A Table Based Parser for Natural Language Processing

David R. L. Davies, 2005

Abstract

This paper addresses the problem of natural language parsing with a strong emphasis on integration with Automated Speech Recognition, 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. The pattern matching algorithm uses data recursion and hashtable techniques as substitutes for stacks and searching to maximise performance and flexibility.

Introduction

The Software Engineering processes and techniques discussed in this paper are part of a Natural Language Agent project. A previous paper [Davies 2005a] described the key elements of the data structure foundations that have been developed to address the demands of large scale, high performance Natural Language Processing.

As a starting point, in line with good Software Engineering design practice, we explicitly address a list of functional requirements derived directly from a Use-Case based description of the system. The principal elements, coming largely from the downstream requirements of the inference engine [Davies 2005c] can be summarised as:

1. Add word definitions to expand the system lexicon.
2. Add labelled part-of-speech (POS) or word patterns to expand the system grammar.
3. Enter facts as elements of a declarative knowledgebase.
4. Enter sequences of commands as labelled elements of a hierarchical command structure to be processed in a procedural manner.
5. Trigger inference action by asserting a query.
6. Trigger a procedural action by asserting a command.
7. Trace an inference chain to verify an inference process.
8. Trace a command sequence to verify a command.
9. A future consideration is the use of speech as an input mode.

The NLP requirements are:

1. Performance: The system should be capable of parsing in excess of one thousand statements per second. This reflects a desire for performance on par with the state of the art [e.g. Abney, 1995].
2. Language: The system should allow for the runtime expansion of the language definition. It should include both declarative and procedural sentence forms.
3. Automated Speech Recognition: The parser should have the flexibility to deal with Chunks in phoneme strings derived in the ASR process with boundaries determined on the basis of the prosodic information available in spoken language. This task also provides a performance benchmark.

The potential for synergy between NLU 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. This perspective was pushed into prominence in the field of parser technology by Abney [1991] and from the direction of ASR in Zechner and Waibel [1998]. Use of the word 'chunk' in the present work reflects a desire for close integration into an ASR system capable of explicit prosodic analysis that may produce chunks that are not strictly aligned with grammatical structure.

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 able to perform this function. The performance demands in this operational context are very high but more so for the inference engine than the parser.

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.

In the following description we look at the staged structure of the parser, techniques developed to deal with ambiguity in the input stream and structures generated to facilitate the inference process. We then review the functionality and performance of the system against the original requirements.

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 (i.e. square brackets) 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 simple 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. The rules are partitioned into a hierarchy of levels (see the parser description below). Currently there are ten levels. Example rules are given in Appendix 1.

Tokenisation and the Part-of-Speech transform

POS information is provided by the system dictionary. 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. 'CAT' as an acronym and 'cat' as the animal), in other situations it is not (e.g. first letter capitalisation at the start of a sentence). In addition to normal English usage the tokeniser also emits hypertext tags as single tokens.

Fortunately, the highly efficient finite-state tokeniser techniques developed for computer languages can be adopted and extended for natural language. The use of state-based procedure look-up ensures that the coding and operational complexity increase minimally 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.

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 patterns such as 'if' and 'and' which are left in the original token form.

Pattern Mapping

A pattern map of recognised word or POS sequences is created at system start-up. 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 addition 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:

{ level + <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 partial match. Starting with a typical rule:

noun_phrase: [article] [adjective]2 noun

we get, on expanding:

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

Although at first sight this may appear excessively memory intensive it is readily accommodated by contemporary hardware. To simplify rule expression, concatenative rules may be included such as '*adjective: adjective adjective'*.

The HashMap class used in the parser is adorned with virtual HashSet structures [Davies 2005a]. The simplest, an array created automatically if the same key is used repeatedly in a store operation allows multiple patterns to be associated with a label. The HashSets share an Iterator class that has the property of masquerading as the currently selected word in the iterator sequence.

This is true for the whole lexicon. The system dictionary uses the same structures to represent lexical ambiguity - either word meaning or multiple POS values[1]. A consequence of this functionality is that in any downstream processing stage a word or parse label can be tested to see if it has ambiguity and the alternatives tested for suitability in the task at hand. Ambiguity is handled largely in the data structures and requires minimal algorithmic overhead in downstream processing.

In the current parser all words in patterns are position sensitive however the HashKey class allows words to be added in a manner (XOR) in which the order is not significant. The value of this to the parser has yet to be explored. To facilitate debugging, a task that is inherently difficult with complex hashtables, the HashKey class has a subclass that stores the sequence of words used to construct it.

Parsing

The pattern substitution process that forms the basis of the parser has no inbuilt grammatical

[1] A possible future extension is phonetic ambiguity passed from an associated ASR system.

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 prosodic information in an ASR system.

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. 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 at that level until no further substitutions can be made and then proceed to higher levels. 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 1.

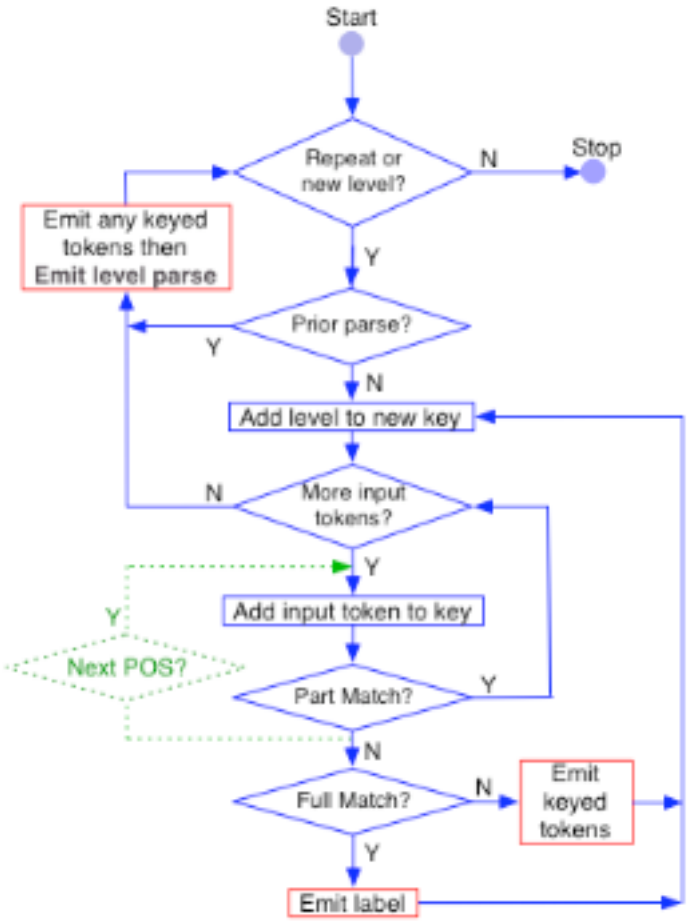

Figure 1: Overview of parser operation. (green/dotted is speculative)

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.

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 embedded in a set of sentences - one from each LR pass of the parser. 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 a parse sentence below it in the parse hierarchy.

The ParseData class has methods for locating a node (label) using 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 the inference engine.

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.

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[2]. 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

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

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.

Evaluation

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.

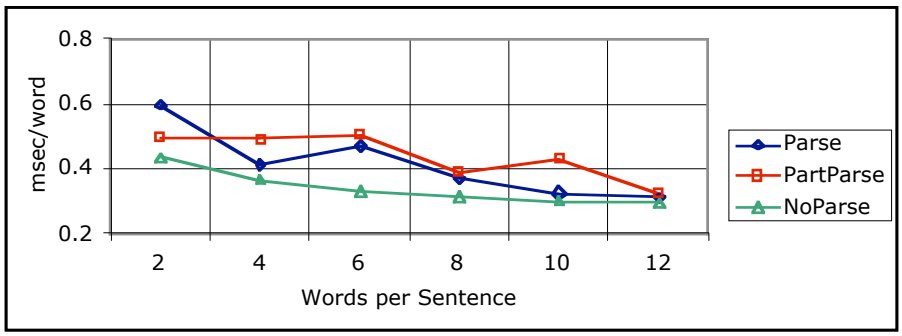

Figure 2: Parser performance as a function of sentence length

Figure 2 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 can improve performance as will semantic analysis when it is introduced to the system.

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 (rather arbitrary) benchmark used in the inference engine for performance evaluation [Davies 2005c] is the invocation of one hundred rules in a single inference. Assuming (again 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.

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 in [Davies 2005a] for meaning and POS ambiguity of words can be 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[3] 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

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

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.

Summary

References

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

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

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

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

Davies, D. R. L. **2005a**, Data Structures for Natural Language Processing, Paper 1 in this series.

Davies, D. R. L. **2005c**, A Table Based Inference Engine for Natural Language Processing, Paper 3 in this series.

Grosjean, F. **1978**, Linguistic Structures and Performance Structures: Studies in
Pause Distribution, in: Dechert and Raupach, Temporal Variables in Speech, Mouton, The Hague (1978).

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

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

Appendix 1: Example grammar rules

Level 1 - word replacements:

be_verb: is

interrogative: which

stop: .

Level 2:

noun_phrase: [article] [adjective]+ noun

Level 3 - a pattern substitution:

subject verb_phrase object: noun_phrase verb_phrase noun_phrase

Level 4 - fact definitions:

class_fact: subject be_verb object

plain_fact: subject verb_phrase object

Level 5 - fact definitions:

if_word condition_fact then_word conditional_fact: if_word fact then_word fact

fact: class_fact

fact: attribute_fact

fact: plain_fact

...

Level 8 - statement definitions:

conditional_statement: if_word condition_fact then_word conditional_fact otherwise fact

Level 9 - sentence definitions:

sentence: statement stop

sentence: command stop

Level 10 - block definitions (for sentence or rule sets):

block: sentence

block: command [semicolon command]+ stop