Langutils: A Natural Language Toolkit for Common Lisp

Ian Eslick
MIT Media Laboratory
20 Ames Street, Rm. 389
Cambridge, MA 02139
+1 617 324-1639
eslick@media.mit.edu

Hugo Liu
MIT Media Laboratory
20 Ames Street, Rm. 320D
Cambridge, MA 02139
+1 617 253-5334
hugo@media.mit.edu

ABSTRACT

In recent years, Natural Language Processing (NLP) has emerged as an important capability in many applications and areas of research. Natural language can be both the domain of application and an important component in the human-computer interface. This paper describes the design and implementation of "langutils," a high-performance natural language toolkit for Common Lisp. We introduce the techniques of real-world NLP and explore tradeoffs in the representation and implementation of tokenization, part-of-speech tagging, and parsing. The paper concludes with a discussion of the use of the toolkit in two natural language applications.

General Terms

Performance, Design, Languages, Human Factors, Algorithms

Keywords

Chunking, Tagging, Tokenization, Natural Language, Parsing

1. INTRODUCTION

Natural Language Processing is becoming an important capability for many modern applications. From e-mail to user interfaces and speech interpretation to text processing, enabling a computer to perform manipulation and interpretation of language can dramatically enhance the usefulness of a program to its user. Major forms of natural language processing in use today include:

- **Dialog or speech systems** use natural language, either written or text, as commands to an application; usually use restricted grammars.
- **Document classification** automatically maps a document into a structured index or ontology.
- **Search and retrieval** indexing of natural language content can be helpful in building richer search interfaces over text documents or web content.
- **Textual analysis** analyzes text for various purposes such as gisting emotional affect, topic spotting, and acquiring user models.
- **Question answering and information retrieval,** still an area of heavy research, use heavy natural language techniques to identify specific kinds of information within larger texts.

This paper describes a Common Lisp toolkit [6] for line- and batch-oriented processing of English language content that can, in part, enable the aforementioned applications. The toolkit was substantially based on the original functionality of the Python-based MontyLingua toolkit [8] targeted specifically for large-scale, high-throughput text analysis.

In Section 2, we motivate the specific algorithms selected for the toolkit and briefly describe their operation. Sections 3 through 5 discuss performance and implementation issues for tokenization, tagging, and chunking, respectively. We also discuss how unique features of LISP simplify the writing and management of these tasks. Section 6 explores the application of the toolkit to three representative applications and Section 7 introduces potential extensions to the system.

2. BACKGROUND AND MOTIVATION

The first stage of processing any unstructured text is to “parse” it into a more structured representation that annotates the key syntactic constituents over which semantic analysis can be performed. The process of parsing text typically consists of *tokenizing* the text into distinct words and punctuation tokens, mapping each token to a corresponding *lexicon* entry, *tagging* the token with an appropriate part of speech given the local context of use, and finally *parsing* or *chunking* the stream of tokens into phrase groups according to part-of-speech type, lexical features (such as tense), and the constraints of the language’s grammar. Each of these major steps is described below. A lexicon is a linguistic dictionary consisting of formal information about the form, use, and parts of speech of specific words. The term ‘lexical’ means “related to elements of a lexicon.” A grammar for natural language, much as for programming languages, describes valid arrangements of words and punctuation and how specific atoms and phrases can be assembled into valid compound forms.

2.1 Tokenization

To allow a part-of-speech tagger to clearly identify the constituent elements of an English sentence, we first need to clearly extract the token sequence that provides the important syntactic building blocks. For example, we need to perform transformations such as:

"…do: first a" => "…do : first a" *Punctuation separation*

"…after. The End." => "…after . The End ." *Period separation*

However, the transformation cannot be implemented as a set of character-local rules, as the following sentence tokenization illustrates:

"Pencil in an appt. for tomorrow at 6:00 o'clock." =>

"Pencil in an appt. for tomorrow at 6:00 o'clock ."

A correct tokenization of the above sentence only separates the period from its adjacent characters. The colon in 6:00 and the period in appt. should not be separated from their surrounding characters.

The process of tokenization in natural language processing remains a bit of a black art. The proper algorithm for tokenization depends greatly on the nature of input you expect: types of errors, internal structure of tokens (such as 6:00), etc. Formal, technical documents often require very different handling than, for instance, text context extracted from a web page or news articles. For most English expressions, a list of common abbreviations along with special sub-forms for time, addresses, and such have to be maintained along with basic language-defined rules.

2.2 Tagging

Once the input has been tokenized, it should consist of linguistically distinct substrings separated by whitespace characters. Each unique substring should now have a specific syntactic role--such as a sentence-ending period or phrase-identifying comma--or it should be a word in the target language's lexicon. The result provides the representation for part-of-speech (POS) tagging.

Many approaches to automated POS tagging exist, and it remains an area of active research. The primary algorithmic approaches that have remained popular in recent years are (1) rule-based and (2) stochastic; both are considered *supervised* algorithms because they require manual engineering, such as manually tagging training corpora and specifying a lexicon. Rule-based approaches such as the Brill tagger [2] encode expertise into rules that make tagging decisions based on *context frames* (i.e. the window of tokens surrounding the current token), and morphological clues (e.g. guess that an unknown word ending in "-s" is a plural noun). Stochastic approaches (cf. [5]) perform tagging based on word-tag frequency (e.g. assign a word the most frequent tag for that word from the training corpus), or based on Bayesian *n-grams* (i.e. a word's best tag is conditioned on a window of *n* words which surround it). More complex stochastic approaches often use Hidden Markov Models to combine information.

The rule-based approach offers arguably greater algorithmic transparency in authoring and debugging, while stochastic taggers tend to behave more opaquely and fail more irrationally. The performance of stochastic taggers also tends to be more closely tied to the genre and nature of the training corpora. Of course, overall evaluated tagger performance is a prime consideration, regardless of methodology. For these reasons, we chose the Brill tagger because as it is the most well known rule-based tagger, variously reported to tag words with an accuracy between 90-96%, depending on the implementation and corpus. Additionally, the Brill tagger is well positioned for NLP interoperability and has established multilingual support; the Brill tagger uses the Penn Treebank tagset [11], a widely used convention in the POS tagging community, and has been trained to tag several other natural languages such as French and Spanish.

The Brill tagger operates in two phases. The first phase uses the lexicon to provide the most likely POS tag for known words. If a word is unknown, default rules are applied along with a set of lexical rules that use suffix, prefix, and other word-local information and a default rule to guess the initial POS tag for all tokens in the document. The default rules are typically that any unknown word is tagged with a 'NN' and a 'NNP' if it is capitalized and not the first word of a sentence. Lexical rules are learned during the supervised training phase and then recorded in a file in the form:

"NN ing fhassuf 3 VBG,"

meaning that if a word is tagged with the default noun tag 'NN' and has a three-character suffix 'ing,' the POS tag should be changed to a verb.

The second pass of the tagger takes the initial tag guesses and transforms the tag of a given word based on the tags surrounding it. These contextual rules are of the form:

"RB JJ NEXTTAG NN"

This line indicates that an adverb (RB) should be changed to an adjective (JJ) if the tag of the next word is a noun (NN).

The result of this process is a probabilistic tagging of the words of the input sentence. The quality of the tagging is a function of how similar the input text is to the original corpus that was used to train the tagging rules.

The reference C implementation of the Brill tagger is available on the web and includes the training program for learning rules. This program is not currently implemented as part of the langutils toolkit; however, the C version can be used to generate files offline that are read by the LISP version.

The training program for the Brill tagger leverages a learning paradigm called transformation-based error-driven learning. This learning process operates by iteratively discovering rules that minimize the error of the automatic tagger vs. a supervised reference set. The training program is initialized with a set of rule templates of which the above lexical and contextual rules are instances (ie a template for the contextual rule would be 'POS1 POS2 NEXTTAG POS3'). The program uses default rules and the lexicon to tag one part of the reference set. Where the reference tags differ from the automatically assigned tags, the program instantiates one or more of the templates, which fixes the error directly. The remaining part of the reference corpus is tagged in independent passes with each newly instantiated rule. The individual rule that decreases the overall tagging error by the largest amount is added to a *rule set*. The process iterates by using the default rules plus the current *rule set* to create the initial state from which new rules are instantiated. During evaluation, the *rule set* is extended with each of the newly instantiated rules and run separately. The best rule is added to the *rule set*. The process terminates when the improvement in the error rate by the best rule in a given stage falls below a pre-determined threshold.

The training program has two stages, one to learn the lexical rules and minimize the per-word tagging error and another to learn the contextual rules to minimize the total tagging error. The two primary reference sets used in the Brill tagger are the hand-tagged Wall Street Journal [11] and Brown corpora [7], but any properly tagged corpus can be used with the training program.

2.3 Parsing and Chunking

Once a high-quality tagging of the tokens is accomplished, a process to parse those words into meaningful groupings is needed before any significant semantic processing can take place. Parsing typically results in the construction of *parse-trees* that represent at the higher levels of the tree syntactic groups, such as sentences, clauses and verb and noun phrases. As you introduce richer subphrases, as well as commas, colons and semi-colons, conjunctions and disjunctions (and/or) and other syntactic words, finding the specific tree arrangement relating one phrase to another becomes quite difficult.

The chunking process is a subset of a complete parser that identifies the basic leaf phrases in a parse tree: the noun, verb, and adverbial and prepositional phrases. They can appear as simple patterns of POS sequences, such as 'DT JJ , JJ NN NN' which identifies a noun phrase such as "the/DT large/JJ ,/, wary/JJ basset/NN hound/NN".

With a chunked form of the sentence available, we can perform lightweight syntactic and semantic analysis without developing a complete parse tree. For example, phrase attachment involves knowledge of specific words in the phrases, which helps to identify high likelihood attachments. We can also train a classifier to identify specific semantic relationships (cause, proceeds, etc) among different chunks.

Identifying full parse trees has exponentially more complexity than identifying the constituent chunks. The popular parsers today, such as Michael Collins' statistical parser [4], train statistical models of phrases and phrase relationships over large corpora. The performance of these systems typically yields a very low throughput system. However, we can trade quality and depth of analysis for speed. Langutils' design allows a developer to trade off performance against the depth and quality of analysis required by the end application.

2.4 Other Important Language Capabilities

There are many other functions a language-oriented library may want to provide to a developer.

Lemmatization. For example, you may want to take all the surface forms of a verb or noun and unify them for purposes of analyzing the semantic content irrespective of time, possession, or plurality. A common linguistic term for this process is lemmatization, which means finding the lemma or root definition of a lexical form. Once you have a lemma, you may also want to generate all valid surface forms to help generate, for instance, all valid sentences that express a given semantic relation between two lemmas in a query expansion (ie "John ran in a marathon," "John runs in a marathon," "John running in a marathon," "John can run in a marathon," "John will run in a marathon," etc).

Spelling correction. When analyzing open text, it is often important to automatically correct the most common spelling mistakes to avoid assuming a misspelled word is in fact an unknown noun or verb.

Stopwords. The analysis of certain large-scale corpora often requires that you focus on the content words such as major verbs or nouns, and ignore words that play a more syntactic role such as 'with' or 'and.' This is called stopword removal.

Semantic analysis. Finally, there are syntactic-semantic analysis tools useful for extracting some basic information from chunked language such as phrase attachment and anaphor resolution (references of he, she, it, that, etc).

3. REPRESENTATION

The intended application of the langutils toolkit was analysis of large bodies of text; therefore, minimization of the performance footprint became a significant constraint on its design. Being sensitive to performance requires thinking first about algorithm and data structure choices and the impact of those choices on cache and CPU behavior. After those decisions are settled, we can focus in on optimizing for local code generation.

In LISP, we aren't forced by default to always consider and make low-level performance choices, as is typical in C++. Because of this, additional knowledge is required to move from writing programs quickly to writing efficient programs. In one sense, optimizing LISP programs is harder than in more primitive languages because the surface syntax does not immediately inform us as to the storage and computation costs at the machine level. Optimization of LISP programs requires new syntactic constructs such as inline type and optimization declarations, as well as consideration of the uses of first order procedures, consing and numbers. Properly cared for, however, LISP programs can perform as well as any popular "high efficiency" language and above all, you only have to expend this effort in the higher level design and in a few low level building blocks – the rest of the program benefits from LISP rapid development model. In the toolkit we highlight on three primary classes of optimization: cache-sensitive data representations, algorithms that optimize for locality, and enabling efficient compilation.

An immediate implementation choice facing a natural language library is choosing how to represent the input strings. You can maintain the original text and add annotations as part of the strings (e.g. tagging 'run' yields 'run/VBD') or choose a less directly readable but more efficient machine representation. In MontyLingua the decision was made to leverage Python's built-in regular expression library and perform all processing directly in the text. This made it easy to describe regular expressions over text such that the code clearly reflected the intended semantics.

In the case of Langutils, throughput was our top priority. We chose to map each unique natural language component to a unique integer token that serves as an index to various table-based resources such as the lexicon, stopword lists, and spelling corrections. This conversion takes place after tokenization and the lexical rule application during tagging, at which point all the original string content is converted into arrays of token integers. We use a CLOS class, the **vector-document** to maintain additional information about the source text, such as origin and the list of POS tags generated by the tagger.

4. TOKENIZATION

In this section *token* is a generic term referring to contiguous sets of non-whitespace characters. Tokenization converts an input token stream to an output token stream separating punctuation from input tokens. As described in section 2, we only want to separate punctuation if it plays a direct syntactic role. Many punctuation characters are part of specific tokens, as in numbers (2.00), times (2:00) and dates (12/24/05).

We observed in Section 2.1 that the process of identifying where to place new whitespace characters requires a contextual window of characters around the present position to be considered. This window is not of fixed size, prohibiting the use of a lookup table. In general we need to have available the information for all the ways in which punctuation-containing tokens might get parsed so that we can separate sentence syntax markers from token syntax markers.

The simplest approach to efficiently solving this problem is to build a simple automaton that directly recognizes all the valid cases of tokens within (i.e. floating point number vs. abbreviation vs. end of sentence). Rather than build the infrastructure for efficient parsing from scratch, we chose to follow the techniques described in Henry Baker's classic paper on META [1] that builds on a parsing technique described in [12].

4.1 META

META is a macro package and design methodology in which it is easy to describe near optimal parsing for a wide variety of languages. Moreover, like the Attributed Grammars that most undergraduate CS majors learn, META expressions can perform computations while they are matching sub-expressions in the input.

META expressions are based on a reader syntax and macro package for building conjunctions and disjunctions, type assertions and in-line procedure calls (acting as terminal or non-terminal productions). The **with-string-meta** macro uses variable capture in the top level macro to provide standard labels (*index, string*) describing the state of the parser so we can easily create local state markers in procedures to implement arbitrary look-ahead and backtracking for sub-expressions. A **meta-match** macrolet is defined by the top-level macro and matches META expressions against the current state, returning t or nil based on whether that expression matched or not. By putting various **meta-match** calls into labels statements and recording the starting value of index, we can easily have our expressions broken up and reused. This improves readability but also enables the parsing of a large class of context-free grammars.

META provides reader syntax for generating compact **meta-match** expressions. In these expressions, [e1 e2 …] means AND, {e1 e2 …} means OR, !(expr) means evaluate the expression 'expr' and @(test var) means to do a type test on var. There is an example of the use of this syntax in section 4.3 below.

We extended the META program by adding a built-in procedure name called **meta-dict** such that !(**meta-dict** name) will determine if the current string matches any string in a list of strings referred to by 'name.' This construction enables us to easily ignore known abbreviations when tokenizing periods.

4.2 Tokenizer Design

The tokenizer proceeds in two stages:

1) Scanning the original string input and extracting all punctuation that is adjacent to other characters by copying the source data into an output buffer and inserting spaces to isolate the character.
2) At the end of each contiguous set of characters, search for errors in the parse (part of a number, abbreviation, etc) and undo the separation when necessary.

Thus the parser has a set of internal functions for scanning, copying, and manipulating the source and destination strings at the 'token' level and a second set of internal functions for testing for specific tokenization sub-cases. The first set of functions takes all punctuation characters that might need to be tokenized, and optimistically separates them. The set of sub-cases recognizes when this should not have happened and removes the whitespace, effectively undoing the first stage's action.

We chose this approach to optimize for the common case, making a trade-off between the cost of cleaning up incorrect tokenization and that of testing every token against all possible parses. Instead, we test for punctuation-including tokens only after we see a token with a punctuation character and undo the tokenization only when it would have been accepted by a regular expression for that particular token type.

The langutils tokenizer supports contractions, possessives, numbers, dates, times, standard abbreviations, common abbreviation, and capitalized proper names. It also detects end of sentence conditions and inserts a newlines after sentences so input can be processed one sentence at a time by reading out lines from the resulting string. The tokenizer interface is intended for use in both batch/string and streaming mode, so you can process chunks from a large file or a continuous network stream. The **tokenize-stream** returns a value list consisting of success-p, final-index, tokenized-string and remainder-string. The **tokenize-string** interface calls **tokenize-stream** on a stream version of the input string.

4.3 Tokenizer META Expressions

The recognizer for a valid case of a time, date, or simple numbers is implemented as follows.

```
(fix-numerics (&aux (old-index index) d)
  (or (meta-match [@(digit d)
                   {#\Space}
                   {#\: #\, #\/}
                   {#\Space}
                   @(digit d)
                   !(delete-spaces (- index 4) 2)
                  ])
      (progn (setq index old-index) nil)))
```

The original string "2:30" would be "2 : 30" after the scanner was applied. So that the token ':' would not be inserted into the token stream, we want to recombine the three tokens into a single token "2:30." The above function tests for this specific failure case and if the test fails, it **setq**'s the current index (in the outer let statement) with the index from when the function was called (*old-index*). The expression beginning with **meta-match** uses the META syntax to test the string buffer at the current *index* for any sequence of: a digit, a space, a colon, comma, or slash followed by another space and any digit. This captures all strings of the form "$2 . 00", "3 : 45" and "12 / 24". The final expression between the square brackets starting with '!' is an imperative that is only run if all the prior expressions match as true. This procedure deletes two spaces between the current index and the current index minus 4.

4.4 Performance Optimization and Results

We have designed the parser to minimize the number of comparisons and the amount of backtracking that has to be done over input tokens. Now, to ensure that the compiler can produce optimal code, we must insert the proper optimization and type declarations for our data structures and performance-critical functions. To ensure that functions can be open-coded by the compiler, their input arguments must have their type properly declared.

Our input array is a string, so we use the **char** function to access characters at specific offsets. **char** is a special case of **aref**. We write processed characters into a simple-array of type base-char using **aref**. Both of these procedures can be open coded as load instructions if the source variables are strings and simple-arrays and the indexes are known to be proper fixnums.

To avoid allocating memory dynamically, we create a single static array for all operations accessible to the parser closure. At the end of a parse session, we transform the array back into a string, requiring a second visit to the array. The array allocation is increased to support the length of any input string. For moderately sized inputs the scratch array is likely to already to be in the L2 cache minimizing the initial cache misses.

The performance data in the rest of this document are based on the Allegro 7.0 LISP compiler and runtime on a 1.67Ghz G4 PowerBook with a 167Mhz front-side bus. Running in a tight loop on a test document, the tokenizer can process approximately 30,000 words, or nearly 200kB, per second using an 8-kilobyte test document.

5. POS TAGGING

The Brill tagger, as previously described, proceeds in two stages:

1) Create an initial tag using the lexicon, default rules and lexical rules

2) Change initial tags based on rules matching the adjacent words and tags

POS tags are represented in memory as LISP symbols. We chose this representation because it is easier to write and debug. We avoided this representation with words because several hundred thousand symbols in the langutils package have a significant storage impact.

5.1 Initial Tagging

The first step of POS tagging scans through the tokenized text string and uses characteristics of the text to guess the first initial POS tag. The algorithm looks up the token id for the word in an EQUAL hash table indexed by the token string. The ID is used to find a lexicon entry. If it exists, the most likely POS tag is pulled from the lexicon and used as the guess for the tag.

If the lexicon entry is empty, the algorithm then defaults to a noun (NN) or proper noun (NNP) depending on whether the word is capitalized. We then apply the set of lexical rules that were learned during training to see if the prefix, suffix, or other lexical characteristic of the unknown word indicates with high probability a particular part of speech.

Lexical rules are represented to the tagger as text lines in a file that parameterizes specific comparison templates as discussed in section 2. For example, "NN ble fhassuf 3 JJ" means that if the last three letters of an unknown word labeled as an NN is 'ble', then it's probably an adjective (JJ). The template referenced by "fhassuf" is created by the function **make-lexical-rule,** which returns closures implementing a specific instance of the template.

The closure template is defined as follows:

```
((string= fname "fhassuf")
 (let ((old-tag (mkkeysym (first list)))
       (suffix (second list))
       (count (read-from-string (fourth list)))
       (new-tag (mkkeysym (fifth list))))
```

```
#'(lambda (pair)
     (let ((token (car pair))
           (tag (cdr pair)))
        (if (and (eq tag old-tag)
                 (strncmp-end2
                   suffix
                   token
                   count
                   (- (length token) count)))
            (progn (setf (cdr pair) new-tag)
                   pair)
            pair)))))
```

The input line is split into substrings, each of which is converted by the let bindings based on the form of the matching closure template. The comparison performed by the inner call to strncmp is a simple EQ between elements of the strings treated as an array.

After the token ID has been looked up and the initial tag has been guessed, the main loop writes the token ID and tag symbol into two temporary arrays used by the second tagging stage.

The code above does not include the declare statements necessary to allow a compiler to open code the primitive calls, but in the actual implementation available online, all types and aggressive optimization switches are included to enable the compiler to open-code primitives and generate as compact code as possible within these closures. The call to **strncmp** is declared inline, which would allow for good additional optimization, but the Allegro compiler ignores the inline declaration except for built-in primitives. This makes sense for development, but is non-ideal for this particular application. (A macro version of **strncmp** would accomplish this objective.)

5.2 Contextual Rule Application

Representing text as a linear sequence of token IDs has tremendous performance benefits over storage as raw strings. The simplification of the contextual matching rules alone will yield a significant payoff in runtime performance. This also significantly compacts the total storage requirements for a given amount of text, implying fewer runtime cache misses as well as fewer load, store and compare operations.

Contextual rules are represented similarly to the lexical rules described above. A **make-contextual-rule** function returns compiled closures. Closure variables capture values that characterize an instance of the rule template defined in the enclosed lambda statement. A contextual rule is represented on disk as:

"RB JJ NEXTTAG NN"

This rule implies that a word tagged as an adverb that precedes a noun should instead be an adjective.

Here we leverage the power of the LISP macro system by building in a level of abstraction over the use of individual case statements above.

5.2.1 *A Simple Template Language*

The full rule template has numerous declarations, may operate over several different possible arrays (tag or word array), and matches against different offsets into those arrays. However, this can be abstracted to a simple set of **labels** and **aref** indices. Thus, we can capture the essential parameters of the complex case, let, and lambda expression as:

```
("NEXTTAG" (match (0 oldtag) (+1 tag1)) => newtag)
```

This rule is labeled "NEXTTAG," and the semantics of the rule indicate that if the current pointer in the document has a POS tag of the type *oldtag* then if the next word's POS tag matches *tag1*, then we should change the current POS tag to *newtag*. The three labels, *oldtag*, *tag1* and *newtag* have a specific meaning to the macro as defined by the following partial table:

```
(list 'tag1 'tags '(mkkeysym (fourth pattern)))
(list 'tag2 'tags '(mkkeysym (fifth pattern)))
(list 'word2 'tokens '(id-for-token (fifth
                                     pattern)))
(list 'oldtag 'tokens '(mkkeysym (first pattern)))
(list 'newtag 'tokens '(mkkeysym (second
                                    pattern))))
```

This table tells the template construction macro to create a **let** binding as specified above for each name referenced in the original rule template and into which array the offset should be computed. The template creation macro function then generates an individual case statement, including optimization declarations, as illustrated below:

```
((string= name "NEXTTAG")
 (let ((tag1 (mkkeysym (fourth pattern))))
   #'(lambda (tokens tags idx)
       (declare (ignore tokens)
           (type (simple-vector symbol) tags)
           (type fixnum idx)
           (optimize speed (safety 0)))
       (if (and (eq (svref tags idx) oldtag)
                (eq (svref tags (+ idx 1)) tag1))
           (progn
              (write-log …)
              (setf (svref tags idx) newtag))))))
```

The *oldtag* and *newtag* variables are bound external to the individual case statements as they are at the same location in the rule expression for all rules. The tokens, tags and current index are passed to the rule through the lambda arguments.

5.2.2 *Performance*

The expression generated by the macro above will be compiled directly into a closure containing code for two comparisons with constants that, upon success, will side effect the new tag directly into the POS tag array. This can be done in a handful of instructions and we pay for our rule abstraction the overhead of a function call. It is then highly efficient to apply a set of rule closures, one at a time, to each location in the document arrays. The rules should all become cached in the instruction cache (barring task switching effects) and the closure frames should mostly be cached in the data cache (there may be some minor interference with array date). We achieve full locality on each fetch of array data into the data cache.

A tokenized string can be fully processed into a vector document at the rate of 15,000 words per second for small (50 word) strings. For our 8-kbyte test document, we end up with a vector document of 1670 tokens IDs. Running the tagger in a tight loop over the tokenized source string, we achieve a throughput of 21,000 tokens per second, or 12 moderately sized documents.

6. PHRASE CHUNKING

The final major capability provided in the langutils toolkit is phrase chunking. Once a tagged **vector-document** object has been created, the chunking process can be very lightweight. Chunking becomes looking for a valid linear sequences of tag types. The total set of valid linear sequences matching a particular phrase type can be expressed as simple regular expression:

```
(defconstant verb-pattern
  '(and (* or RB RBR RBS WRB)
        (? or MD)
        (* or RB RBR RBS WRB)
        (or VB VBD VBG VBP VBZ)
        (* or VB VBD VBG VBN VBP VBZ RB RBR RBS)
        (? or RP)
        (? and (* or RB) (or VB VBN) (? or RP))))
```

In this simple regular expression, '*' indicates to match zero-or-more of the following expression, '?' for zero or one and '+' for one-or-more. The absence of a modifier means to match exactly one instance of the expression. The and/or labels at the head of an expression indicate whether all or any of the following constants should match for the full expression to match.

It is easy enough to write specific code to support a given regular expression, or to use the META infrastructure to create the parser by hand. However, we do not need the full generality of META based parsers, and instead directly and efficiently support this syntax with a simple macro system that compiles these expressions into custom parsers that operate directly on the text and tag arrays.

6.1 Regular Expression Compiler for Arrays

The regular expression syntax and semantics provided above are intended to operate on arrays of data. The result of applying the expression to an array is a pair of offsets into the array indicating where the matched pattern begins and ends. A generic vector-match macro abstraction was developed for this purpose and used in support of the phrase-chunking implementation. This package provides a procedure, **match-array,** that recursively generates the direct **and**, **or,** and **if** expressions implementing the regular expression pattern. The generated code keeps track of the starting offset and finishing offset of the match and can be called incrementally at each incremental offset into an array to search for any match of the regular expression it implements.

A number of utility macros were constructed around **match-array** such as **compile-pattern, find-all-patterns, vector-match1** and **do-collect-vector-matches.** The function **compile-pattern** calls the LISP compiler on an interpreted lambda to generate a compiled closure that takes as input an array and offset and returns the start and end indices upon finding a match. **do-collect-vector-matches** is a context-creating macro that takes as its arguments two labels, a pattern expression, an array object, and a body expression. The body is executed whenever there is a match of the pattern against the vector; the labels are set to the beginning and ending indices of the matched region.

The implementation of a specific phrase chunker requires only that we call **do-collect-vector-matches** over the appropriate array within the **vector-document** we are analyzing. The body creates a **phrase** object that stores the beginning and ending index of a particular match as well as the source document it references. Here is the chunker that finds phrases described by the verb-pattern above:

```
(defmethod get-verb-chunks ((doc vector-document))
  (do-collect-vector-matches (s e verb-pattern)
                                ((document-tags doc))
      (make-instance 'phrase
                     :type :verb
                     :document doc
                     :start s
                     :end e)))
```

6.2 Chunking Performance

There are strong locality benefits to these compiled regular expressions. The constants and matching code are resident in the instruction cache and the data flows linearly through the data cache. A streaming pre-fetching CPU should successfully hide all of the latency to main memory for this application, leaving the pipeline performance as the limiting attribute.

Using another tight-loop test where we extract verb phrases from our reference document, we can achieve a throughput of 250k tokens per second, or dozens of news stories per second. Tokenization and tagging, because of the data conversion and copying involved, are currently the bottlenecks for langutils processing throughput. Because of this, we can easily perform several extraction passes for different phrase types over small documents with minimal overall performance impact.

7. APPLICATION EXAMPLES

There are a number of good applications for the langutils library, a few of which are illustrated in the following section.

7.1 Command and Control

Command and Control is a paradigm for a speech or written natural language interface which allows a user to control a high-level programmatic API by speaking or typing simple sentences of the imperative form (e.g. "Show me all the coats for winter") or declarative form (e.g. "The program should convert PDF into HTML"). Within these two simple constructions, it is relatively straightforward to map subject-verb-object roles onto a langutils chunked representation, and then subsequently to translate the syntactic frame to a high level programmatic API command. For example,

```
Show me all the coats for winter ➔
(VX Show VX) (NX me NX) (NX all the coats NX) (PX
for winter PX) ➔
verb: Show; obj1: me; obj2: all the coats; obj3:
for winter ➔
show_webpage(user='user',
   sql_command='select all coats from table
   where season=winter')
```

To achieve end-to-end command and control, a speech recognition package interprets user utterances into text, and langutils maps this text into subject-verb-object roles that could then map more cleanly into a particular application's API. Using this approach, for example, an e-commerce website could allow users to navigate using natural language utterances.

In a variation on the command and control idea, the *Metafor* code visualization system [10], uses MontyLingua to map an English narrative written in declarative/imperative form into programmatic form, i.e. object.function(argument). It would be straightforward to build this functionality on top of langutils.

7.2 Extracting Common Sense

Knowledge or expertise is often more easily elicited through unrestricted natural language utterances, especially when engaging end-user authors who are casual non-programmers. However, to make computation facile, knowledge needs to be expressed more concisely and uniformly than free English sentences. Here, langutils/MontyLingua has an opportunity to bridge a gap, allowing knowledge to be authored as unrestricted natural language, and automatically extracting from the free text a more concise Subject-Verb-Object* representation, with words morphologically lemmatized and auxiliary words stripped.

The Open Mind Common Sense (OMCS) knowledge base [13] is a corpus of 800,000 English sentences imparting social, causal, and physical common sense knowledge about the everyday world. Some of the sentences are pre-structured as they were gathered through a website as a fill-in-the-blanks activity (e.g. "The effect of ____ is ____"). However, users routinely fill these blanks with complex utterances. Using langutils/MontyLingua, the OMCS sentences were parsed into an ontology of syntactically regularized forms such as Noun Phrase (e.g. "sandwich"), Adjective Phrase ("red"), Verb ("eat"), and Verb-Argument ("eat :: sandwich :: in restaurant") compounds. These regularized forms are taken as a computation-friendly knowledge representation and used to constitute the 300,000 nodes of ConceptNet [9], the common sense reasoning engine generated from the OMCS corpus.

In addition to extracting machine-computable common sense from the OMCS corpus, langutils/MontyLingua is also an integral part of ConceptNet's high-level document processing API, enabling it to perform topic spotting, textual affect sensing, summarization, and other tasks by parsing any input text into the same ontology of syntactically regularized forms so that ConceptNet can 'recognize' concepts in free text.

7.3 Large-scale Web Mining

Numerous efforts today use large-scale statistics over the web to mine for both linguistic and topic-specific content. In our commonsense reasoning project, we were investigating the relationship between concepts on a specific topic. Specifically, we were mining the web to discover neighborhoods of related 'event' concepts in the form of Verb-Object and Verb-Argument phrases such as "won a marathon" and "fell onto the ground."

The application used the public Google API to find pages related to a target query concept such as "ran a marathon." The top 100 pages returned by Google were downloaded, stripped, tokenized, tagged, and chunked. The extracted chunks from all the pages were combined and ranked against each other according to several ranking heuristics, yielded a ranked list of matching concepts. Initial evaluations indicate that the system could achieve 30% high quality conceptually related phrases with 70% noise. Ongoing work on ranking and extraction seeks to boost the signal to noise ratio of the ranking system.

The aggregate throughput of the system in a sustained mining context is 200 web pages per minute on a Dual PowerPC G5 with an average page size of 5kBytes of html-stripped text. If we factor out the time spent accessing the Google API and downloading the HTML pages, the system achieves a throughput of nearly 400 pages per minute.

8. ENHANCEMENTS AND EXTENSIONS

There are a number of important enhancements and extensions to the current toolkit implementation we have considered.

Token representation. The current token representation is flawed, as all tokens are assigned an integer value, whether a number, date or word. The code space of valid tokens in a given natural language lexicon is bounded to hundreds of thousands, but the code space of numeric or other structured tokens is unbounded. An escape coding mechanism in the vector-document abstraction could be used to indicate special token types that could then be looked up separately in a table, perhaps in an appropriate pre-processed form based on their specific type.

Tokenization. There is an opportunity to improve performance by writing a compiler for our current implementation of **meta-dict** by factoring out common prefixes of the dictionary strings and ordering the prefix matching to minimize the total number of characters tested against a random language string. There are a number of ways to handle the tokenization, and we are not convinced that the approach we chose is optimal within the overall META approach.

Improved tagger support. We intend to add the Brill training program to the toolkit in the future. We may also investigate the ability to tune a rule set based on the discovery of near misses by a semantic interpretation layer (i.e. an unknown token is a person, etc). This will be important for applications that want to parse text containing poor grammar, such as personal e-mail, or for expert domains for which there are no hand-labeled corpora.

Multilingual support. The parsing approach we chose allows for relatively gentle scaling to multiple languages. Tagging scales directly as the Brill tagger has been trained on a few other languages, and only regular expression chunking and linking rules need to be written for other languages. Syntactic parsing into Subject-Verb-Object* form is easier in some languages; for example, Korean, Latin, and Japanese employ morphology or particles to explicitly mark a noun phrase as a subject or object, direct or indirect.

The potentially difficult step in adding a new language would be updating the tokenizer. For some symbol-based languages, this is straightforward, but for many languages based on the Arabic alphabet, an alternate tokenizer for each language or a more general, customizable tokenizer would be needed.

9. SUMMARY

We have presented the motivation, features, and optimizations used in implementing a natural language processing library in LISP. Several of the unique features of LISP were presented as an illustration of how we can use

LISP's mechanisms for higher-order syntactic abstractions without losing the capacity for competitive run-time performance.

We further presented applications of natural language processing to several domains and we hope that the availability of this library will encourage the use of natural language within the LISP community.

Online references for the full source of langutils and a release of the Python-based MontyLingua toolkit can be found in [6] and [8].

10. REFERENCES

[1] Henry Baker. 1991. Unpublished manuscript. http://home.pipeline.com/~hbaker1/Prag-Parse.html

[2] Eric Brill. 1990. "A simple rule-based part of speech tagger", In Proceedings of the third conference on Applied Natural Language Processing. Association for Computational Linguistics, Trento, Italy. pp152-155.

[3] Eric Brill, David Magerman, Mitchell Marcus. 1990. Deducing linguistic structure from the statistics of large corpora. In *DARPA Speech and Natural Language Workshop*. Morgan Kaufmann, Hidden Valley, Pennsylvania, June.

[4] Michael Collins. Head-Driven Statistical Models for Natural Language Parsing. PhD Dissertation, University of Pennsylvania, 1999.

[5] Evangelos Dermatas, & George Kokkinakis, 1995. Automatic stochastic tagging of natural language texts. *Computational Linguistics 21.2*: 137-163.

[6] Ian Eslick. Web based resource. http://www.media.mit.edu/~eslick/langutils/

[7] W. Francis and H. Kucera. Frequency Analysis of English Usage. Houghton Mifflin, Boston, 1982.

[8] Hugo Liu. Web based resource. http://www.media.mit.edu/~hugo/montylingua/

[9] Hugo Liu and Push Singh (2004). "ConceptNet: a practical commonsense reasoning toolkit". *BT Technology Journal*, 22(4):211-226

[10] Hugo Liu and Henry Lieberman (2005) Programmatic Semantics for Natural Language Interfaces. Proceedings of the ACM Conference on Human Factors in Computing Systems, CHI 2005, April 5-7, 2005, Portland, OR, USA, to appear. ACM Press.

[11] M. Marcus, Beatrice Santorini and M.A. Marcinkiewicz: Building a large annotated corpus of English: The Penn Treebank. In *Computational Linguistics*, volume 19, number 2, pp313-330.

[12] Schorre, D.V. "META II: A Syntax-Oriented Compiler Writing Language". Proceedings of the 19th National Conference of the ACM (Aug. 1964), D1.3-1 - D1.3-11.

[13] Singh, P., Lin, T., Mueller, E. T., Lim, G., Perkins, T., & Zhu, W. L. (2002). Open Mind Common Sense: Knowledge acquisition from the general public. Proceedings of the First International Conference on Ontologies, Databases, and Applications of Semantics for Large Scale Information Systems. Lecture Notes in Computer Science (Volume 2519). Heidelberg: Springer-Verlag.