A Software Architecture for Natural Language Processing
Dave Davies, 2005

Abstract
The implementation of Natural Language Processing (NLP) systems poses some interesting problems for the Software Engineer. In addition to the algorithmic problems associated with parsing and inference in a loosely structured and open-ended language there is the challenge of providing the diverse range of data structures needed to deal with issues of complexity, efficiency usability and reliability. Conversely, for the computational linguist, the implementation details can influence both the scope and style of the practical application of language theory.

We look at several data structures that have been designed to facilitate NLP. Primarily, we describe the Link Matrix (LM), a novel structure designed to provide efficient lexical association. We also describe a set of structures forming a Content Addressable Memory (CAM) based on the use of hashtables. The system also employs a range of iterator structures that facilitate dealing with the complexity and ambiguity inherent in NLP. We then look at a parser and an inference engine built on these structures.

The parser utilises the CAM for grammar pattern templates and storage of results. It uses iterator structures, at both word and sentence levels, to deal with lexical ambiguity. Parser performance issues are discussed and performance figures provided. The inference engine uses the LM to provide indexing and pattern matching in the knowledgebase and CAM structures as a whiteboard for rapid access to partial and complete inference solutions.

Of particular interest in this work is the potential for interdependence between NLP and Automated Speech Recognition (ASR). NLP is an essential component in effective ASR while wide scale adoption of ASR provides the potential for greatly expanding the application domain for NLP. Specifically, in this context, we consider the flexibility of the parser and the performance of the inference engine.

1. Introduction

1.1 Overview
If computer programs can be viewed as consisting of algorithms plus data structures (Winograd 1979) then much of the evolution of Computer Science can be seen as the development of specialised data structures to simplify and improve the efficiency of algorithms. The complexity of NLP provides a particularly demanding domain for software development and the work described here develops new structures and extends conventional structures to suit it.

The software system described is a medium grained (word level) object-oriented design implemented in Java. Finer granularity, taking it to the character level, was not considered necessary since the parser and inference engine operate strictly at a word level. The layered system architecture is based on the model-view-controller design pattern. Here we consider only the model layer.

In Section 2 we look at the data structures that have been developed or adapted to the NLP task. Section 3 describes the parser, its use of the CAM structure and the data structures it provides for syntactic reference in the inference engine. Section 4 describes the inference engine.

1.1 History of the project

The processes and techniques discussed here form part of a Natural Language Agent project. The principal data structure, the Link Matrix, was initially developed for application to abstract symbol matching in a program for playing the board game Go. Its application to NLP (Davies 1983) had the primary objectives of text compression and indexing. The LM formed the basis of The Word Machine which was released commercially as an ideas processor. The Word Machine Inference Engine was not released commercially, primarily due to the restrictions posed by hardware limitations of the time. Contemporary desktop hardware provides ample memory and speed for large scale text processing and inference on large rule sets. While the project is aimed at commercial applications it is hoped that it will also provide a useful tool for natural language research.

While the primary interests of the author are NL rule-based systems and the integration of these with Automated Speech Recognition, the interests of students involved in the project have lead to another, complementary, application area: semantic web search. Since much of the work toward the latter goal has been in the interface or view layer of the system it will not be treated in detail in this report.

1.2 Text indexing

For an inference engine to work efficiently on a large knowledgebase (KB) it must have rapid access to just those statements in the KB that share words with the statements involved in an inference chain. Many techniques for text indexing or file inversion have been discussed in the literature, particularly in relation to searching large text databases or the Web. Reviews of indexing and search techniques can be found in Heinz and Zobel (2003) and de Moura et.al. (2000).

The problem domain addressed in this report differs significantly from those usually targeted in text search engines. Here we are interested in providing an infrastructure for mechanical inferencing in text collections (knowledgebases) of up to a gigaword rather than the terabytes typically indexed in search engines. We want all the text to be in memory, all words indexed, and the sentence structure to be accessible through the index. This problem is addressed in Section 2.1.

1.3 Automated Speech Recognition

The potential for synergy between NLP and ASR systems is a primary motivating force in this work. Grosjean (1978) showed that a hierarchy of pause structures in speech was closely related to the parse tree, the distinction being largely a matter of the speech conforming to isochronal metrical constraints. Use of the word ‘chunk’ in the present work reflects a desire to deal with arbitrary temporal partitioning of an information stream allowing close integration into an ASR system capable of explicit prosodic analysis which may produce

chunks that are not strictly aligned with grammatical, or even lexical, structure. This perspective was pushed into prominence in the field of parser technology by Abney (1991) and from the direction of ASR by Zechner and Waibel (1998).

Davies (2002) outlines a multi-modal approach to ASR that allows runtime reconfiguration of recognition strategies as information on a speech segment is accumulated. To achieve this involves inclusion of a smart controller system. The procedural and declarative rule-based functions of the system described here are potentially capable of performing this function. The performance demands in this operational context are very high but more so for the inference engine than the parser which will usually only need to perform a table lookup for a prior parse.

ASR systems are generally heavily dependent on assistance from language models. These can be statistical models of word N-grams or, as anticipated here, syntactic and semantic models. The ability to process in the order of one thousand chunks per second in the parser is likely to be sufficient. Pronunciation models are also needed and in our case are readily incorporated into a dictionary that is capable of dealing efficiently with multiple meanings and POS. An associated project (ReadRight 2004) is working on multi-dialect pronunciation models for English. The ASR problem is discussed further in Section 3.2.3.

1.4 Design

Best practice in Software Engineering demands the specification of a set of requirements that define the scope of the task. General system requirements for the parser and inference engine are:

1. Size: The system should allow the processing of millions of statements but be capable of restricting algorithmic complexity to just those statements that are relevant to the inference task at hand.
2. Performance: The system should be capable of parsing in the order of one thousand statements per second and performing approximately one hundred inference steps per second.
3. Expansion: The system should allow for the runtime expansion of the language definition and inference policy.
4. Dictionary: The system should be capable of efficiently handling an English lexicon exceeding 100,000 words. It should be capable of efficiently dealing with variations in meaning and Part-of-Speech (POS) and should be runtime extensible.

In first section we outline the constraints placed on the system by these requirements and the principal design solutions: the Link Matrix, Content Addressable Memory and the use of nested iterators.

2. Data Structures

2.1 The Link Matrix

In this section we deal with the problem of text indexing and lexical association. Alternatively we can view the process as the construction of a semantic network. The task is to efficiently associate all statements in a knowledgebase that contain instances of a particular word. In

doing so we can reduce the complexity of inference steps from an order related to the size of the complete KB, to one related to the number of statements that may have relevance due to direct lexical or indirect semantic associations. To clearly distinguish this task from conventional text indexing we derive the structure in a systematic manner starting from a simple tokenised KB representation and progressing in stages to the Link Matrix.

Symbols used in the analysis are:

- W Word object: stores the ASCII form of the word and other information (e.g. POS)
- D Dictionary object: stores word objects and instance counts
- T Token object: pointer to a word in the dictionary
- S Sentence object: an array of words, tokens or links
- L Link object: a pointer to another word instance in sentence
- I Index object: pointer to the first occurrence of word in the KB

We start with a simple four word sentence (S in Figure 1). We then create a dictionary and tokenise the text in the conventional manner replacing each word in the KB with a pointer into the dictionary. By tokenising text, moving from a character based level to the word level, we can increase temporal efficiency however we still have to search the complete data set to investigate the associations between different instances of a word - referred to here as lexical associations.

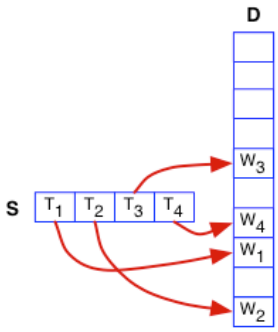

Fig. 1. Tokenised text

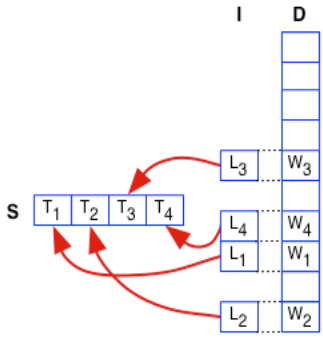

Fig. 2. Indexed Sentence

The simplest approach to lexical association is an index. In a conventional index we reference every instance of a particular word in the KB from an instance array (L3... in Figure 2). An index is effective from the point of view of speed, but we have introduced an inefficiency in memory usage. By having an index entry for each word in the data set we have doubled the memory used for each word instance.

To progress from this point it is helpful to analyse the nature of the information that we are storing. One requirement is that we retain the structure of the sentence. In Figure 2 this is done in the sentence array S. We can absorb this information into the index by replacing links to the sentence array with links to the corresponding items in the index.

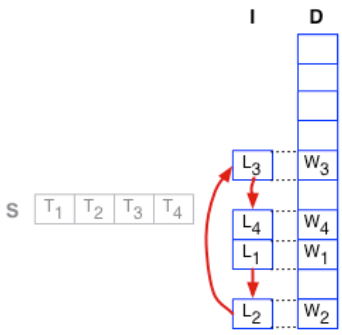

Fig. 3. Linked Index

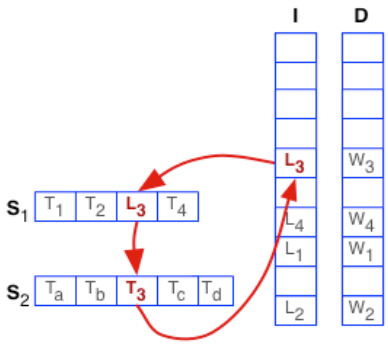

Fig. 4. Link Matrix

This structure is illustrated in Figure 3. The original sentence S is now represented by the link sequence (L1, ... L4) with the ends of the list joined to form a loop allowing access to the complete list starting from any element. The original sentence is now redundant so we have returned to the memory usage of the tokenised sentence but have retained the lexical association information in an efficient form.

Pushing the analysis of the information content of the data structure further we can view the new Linked Index form of the sentence as a two dimensional hyperplane consisting of the index array in one dimension and a linked list in the other. In both of these dimensions there is information embedded in the order of the elements. The order in the linked list is used to represent the word order of the sentence. The order of the index array is unused.

Fig. 4. Link Matrix (instances of only one word are linked)

One way that we might be able to utilise this information is to represent priorities for lexical associations. From a software engineering perspective, however, the structure is not optimal. Arrays are relatively inflexible in expansion and changes in set order whereas linked lists are readily expanded and provide a flexible representation of set order. What we need is an array representing the sentence and word instances represented in a linked list. We need to invert the structure.
Figure 4 shows how this is achieved for one word, W3. The remaining word instances have been left in token form to simplify the diagram. To better demonstrate the structure a second sentence has been included. The index structure has been inverted with the sentences now in arrays and the lexical associations represented in the linked list (arrows). As with the linked index, each instance list is joined at the ends by a token (T3) pointing back to the index to form a loop so the whole list can be accessed from any starting point within it. We have returned to the memory usage of the tokenised text (KB size) plus the size of the index which is now proportional to the dictionary size rather than the KB size.

2.1.2 Optimisation

We have seen that the LM, by using a linked list structure for the lexical associations, provides a means of prioritising the order in which instances are accessed in the list. In practice this can be achieved by placing an instance at the front of the list each time the statement containing it is used successfully in an inference. This has the effect of spreading

the optimisation over all inference tasks. To reduce the impact of this activity on inference performance it can be allocated to a low priority background task.

The lexical associations discussed so far apply to individual words. Accessing the KB for a target pattern consisting of several words can be optimised with the aid of accumulated instance counts for each word. If the word with the lowest instance count is used as the start word of the pattern matching process, the number of sentences tested is minimised.

2.1.3 Embedded structures

So far we have discussed only word associations in the LM. At little additional cost we can also link punctuation which has some interesting consequences. If punctuation is linked, and we control the scope of these links, we can create a variety of structures such as tables and tree structures within documents. Any sentence with components delimited by tabs, commas or semicolons can be viewed as a row in a table. Multiple rows with comparable structure can be associated through the links of these delimiters. The table can be distributed arbitrarily through a document or even across documents. Linked together, such tables can be viewed as records of a distributed object database structure.

The outliner format commonly provided by some word processors provides a tree structured hierarchy for document contents. Multiple consecutive tabs or spaces leading a sentence or paragraph can provide this structure naturally within the LM. The Nth tab, if linked only to other Nth tabs within a predefined scope, defines the Nth sublevel for the tree structure and, used recursively, can form the head of a substructure.

The structure of documents can also be represented using XML. The tokeniser optionally parses HTML and XML tags as a single word. Identical tags are are linked to provide a content type superstructure for document sets. The use of XML is discussed further in the next section.

In a NLP these embedded structures can provide an efficient foundation for implementing structures for semantics and ontologies for grammar rules and the KB generally.

2.2 The CAM and Hash Key Grammars

Hashtables can be viewed as providing a form of associative or content addressable memory implemented in software. Any information that can be represented by a unique integer can be combined in arbitrarily complex ways to form a unique identifier for the combination and used as a key to access information stored in the table.

In practice it is desirable to assert some structure to the formation of keys. It is this structure that we refer to as key grammars. Having formalised patterns for key structures provides a level of generalisation in the use of tables that enables a system to create new data structures at run-time based on novel inputs or discovered patterns. By providing a uniform structure that is used consistently across the system we can provide an answer to the question: '*If some other process has already solved this task and stored information, how was it likely to have been stored?*'.

The simplest form of compound Hash Key used in this work is is a binary form:

key = {<set label> <value label>}

which represents a set with name <set label> containing elements referenced by <value label>.

Structured keys can be viewed as representing a variety of abstract data structures such as sets, trees and arrays in the CAM. A toolkit is provided for efficiently traversing and manipulating these structures. The CAM thus provides a layer of abstraction over the LM.

A simple example of the use of CAM is dealing with lexical ambiguity. Many words exhibit ambiguity in the written form. Here we represent words as a HashSet consisting of a compact HashArray, the elements of which represents alternative meanings for the word (Figure 5). A second dimension to the structure formed using HashLists is used to store the possible Part of Speech alternatives associated with each meaning.

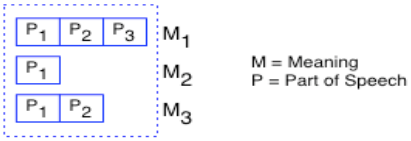

Figure 5. Typical word variant structure for a single word

Using this set structure the Word class is sub-classed as a MultiWord form that uses the HashSet and its Iterators to allow two dimensional scanning of meaning and POS in any context that the word instance appears. The Iterators can be optionally locked to a particular value.

As a consequence of this treatment of lexical ambiguity, sentences containing MultiWords can take multiple forms. A VariantSentence class is provided to allow iteration through the possible sentence forms. An algorithm using grey-scale coding was developed to ensure the uniqueness of sentence representations generated by multiple iterators and to ensure that the most common meanings were presented first for each word in a sentence (assuming that the meanings have previously been prioritised in the dictionary).

A more complex example is provided by the use of XML. Adding elements from the XML tag hierarchy incrementally to the CAM creates a tree structure in the CAM that maps the XML structure. Elements sharing a common partial path are automatically collected in the CAM as a HashSet that can be traversed using its iterator.

To increase the flexibility of access, word sequences used as keys can be added to the LM allowing searches using arbitrary key components, as opposed to the partial paths (the complete sequence from an element to the root) required in CAM lookup. For example, taking the XML path:

```
PubMedExample.xml: PubmedArticle: MedlineCitation [owner=nlm
status=medline]: article [PubModel=print]: journal: JournalIssue:
PubDate: month:
```

reference can be made to data contained under {`status=medline`, `PubDate`} to access specific publication dates. URLs are treated similarly allowing specification of data source such as {`mySite`, `examples`}.

We thus have structures for specifying context using a document's source and its internal structure. The context can be used to construct semantic and ontological constraints and extend the system's internal structures to include public structures available on the Web (W3 Consortium 1999, 2004, 2005).

3. The Parser

This section addresses the problem of natural language parsing with a strong emphasis on performance and reliability oriented Software Engineering and, specifically, table based techniques. We describe a bottom-up, cascading, chunk parser that generates a parse tree by iteratively replacing recognised part-of-speech patterns by a label or alternate pattern. To maximise performance and flexibility the pattern matching algorithm uses data recursion and hashtable techniques as substitutes for stacks and searching respectively.

3.1 The Parsing Process

Prior to reaching the parser, text is tokenised and the possible POS assignments are made. The subsequent parsing process is divided into three stages: the POS transform, mapping and parsing.

The grammar definitions are represented in a conventional manner with a grep-like syntax available for specifying repetitions of sub-patterns. The structure for simple rules is *label: pattern*. For pattern transforms the structure is *pattern2: pattern1*.

The system is seeded with a basic English grammar consisting of 179 rules - 70 of which are simple word level substitutions. The current test grammar does not attempt to provide coverage, rather it contains examples of constructs chosen to test the parser and inference engine. Examples of rules are given in Appendix 1.

3.1.1 Tokenisation

Before the dictionary is accessed the text must be tokenised. This process is more complex for natural language than it is for computer languages. Typical problems are the inclusion of commas and decimal points in numbers and apostrophes in possessive nouns and abbreviations. The treatment of character case also requires more flexibility in natural language than in computer languages. Sometimes it is significant (e.g. 'TOM' as an acronym and 'tom' as the male animal and 'Tom' as a proper noun), in other situations, such as first letter capitalisation at the start of a sentence, it is not. In addition to normal English usage the tokeniser also optionally emits hypertext and XML tags as single tokens.

Fortunately, the highly efficient finite-state tokeniser techniques developed for computer language tokenisers can be adopted and extended for natural language. The use of state-based procedure look-up ensures that the coding and operational complexity increase only moderately with increased functional complexity. Here the tokeniser state is defined by the current word type (word, number etc.) and the current character type (alpha, digit or

punctuation subclass). The action to be performed in each state is stored in a look-up table referenced by state codes. The current implementation has 144 (12x12) states.

3.1.2 The Part of Speech transform

POS information is provided by the system dictionary. The tokenised sentence is scanned and a new representation built using POS tokens to represent most words. The exceptions are words used explicitly in the grammar such as ‘if’ and ‘and’ which are left in the original token form.

3.1.3 Pattern Mapping

A pattern map of recognised word or POS sequences is created at system start-up from grammar definitions. It consists of a hashtable with the keys constructed using the pattern sequences. The data item referenced by the key is the label of the grammar rule.

The HashKey class allows concatenation of a sequence of elements (e.g. words). The structure used to order elements into a key is referred to as the key grammar. For the parser pattern matching operation described here the key syntax:

{ GrammarLevel + <pattern sequence> + KeyLabel }

is used.

The KeyLabels ‘HeadKey’ and ‘PartKey’ are used for the parser patterns. Other labels are used to store the parse results and to access parser rules by label or by level and rule number. Inclusion of the parse level in the key assures that only rules from the desired level are accessed.

Repeat patterns in the rule may be indicated by grep-like use of square brackets. They are expanded as separate rules in the map up to a system specified number of repeats if a ‘+’ follows the closing bracket. Alternatively, the number of possible repeats can be explicitly declared in the rule by placing a digit after the closing bracket. Note that brackets, ‘+’ or integers in the original sentence are represented by special punctuation or numeric tokens so will not appear explicitly in the patterns.

Partial patterns are also entered in the map and reference a token (*PartKey*) that indicates a valid partial match. Starting with a typical rule:

noun_phrase: [article] [adjective]2 noun

and expanding we get the full patterns:

noun_phrase: noun
noun_phrase: article noun
noun_phrase: adjective noun
noun_phrase: article adjective noun
noun_phrase: adjective adjective noun
noun_phrase: article adjective adjective noun

and the partial patterns:

PartKey: article
PartKey: adjective
PartKey: article adjective
PartKey: adjective adjective
PartKey: article adjective adjective.

To simplify rule expression, concatenative rules may be included such as '*adjective: adjective adjective*'.

3.1.4 The parser matching stage

The pattern substitution process that forms the basis of the parser has no inbuilt grammatical knowledge. It simply builds a hierarchy of patterns as laid out in the mapping process. In addition to the POS based patterns described above it could also be used, with the appropriate pattern mappings, for detecting patterns of tense or number in a sequence of words to verify agreement, or patterns of phonemic or prosodic information in an ASR system.

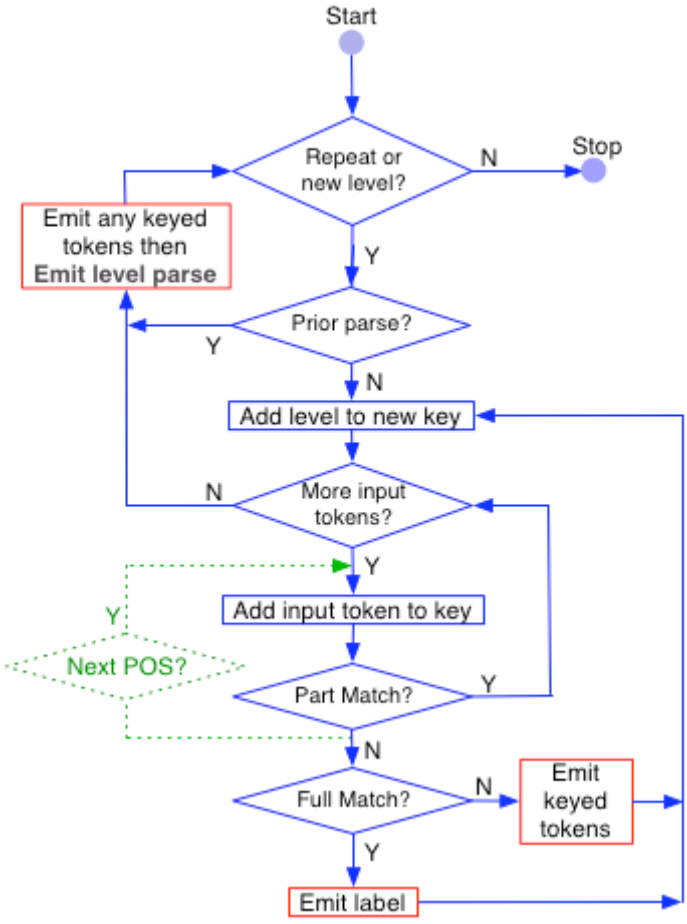

Figure 6: Overview of parser operation. (green/dotted is unimplemented)

The matcher operates left-to-right on an incoming tokenised sentence looking for patterns. Each pass produces a new version of the sentence with recognised patterns replaced by their

labels (rule heads). Clustering the pattern rules into levels allows control of the order in which rules are applied. Within a level no ordering of the rules is enforced which allows flexibility.

Starting at the lowest pattern level we test for patterns until no further substitutions can be made and then proceed to the next level. The output from each pass, in the form of a sentence, is used as the input to the next pass. Parser operation is illustrated in Figure 6.

To detect a pattern, a hash key is constructed starting with a context identifying token, in this case the parse level, then successively adding tokens from the input sentence, testing the key on the pattern map and continuing this sequence as long as the key returns a partial match token. If no partial match is returned by the key a check is made for a full match. If the full match key retrieves a label token then this is emitted to the output stream of this pass. If no pattern is detected then the original tokens are emitted into the output sentence.

Not shown in Figure 1 is an initial check to see if the sentence has been parsed previously and the mechanism for dealing with ambiguous word POS. An outer loop iterates through all POS combinations for the sentence and attempts to parse each in turn. The 'Next POS' loop in Figure 1 illustrates an alternative, and possibly more efficient, approach.

3.1.5 The parser output

The parser always produces a parse tree. A successful parse is indicated by progress to a single label at the highest level. The tree is formed from the data-recursive structure of the label object. While masquerading as a word token, the label also holds a mapping onto the pattern that it represents in the grammar rule that defines it. The tree is thus embedded in a set of sentences - one from each LR pass of the parser.

Efficient guided and unguided tree search methods are provided. A ParseData class has methods for locating a node (label) described by a full path through the tree from the root or, more generally, searching for a specified label starting from a partial path - for example, a search for '*the subject phrase of the condition fact in a conditional statement*' is possible.

In addition to the parse tree the ParseData class stores a reduced form of the sentence parse for rapid context checking in the inference engine. This ParseArray structure has an integer representing each word in the sentence. The parse levels are reduced to four major levels corresponding in the current grammar to phrase, fact, clause and sentence levels. An identifier for the last rule to act on this word at each major level is bit-coded into the integer representing the word. Context checking thus becomes a simple integer comparison - with bit-masking where necessary to generalise the context.

3.1.6 An example

The above description gives an overview of the parser. To see how it operates in detail we will look at some examples of problems that illustrate allocation of tasks to different levels of the system.

The first example, the word *isn't*, involves concatenation and abbreviation. It is fundamentally a character based problem because of the concatenation. The tokeniser creates three tokens

(*isn*, *'*, *t*). This could be handled most efficiently in the tokeniser if it was simply a matter of converting the sequence into '*is not'*. Unfortunately this approach does not generalise (as can be seen from the word *can't* where an *n* has been dropped) and adding complex morphological functionality to the tokeniser is not desirable since it is hard-coded and relatively inflexible.

The problem could also be handled in the dictionary which deals with inflectional morphology. The dictionary, however, works on a single word in, single word out basis and changing this model would have undesirable systemwide ramifications. The most flexible option is to handle the problem in the parser.

The relevant parser rule is: *is not (1 2 to 3): isn ' t.* The numbers in brackets are a template for mapping onto the pattern - '*is*' maps onto the first word in the pattern, '*isn'*. The word '*not*' maps onto words two to three - the apostrophe and '*t*'. The resulting parse can be seen in Example A.

Level	Level parse
0	n isn ' t art participle n .
1	Noun is not article part_word noun stop
1	Noun be_word not_word article part_word noun stop
2	Noun_phrase negative_be_word noun_phrase stop
2	Noun_phrase be_word noun_phrase stop
3	Subject be_word object stop
4	Class_fact stop
5	Fact stop
9	Statement stop
9	Sentence
10	Block

Example A: Tom isn't a sleeping cat.

Also illustrated in Example A is the participle *sleeping*. This problem - using a verb as an adjective - is treated in the dictionary with the POS option either passed in with the original dictionary definitions or, where it follows a regular structure, constructed by applying morphotactic rules to the stem *sleep*. Either way it is passed on within the Word object as an alternate POS and available everywhere downstream. Use of a grammar rule, however, allows us to specify a precise context for its use. We use the rules:

Level	Rule
1	part_word: participle
2	noun_phrase: article part_word noun

for the conversion. The level 1 rule is redundant in this case but exists to maintain consistency across the rule structure.[1] The level 2 rule specifies the context in which the participle is recognised as part of a noun phrase.

In the dictionary all words are assigned a main POS and can also be assigned a secondary sub-POS. In the case of *sleeping* the sub-POS is *participle*. In the parser's POS conversion stage the sub-POS is used for verbs if they have one assigned. They are converted to their main POS form (*verb*) after the parser has tested for the presence of rules that use the sub-POS.

The level 3 mapping to the subject-object form of the sentence uses the rule:

Level	Rule
3	subject be_word object: noun_phrase be_word noun_phrase

This rule does not need a template since it is a direct one-to-one mapping which is assumed as a default. The transform is included as a requirement of the inference engine that needs to distinguish between the subject and object of a statement where they exist.

3.2 Parser evaluation

3.2.1 Performance

Performance measurements reported here are for a 400MHz PowerPC running Java 1.4. We look at performance for sentence lengths between 2 and 12 words. To simplify the analysis only one POS was allowed for words in the test sentences.

Three sets of sentences were used. The first set consisted of grammatically correct, parsable sentences. The second set had the last noun of each sentence replaced by *x* which was assigned a POS that the parser did not recognise (i.e. wasn't in the grammar) so the sentences were only partially parsable. The third set consisted entirely of repetitions of *x* and so were totally unparsable.

An analysis of the parser algorithm would suggest that the unparsable set, with all words retained through all level cascades would have a computational complexity of O(*w*) where *w* is the number of words in the sentence. For the parsable set the number of words processed at each level drops with increasing level. If the number of active rules and the mean pattern size are evenly distributed through the levels the word reduction is linear with respect to the number of levels so complexity is still O(N) but the mean number of words per sentence is approximately *w*/2. We might expect partial parses to sit somewhere in-between.

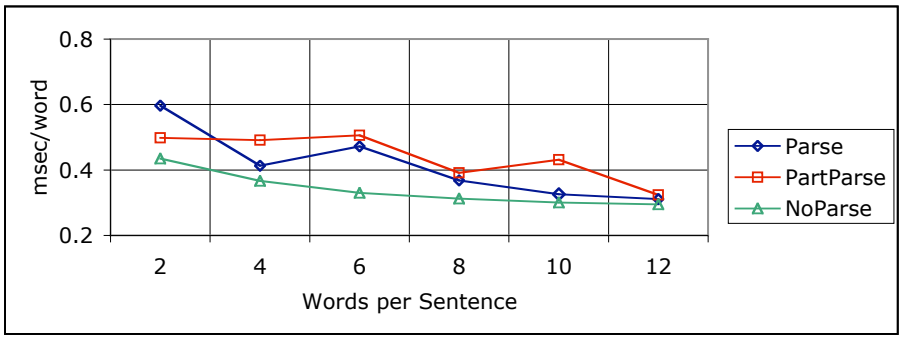

Figure 7: Parser performance as a function of sentence length

Figure 7 shows the results of timing tests. The performance can be seen to approach linear for both the parsable and unparsable sentence sets and larger values of *w*. The slower than expected performance for the partially parsed sentences is likely to arise from a combination of larger average sentence size and an increase in failed pattern match attempts. Word rates are around 2,000 to 3,000 words per second which satisfies the initial performance requirements.

Introduction of multi-POS words reduces performance, in the worst case, by a factor proportional to the combinatorial expansion of POS alternates across the sentence. Prioritisation of POS alternates in the dictionary improves performance as will semantic analysis when it is introduced.

3.2.2 Language

Creating a flexible language implementation requires two main functions. First we need an interface to the grammar that allows users to extend the language and evaluate the impact of changes. This task is work in progress and beyond the scope of this report.

The second need is the ability to rapidly re-evaluate past parses and inference results in the light of grammar changes. It is this task that will place the greatest demands on the parser. While much of the re-evaluation can be achieved in a background processing mode it is necessary to consider the situation where an immediate re-parse of sets of sentences is called for.

A benchmark used in the inference engine for performance evaluation is the invocation of one hundred rules in a single inference. Assuming (rather arbitrarily) a typical rule length of twelve words and a system response time requirement of less than one second gives us a parser performance requirement of 1200 words per second if all rules used in the inference need to be reparsed. This level of performance is achievable in the parser.

3.2.3 Automated Speech Recognition

The main requirement placed on the parser by an ASR system is the ability to move from word level pattern matching to a subword domain. We want to process input streams of phoneme hypotheses and make lexical hypotheses. In this process we introduce high degrees of ambiguity in the phoneme patterns presented to the parser.

The parser described here is able to adapt to this regime with minimal modification. If a dictionary is primed with typical phonemic alternates for each word the iterative structures described above for meaning and POS ambiguity of words are used to handle phonemic ambiguity. The grammar rules can be augmented with hand crafted or statistically generated and weighted phoneme to word or phoneme to syllable transforms.

If we take a moderate word rate of two words per second and an average of ten phonemes per word[2] we need to match around 20 phonemes per second for real-time processing or over 40 p/s if we allow 50% of the available time for the signal processing stages. Processing 2000 symbols per second allows approximately 50 scans per word.

Preliminary ASR results (Davies 2002) suggest that up to four alternates need to be considered for each phoneme. If we assume an average rate of two alternates per phoneme then for a ten phoneme sequence we have a combinatorial total of around 1000 scans per word. If we match at a syllable level, or approximately 5 phonemes per sequence, we get 32 tests per word so we are able to match speech input in approximately real time.

Of course, the nature of combinatorial problems is such that performance will rapidly deteriorate if larger numbers of alternates are involved. At four alternates per phoneme we are up to around 1000 scans per syllable. For difficult words an alternative strategy will be needed. The likelihoods estimated by the ASR system for each phoneme alternate can be used in a stochastic matcher to locate islands of relative certainty, providing a more stable context for identification of neighbouring phonemes.

Because of its left-right operation the parser can provide grammatical support for word hypotheses generated by the ASR system. The parser is also capable of analysing prosodic patterns to add recognition evidence at phoneme the level and disambiguation at the word level.

4. The Inference Engine

This section addresses the problem of Natural Language Inference with a strong emphasis on Software Engineering and, specifically, table based techniques and application of the Link Matrix data structure. We look primarily at the resolution or verification process and issues related to flexibility, efficiency and performance in large Knowledgebases.

4.1 Introduction

There has long been a recognition that computer languages should be developed to describe problems rather than just their solutions (eg. Winograd 1979). Natural Language systems are an obvious candidate for this role. Algorithms for NP parsing and inference have been extensively discussed in the literature and many NL systems have been constructed and reported (Cunningham 1997). Despite this extensive body of work NL systems have still not reached the point where they form a pervasive component of Information and Knowledge Systems. A primary motivating force behind this work is to enhance the usability, flexibility and connectivity of NLP in a way that can enable it to perform such a role.

Consequently, Software Engineering for the Natural Language inference problem is viewed here as having four principal objectives or requirements:

5.Language scope: The language definition, a subset of English in our case, should be broad enough to adequately describe the problem domain in forms that are intuitively accessible to the end user.

6.Flexibility: The parser grammar and production system policy settings should be directed by rule sets that are human readable and editable. They should be capable of runtime update or modification. Code modules should conform to a standard plugin format allowing alternative inference policies or algorithms to be selected to suit a

particular inference task or problem domain and provide connectivity with the underlying operating system and networks.

7.Performance: A general requirement for efficient inference is ‘minimal searching’. Only those statements in the Knowledgebase (KB) that are directly relevant to the current inference task should be accessed. A form of associative memory should be used to optimise access to transient information in the inference process.

8.Stability: Both the Parser and Inference Engine should be robust, coping in an orderly manner with ill-formed statements or queries.

We look at the inference resolution or, more specifically, the unification process and issues related to efficiency and performance in large Knowledgebases. Data and architectural structures have been developed to specifically address key issues of NLP. The algorithms used have been designed, or modified, to work closely with the underlying structures. This process has evolved through several iterative stages with each iteration tightening the partnership between algorithm and structure.

4.2 System Architecture

The current version of this system has been constructed using Java technology. This choice was based on Java’s cross-platform capability but also on considerations such as system and network connectivity, security, Java’s large library infrastructure and the availability of the JavaSpeech API that defines interface conventions for rule-based systems and speech interfaces.

In overview, the system uses the Model-View-Controller pattern to provide a layered architecture. Here we consider only the Model layer. Within the Model layer a sublayer structure is used along established lines (eg. Mizoguchi 1982) with text conversion, parsing, inference and dialogue control layers. A review of NL architecture is provided in (Cummingham 1997)

To provide a flexible runtime ‘plugability’ a general Action class defines common inputs, output and activation for inbuilt Actions such as Parse, Respond (to query), Resolve (inference) and Transform (statement form). This approach allows for the future inclusion of a runtime choice of Parser and other Actions. Java’s Reflection capability allows new Action subclasses to be defined and instantiated at runtime from library classes providing a flexible interface with the GUI, operating system or network functionality provided by the libraries.

The Link Matrix is a structure that links all instances of a given word as a linked list and, in doing so, links all statements in the Knowledgebase that contain that word. All words are linked. The link structure allows the performance of KB access in the inference process to be optimised. Only statements that are directly related to the inference process are accessed.

A Whiteboard structure is provided in the system as an Object Oriented and adorned variant on the conventional Blackboard structures that have been used in many rule-based systems. Some of the adornments provided are:

- structured HashKey objects that facilitate tracing and debugging

· hash key grammar conventions that allow complex keys to be constructed in a systematic manner
· infrastructure for hashtable based arrays, lists and trees
· nested Iterators that are particularly useful in dealing with ambiguity.

The Dialogue Manager

In the dialog manager the inference problem is posed as the verification of a *query_statement* that has been derived through a transform, or sequence of transforms, from an initial *query*. For example the *query* 'Does Tom eat meat?' is transformed to the *query_statement* 'Tom does eat meat.'.

Implementation af the Inference Engine

The Knowledgebase (KB) used in the inference process consists of lexical, logical and semantic rules such as: 'Devours means eats.' or 'A cat is an carnivore' and specific data facts such as 'Tom is a black cat' or 'Tom eats meat'. The inference process we are addressing is the use of known statements about the world to unify, through either logical or semantic equivalences, two specific statements: a statement to be tested and a set of known facts.

The unification process consists of two main phases: Expansion and Test. In the Expansion phase alternative interpretations of each phrase of the '*subject verb_phrase object'* form of the query statement are derived from either equivalence statements (e.g. 'A means B') or class definitions (e.g. 'A is a B') from the KB. The test phase combines alternate phrases to form new forms of the original statement and tests to see if they are present in the KB providing either logical or semantic support for the inference

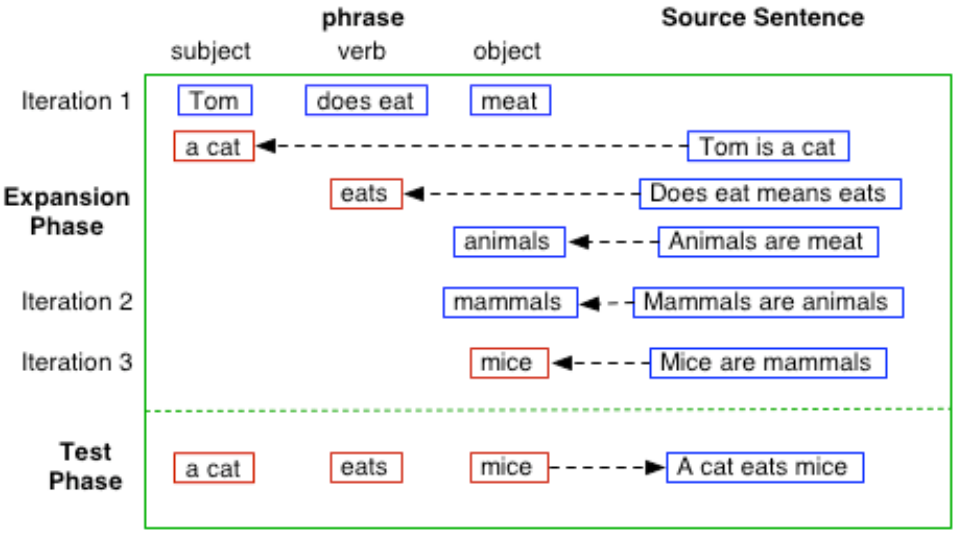

Figure 8: The expansion and test phases for the statement: '*tom does eat meat*'

The Expansion Phase

The search for alternate phrases in the KB is performed using the Link Matrix. It is a performance and space efficient representation of the KB. It consists of three parts, a dictionary, an index and the Knowledgebase. Word information (ASCII, POS etc.) is stored in the dictionary table. For each word in the dictionary a second structure, the index, stores a link to an instance in the KB.

Each sentence in the KB is represented by an array of links - one for each word in the sentence. Each link points to a previous instance of that word which, in turn, links to another instance in a circular list. The dictionary index entry for the word is included in this list. This provides the search with linked lists of the instances of each word in the target pattern. The word lists link all sentences in the Matrix that contain the target words and the set of sentences in each word set is the target set.

To optimise phrase matching performance, the target word with the lowest instance count in the KB is used as the start word in the match. The sentences containing that word are tested in turn for the target phrase. The integer parse array generated by the parser is used to test the suitability of each target word instance to ensure that it is in the correct syntactic position in the correct sentence type. If the context is correct then pattern matching is attempted using a conventional algorithm. The complexity of the search is thus proportional to the product of the smallest instance count in the target and the number of words in the target.

The Test Phase.

In the Test phase the sets of alternate phrases generated in the Expansion phase are combined exhaustively to form candidate solution statements, logically or semantically equivalent representations of the target statement. The KB is then accessed to see if any of these statements are presented as fact. For example, using the phrases shown in Figure 2, a statement such as '**A cat + eats + mice**' can be formed and used to generate a key. This key can then be used to access the statement in the KB if it is present.

When a candidate is found in the KB the inference has been confirmed and the process is either terminated or allowed to continue to find further solutions. The two phases may be repeated iteratively until all solutions have been found. If conditional statements are encountered the *condition fact* is verified recursively. Under the current inference policy the verification is delayed until the *consequent statement* is confirmed as needed in a resolution.

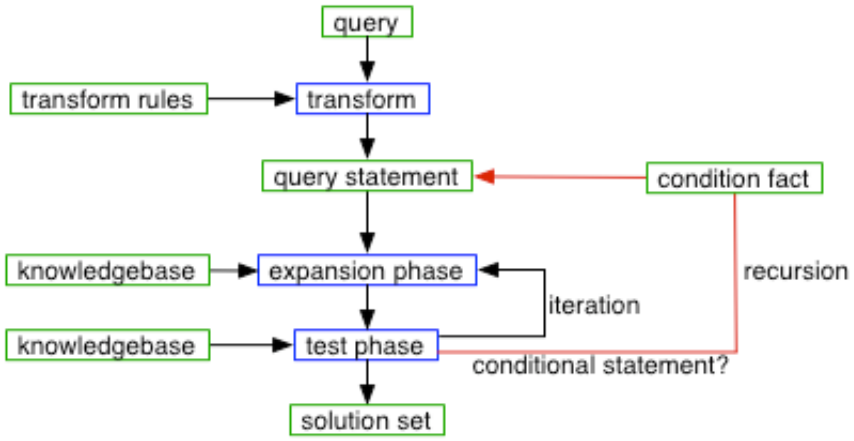

Figure 4: Overview of the inference process

In both phases the whiteboard is used to store intermediate results. In the case of recursion the whiteboard is shared by each recursive step so that the partial results of one recursion can be accessed by any other to avoid duplication or circular referencing.

Output

The full resolution process is outlined in Figure 4. The Test phase returns a solution set to the caller or, alternatively, any object can register as an Observer to receive messages on the progress of the inference process and control it. This provides the potential for terminating runaway inferences or, more generally, runtime control of inference policy.

The solution set contains one item for each solution found. A solution is presented in the form of a linked list of the antecedents for each phrase as shown in Figure 2. This enables tracing and display of the inference process to be performed retrospectively.

Conclusions

Acknowledgements

I would like to take this opportunity to express my appreciation to the many people who have helped bring this project to its current state. It's inception was made possible by the entrepreneurial vision of Bill Fisher and the financial support of the Australian Industrial Research and Development Incentives Board. Its development has been greatly facilitated by the skill and dedication of Arthur Zawadski, Xiaoming Zhang and Darren Hitchman though many others, particularly the graduate students in the University of Canberra IT Project course, have provided both energy and valuable insights.

Notes

[1] Initial experience suggests that some parser efficiency should be sacrificed to provide a consistent and intuitively structured grammar.

[2] Typical word lengths are more like eight phonemes per word but with continuous speech we need to consider overlap between words.

References

9.Abney, Steven 1991. Parsing by Chunks, in Robert Berwick, Stephen Abney, Carol Tenny (eds.), *Principle-Based Parsing*, Klewer Academic Publishers, 1991.

10.Abney, Steven 1995. Partial Parsing via Finite-State Cascades, *Natural Language Engineering* 1 (1), Cambridge University Press

11.Ciravegna, Fabio and Lavelli, Alberto 2000. Grammar Organization for Cascade-Based Parsing in Information Extraction, *Sixth International Workshop on Parsing Technologies*, Trento Italy.

12.Cunningham, Hamish, Humphries, Kevin, Gaizauskas, Robert, Wilkes, Yorick 1997. Software Infrastructure for Natural Language Processing, *Proceedings of the fifth conference on Applied Natural Language Processing*, Morgan-Kaufmann

13.Davies, D.R.L. 1983. A Data Structure for Text Compression, *Australian Microcomputer Software Conference*, ACS, Canberra.

14.Davies, D.R.L. 2002. Representing Time in Automated Speech Recognition, Ph.D. thesis. http://www.blis.canberra.edu.au/hccl/StaffPages/DaveD/index.htm

15.de Moura, Edleno Silva, Navarro, Gonzalo, Ziviani, Nivio, Baeza-Yates, Ricardo 2000. Fast and Flexible Word Searching on Compressed Text, *ACM Transactions on Information Systems*, Vol. 18, No. 2, April 2000, Pages 113–139.
16.Grosjean, F. 1978. Linguistic Structures and Performance Structures: Studies in Pause Distribution, in: Dechert and Raupach, *Temporal Variables in Speech*, Mouton, The Hague.
17.Heinz, S. and Zobel, J. 2003. Efficient single-pass index construction for text databases, *Journal of the American Society for Information Science and Technology*, 54(8):713-729, 2003.
18.Mizoguchi, Fumio and Kondo, Shozo 1982. A Software Environment for Developing a Natural Language Understanding System, *COLING'82*, North-Holland.
19.ReadRight 2004. Models for British, US and Australian English are in preparation in The ReadRight Project, see http://home.netspeed.com.au/dave.davies/ReadRight.html
20.W3 Consortium, 1999. Namespaces on XML, http://www.w3.org/TR/REC-xml-names/
21.W3 Consortium, 2004. Web Ontology Language, http://www.w3.org/2004/OWL/
22.W3 Consortium, 2005. Simple Knowledge Organisation System, http:// www.w3.org/ 2004/02/skos/
23.Winograd, Terry 1979. Beyond Programming Languages, *Communications of the ACM*, 22(7).
24.Zechner, Klaus and Waibel, Alex 1998. Using Chunk Based Partial Parsing of Spontaneous Speech in Unrestricted Domains for Reducing word Error Rate in Speech Recognition, *Proceedings of COLING/ACL 98*, Montreal Canada