals, adverbs, "to" and "that"
vg_word( _/md ).
vg_word( _/(md/_) ).
vg_word( _/to ).
vg_word( _/comp ).

% existential "there"
% this is now specified more subtlely in vg_syn.pl
% vg_word( there/ex ).
```

pp.pl

```
%%------------------------------------------------------
%% pp.pl
%%
%% Contains: predicates for making & attaching
%%           prepositional phrases (PPs)
%%------------------------------------------------------

%%------------------------------------------------------
%% make_pps/2
%% make_pps( +RawSentList, ?CookedSentList )
%%
%% Description:
%%   CookedSentList is the result of forming
%%   (optionally) PPs from [prep, NP]
%% Succeeds: 1*
%% Side Effects: none
%%-------------------------------------------------------

% base case
make_pps( [], [] ).

% recurse, checking within compound NPs
make_pps( [ng:List1 | Tail1], [ng:List2 | Tail2] ) :-
        !,
        make_pps( List1, List2 ),
        make_pps( Tail1, Tail2 ).

% recurse, applying PP <- P NP
% and checking within compound NPs
make_pps( [Prep/prep, NP:List1 | Tail1],
            [pp:[Prep/prep, NP:List2] | Tail2] ) :-
        np( NP:List1 ),
        !,
        make_pps( List1, List2 ),
        make_pps( Tail1, Tail2 ).

% or recurse, doing nothing
make_pps( [Head | Tail1], [Head | Tail2] ) :-
        make_pps( Tail1, Tail2 ).


%%------------------------------------------------------
%% attach_np_pps/2
%% attach_np_pps( +RawSentList, ?CookedSentList )
```

```
%%
%% Description:
%%   CookedSentList is the result of (usually optionally)
%%   attaching NP-type PPs to neighbouring NPs
%% Succeeds: 1*
%% Side Effects: none
%%------------------------------------------------------

% base case
attach_np_pps( [], [] ).

% recurse, forcibly applying NP <- NP's NP
attach_np_pps( [NP1, NP2 | Tail1],
               [ng:Feat:[NP1, NP2] | Tail2] ) :-
        np( NP1 ),
        np( NP2 ),
        sem( NP1, pos ),
        feat( NP2, Feat ),
        !,
        attach_np_pps( Tail1, Tail2 ).

% recurse, optionally applying (NP (P (NP PP))) <- (NP (P NP)) PP
% for non-VG-type PPs
attach_np_pps( [ng:F:[NP, pp:F1:[P1, NP1]], pp:PP2 | Tail1], Tail2 ) :-
        np( NP ),
        np( NP1 ),
        feat( NP1, Feat ),
        \+ vg_pp( pp:PP2 ),
        attach_np_pps( [ng:F:[NP, pp:F1:[P1, ng:Feat:[NP1, pp:PP2]]] |
                          Tail1],
                          Tail2 ).


% recurse, optionally applying NP <- NP PP for non-VG-type PPs
attach_np_pps( [NP, pp:PP | Tail1],
               Tail2 ) :-
        np( NP ),
        feat( NP, Feat ),
        \+ vg_pp( pp:PP ),
        attach_np_pps( [ng:Feat:[NP, pp:PP] | Tail1], Tail2 ).

% recurse, optionally applying (P NP) PP <- (P (NP PP))
% for non-VG-type PPs
attach_np_pps( [pp:Sem:[Prep/prep, NP], pp:PP | Tail1],
               [pp:Sem:[Prep/prep,
                              ng:Feat:[NP, pp:PP]] | Tail2] ) :-
        np( NP ),
        feat( NP, Feat ),
        \+ vg_pp( pp:Sem:[Prep/prep, NP] ),
        \+ vg_pp( pp:PP ),
        attach_np_pps( Tail1, Tail2 ).

% or recurse, doing nothing
attach_np_pps( [Head | Tail1], [Head | Tail2] ) :-
        attach_np_pps( Tail1, Tail2 ).


%%------------------------------------------------------
%% attach_vg_pps/2
%% attach_vg_pps( +RawSentList, ?CookedSentList )
%%
%% Description:
%%   CookedSentList is the result of (forcibly)
%%   attaching VG-type PPs to nearest VGs
%% Succeeds: 1*
```

```
%% Side Effects: none
%%-----------------------------------------------------

% if we can find a VG-type PP, attach to nearest VG
% and recurse
attach_vg_pps( In, Out ) :-
        nth( N, In, pp:PP, Rest ),
        vg_pp( pp:PP ),
        nearest( N, vg:Feat:VGList, Rest, M ),
        append( VGList, [pp:PP], VGList2 ),
        subst_nth( M, vg:Feat:VGList2, Rest, Mid ),
        !,
        attach_vg_pps( Mid, Out ).

% in queries only, do the same for whq-words
attach_vg_pps( In, Out ) :-
        process_type( query ),
        nth( N, In, WhWord, Rest ),
        whq( WhWord, _, _ ),
        vg_whq( WhWord ),
        nearest( N, vg:Feat:VGList, Rest, M ),
        append( VGList, [WhWord], VGList2 ),
        subst_nth( M, vg:Feat:VGList2, Rest, Mid ),
        !,
        attach_vg_pps( Mid, Out ).

% in answers only, stop if no more VG-type PPs found
attach_vg_pps( In, In ) :-
        process_type( answer ),
        \+ (
             member( pp:PP, In ),
             vg_pp( pp:PP ),
             member( vg:_, In )
           ).

% in queries only, stop if no more VG-type PPs and no
% more whq-words found
% we really need something to prevent "how long" being moved here
% when it's a length, not a duration
attach_vg_pps( In, In ) :-
        process_type( query ),
        \+ (
             member( pp:PP, In ),
             vg_pp( pp:PP ),
             member( WhWord, In ),
             whq( WhWord, _, _ ),
             vg_whq( WhWord ),
             member( vg:_, In )
           ).


%%-----------------------------------------------------
%% vg_pp/1
%% vg_pp( +PP )
%%
%% Description:
%%   Succeeds if PP has semantic class suitable for
%%   attaching to a VG
%% Succeeds: 0-1
%% Side Effects: none
%%-----------------------------------------------------

% when, how, why-type PPs
vg_pp( pp:PP ) :-
        sem( pp:PP, Sem ),
```

```
        member( Sem, [tim, man, rea] ).


%%------------------------------------------------------
%% vg_whq/1
%% vg_whq( +WhWord )
%%
%% Description:
%%   Succeeds if WhWord has semantic class suitable for
%%   attaching to a VG
%% Succeeds: 0-1
%% Side Effects: none
%%-------------------------------------------------------

% when, how, why
vg_whq( WhWord ) :-
        sem( WhWord, Sem ),
        member( Sem, [tim, man, rea] ).
```

preparse.pl

```
%%------------------------------------------------------
%% preparse.pl
%%
%% Contains: predicates for low-level sentence pre-processing
%%------------------------------------------------------

%%------------------------------------------------------
%% pre_process/2
%% pre_process( +RawSentList, ?CookedSentList )
%%
%% Description:
%%   CookedSentList is the result of applying rewrite
%%   rules, stemming and adjusting simple NP bracketing
%% Succeeds: 1*
%% Side Effects: none
%%-------------------------------------------------------

pre_process( RawSent, CookedSent ) :-
        pp_rewrite( RawSent, Sent0 ),
        stem( Sent0, Sent1 ),
        fix_adv_adj( Sent1, Sent2 ),
        fix_ex_there( Sent2, Sent3 ),
        remove_final_punct( Sent3, Sent4 ),
        check_verb( Sent4, CookedSent ).


%%------------------------------------------------------
%% pp_rewrite/2
%% pp_rewrite( +RawSentList, ?CookedSentList )
%%
%% Description:
%%   CookedSentList is the result of applying rewrite
%%   rules to RawSentList
%% Succeeds: 1*
%% Side Effects: none
%%-------------------------------------------------------

% base case
pp_rewrite( [], [] ).

% recurse, optionally applying rewrite rules
pp_rewrite( Sent1, Sent2 ) :-
        rewrite( From, To ),
        append( From, Tail1, Sent1 ),
```

```
        append( To, Tail2, Sent2 ),
%       !,  % let's make rewriting optional
        pp_rewrite( Tail1, Tail2 ).

% recurse, forcibly expanding abbreviations
pp_rewrite( [Head | Tail1], Sent2 ) :-
        abbrev( Head, Expand ),
        append( Expand, Tail2, Sent2 ),
        !,
        pp_rewrite( Tail1, Tail2 ).

% recurse, forcibly expanding abbreviations already (wrongly)
% expanded as possessives
pp_rewrite( [Head, np:[Pos/pos] | Tail1], Sent2 ) :-
        abbrev( [Head, np:[Pos/pos]], Expand ),
        append( Expand, Tail2, Sent2 ),
        !,
        pp_rewrite( Tail1, Tail2 ).

% otherwise recurse, doing nothing
pp_rewrite( [Head | Tail1], [Head | Tail2] ) :-
        pp_rewrite( Tail1, Tail2 ).


%%-----------------------------------------------------
%% remove_empties/2
%% remove_empties( +RawSentList, ?CookedSentList )
%%
%% Description:
%%   CookedSentList is the result of removing empty
%%   phrases (left by pp_remove/2) from RawSentList
%%   *** OBSOLETE ***
%% Succeeds: 1
%% Side Effects: none
%%------------------------------------------------------

% base case
remove_empties( [], [] ).

% recurse, removing any empty phrases
remove_empties( [_Type:[] | Tail1], Tail2 ) :-
        !,
        remove_empties( Tail1, Tail2 ).

% recurse, recursing within sub-phrases
remove_empties( [Type:List1 | Tail1], [Type:List2 | Tail2] ) :-
        !,
        remove_empties( List1, List2 ),
        remove_empties( Tail1, Tail2 ).

% otherwise, recurse doing nothing
remove_empties( [Head | Tail1], [Head | Tail2] ) :-
        remove_empties( Tail1, Tail2 ).


%%-----------------------------------------------------
%% fix_adv_adj/2
%% fix_adv_adj( +RawSentList, ?CookedSentList )
%%
%% Description:
%%   CookedSentList is the result of incorporating
%%   adverbs which are modifying adjectives into the
%%   appropriate NPs (shallowproc leaves them out)
%% Succeeds: 1
%% Side Effects: none
```

```
%%-----------------------------------------------------

% base case
fix_adv_adj( [], [] ).

% recurse, forcibly combining NP Adv NP -> NP as long as
% second NP starts with an adjective
fix_adv_adj( [np:NP, Adv/rb, np:[Adj/Type | NPTail] | Tail1],
             [np:BigNP | Tail2] ) :-
        adj_type( Type ),
        !,
        append( NP, [Adv/rb, Adj/Type | NPTail], BigNP ),
        fix_adv_adj( Tail1, Tail2 ).

% recurse, forcibly combining Adv NP -> NP as long as
% NP starts with an adjective
fix_adv_adj( [Adv/rb, np:[Adj/Type | NPTail] | Tail1],
             [np:[Adv/rb, Adj/Type | NPTail] | Tail2] ) :-
        adj_type( Type ),
        !,
        fix_adv_adj( Tail1, Tail2 ).

% otherwise recurse doing nothing
fix_adv_adj( [Head | Tail1], [Head | Tail2] ) :-
        fix_adv_adj( Tail1, Tail2 ).


%%-----------------------------------------------------
%% fix_ex_there/2
%% fix_ex_there( +RawSentList, ?CookedSentList )
%%
%% Description:
%%   CookedSentList is the result of removing the
%%   existential "there" from NPs (shallowproc puts it in)
%% Succeeds: 1
%% Side Effects: none
%%-----------------------------------------------------

% base case
fix_ex_there( [], [] ).

% remove "there" from its own NP
fix_ex_there( [np:[Ex/ex] | Tail1], [Ex/ex | Tail2] ) :-
        !,
        fix_ex_there( Tail1, Tail2 ).

% remove a head "there" from NP
fix_ex_there( [np:[Ex/ex | NPTail] | Tail1],
              [Ex/ex, np:NPTail | Tail2] ) :-
        !,
        fix_ex_there( Tail1, Tail2 ).

% remove a trailing "there" from NP
fix_ex_there( [np:NPList1 | Tail1], [np:NPList2, Ex/ex | Tail2] ) :-
        append( NPList2, [Ex/ex], NPList1 ),
        !,
        fix_ex_there( Tail1, Tail2 ).

% and otherwise recurse
fix_ex_there( [Head | Tail1], [Head | Tail2] ) :-
        fix_ex_there( Tail1, Tail2 ).


%%-----------------------------------------------------
```

```
%% remove_final_punct/2
%% remove_final_punct( +RawSentList , ?CookedSentList )
%%
%% Description:
%%   CookedSentList is the result of removing a final
%%   full stop, ? or ! from the end of RawSentList: it
%%   doesn't help us with parsing, and it gets in the way
%%   of inverting wh-questions
%% Succeeds: 1
%% Side Effects: none
%%-------------------------------------------------------

remove_final_punct( RawSent , CookedSent ) :-
        append( CookedSent , [_/'.'], RawSent ),
        !.

remove_final_punct( RawSent , RawSent ).


%%------------------------------------------------------
%% check_verb/2
%% check_verb( +RawSentList , ?CookedSentList )
%%
%% Description:
%%   CookedSentList is the result of ensuring that at
%%   least one verb in the sentence has shallowproc-style
%%   subject information *** OBSOLETE ***
%% Succeeds: 1*
%% Side Effects: none
%%-------------------------------------------------------

% if there's a fully detailed verb on the top level, fine
check_verb( Sentence, Sentence ) :-
        member( _/(_/_), Sentence ),
        !.

% if not, promote a non-detailed verb
check_verb( Sentence, Sentence2 ) :-
        member( Verb/Type, Sentence ),
        atom_concat( 'vb', _, Type ),
        !,
        subst_first( Verb/Type, Sentence, Verb/(Type/1), Sentence2 ).

check_verb( Sentence, Sentence ).


%%------------------------------------------------------
%% rewrite/2
%% rewrite( ?FromList, ?ToList )
%%
%% Description:
%%   Optional rewriting rules - both arguments are lists
%%   of tagged words
%% Succeeds: 0-1
%% Side Effects: none
%%-------------------------------------------------------

% multi-word units
rewrite( [such/_, as/_], ['such_as'/prep] ).
rewrite( [for/_, np:[example/_]], ['for_example'/prep] ).
rewrite( [for/_, np:[instance/_]], ['for_instance'/prep] ).
rewrite( [that/_, is/_], ['that_is'/prep] ).

% for nulti-word dets, attach to following NPs
rewrite( [np:[a/_, lot/_], of/_, np:NP], [np:['a_lot_of'/det | NP]] ).
```

```
rewrite( [np:[lots/_], of/_, np:NP], [np:['lots␣of'/det | NP]] ).
rewrite( [np:[a/_, few/_], of/_, np:NP], [np:['a␣few␣of'/det | NP]] ).
rewrite( [np:[few/_], of/_, np:NP], [np:['few␣of'/det | NP]] ).
rewrite( [np:[one/_], of/_, np:NP], [np:['one␣of'/det | NP]] ).
rewrite( [np:[each/_], of/_, np:NP], [np:['each␣of'/det | NP]] ).
rewrite( [np:[some/_], of/_, np:NP], [np:['some␣of'/det | NP]] ).

rewrite( [as/_, well/_, as/_], ['as␣well␣as'/cc] ).
rewrite( [and/_, so/_, on/_], [and/cc, np:['so␣on'/np]] ).
rewrite( [in/_, order/_, to/_], ['in␣order␣to'/to] ).
rewrite( [in/_, np:[order/_], to/_], ['in␣order␣to'/to] ).
rewrite( [so/_, that/_], ['so␣that'/subconj] ).

rewrite( [np:[the/det], like/prep], [np:[the/det, like/nn]] ).

rewrite( [np:[most/jjs], Adv/rb], [most/rbs, Adv/rb] ).
rewrite( [np:[more/jjr], Adv/rb], [more/rbr, Adv/rb] ).

% common mistaggings
%rewrite( [one/cd], [one/pp] ).
%rewrite( [np:[one/cd]], [np:[one/pp]] ).
rewrite( [sure/jj], [sure/rb] ).
rewrite( [np:[sure/jj]], [sure/rb] ).

% remove whq-words from own NPs (so they can't get mixed up
% in anaphora later)
rewrite( [np:[WhWord]], [WhWord] ) :-
       whq( WhWord, _, _ ).


%%-------------------------------------------------------
%% abbrev/2
%% abbrev( ?FromWordOrList, ?ToList )
%%
%% Description:
%%   ToList is a list of tagged words corresponding to the
%%   expansion of FromWordOrList. This can be either a
%%   tagged single-word abbreviation or a tagged two-word
%%   abbreviation (second word "'s")
%% Succeeds: 0-1
%% Side Effects: none
%%-------------------------------------------------------

% "n't" abbreviations (in OTA's GG class)
abbrev( Abbrev/Tag, [Verb/Tag, not/rb] ) :-
        atom_concat( Verb, 'n\'t', Abbrev ),
        word( Abbrev, Details ),
        memberchk( 'gg', Details ),
        word( Verb, _ ).

% and exceptions to this rule
abbrev( 'ain\'t'/Tag, [is/Tag, not/rb] ).
abbrev( 'can\'t'/Tag, [can/Tag, not/rb] ).
abbrev( 'cannot'/Tag, [can/Tag, not/rb] ).
abbrev( 'shan\'t'/Tag, [shall/Tag, not/rb] ).
abbrev( 'won\'t'/Tag, [will/Tag, not/rb] ).

% other abbreviations containing "'" (in OTA's GF class)
abbrev( Abbrev/Tag, [Word/Tag, Verb] ) :-
        atom_concat( Word, End, Abbrev ),
        atom_concat( '\'', _, End ),
        v_abbrev( End, Verb ),
        word( Abbrev, Details ),
        memberchk( 'gf', Details ),
        word( Word, _ ).
```

```
% "'s" abbreviations have been undone by tokenise, then tagged POS
% this actually gets "let's" wrong, but I can't see it being a problem
% when Word outside NP (e.g. where's, here's)
abbrev( [Word/Tag, np:['\'s'/pos]], [Word/Tag, is/vbz] ) :-
        atom_concat( Word, '\'s', Cat ),
        word( Cat, Details ),
        (
          memberchk( 'gf', Details );
          memberchk( 'gh', Details )
        ),
        word( Word, _ ).

% must do the same for versions where it's been inclued in an NP
% e.g he's, it's, there's
abbrev( A, [np:[Word/Tag], is/vbz] ) :-
        (
            A = [np:[Word/Tag], np:['\'s'/pos]];
            A = np:[Word/Tag, '\'s'/pos]
        ),
        atom_concat( Word, '\'s', Cat ),
        word( Cat, Details ),
        (
          memberchk( 'gf', Details );
          memberchk( 'gh', Details )
        ),
        word( Word, _ ).


%%-----------------------------------------------------
%% v_abbrev/2
%% v_abbrev( ?FromAtom, ?ToWord )
%%
%% Description:
%%   ToWord is the tagged full-word expansion of the
%%   common untagged abbreviation for a verb FromAtom
%% Succeeds: 0-1
%% Side Effects: none
%%------------------------------------------------------

% cover all the OTA GF class examples except "'twas" etc.
v_abbrev( '\'ll', will/md ).
v_abbrev( '\'d', would/md ).
v_abbrev( '\'re', are/vbp ).
v_abbrev( '\'ve', have/vb ).
v_abbrev( '\'m', am/vbp ).
```

stem.pl

```
%%-----------------------------------------------------
%% stem.pl
%%
%% Contains: predicates for verb stemming
%%-----------------------------------------------------

%%-----------------------------------------------------
%% stem/2
%% stem( ?Stem, ?Word )
%% stem( ?StemList, ?WordList )
%%
%% Description:
%%   Stem is the (tagged) stem of (tagged) Word. Alternatively
%%   StemList, WordList are the sentence/phrase equivalents
%% Succeeds: 1*
%% Side Effects: none
```

```
%%-----------------------------------------------------

% base case
stem( [], [] ).

% recurse, stemming head
stem( [Head1 | Tail1], [Head2 | Tail2] ) :-
        stem( Head1, Head2 ),
        stem( Tail1, Tail2 ).


% if it's a featured phrase, recurse inside
% (while these don't exist to begin with, we may want to
% call stem/2 backwards to return to surface form later)
stem( Type:Feat:List1, Type:Feat:List2 ) :-
        !,
        stem( List1, List2 ).

% same for unfeatured phrase
stem( Type:List1, Type:List2 ) :-
        !,
        stem( List1, List2 ).

% verb: call verb_stem/3
stem( Verb/(Form/Details), Stem/(Form/Details) ) :-
        !,
        verb_stem( Verb, Form, Stem ).

% verbs not given full _/(_/_) form
stem( Verb/Form, Stem/Form ) :-
        atom_concat( 'vb', _, Form ),
        !,
        verb_stem( Verb, Form, Stem ).

% noun: if plural, call noun_stem/2
stem( Noun/nns, Stem/nns ) :-
        !,
        noun_stem( Noun, Stem ).

% noun: in case a plural's been mistagged, use rules & correct tag
stem( Noun/nn, Stem/nns ) :-
        noun_stem( Noun, Stem ),
        \+ Noun = Stem,
        !.

% adjective: if comparative/superlative, call ad_stem/3
stem( Ad/Form, Stem/Form ) :-
        member( Form, [jjr, jjs, rbr, rbs] ),
        !,
        ad_stem( Ad, Form, Stem ).

% if not a phrase, verb or plural noun, leave it alone
stem( Word, Word ) :-
        \+ is_list( Word ).


%%----------------------------------------------------
%% verb_stem/3
%% verb_stem( ?Word, ?Tag, ?Stem )
%%
%% Description:
%%   Stem is the stem of Word according to PoS tag Tag.
%%   At least 2 of the arguments must be instantiated
%% Succeeds: 1*
%% Side Effects: none
```

```
%%-----------------------------------------------------

% verb: if already the VB form, leave it alone
verb_stem( Verb, vb, Verb ) :-
        !.

% verb: if a modal, leave it alone
verb_stem( Verb, md, Verb ) :-
        !.

% verb: if we can find it in the irregular list
verb_stem( Verb, Form, Stem ) :-
        irreg_verb_stem( Verb, Form, Stem ),
        !.  % if irreg_verb_stem/3 succeeds, don't want later backtracking

% verb: if not, use regular rules
verb_stem( Verb, Form, Stem ) :-
        reg_verb_stem( Verb, Form, Stem ).


%%-----------------------------------------------------
%% irreg_verb_stem/3
%% irreg_verb_stem( ?Word, ?Tag, ?Stem )
%%
%% Description:
%%   Stem is the stem of irregular verb Word according
%%   to PoS tag Tag.
%%   At least 2 of the arguments must be instantiated
%% Succeeds: 1*
%% Side Effects: none
%%-----------------------------------------------------

%  just look up entry: verb(VB,VBP,VBZ,VBG,VBD,VBN).
irreg_verb_stem( Verb, vbp, Stem ) :-
        irreg_verb( Stem, Verb, _, _, _, _ ).

irreg_verb_stem( Verb, vbz, Stem ) :-
        irreg_verb( Stem, _, Verb, _, _, _ ).

irreg_verb_stem( Verb, vbg, Stem ) :-
        irreg_verb( Stem, _, _, Verb, _, _ ).

irreg_verb_stem( Verb, vbd, Stem ) :-
        irreg_verb( Stem, _, _, _, Verb, _ ).

irreg_verb_stem( Verb, vbn, Stem ) :-
        irreg_verb( Stem, _, _, _, _, Verb ).


%%-----------------------------------------------------
%% reg_verb_stem/3
%% reg_verb_stem( ?Word, ?Tag, ?Stem )
%%
%% Description:
%%   Stem is the stem of regular verb Word according to
%%   PoS tag Tag.
%%   At least 2 of the arguments must be instantiated
%% Succeeds: 1*
%% Side Effects: none
%%-----------------------------------------------------

% regular VBP should not need stemming
reg_verb_stem( Verb, vbp, Verb ) :-
        word( Verb, _TypeList ).
```

```
% otherwise check conjugation type and call reg_stem/3
reg_verb_stem( Verb, Form, Stem ) :-
        check_verb_type( Verb, FormType ),
        equiv_form( Form, FormType ),
        reg_stem( Verb, Form, Stem ).


%%-------------------------------------------------------
%% std_stem/3
%% std_stem( ?Word, ?Tag, ?Stem )
%%
%% Description:
%%   Stem is the stem of regular verb Word according to
%%   PoS tag Tag, if the standard rules specified by
%%   ending/2 are followed.
%%   At least 2 of the arguments must be instantiated
%% Succeeds: 0-1
%% Side Effects: none
%%--------------------------------------------------------

% like type 0: just remove ending to give stem
std_stem( Verb, Form, Stem ) :-
        ending( Form, Ending ),
        atom_concat( Stem, Ending, Verb ).


%%-------------------------------------------------------
%% reg_stem/3
%% reg_stem( ?Word, ?Tag, ?Stem )
%%
%% Description:
%%   Stem is the stem of regular verb Word according to
%%   PoS tag Tag, derived by following conjugation type
%%   rules from OTA dictionary.
%%   At least 2 of the arguments must be instantiated
%% Succeeds: 1*
%% Side Effects: none
%%--------------------------------------------------------

% type 1 VBZ: +es rather than +s
reg_stem( Verb, vbz, Stem ) :-
        std_stem( Verb, vbz, RawStem ),
        atom_concat( Stem, 'e', RawStem ),
        check_verb_type( Stem, '1' ).

% type 2: stem-final e removed for all except VBZ
reg_stem( Verb, Form, Stem ) :-
        \+ Form = vbz,
        std_stem( Verb, Form, RawStem ),
        atom_concat( RawStem, 'e', Stem ),
        check_verb_type( Stem, '2' ).

% type 3 VBZ: stem-final y replaced by ie
reg_stem( Verb, vbz, Stem ) :-
        std_stem( Verb, vbz, RawStem ),
        atom_concat( TmpStem, 'ie', RawStem ),
        atom_concat( TmpStem, 'y', Stem ),
        check_verb_type( Stem, '3' ).

% type 3 VBD, VBP, VBN: stem-final y replaced by i
reg_stem( Verb, Form, Stem ) :-
        member( Form, [vbd, vbp, vbn] ),
        std_stem( Verb, Form, RawStem ),
        atom_concat( TmpStem, 'i', RawStem ),
        atom_concat( TmpStem, 'y', Stem ),
```

```
        check_verb_type( Stem, '3' ).

% type 4: stem-final letter doubled for all except VBZ
reg_stem( Verb, Form, Stem ) :-
        \+ Form = vbz,
        std_stem( Verb, Form, RawStem ),
        sub_atom( RawStem, 0, _, 1, Stem ),
        check_verb_type( Stem, '4' ).

% if we haven't matched the categories above, just remove ending
reg_stem( Verb, Form, Stem ) :-
        std_stem( Verb, Form, Stem ),
        check_verb_type( Stem, _Type ).


%%-----------------------------------------------------
%% check_type/3
%% check_type( ?Word, ?CatType, ?ConjType )
%%
%% Description:
%%   CatType is the word category type and ConjType the
%%   conjugation/inflectional type of word Word as defined
%%   in the OTA dictionary
%% Succeeds: 1*
%% Side Effects: none
%%------------------------------------------------------

check_type( Word, CatType, ConjType ) :-
        word( Word, TypeList ),
        member( Type, TypeList ),
        atom_concat( CatType, ConjType, Type ),
        atom_length( CatType, 1 ).


%%-----------------------------------------------------
%% check_verb_type/2
%% check_verb_type( ?Word, ?ConjType )
%%
%% Description:
%%   ConjType is the conjugation type of verb Word
%%   as defined in the OTA dictionary
%% Succeeds: 1*
%% Side Effects: none
%%------------------------------------------------------

check_verb_type( Word, ConjType ) :-
        check_type( Word, CatType, ConjType ),
        verb_cat( CatType ).


%%-----------------------------------------------------
%% check_noun_type/2
%% check_noun_type( ?Word, ?ConjType )
%%
%% Description:
%%   ConjType is the inflection type of noun Word
%%   as defined in the OTA dictionary
%% Succeeds: 1*
%% Side Effects: none
%%------------------------------------------------------

check_noun_type( Word, ConjType ) :-
        check_type( Word, CatType, ConjType ),
        noun_cat( CatType ).
```

```
%%----------------------------------------------------
%% check_ad_type/2
%% check_ad_type( ?Word, ?ConjType )
%%
%% Description:
%%   ConjType is the inflection type of adjective/adverb
%%   Word as defined in the OTA dictionary
%% Succeeds: 1*
%% Side Effects: none
%%-----------------------------------------------------

check_ad_type( Word, ConjType ) :-
        check_type( Word, CatType, ConjType ),
        ad_cat( CatType ).


%%----------------------------------------------------
%% equiv_form/2
%% equiv_form( ?Tag, ?Atom )
%%
%% Description:
%%   Atom is the equivalent OTA code for PoS tag Tag
%% Succeeds: 1*
%% Side Effects: none
%%-----------------------------------------------------

equiv_form( vbz, a ).
equiv_form( vbg, b ).
equiv_form( vbd, c ).
equiv_form( vbn, d ).
%equiv_form( vbp, e ).


%%----------------------------------------------------
%% verb_cat/1
%% verb_cat( ?Atom )
%%
%% Description:
%%   Atom is the OTA code for a verb type
%% Succeeds: 1*
%% Side Effects: none
%%-----------------------------------------------------

verb_cat( g ).
verb_cat( h ).
verb_cat( i ).
verb_cat( j ).


%%----------------------------------------------------
%% noun_cat/1
%% noun_cat( ?Atom )
%%
%% Description:
%%   Atom is the OTA code for a noun type
%% Succeeds: 1*
%% Side Effects: none
%%-----------------------------------------------------

noun_cat( k ).
noun_cat( l ).
noun_cat( m ).
noun_cat( n ).
```

```
%%---------------------------------------------------------
%% ad_cat/1
%% ad_cat( ?Atom )
%%
%% Description:
%%   Atom is the OTA code for a adjective/adverb type
%% Succeeds: 1*
%% Side Effects: none
%%----------------------------------------------------------

ad_cat( o ).
ad_cat( p ).


%%---------------------------------------------------------
%% ending/2
%% ending( ?Tag, ?Atom )
%%
%% Description:
%%   Atom is the regular ending for the PoS tag Tag
%% Succeeds: 1*
%% Side Effects: none
%%----------------------------------------------------------

% verb endings
ending( vbz, 's' ).
ending( vbg, 'ing' ).
ending( vbd, 'ed' ).
ending( vbn, 'ed' ).
ending( vbp, 'ed' ).

% adjective/adverb endings
ending( jjr, 'r' ).
ending( jjs, 'st' ).
ending( rbr, 'r' ).
ending( rbs, 'st' ).


%%---------------------------------------------------------
%% noun_stem/2
%% noun_stem( ?Noun, ?Stem )
%%
%% Description:
%%   Stem is the singular of the plural noun Noun
%%   according to OTA dictionary info
%% Succeeds: 1*
%% Side Effects: none
%%----------------------------------------------------------

% type 6: +s
noun_stem( Noun, Stem ) :-
        atom_concat( Stem, 's', Noun ),
        check_noun_type( Stem, '6' ).

% type 7: +es
noun_stem( Noun, Stem ) :-
        atom_concat( Stem, 'es', Noun ),
        check_noun_type( Stem, '7' ).

% type 8: +y -> +ies - forwards version
noun_stem( Noun, Stem ) :-
        var( Stem ),
        atom_concat( TmpStem, 'ies', Noun ),
        atom_concat( TmpStem, 'y', Stem ),
```

```
        check_noun_type( Stem, '8' ).

% type 8: +y -> +ies - backwards version
noun_stem( Noun, Stem ) :-
        var( Noun ),
        atom_concat( TmpStem, 'y', Stem ),
        atom_concat( TmpStem, 'ies', Noun ),
        check_noun_type( Stem, '8' ).

% type 9: no change
noun_stem( Noun, Noun ) :-
        check_noun_type( Noun, '9' ).

% irregular type I: try various possibilities - forwards
noun_stem( Noun, Stem ) :-
        var( Stem ),
        plur_suffix( Sing, Plur ),
        atom_concat( TmpStem, Plur, Noun ),
        atom_concat( TmpStem, Sing, Stem ),
        check_noun_type( Stem, 'i' ).

% irregular type I: try various possibilities - backwards
noun_stem( Noun, Stem ) :-
        var( Noun ),
        plur_suffix( Sing, Plur ),
        atom_concat( TmpStem, Sing, Stem ),
        atom_concat( TmpStem, Plur, Noun ),
        check_noun_type( Stem, 'i' ).

% otherwise look up in irregular dictionary
noun_stem( Noun, Stem ) :-
        irreg_noun( Stem, Noun ).


%%-----------------------------------------------------
%% plur_suffix/2
%% plur_suffix( ?Sing, ?Plur )
%%
%% Description:
%%   Sing is the singular equivalent ending of the plural
%%   Plur - these are some regular types NOT described
%%   in the OTA dictionary
%% Succeeds: 1*
%% Side Effects: none
%%-----------------------------------------------------

plur_suffix( Sing, Plur ) :-
        nth( N, ['men', 'ves', 'a' , 'i' , 'ae', 'es', 'eaux',
                 'ices', 'ices', 'ves', 'a' , 'i', 'ice' ,
                 'eet', 'eeth', 'children'], Plur ),
        nth( N, ['man', 'f'  , 'um', 'us', 'a' , 'is', 'eau' ,
                 'ix'  , 'ex'  , 'fe' , 'on', 'o', 'ouse',
                 'oot', 'ooth', 'child'], Sing ).


%%-----------------------------------------------------
%% ad_stem/2
%% ad_stem( ?Ad, ?Stem )
%%
%% Description:
%%   Stem is the standard form of the comparative/superlative
%%   adjective/adverb Ad according to OTA dictionary info
%% Succeeds: 1*
%% Side Effects: none
%%-----------------------------------------------------
```

```
% type B: +r, +st
ad_stem( Ad, Form, Stem ) :-
        std_stem( Ad, Form, Stem ),
        check_ad_type( Stem, 'b' ).

% type C: +er, +est - forwards
ad_stem( Ad, Form, Stem ) :-
        var( Stem ),
        std_stem( Ad, Form, TmpStem ),
        atom_concat( Stem, 'e', TmpStem ),
        check_ad_type( Stem, 'c' ).

% type C: +er, +est - backwards
ad_stem( Ad, Form, Stem ) :-
        var( Ad ),
        check_ad_type( Stem, 'c' ),
        atom_concat( Stem, 'e', TmpStem ),
        std_stem( Ad, Form, TmpStem ).

% type D: +y -> +ier, +iest - forwards
ad_stem( Ad, Form, Stem ) :-
        var( Stem ),
        std_stem( Ad, Form, TmpStem1 ),
        atom_concat( TmpStem2, 'ie', TmpStem1 ),
        atom_concat( TmpStem2, 'y', Stem ),
        check_ad_type( Stem, 'd' ).

% type D: +y -> +ier, +iest - backwards
ad_stem( Ad, Form, Stem ) :-
        var( Ad ),
        check_ad_type( Stem, 'd' ),
        atom_concat( TmpStem2, 'y', Stem ),
        atom_concat( TmpStem2, 'ie', TmpStem1 ),
        std_stem( Ad, Form, TmpStem1 ).

% irregular comparatives
ad_stem( Ad, Form, Stem ) :-
        atom_concat( _, 'r', Form ),
        irreg_ad( Stem, Ad, _ ).

% irregular superlatives
ad_stem( Ad, Form, Stem ) :-
        atom_concat( _, 's', Form ),
        irreg_ad( Stem, _, Ad ).
```

io.pl

```
%%-----------------------------------------------------
%% io.pl
%%
%% Contains: predicates for shallow processor interface
%%-----------------------------------------------------

%%-----------------------------------------------------
%% call_tagger/2
%% call_tagger( +String, -Stream )
%%
%% Description:
%%   Calls external tagger and opens stream to read
%%   the result. String can be either a filename containing
%%   a sentence, or the sentence itself
%% Succeeds: 1
%% Side Effects: none
%%------------------------------------------------------
```

```
% calls ./shallowproc and pipes to Stream
call_tagger( String, Stream ) :-
        build_command_string( String, Command ),
        popen( Command, read, Stream ).


%%-----------------------------------------------------
%% build_command_string/2
%% build_command_string( +String, ?CommandString )
%%
%% Description:
%%   Creates CommandString to call shallowproc, piping
%%   String in (if it's a filename) or with String
%%   as a command-line argument (if it's a sentence)
%% Succeeds: 0-1
%% Side Effects: none
%%------------------------------------------------------

% if arg1 is a filename, pipe into shallowproc
build_command_string( FileName, Command ) :-
        file_exists( FileName ),
        !,
        atom_concat( './shallowproc.sh␣<', FileName, Command ).

% otherwise assume it's a sentence
build_command_string( Text, Command ) :-
        atom_concat( './shallowproc.sh␣"', Text, Tmp ),
        atom_concat( Tmp, '"', Command ).


%%-----------------------------------------------------
%% get_sentence/2
%% get_sentence( +FileName, ?SentList )
%%
%% Description:
%%   Opens file FileName and reads in the tagged sentence
%%   into the Prolog list SentList
%% Succeeds: 0-1
%% Side Effects: none
%%------------------------------------------------------

get_sentence( FileName, Sentence ) :-
        open( FileName, 'read', Stream ),
        read_sentence( Stream, Sentence ),
        close( Stream ),
        !.

%%-----------------------------------------------------
%% read_sentence/2
%% read_sentence( +FileName, ?SentList )
%%
%% Description:
%%   Opens file FileName and reads in the tagged sentence
%%   into the Prolog list SentList
%% Succeeds: 0-1
%% Side Effects: none
%%------------------------------------------------------

% base case: stop at end of file
read_sentence( Stream, [] ) :-
        at_end_of_stream( Stream ).

% base case: stop at end of bracketed phrase
read_sentence( Stream, [] ) :-
```

```
        peek_code( Stream, Code ),
        rbracket( Code ),
        get_next_word( Stream, _ ).

% if start of bracketed phrase, get type,
% recurse on phrase and then recurse on rest of sentence
read_sentence( Stream, [Type:List | Tail] ) :-
        peek_code( Stream, Code ),
        lbracket( Code ),
        get_next_word( Stream, Tmp ),
        check_for_type( Tmp, Type ),
        read_sentence( Stream, List ),
        read_sentence( Stream, Tail ).

% if a word, get it, and the tag, and recurse
read_sentence( Stream, [Word/Tag | Tail] ) :-
        get_next_word( Stream, Word ),
        get_next_word( Stream, RawTag ),
        convert_verb_tag( RawTag, Tag ),
        read_sentence( Stream, Tail ).


%%-----------------------------------------------------
%% check_for_type/2
%% check_for_type( +String, ?TypeAtom )
%%
%% Description:
%%   If we're using shallowproc's more advanced features,
%%   find the CNP or NP marker - otherwise assume NP
%% Succeeds: 0-1
%% Side Effects: none
%%------------------------------------------------------

% just '(' -> NP
check_for_type( '(', np ).

% otherwise get type
check_for_type( Tmp, Type ) :-
        atom_concat( '(', Type, Tmp ),
        \+ Type = ''.


%%-----------------------------------------------------
%% convert_verb_tag/2
%% convert_verb_tag( +String, ?CompoundTag )
%%
%% Description:
%%   If we're using shallowproc's more advanced features,
%%   convert e.g. "VBZ*Verb-1-NormalSubj" -> (vbz/1)
%% Succeeds: 1
%% Side Effects: none
%%------------------------------------------------------

% if not a complex verb tag, do nothing
convert_verb_tag( NonVerbTag, NonVerbTag ) :-
        \+ sub_atom( NonVerbTag, _, _, _, * ).

% if it is, pull out the subject pos info
convert_verb_tag( Raw, Tag/Subj ) :-
        name( Raw, RawCodes ),
        asterisk( Star ),
        append( TagCodes, [Star | _], RawCodes ),
        name( Tag, TagCodes ),
        minus( Minus ),
        append( _, [Minus, SubjCode, Minus | _], RawCodes ),
```

```
        name( Subj, [SubjCode] ).


%%------------------------------------------------------
%% get_next_word/2
%% get_next_word( +Stream, ?Word )
%%
%% Description:
%%   Gets the next word Word from Stream
%% Succeeds: 0-1
%% Side Effects: none
%%------------------------------------------------------

get_next_word( Stream, Word ) :-
        get_next_word_code( Stream, Code ),
        name( Word, Code ).


%%------------------------------------------------------
%% get_next_word_code/2
%% get_next_word_code( +Stream, ?WordCodeList )
%%
%% Description:
%%   Gets the next word's ASCII representation
%%   WordCodeList from Stream
%% Succeeds: 0-1
%% Side Effects: none
%%------------------------------------------------------

% base case: stop if space
get_next_word_code( Stream, [] ) :-
        peek_code( Stream, Code ),
        space( Code ),
        get_code( Stream, Code ).

% base case: stop if newline
get_next_word_code( Stream, [] ) :-
        peek_code( Stream, Code ),
        newline( Code ),
        get_code( Stream, Code ).

% base case: stop if CR/LF ("bracket" uses this)
get_next_word_code( Stream, [] ) :-
        peek_code( Stream, Code1 ),
        carriage( Code1 ),
        get_code( Stream, Code1 ),
        peek_code( Stream, Code2 ),
        newline( Code2 ),
        get_code( Stream, Code2 ).

% otherwise get lower-case ASCII char & recurse
get_next_word_code( Stream, [Code | Word] ) :-
        get_lower_code( Stream, Code ),
        get_next_word_code( Stream, Word ).


%%------------------------------------------------------
%% get_lower_code/2
%% get_lower_code( +Stream, ?Code )
%%
%% Description:
%%   Code is the lower-case version of the next char's
%%   ASCII representation from Stream
%% Succeeds: 0-1
%% Side Effects: none
```

```
%%-------------------------------------------------------

get_lower_code( Stream, Code ) :-
        get_code( Stream, RawCode ),
        lower( Code, RawCode ).


%%------------------------------------------------------
%% lower/2
%% lower( +Code, ?LowerCode )
%%
%% Description:
%%   LowerCode is the lower-case version of the ASCII code
%%   Code
%% Succeeds: 0-1
%% Side Effects: none
%%-------------------------------------------------------

% if it's not lower-case alphabetic, do nothing
lower( L, L ) :-
        L > 90;
        L < 65.

% if it is, add 32
lower( Lower, Upper ) :-
        Upper < 91,
        Upper > 64,
        Lower is Upper + 32.

%%------------------------------------------------------
%% List of ASCII codes for useful characters
%%------------------------------------------------------

space( 32 ).
carriage( 13 ).
newline( 10 ).
questmark( 63 ).
exclmark( 33 ).
fullstop( 46 ).
comma( 44 ).
semicolon( 59 ).
colon( 58 ).
lbracket( 40 ).
rbracket( 41 ).
asterisk( 42 ).
minus( 45 ).
```

B.3.5 Miscellaneous

debug.pl

```
%%----------------------------------------------------
%% debug.pl
%%
%% Contains: predicates for toggling debug info display
%%----------------------------------------------------

:- dynamic no_debug/1.

%%----------------------------------------------------
%% debug_on/0
%%
%% Description:
%%   Turns debug display on
%% Succeeds: 1
%% Side Effects: retracts all no_debug/1 clauses
%%-----------------------------------------------------

debug_on :-
        retractall( no_debug( _ ) ).


%%----------------------------------------------------
%% debug_off/0
%%
%% Description:
%%   Turns debug display off
%% Succeeds: 1
%% Side Effects: asserts a general no_debug/1 clause
%%-----------------------------------------------------

debug_off :-
        assert( no_debug( _ ) ).

% turn debugging off as default
:- debug_off.


%%----------------------------------------------------
%% no_debug/1
%% no_debug( ?PredName )
%%
%% Description:
%%   If true, the display predicate display_(PredName)
%%   will have no effect
%% Succeeds: 0-1
%% Side Effects: none
%%-----------------------------------------------------

% no_debug( trace ).
% no_debug( struct ).
% no_debug( answer ).
% no_debug( simple ).
```

display.pl

```
%%----------------------------------------------------
%% display.pl
%%
%% Contains: predicates for displaying results & debug info
%%----------------------------------------------------
```

```
%%-----------------------------------------------------
%% display_simple/2
%% display_simple( +TextAtom, +ThingToDisplay )
%%
%% Description:
%%   Displays simple (no phrase markers/tags) version
%%   of anything, with TextAtom displayed first as info.
%% Succeeds: 1
%% Side Effects: none
%%------------------------------------------------------

% if no_debug set, do nothing
display_simple( _, _ ) :-
        no_debug( simple ),
        !.

% call strip_markers/2 and print result
display_simple( Text, A ) :-
        strip_markers( A, A1 ),
        Indent = 10,
        format( "~n~p:~*|~p", [Text, Indent, A1] ),
        !.

% ensure always succeeds
display_simple( _, _ ) :-
        print( 'display_simple problem' ).


%%-----------------------------------------------------
%% display_result/4
%% display_result( +Message, +Filename, +Ans, +Highlight )
%%
%% Description:
%%   Displays filenames and associated answers
%% Succeeds: 1
%% Side Effects: none
%%------------------------------------------------------

display_result( Message, File, A, H ) :-
        strip_markers( A, A1 ),
        strip_markers( H, H1 ),
        Indent1 = 2,
        Indent2 = 13,
        Indent3 = 25,
        format( "~n~*|~p:~*|~p~*|~p~n~*|~p",
                [Indent1, Message, Indent2, File,
                 Indent3, A1, Indent3, H1] ).

% ensure always succeeds
display_result( _, _, _, _ ) :-
        print( 'display_result problem' ).


%%-----------------------------------------------------
%% display_answer/0
%%
%% Description:
%%   Displays current answer and text for highlighting
%% Succeeds: 1
%% Side Effects: none
%%------------------------------------------------------

% if no_debug set, do nothing
display_answer :-
```

```
        no_debug( answer ),
        !.

% otherwise display
display_answer :-
        current_answer( A ),
        strip_markers( A, A1 ),
        Indent = 2,
        format( "~nAnswer:~n~*|~p~n", [Indent, A1] ),
        current_answer_list( H ),
        strip_markers( H, H1 ),
        format( "~nHighlight:~n~*|~p~n", [Indent, H1] ).

% ensure always succeeds
display_answer :-
        print( 'display_answer problem' ).


%%------------------------------------------------------
%% display_match/3
%% display_match( +A, +B, +RuleNumber )
%%
%% Description:
%%   Displays debug information about one matching rule
%% Succeeds: 1
%% Side Effects: none
%%------------------------------------------------------

% display
display_match( A, B, Rule ) :-
        strip_markers( A, A2 ),
        strip_markers( B, B2 ),
        Indent = 2,
        format( "~n~*|~p matches ~p (by rule ~p)",
                [Indent, A2, B2, Rule] ),
        !.

% ensure always succeeds
display_match( _, _, _ ) :-
        print( 'display_match problem' ).


%%------------------------------------------------------
%% display_trace/2
%% display_trace( +Message, +TraceList )
%%
%% Description:
%%   Displays debug information about a whole list of
%%   matching rules
%% Succeeds: 1
%% Side Effects: none
%%------------------------------------------------------

% if no_debug set, do nothing
display_trace( _, _ ) :-
        no_debug( trace ),
        !.

% otherwise display message and call display_trace/1
display_trace( Text, Trace ) :-
        nl, nl, print( Text ),
        display_trace( Trace ).

% ensure always succeeds
display_trace( _, _ ) :-
```

```
        print( 'display_trace problem' ).


%%-----------------------------------------------------
%% display_trace/1
%% display_trace( +TraceList )
%%
%% Description:
%%   Displays debug information about a whole list of
%%   matching rules
%% Succeeds: 1
%% Side Effects: none
%%------------------------------------------------------

% base case
display_trace( [] ) :-
        !.

% recurse, calling display_match/3
display_trace( [[A, B, Rule] | Tail] ) :-
        display_match( A, B, Rule ),
        display_trace( Tail ),
        !.

% ensure always succeeds
display_trace( _ ) :-
        print( 'display_trace problem' ).


%%-----------------------------------------------------
%% display_struct/1
%% display_struct( +Struct )
%%
%% Description:
%%   Nicely displays a sentence, structure or phrase
%% Succeeds: 1
%% Side Effects: none
%%------------------------------------------------------

% if no_debug set, do nothing
display_struct( _ ) :-
        no_debug( struct ),
        !.

% call display_struct/2 with initial indent of zero
display_struct( S ) :-
        nl,
        display_struct( S, 2 ),
        !.

% ensure always succeeds
display_struct( _ ) :-
        print( 'display_struct problem' ).


%%-----------------------------------------------------
%% display_struct/2
%% display_struct( +Struct, +Indent )
%%
%% Description:
%%   Nicely displays a sentence, structure or phrase
%% Succeeds: 1
%% Side Effects: none
%%------------------------------------------------------
```

```
% structure: remove marker & treat as list
display_struct( s:S, Tab ) :-
        display_struct( S, Tab ).

% base case
display_struct( [], _Tab ).

% recurse, displaying whole structures nicely
display_struct( [s:S | Tail], Tab ) :-
        display_struct( S, Tab ),
        display_struct( Tail, Tab ).

% recurse, printing VGs whole
display_struct( [vg:VG | Tail], Tab ) :-
        ( (Tab = 2) -> print( '* ' ); tab( Tab ) ),
        print( vg:VG ),
        nl,
        display_struct( Tail, Tab ).

% recurse, displaying NGs and PPs nicely
display_struct( [Type:Sem:List|Tail], Tab ) :-
        member( Type, [ng, pp] ),
        ( (Tab = 2) -> print( '* ' ); tab( Tab ) ),
        print( Type:Sem ),
        print( ':' ),
        nl,
        NewTab is Tab + 2,
        display_struct( List, NewTab ),
        display_struct( Tail, Tab ).

% recurse, printing anything else out whole
display_struct( [Head|Tail], Tab ) :-
        ( (Tab = 2) -> print( '* ' ); tab( Tab ) ),
        print( Head ),
        nl,
        display_struct( Tail, Tab ).

% ensure always succeeds
display_struct( _, _ ) :-
        print( 'display_struct problem' ).
```

lists.pl

```
%%-----------------------------------------------------
%% lists.pl
%%
%% Contains: miscellaneous list tools
%%-----------------------------------------------------

%%-----------------------------------------------------
%% strip_markers/2
%% strip_markers( +SentList, ?CleanSentList )
%%
%% Description:
%%   CleanSentList is the result of removing all phrase
%%   markers, semantic features and PoS tags from
%%   SentList
%% Succeeds: 0-1
%% Side Effects: none
%%------------------------------------------------------

% remove type markers from structure or (unfeatured) phrase
% (if featured, we'll recurse here anyway)
strip_markers( _Type:List1, List2 ) :-
        !,
```

```
        strip_markers( List1, List2 ).

% remove tags from words
strip_markers( Word/_Type, Word ):-
        !.

% leave words alone
strip_markers( Word, Word ) :-
        \+ is_list( Word ),
        !.

% base case
strip_markers( [], [] ).

% recurse, on both head and tail
strip_markers( [Head1 | Tail1], [Head2 | Tail2] ) :-
        strip_markers( Head1, Head2 ),
        strip_markers( Tail1, Tail2 ),
        !.

% if all else fails, return as is
strip_markers( A, A ).


%%-------------------------------------------------------
%% strip_punct/2
%% strip_punct( +SentList, ?CleanSentList )
%%
%% Description:
%%   CleanSentList is the result of removing all
%%   punctuation marks from SentList
%% Succeeds: 0-1
%% Side Effects: none
%%--------------------------------------------------------

% base case
strip_punct( [], [] ).

% recurse, removing a punctuation char
strip_punct( [_P1/P2 | Tail1], Tail2 ) :-
        punct( P2 ),
        !,
        strip_punct( Tail1, Tail2 ).

% recurse, checking within phrases (with semantic features)
strip_punct( [Type:Feat:List1 | Tail1], [Type:Feat:List2 | Tail2] ) :-
        !,
        strip_punct( List1, List2 ),
        strip_punct( Tail1, Tail2 ).

% recurse, checking within phrases (without features)
% or structures
strip_punct( [Type:List1 | Tail1], [Type:List2 | Tail2] ) :-
        !,
        strip_punct( List1, List2 ),
        strip_punct( Tail1, Tail2 ).

% recurse, leaving anything else alone
strip_punct( [Head | Tail1], [Head | Tail2] ) :-
        strip_punct( Tail1, Tail2 ).


%%-------------------------------------------------------
%% punct/1
%% punct( ?PunctChar )
```

```
%%
%% Description:
%%   Succeeds if PunctChar is a possible punctuation-
%%   type PoS tag
%% Succeeds: 1*
%% Side Effects: none
%%-----------------------------------------------------

% .!? get . tag
punct( '.' ).

% ;:- get : tag
punct( ':' ).

% , gets , tag
punct( ',' ).


%%-----------------------------------------------------
%% subst_first/4
%% subst_first( ?A, +List1, +B, ?List2 )
%%
%% Description:
%%   List2 is the result of replacing the first instance
%%   of A in List1 with B. Like substitute/4 but only
%%   changes the first instance, and fails if no instance
%%   exists
%% Succeeds: 0-1
%% Side Effects: none
%%-----------------------------------------------------

% subst A with B
subst_first( A, [A | Tail], B, [B | Tail] ).

% otherwise recurse until we find A - no [] case!
subst_first( A, [C | Tail1], B, [C | Tail2] ) :-
        \+ A = C,
        subst_first( A, Tail1, B, Tail2 ).


%%-----------------------------------------------------
%% tree_member/2
%% tree_member( ?A, ?List )
%%
%% Description:
%%   Succeeds if A is a member of List, or a member of
%%   something in the tree of List
%% Succeeds: 1*
%% Side Effects: none
%%-----------------------------------------------------

% true if it's a member of a list
tree_member( A, B ) :-
        member( A, B ).

% true if it's a tree_member of a member of a list
tree_member( A, B ) :-
        member( C, B ),
        tree_member( A, C ).

% true if it's a tree_member of a phrase or structure
tree_member( A, _Type:List ) :-
        tree_member( A, List ).
```

```
%%------------------------------------------------------
%% tree_select/3
%% tree_select( ?A, +List1, ?List2 )
%%
%% Description:
%%   List2 is the result of removing an instance of A
%%   from within the tree of List1
%% Succeeds: 0-1
%% Side Effects: none
%%-------------------------------------------------------

% remove a member of a list
tree_select( X, [X | T], T ).

% remove from a struct or (unfeatured) phrase within a list
tree_select( X, [Type:List1 | T], [Type:List2 | T] ) :-
        tree_select( X, List1, List2 ).

% remove from a (featured) phrase within a list
tree_select( X, [Type:Feat:List1 | T], [Type:Feat:List2 | T] ) :-
        tree_select( X, List1, List2 ).

% or recurse to find another instance
tree_select( X, [H | T1], [H | T2] ) :-
        tree_select( X, T1, T2 ).


%%------------------------------------------------------
%% phrase_select/3
%% phrase_select( ?A, +Phrase1, ?Phrase2 )
%%
%% Description:
%%   Phrase2 is the result of removing an instance of A
%%   from within the tree of Phrase1. If this turns out
%%   to be a phrase with only one member, the outer phrase
%%   envelope is removed
%% Succeeds: 0-1
%% Side Effects: none
%%-------------------------------------------------------

% call tree_select/3 and just return a single element
phrase_select( X, _T1:_F1:List1, T2:L2 ) :-
        tree_select( X, List1, [T2:L2] ).

% otherwise just call tree_select/3 on list
phrase_select( X, Type:Feat:List1, Type:Feat:List2 ) :-
        tree_select( X, List1, List2 ),
        \+ List2 = [_].


%%------------------------------------------------------
%% tree_subst/4
%% tree_subst( ?A, +List1, ?B, ?List2 )
%%
%% Description:
%%   List2 is the result of replacing all instances
%%   of A within the tree of List1 with B. Like
%%   substitute/4 but deals with instances at all tree levels
%% Succeeds: 0-1
%% Side Effects: none
%%-------------------------------------------------------

% look within a structure or (unfeatured) phase
% (if featured, we'll recurse here anyway)
tree_subst( A, Type:ListA, B, Type:ListB ) :-
```

```
        !,
        tree_subst( A, ListA, B, ListB ).

% base case
tree_subst( _A, [], _B, [] ).

% remove A and recurse
tree_subst( A, [A | LA], B, [B | LB] ) :-
        !,
        tree_subst( A, LA, B, LB ).

% recurse within a structure or (unfeatured) phrase and recurse
% (if featured, we'll recurse on the top clause anyway)
tree_subst( A, [Type:ListA | LA], B, [Type:ListB | LB] ) :-
        !,
        tree_subst( A, ListA, B, ListB ),
        tree_subst( A, LA, B, LB ).

% otherwise recurse doing nothing
tree_subst( A, [C | LA], B, [C | LB] ) :-
        tree_subst( A, LA, B, LB ).


%%-----------------------------------------------------
%% flatten_np/2
%% flatten_np( +RawNP, ?CookedNP )
%%
%% Description:
%%   CookedNP is the result of flattening the structure
%%   of RawNP (content of sub-NPs are put on top level -
%%   PPs unaffected). Interface to flatten_nps/2
%% Succeeds: 1
%% Side Effects: none
%%------------------------------------------------------

% remove phrase marker & features, call flatten_nps/2
flatten_np( ng:Feat:List, ng:Feat:FlatList ) :-
        flatten_nps( List, FlatList ).


%%-----------------------------------------------------
%% flatten_nps/2
%% flatten_nps( +RawNPList, ?CookedNPList )
%%
%% Description:
%%   CookedNPList is the result of flattening the structure
%%   of RawNPList (content of sub-NPs are put on top level -
%%   PPs unaffected)
%% Succeeds: 1
%% Side Effects: none
%%------------------------------------------------------

% base case
flatten_nps( [], [] ).

% recurse on both head & tail, then append result
flatten_nps( [Head | Tail1], Result ) :-
        flatten_nps( Head, FlatHead ),
        flatten_nps( Tail1, FlatTail ),
        append( FlatHead, FlatTail, Result ).

% simple NP is already flat - just take list
flatten_nps( np:_:List, List ).

% complex NP needs to be flattened
```

```
flatten_nps( ng:_:List, FlatList ) :-
        flatten_nps( List, FlatList ).

% any word, PP etc gets returned as list (so it can be appended)
flatten_nps( X, [X] ) :-
        \+ is_list( X ),
        \+ np( X ).


%%----------------------------------------------------------
%% end_member/2
%% end_member( ?A, ?List )
%%
%% Description:
%%   Just like member/2, succeeds if A is a member of List.
%%   However, if A is uninstantiated, it finds the members
%%   in the opposite order: RH end first
%% Succeeds: 1*
%% Side Effects: none
%%-----------------------------------------------------------

end_member( A, List ) :-
        reverse( List, RList ),
        member( A, RList ).


%%----------------------------------------------------------
%% all_members/2
%% all_members( ?A, ?List )
%%
%% Description:
%%   Succeeds if A can be instantiated to match all
%%   members of List. A is copied at each recursion level
%%   so does not become instantiated:
%%   all_members( _/x, [a/x, b/x, c/x] ) will succeed
%% Succeeds: 1*
%% Side Effects: none
%%-----------------------------------------------------------

% base case
all_members( _A, [] ).

% recurse, checking head matches (without instantiating)
all_members( A, [B | Tail] ) :-
        copy_term( A, B ),
        all_members( A, Tail ).


%%----------------------------------------------------------
%% nearest/4
%% nearest( +N, ?A, +List, ?Pos )
%%
%% Description:
%%   Pos is the nearest position in List to N which
%%   can be instantiated to A. A becomes instantiated if
%%   not already.
%%   nearest( 3, vg:VG, [vg:[a], x, y, vg:[b]], Pos )
%%     -> Pos = 4, VG = [b]
%% Succeeds: 1*
%% Side Effects: none
%%-----------------------------------------------------------

nearest( N, A, List, Pos ) :-
        findall( M, nth( M, List, A ), PosList ),
        diff( N, PosList, RemList ),
```

```
        min_list( RemList, Min ),
        nth( Tmp, RemList, Min ),
        nth( Tmp, PosList, Pos ),
        nth( Pos, List, A ).  % to instantiate A


%%-----------------------------------------------------
%% diff/3
%% diff( +N, +List1, ?List2 )
%%
%% Description:
%%   List2 is the absolute value of the result of
%%   subtracting N from each member of List1
%% Succeeds: 0-1
%% Side Effects: none
%%------------------------------------------------------

% base case
diff( _X, [], [] ).

% recurse subtracting X
diff( X, [Head | Tail1], [Diff | Tail2] ) :-
        Tmp is Head - X,
        abs( Tmp, Diff ),
        diff( X, Tail1, Tail2 ).


%%-----------------------------------------------------
%% abs/2
%% abs( +N, ?X )
%%
%% Description:
%%   X is the absolute value of N
%% Succeeds: 0-1
%% Side Effects: none
%%------------------------------------------------------

% if X positive, leave alone
abs( X, X ) :-
        X >= 0.

% if negative, return positive equivalent
abs( X, Y ) :-
        X < 0,
        Y is 0 - X.


%%-----------------------------------------------------
%% subst_nth/4
%% subst_nth( +N, ?X, +List1, ?List2 )
%%
%% Description:
%%   List2 is the result of replacing the Nth member of
%%   List1 with X
%% Succeeds: 0-1
%% Side Effects: none
%%------------------------------------------------------

% reverse and call subst_nth_r/4
subst_nth( N, X, In, Out ) :-
        reverse( In, RevIn ),
        subst_nth_r( N, X, RevIn, RevOut ),
        reverse( RevOut, Out ).
```

```
%%-----------------------------------------------------
%% subst_nth_r/4
%% subst_nth_r( +N, ?X, +List1, ?List2 )
%%
%% Description:
%%   List2 is the result of replacing the Nth member
%%   (counting from RH end!) of List1 with X. Does the
%%   work for subst_nth/4
%% Succeeds: 0-1
%% Side Effects: none
%%-----------------------------------------------------

% if not there yet, keep recursing
subst_nth_r( N, X, [Head | Tail1], [Head | Tail2] ) :-
        length( Tail1, T ),
        T >= N,
        subst_nth_r( N, X, Tail1, Tail2 ).

% if tail is N-1 long, make the substitution
subst_nth_r( N, X, [_Head | Tail], [X | Tail] ) :-
        length( Tail, T ),
        T is N - 1.


%%-----------------------------------------------------
%% append/4
%% append( ?List1, ?List2, ?List3, ?List123 )
%%
%% Description:
%%   List123 is the result of appending List1, List2, List3
%%   (like append/3).
%% Succeeds: 0-1
%% Side Effects: none
%%-----------------------------------------------------

% if ABC instantiated, use it first
append( A, B, C, ABC ) :-
        nonvar( ABC ),
        append( Tmp, C, ABC ),
        append( A, B, Tmp ).

% if ABC uninstantiated, use it last
append( A, B, C, ABC ) :-
        var( ABC ),
        append( A, B, Tmp ),
        append( Tmp, C, ABC ).


%%-----------------------------------------------------
%% reverse_append/3
%% reverse_append( -List1, -List2, +List3 )
%%
%% Description:
%%   Just like append/3 but will start with List1 = List3,
%%   List2 = [] and then fill List2, rather than the
%%   other way around
%% Succeeds: 1*
%% Side Effects: none
%%-----------------------------------------------------

reverse_append( A, B, AB ) :-
        var( A ),
        var( B ),
        is_list( AB ),
        reverse( AB, BA ),
```

```
        append( RB, RA, BA ),
        reverse( RB, B ),
        reverse( RA, A ).


%%-------------------------------------------------------
%% check_slash/2
%% check_slash( +String1, ?String2 )
%%
%% Description:
%%   String2 is String1 with a '/' directory separator
%%   appended if it didn't already have one
%% Succeeds: 0-1
%% Side Effects: none
%%--------------------------------------------------------

check_slash( S, S ) :-
        atom_concat( _, '/', S ),
        !.

check_slash( S1, S2 ) :-
        atom_concat( S1, '/', S2 ).
```

prolog.pl

```
%%-------------------------------------------------------
%% prolog.pl
%%
%% Contains: replacements for built-in predicates that
%%           are not available in older SICStus versions
%%-------------------------------------------------------

%%-------------------------------------------------------
%% atom_concat/3
%% atom_concat( ?Atom1, ?Atom2, ?Atom3 )
%%
%% Description:
%%   Atom3 is the result of concatenating Atom1 and Atom2.
%%   Either Atom3 or both Atom1, Atom2 must be instantiated
%% Succeeds: 0-1
%% Side Effects: none
%%--------------------------------------------------------

% if AB uninstantiated
atom_concat( A, B, AB ) :-
        atom( A ),
        atom( B ),
        name( A, ACodes ),
        name( B, BCodes ),
        append( ACodes, BCodes, ABCodes ),
        name( AB, ABCodes ).

% if AB instantiated
atom_concat( A, B, AB ) :-
        atom( AB ),
        name( AB, ABCodes ),
        append( ACodes, BCodes, ABCodes ),
        name( A, ACodes ),
        name( B, BCodes ).


%%-------------------------------------------------------
%% peek_code/2
%% peek_code( +Stream, ?Code )
%%
```

```
%% Description:
%%   Interface for the renamed peek_char/2
%% Succeeds: 0-1
%% Side Effects: none
%%-------------------------------------------------------

peek_code( Stream, Code ) :-
        peek_char( Stream, Code ).


%%-------------------------------------------------------
%% get_code/2
%% get_code( +Stream, ?Code )
%%
%% Description:
%%   Interface for the renamed get0/2
%% Succeeds: 0-1
%% Side Effects: none
%%-------------------------------------------------------

get_code( Stream, Code ) :-
        get0( Stream, Code ).


%%-------------------------------------------------------
%% sub_atom/5
%% sub_atom( +Atom1, ?Before, ?Length, ?After, ?Atom2 )
%%
%% Description:
%%   Atom2 is a sub-atom of Atom1, with length Length.
%%   Before is number of chars before Atom2, After is
%%   number of chars after. Atom1 must be instantiated
%% Succeeds: 0*
%% Side Effects: none
%%-------------------------------------------------------

sub_atom( A1, B, L, A, A2 ) :-
        atom( A1 ),
        name( A1, A1Codes ),
        append( BCodes, A2Codes, ACodes, A1Codes ),
        length( BCodes, B ),
        length( A2Codes, L ),
        length( ACodes, A ),
        name( A2, A2Codes ).


%%-------------------------------------------------------
%% atom_length/2
%% atom_length( +Atom, ?Length )
%%
%% Description:
%%   Length is the length in chars of Atom
%% Succeeds: 0-1
%% Side Effects: none
%%-------------------------------------------------------

atom_length( A, L ) :-
        atom( A ),
        name( A, ACodes ),
        length( ACodes, L ).
```