

Chapter 3: Parsing

Objectives:

- Understand why token recognition is crucial to effectiveness and efficiency of information retrieval systems
- Understand popular stemming rule-based stemming algorithms: Porter, Lovins,
- Understand language-independent stemmers
- Learn the basics of using JavaCC to build a parser

1 Overview of Parsing

Parsing is one of the most overlooked parts of most information retrieval systems. Many systems describe their proprietary techniques designed to find the “perfect” documents and focus entirely on how they find relevant documents. Parsing refers to the process of identifying tokens in a stream of text. For a string “the big dog jumped up the hill” we can agree that the tokens are “the”, “big”, “dog”, “jumped”, “up”, and “hill”. In languages like Chinese this isn’t so easy since words are not separated by spaces, but for this book, we’ll stick to parsing English. Now, the problem becomes harder when we try to parse something like “The F-15 flew beneath the 3G69 Radar of the U.S. Army.”

The document parser is used to accept text prior to indexing and the query parser is used to identify tokens prior to implementing the query. Since parsing runs on *every* term encountered, any special processing must be done extremely quickly as it will have a significant effect on run –time performance.

1.1 Handling Special Characters

What do we do with tokens like “F-15”? Do we turn them into “F” and “15” or do we try to put them together as “F-15”. If we do choose to parse this as “F-15” then what rules shall we use for hyphens? Should all hyphenated terms be treated as a single term? Suppose we read in the paper “and then he took the oh-so-long-awaited-jump”, do we really want to treat the hyphenated token as one long token.

The reason parsing is so crucial is that it drives query accuracy. If hyphenated terms like “F-15” are treated as a single token, we will be able to respond to a query that says, “find all documents about the F-15”. If we split this term into two tokens “F” and “15”, we are creating two terms. Now the query will indeed retrieve documents about the “F-15” as desired, however, it will also retrieve queries about the letter “F” and any students who received a grade of “15” on a test.

For each special character, a decision must be made what to do with that character. It is tempting to simply say, “let’s remove all punctuation marks from a string and lowercase all tokens”. This might be appealing, but now a token representing the United States (U.S.) is transformed into the term “us” and will match documents with any occurrence of “us”.

Given all this, very complex rules can be developed for processing tokens. Rules that will recognize tokens like “\$999,999.00” and parse them correctly. The problem is that we are not parsing a few words in a document; we are parsing literally *billions* of terms. Parsing time can have a dramatic impact on the entire indexing process. The time taken to index documents directly impacts the delay between when a document is created to when it can be retrieved. For a static collection, this is not a big issue, but for a

dynamic collection (e.g., such as the web or a large intranet), lengthy indexing times are simply unacceptable.

1.2 **Phrase Processing**

Phrase processing is the topic of chapter xx, but common wisdom states that the identification of phrases at parse time helps the accuracy of an information retrieval system. A query on information about “bicycle riders in new york” is interested in documents that contain “new york” as a phrase. A typical information retrieval system, however, treats the query as a bag of words and matches documents with terms in any particular order. This query might locate a document such as “*York* Peppermint patties are a *new* food that is often eaten by *riders* who absolutely hate to ride a *bicycle*”. Identifying that the phrase “new york” occurs in this query can be done in many different ways. The simplest solution is to just take overlapping two term pairs as they occur in the query. In this case we identify “bicycle riders”, “riders in”, “in new”, and “new york”. We do, indeed, identify “new york” and treat it as a phrase, but we also find many phrases that are never used as phrases. The argument goes that this is not a problem, as they do not result in retrieval of spurious documents. The problem is, however, that an inverted index grows dramatically when arbitrary phrases are used. Using this approach and filtering phrases that occur at least 25 times resulted in a set of over seven million phrases for a ten gigabyte test collection used by our research laboratory. Better phrase identification dramatically reduces the size of the inverted index with, if anything, an increase in effectiveness of retrieval. The problem again is that elaborate phrase identification can significantly slow down parsing, and prolong the entire indexing

process. Hence, solutions such as overlapping phrases are often used (usually with the added catch that phrases do not cross punctuation marks or very frequently used words).

1.3 Stop Words

George Zipf used to count words occurring in text. He found that most of text is composed of the same few words. In many studies, the term “a” occurs in **seven percent** of text. It is really an astonishing result – the fact that every seven out of one hundred words we use is “the.” After that, words like “a, of, to, in, at, etc.” are very commonly used. Francis and Kucera did one of the first studies of word frequency using computers on a collection of documents called the *Brown Corpus*. This collection contained many different types of documents designed to measure their word frequencies. Repeatedly, it was determined that a few words cover most occurrence in English. In fact, the frequency distribution of text follows a curve called the Zipfian distribution **[REF]** and it is shown in Figure **xxx**. The x-axis indicates the rank of the term in the collection (e.g.; the most frequent term is represented with a rank of 1). The y-axis indicates the frequency of the term in the collection. It can be seen that the curve descends very quickly. Typically, the most frequent terms in English account for 50 percent of all terms. So what does all this mean for an information retrieval system? Well, it says that a few terms occur so often that indexing them and retrieving them is probably meaningless. The odds of a term “the” randomly occurring in a document and a query is much higher than the odds of a more infrequent term. These frequently used terms are merely noise and can be eliminated from an information retrieval system. Typically, an information retrieval system contains a list of “stop words” and the parser, after

tokenizing the word and dealing with special characters, will check the list of stop words. If the term appears in the list of stop words, it will be removed. A list of about four hundred commonly used stop words in information retrieval was developed by Chris Buckley and used since the early 1980's is found in Appendix A. It is also in the file **stopwords.txt** on the book website.

It turns out that it is not all that clear exactly which stop words should be removed. Removing lots of them is tempting as they dramatically reduce the size of the inverted index and thus speed up the entire indexing process. Checking for them can be done very quickly with a binary search or a hashing algorithm. The stop word list is very small; so, it easily fits in memory making it easy to detect. Thus, what is the downside? Well, the classic counterexample is to search for the famous quotation "to be, or not to be". As luck would have it, this phrase contains only stop words and, worse, is the most famous quotation from William Shakespeare. For fun, go try the query "to be or not to be" on web search engines. You will find that many engines cannot find this phrase because they eliminate stop words. Others have bitten the bullet and index stop words simply because of examples like this. Often single letters are treated as noise occurrences. This is fine except for when a user issues a query such a "find all documents about Vitamin C." And if that is not enough, a reasonably common last name in Chinese is transliterated as "The". So to find all documents about "Major The" is a valid request.

For many systems, the few counterexamples that indicate the downside of stop words are not sufficient to justify indexing all terms. However, some systems index stop words simply to handle these counterexamples.

2 Implementation Details: A simple tokenizer

Our SimpleIR parser is implemented in the TRECParse class in the readDocuments() method. The readDocuments() method reads a file and outputs a list of Document objects. The TRECParse is designed for SGML TREC documents. The idea is that we need parsers to handle many different types of documents. Ultimately, we can develop a generic abstract Parser class, and TRECParse will be only be a subtype of this class. For now, there is no abstract parser class (to keep things simple).

```
public class TRECParse {  
  
    private static final int BEGIN_DOC      = 0;  
    private static final int END_DOC        = 1;  
    private static final int BEGIN_TITLE    = 2;  
    private static final int END_TITLE      = 3;  
    private static final int BEGIN_TEXT     = 4;  
    private static final int END_TEXT       = 5;  
    private static final int BEGIN_DOC_NAME = 6;  
    private static final int END_DOC_NAME   = 7;  
    private static final int BEGIN_DATELINE = 8;  
    private static final int END_DATELINE   = 9;  
    private static final int WORD           = -3;  
  
    private HashMap          distinctTerms;  
    private SimpleStreamTokenizer in;  
    private String           inputFileName;  
    private StopWordList     stopWords;  
    private PorterStemmer    porterStemmer;  
    private LovinsStemmer    lovinsStemmer;  
  
    TRECParse (String ifn, String swfn) {  
        inputFileName = ifn;  
        in = parserInit();  
        stopWords = new StopWordList(swfn);  
  
        /* activate the Porter stemmer */  
        porterStemmer = new PorterStemmer();  
  
        /* activate the Lovins stemmer */  
        lovinsStemmer = new LovinsStemmer();  
    }  
}
```

A TREC document may have various “zoned” fields such as *title*, *documentID*, *documentDate*, and *dateline*. We actually generate a document id, but the document name is a unique identifier assigned to every TREC document (e.g.; WSJ-1987-001). The constructor for the class TRECParser initializes a StopWordList object (which just consists of a HashMap with all of the stop words). Different parsers with different stop word lists are easily incorporated as the name of the file containing the stop word list is a parameter to this constructor.

We now describe the **parserInit** method given below. We open the input text file, launch a reader to read the file, identify the SGML tags of interest, and start up a SimpleStreamTokenizer to do the actual tokenization from the file. We will not include a description of the SimpleStreamTokenizer as its gory details are beyond the scope of this book. Suffice to say that we modified the SteamTokenizer that comes with Java to be a little more efficient and easy to read. The source code is on the CD that comes with this book. The key is that a call to nextToken() from the SimpleStreamTokenizer results in the next whitespace delimited token. No special handling of things like F-16 is done (we wanted it to be Simple!). Also, SGML tags are recognized by the tokenizer that returns an indicator as to which tag was encountered.

```

/* sets up for reading the inverted index: opens the input stream and */
/* reads in the stop word list                                     */
private SimpleStreamTokenizer parserInit() {

    SimpleStreamTokenizer in = null;

    /* see if we can open the input file */
    try {
        /* open the input stream for the input file */
        InputStream is = new FileInputStream(inputFileName);

        /* now make sure the input reading is buffered */
        Reader r = new BufferedReader ( new InputStreamReader ( is));

        /* now set up the stream tokenizer which will tokenize this */
        HashMap tags = initTags();
        in = new SimpleStreamTokenizer ( r, tags);
    }
    catch (FileNotFoundException e) {
        System.out.println("Could not open file --> "+inputFileName);
        System.exit(1);
    }
    return in;
}

```

Now we discuss the driver of the parser: the **readDocument** method. Essentially, a token is obtained from the **nextToken** method and the **ttype** (token type) is used to drive the parsing process. The variable **ttype** is defined in the **StreamTokenizer** class that comes with the JDK. In our implementation, **SimpleStreamTokenizer** was extended to include various types to indicate the type of SGML tag encountered. Different tags can be easily mapped to the same type. The <TITLE> and <TL> tags both map to the BEGIN_TITLE tag type. The mapping of titles to tags is done in the **initTags** method of the parser. Once the tag type is obtained, a **switch** statement is used to determine the correct action based on the tag. The END_DOC tag </DOC> triggers the creation of a document object. The BEGIN_DOC tag triggers the resetting of the existing document

```

String      dateline      = null;
String      docName       = null;
ArrayList   documentList  = new ArrayList();
boolean     done          = false;
int         documentID    = 0;
int         length        = 0;
int         offset        = 0;
String      title         = null;

boolean endOfFile = false;
while (! done) {
    try {
        if (in.nextToken() == in.TT_EOF) {
            done = true;
            endOfFile = true;
            continue;
        }
        switch (in.ttype) {
            case END_DOC:
                length = in.currentPosition - offset;
                Document d = new Document (documentID, docName, title, dateline,
                                           inputFileName, offset, length, distinctTerms);
                ++documentID;
                documentList.add(d);
                break;

            case BEGIN_DOC:
                offset = in.currentPosition;
                docName = null;
                dateline = null;
                title = null;
                distinctTerms = new HashMap();
                break;

            case BEGIN_DOC_NAME:
                docName = readNoIndex(in, END_DOC_NAME);
                break;

            case BEGIN_TITLE:
                title = readNoIndex(in, END_TITLE);
                break;

            case BEGIN_DATELINE:
                dateline = readNoIndex(in, END_DATELINE);
                break;

            case BEGIN_TEXT:
                readText(in, stopWords);
                break;

            case WORD:
                break;

            default: {
                System.out.println("Unrecognized tag: "+in.sval);
                System.exit(-1);
            }
        }
    } catch (IOException e) {
        System.out.println("Exception while reading doc ");
        e.printStackTrace();
    }
}
return documentList;
}

```

variables. Only text within the <TEXT> tags is actually indexed. The **readNoIndex** method is called for both the TITLE and DATELINE tags as these tagged fields indicate text that we do not wish to index or apply stopwords. This method reads tokens until the appropriate END_TAG is encountered. For web search engines, much text is ill formed and does not always have END_TAGS. For this case, a more sophisticated **readNoIndex** method is needed. For now, we assume that each start tag maps to a corresponding end tag. The **readText** method is given below:

```
private void readText(SimpleStreamTokenizer in, StopWordList stopWords) throws IOException
{
    TermFrequency tf;
    String token;

    in.nextToken();
    token = in.sval;
    while (in.ttype != END_TEXT) {
        token = token.toLowerCase();
        if (! stopWords.contains(token)) { // skip stop words

            // now stem the token if necessary
            String porterStem = porterStemmer.runStemmer(token);
            String lovinsStem = lovinsStemmer.stemString(token);
            if (distinctTerms.containsKey(token)) {
                tf = (TermFrequency) distinctTerms.get(token);
                tf.Increment(); // if we have it just increment tf
                distinctTerms.put(token, tf);
            } else {
                tf = new TermFrequency(); // new token, init the tf
                distinctTerms.put(token, tf);
            }
        }
        in.nextToken();
        token = in.sval;
    }
    return;
}
```

The **readText** method computes the term frequency of a term in a document.

Using the **distinctTerms** hashmap does this. Each term is mapped to its term frequency.

We start with an empty hashmap. Each term is first checked against the stop list. If it does not appear in the stoplist, it is then checked against the HashMap. If the term is not in the distinctTerms hashmap, it is added with a frequency of one; if it already appears, the frequency is updated. This continues until we hit the end of text tag.

The parser continues processing until the end of file is encountered. We note that the parser shown here can handle key tags occurring in any order. There is no requirement that DATELINE appear before or after TITLE. Text may appear at any time. Also text may appear multiple times. However, the parser shown here does not support nested tags. We assume that within a DATELINE tag there is no TEXT tag. If there is, it is treated as part of the DATELINE.

Removal of common endings such as “ing” or “ed” is called *stemming*. Numerous stemming algorithms exist. Regardless of the algorithm, in SimpleIR, the stemmer can be called in the **readText method**. We now briefly describe current stemming algorithms and then provide some implementation details on one approach that is commonly used, namely the Porter stemmer [\[REF\]](#).

3 Stemming Algorithms

Stemming refers to the process of removing common prefixes and suffixes from query and document terms. The notion is that a user searching for the term *book* will not mind finding documents with the term *books*. Numerous laboratory experiments were conducted since the late 1960’s, and they consistently show that prefix and suffix removal often has some impact on average precision (at best 10 percent). The reality is that *for many queries* stemming makes little difference. If we search for “Abraham Lincoln” we are going to be unaffected by any stemming algorithm and neither query

term can be ambiguously stemmed. However, if we search for “organization” and our stemmer removes the reasonably common suffix “ization” we end up with documents about “organ”. This kind of disaster is what stemmers try their best to avoid. The reality is that improving a stemmer typically increases its complexity. The other advantage to stemmers is efficiency. They dramatically reduce the number of distinct terms in the collection. This affects the size of the entire inverted index. In the 1970’s and 1980’s, this reduction in size was crucial as the term dictionary really needed to fit into memory and stemming made this possible. Today, stemming is done more for effectiveness, but given the size of the web, the temptation to reduce the number of terms is still pretty strong.

Stemmers can be categorized as language-specific, rule-based stemmers. The Lovins stemmer [REF] and the Porter stemmer are both rule based stemmers that specify a set of rules to apply to a given English term. Prefixes and suffixes are removed based on these rules. For example, a rule might be something like “if the term is over 4 characters and ends in ‘s’, then remove the ‘s’ and create a stem”. These rules are created with very specific types of words in mind, and the hope is that they will affect only the words that they are designed to affect.

Co-occurrence based stemmers [REF] appeared this past decade, and they work by identifying equivalence classes of terms that occur in the same documents and share common stems. The notion is that the terms “book” and “books” will frequently co-occur together in the same documents while “organ” and “organization” will likely co-occur less frequently (say in organ donation documents).

3.1 Rule-Based Stemmers

Rule-based stemmers use handcrafted rules while co-occurrence stemmers use corpus statistics about co-occurrence of terms. The Lovins stemmer and Porter stemmers essentially have large sets of handcrafted suffixes that are applied to the term in question. Once applied, “cleanup” rules are invoked to attempt to make sure the “stem” still works.

3.1.1 Lovins

The Lovins stemmer was developed in the late 1960’s [xxxx]. It starts with a set of suffixes that are eleven characters long. If any of these suffixes match, the suffix is removed and the algorithm moves to the cleanup phase. If not, the ten character suffixes are tried, then nine, etc. Essentially, the longest-matching suffix is applied to the term.

Once the cleanup phase starts, 34 rules are followed to change a suffix to another suffix.

The rules are described in Table 1:

Current Ending	Replacement	Example
Bb,dd, gg, ll, mm, pp, rr, ss, tt	B, d, g, l, m, p, r, s, t	
iev	ief	believe → belief
uct	uc	
umpt	um	
rpt	rb	
urs	ur	nurse → nur
istr	ister	
metr	meter	metric → meter
olv	olut	solvent → solut

[c]ul, [c] != {a,i,o}	[c]ll	
bex	bic	
dex	dic	
pex	pic	
tex	tic	
ax	ac	
ex	ec	
ix	ic	
lux	luc	
uad	uas	
vad	vas	
cid	cis	
lid	lis	
erid	eris	
pand	pans	
end	ens	
ond	ons	
lud	lus	
rud	rus	
[c]her, [c] != p,t	hes	
mit	mis	
[c]end, [c] != m	ens	
ert	ers	

[c]et, [c]!n	es	
yt	ys	
yz	ys	

A recent java implementation is available at <http://www.gnu.org/software/libstemmer/> in the GNU distribution. The stemmer works for many terms in that “book” and “books” have a common stem, as do numerous other terms. It only takes a few minutes to run the stemmer interactively before some humorous flaws come to light: “fatal” and “fat” share the common stem “fat”; “base”, “bass”, and “basically” all share the stem “bas”; “paste” and “past” have a common stem “past”; “passion” and “passive” (two opposites) actually share the same stem “pas”.

3.1.2 Porter

The Porter stemmer is similar to the Lovins stemmer. The general goal of the porter stemmer is to reduce the number of steps of Lovins and provide comparable performance.

The Porter stemmer essentially does the following steps:

Step 1a: Remove plurals: SSES → SS, IES → I, S → null

Step 1b: Normalize Tense: ED → null, ING → null, as long as characters preceding these suffixes contains a vowel.

Step 1c: Remove plurals: SSES → SS, IES → I, S → null

Step 1d: Y → I, as long as stem contains a vowel

Steps 2-5 contain sequences of replacements very similar to those found in Lovins (e.g.;

TIONAL → TION). The difference is the decision to replace is not based on an ever-

decreasing length of string. Instead, the potential for stemming is estimated based on the number of vowel-consonant shifts in the string. Consider the terms “pirate” and “necessitate”. The rule $ATE \rightarrow \text{null}$ exists but does not fire for “pirate”. Prior to the “ate”, the remaining characters “pir” contain only one vowel-consonant shift after an initial sequence of consonants. Conversely, “necessitate” has the stem “necessit” which has “ec”, “es”, “it” or three vowel-consonant shifts. The notion is that testing for vowel-consonant shifts is less computationally demanding than applying an ever decreasing set of rules.

3.2 *Dictionary-Based (KSTEM)*

A dictionary based stemmer proposed by Kroevetz [Kroe93] works to ensure that all stems are actually words in a dictionary. This stemmer was shown to outperform the Porter stemmer, but the need for an accurate dictionary is crucial to its success.

3.2.1 **Inflectional**

The inflectional stemmer proposed by Kroevetz has three components. First plurals are transformed to singulars, past tense to present, and ‘ing’ is removed. Plurals are partitioned into “ies”, “es”, and “s”. For “ies” the “s” is removed, and we check the dictionary to see if the new stem is found. If not, the ending is replaced with a “y”. For “es” and “s”, the “s” is simple removed. Exception lists are also used to handle processing of “ed” and “ing” endings for words that should be handled differently. Interestingly, Kroevetz shows [REF] that the inflectional stemmer outperforms the standard Porter stemmer for a variety of small collections. Unfortunately, a reasonable collection like TREC was not used for these results. Improvements tended to be small

with most improvements ranging from 5-10 percent in average precision. **[WERE they statistically significant at all – if not, say so, if yes, say so and change wording,]**

3.2.2 Derivational

The derivational stemmer **[REF]** only applied a stemming process to terms that were shown to be “related” to the root term. The dictionary shows that entries that are related are found in the same definition of the term. An analysis of common suffixes was done, and it was found that –er, -or, -ion, -ly, -ity, -al, -ive, -ize- and –ment, and –bl were the most common. The inflectional stemmer was modified to include these suffixes. It was found that the derivational stemmer outperformed the inflectional stemmer in every case. We note that both of these dictionary-based stemmers are intuitively appealing, but they rely heavily upon the presence and accuracy of a comprehensive dictionary. Also, to our knowledge, they remain untested on a large document collection.

3.3 Co-occurrence Based

The KSTEM stemmer **[REF]** is based on the quality of the term dictionary. Another approach is to base stemming on the co-occurrence of the original word and its stem **[Xuxx]**. The idea is that if a term and its stem occur frequently in the same documents across a document collection, it is desirable to stem the term. Consider the term “stocks”, in a document collection about business; stemming this term to “stock” makes good sense. In a collection about witch trials, “stock” may well refer to a device to hold prisoners. Depending on the co-occurrence in the document collection, we may or may not want to apply a given stem.

The measure of significance of co-occurrence used between term a and stem s is $(tf(a,s) - E(a,b)) / (tf(a) + tf(s))$, where $tf(as)$ is the number of co-occurrences of as within some window w of words, and $tf(a)$ is the number of occurrences of a , $tf(s)$ is the number of occurrences of s , and $E(a,b)$ is the expected number of occurrences of a with s . The algorithm is straightforward:

Step 1: Identify all unique tokens

Step 2: Construct the initial equivalence class of tokens based on an aggressive stemmer.

The Porter stemmer was used so if “jumping”, “jumped”, “jump” all end up with the same stem “jump” they are placed into an equivalence class. Note that the more aggressive the stemmer (the more often it stems words), the more terms that will be found in the equivalence class.

Step 3: Identify the co-occurrence of terms in the same equivalence class; compute the estimated co-occurrence here as well.

Step 4: For all “interesting” pairs, keep them in the equivalence class. We note that two algorithms (one based on connected components the other based on optimal partitioning) may be used to refine the equivalence class of terms.

For the wall street journal collection of TREC documents, it was found that the Porter algorithm outperformed KSTEM by 3.4 percent while the co-occurrence based stemmer outperformed KSTEM by 4.0. We note that another key advantage of this stemmer is that it is language independent; co-occurrence can be used on arbitrary languages.

4 Using Grammars for Parsing

Compilers for parsing input text commonly use parser generators based on grammars and tokenizing based on a programming language. It is reasonable to consider them as a potential tool for parsing a search engine. The general idea is to define an acceptable *language* as a set of valid strings defined by a *grammar*. Once the grammar is defined, a parser generator accepts the grammar as input and builds a parser. The parser may then be referenced from other routings. For the C programming language *lex* and *yacc* are used to generate parsers, for java, *javacc* is used.

The basic idea behind a language is a set of rules of the form $S \rightarrow a S b \mid \lambda$. where λ is the null string. Upper case letters refer to *non-terminals*, and lowercase letters are *terminals*. The upper case letters can be replaced by any rule. Hence the string *aaabbb* may be formed by applying the $S \rightarrow a S b$ rule three times and finally applying the $S \rightarrow \lambda$ rule one time. Detailed descriptions of numerous grammars and types of languages may be found in [xxx, yyy, zzz]. Grammars are so common in Computer Science that they are often used to specify programming languages. A grammar for Java is found in [xx].

Grammars may also make use of regular expressions. See [] for details on regular expressions. The expression a^* indicates zero or more *a*'s. The expression a^+ indicates one or more *a*'s. Some useful regular expressions for document parsing are given in Figure 7.

Figure 7: Regular Expressions

Acronym: `(["A"-"Z"])(["A"-"Z"])*` Ex: NCR, IBM, etc.

Abbreviation: `(["A"-"Z"] ".")*>` Ex: U.S.A.

Model: `["a"-"z","A"-"Z"] "-"(["0"-"9"])*>` Ex: F-16, C-25

Word: `["a"-"z","A"-"Z"](["a"-"z","A"-"Z"])*>` Ex: hippo, Hippo

Integer: `["0"-"9"](["0"-"9"])*>` Ex: 123

Decimal: `(["0"-"9"])* "."(["0"-"9"])+>` Ex: 123.45

Figure 8 gives some grammar rules to define a TREC document. The general rule of $S \rightarrow \langle \text{document} \rangle | \text{EOF}$ indicates that the starter symbol may either be replaced by zero or more documents followed by EOF. In JavaCC this is coded as the *parseDocument* routine and is called zero or more times (using the *kleene star* – the real name for it) operator described earlier) followed by and end-of-file marker.

Figure 8: Parsing a File

File consists of zero or more documents followed by an EOF marker.

S --> <document> | EOF.

void Input() :

```
    int DocId = 0;
    Document d = new Document(0);

    (parseDocument(d)

        d.output();
        ++DocId;
        d = new Document(DocId);

    )* <EOF>
```

For document parsing, the grammar would be of the form shown in Figure 9. This shows that a Document may optionally consist of a headline, a dateline (the order is not restricted) and optionally some text. If the text occurs, it will consist of one or more words. Finally, a given word will either be a model (e.g.; F-14), acronym, integer, or some other terminal.

Figure 9: Parsing a Document

Document consists of an optional headline or dateline followed by some text.

<document> → <headline> | <dateline> | <textbegin>

<textbegin> → <text_begin> <word> <text_end>

<word> → <model> | <acronym> | <integer>, etc.

A simple JavaCC parser for TREC documents is included on the distribution CD in the file `trecParser.jj`. This file may be used as input to `javacc` and a `.java` output file will be produced. This `.java` file has sufficient class libraries to use for parsing documents.

JavaCC supports very complex parsing of SGML and HTML documents. HTML and SGML specific grammars exist on the web for different types of documents, and they can easily be incorporated and modified. We note that the generic nature of the tool comes at the cost of performance. A JavaCC parser will be slower than the custom built parser we provide as part of SimpleIR. However, JavaCC performance is getting better as the code for parser generation is constantly being improved. Also, for a sophisticated document collection, it may well be worth a slower parser to have code that is more easily modifiable. At present, we are unaware of a web search engine that uses JavaCC.

However, it is reasonable to expect that ultimately this will occur as more complex document parsing becomes mandatory, and the parser generation continues to be

optimized. For now, we recommend using JavaCC for indexing small to mid-sized document collections that contain sophisticated document tags. For large document collections, you probably should resort to building your own custom parser.

5 Sample Implementation

```
private boolean hasSuffix( String word, String suffix, String stem
) {
    String tmp = "";

    if ( word.length() <= suffix.length() )
        return false;
    if (suffix.length() > 1)
        if ( word.charAt( word.length()-2 ) !=
            suffix.charAt( suffix.length()-2 ) )
            return false;

    stem = "";

    for ( int i=0; i<word.length()-suffix.length(); i++ )
        stem += word.charAt( i );
    tmp = stem;

    for ( int i=0; i<suffix.length(); i++ )
        tmp += suffix.charAt( i );

    if ( tmp.compareTo( word ) == 0 )
        return true;
    else
        return false;
}
```

We already provided a detailed description of the SimpleIR parser. We now describe the implementation of a Porter stemmer. First we start with some utility methods. The `hasSuffix` method above simply identifies whether or not a suffix exists in a given term. The `cvc` method below identifies whether or not we have had a switch from consonant to vowel to consonant in the method described. The initial if statement essentially checks

the string at position (length-1) and checks to see if we have a non-vowel. If so, we continue checking at (length-2) for a vowel. If both of these match, we make sure the string has at least three characters and then we check to see if we have a vowel at (length - 3).

```
private boolean cvc( String str ) {
    int length=str.length();

    if ( length < 3 )
        return false;

    if ( (!vowel(str.charAt(length-1),str.charAt(length-2)) )
        && (str.charAt(length-1) != 'w')
        && (str.charAt(length-1) != 'x') && (str.charAt(length-1) != 'y')
        && (vowel(str.charAt(length-2),str.charAt(length-3))) ) {

        if (length == 3) {
            if (!vowel(str.charAt(0),'?'))
                return true;
            else
                return false;
        }
        else {
            if (!vowel(str.charAt(length-3),str.charAt(length-4)) )
                return true;
            else
                return false;
        }
    }

    return false;
}
```

The notion is that if we have a consonant-vowel-consonant ending to the string, this will drive the stemming decisions. A *measure* method given below identifies the number of cvc switches we have had in a term. The notion is that the more cvc switches, the longer the word is, and hence, the more likely that the term should be stemmed.

```

private int measure( String stem ) {

    int i=0, count = 0;
    int length = stem.length();

    while ( i < length ) {
        for ( ; i < length ; i++ ) {
            if ( i > 0 ) {
                if ( vowel(stem.charAt(i),stem.charAt(i-1)) )
                    break;
            }
            else {
                if ( vowel(stem.charAt(i),'a') )
                    break;
            }
        }

        for ( i++ ; i < length ; i++ ) {
            if ( i > 0 ) {
                if ( !vowel(stem.charAt(i),stem.charAt(i-1)) )
                    break;
            }
            else {
                if ( !vowel(stem.charAt(i),'?') )
                    break;
            }
        }
        if ( i < length ) {
            count++;
            i++;
        }
    } //while

    return(count);
}

```

Measure is straightforward. We keep scanning text until we hit a vowel. Then, we scan until we hit a non-vowel. Once this happens, we increment a counter. Hence, the number of shifts between a vowel and a consonant is returned. The driver for the stripSuffixes method in the porter stemmer is given below. As mentioned earlier, stemming is a five-step process, where the stems for each step are sent to the stem of the next step. To give an idea of how the steps are implemented, step 2 is given below (step 1 has a bunch of special cases).

In step 2, a list of suffixes are initialized, and they are checked sequentially against the input string. If the suffix matches and we have a measure greater than zero (implying that we have at least one consonant-vowel shift), we replace the suffix with the corresponding stem found in the suffixes table.

```

private String stripSuffixes( String str ) {

    str = step1( str );
    if ( str.length() >= 1 )
        str = step2( str );
    if ( str.length() >= 1 )
        str = step3( str );
    if ( str.length() >= 1 )
        str = step4( str );
    if ( str.length() >= 1 )
        str = step5( str );
}

```

```

private String step2( String str ) {

    String[][] suffixes = { { "ational", "ate" },
                            { "tional", "tion" },
                            { "enci", "ence" },
                            { "anci", "ance" },
                            { "izer", "ize" },
                            { "iser", "ize" },
                            { "abli", "able" },
                            { "alli", "al" },
                            { "entli", "ent" },
                            { "eli", "e" },
                            { "ousli", "ous" },
                            { "ization", "ize" },
                            { "isation", "ize" },
                            { "ation", "ate" },
                            { "ator", "ate" },
                            { "alism", "al" },
                            { "iveness", "ive" },
                            { "fulness", "ful" },
                            { "ousness", "ous" },
                            { "aliti", "al" },
                            { "iviti", "ive" },
                            { "biliti", "ble" } };

    String stem = new String();

    for ( int index = 0 ; index < suffixes.length; index++ ) {
        if ( hasSuffix ( str, suffixes[index][0], stem ) ) {
            if ( measure ( stem ) > 0 ) {
                str = stem + suffixes[index][1];
                return str;
            }
        }
    }

    return str;
}

```

6 Summary

We described an overview of the SimpleIR parser, a survey of stemming algorithms, and grammar based parsing. Stemming is crucial to average precision as a query term may well be very semantically different from its stem. Avoiding stemming can also be costly because queries with common suffixes such as “-ed” and “-ing” will end up not matching document terms that mean the same but have a trivial suffix. Dictionary based stemmers tend to out-perform simplistic stemmers, but they pay the price of a dictionary lookup plus many words are not in the dictionary. Co-occurrence based stems are language-independent, but require substantial processing time to compute co-occurrences, and they have not shown tremendous performance improvements over simpler techniques.

Grammars are a nice way to handle complex documents. JavaCC is a tool that automatically builds a parser for an arbitrary grammar. The only concern is that for large document collections, run-time performance may be so important that it rules out the luxury of using a parser generator.

7 Exercises

1. Show how the Porter stemmer can be built with javaCC.
2. Consider a stemmer that did not include the “measure” check as given in the Porter stemmer. Provide five examples of terms that would be incorrectly stemmed. Show how these examples are “fixed” by the measure indicator.
3. In Japanese and Chinese, a single word is represented by a character. Is there a role for stemmers in this language?

4. Suggest a means for building an Arabic stemmer. Arabic has hundreds of common word endings.

8 References

[Kroe93] Viewing morphology as an inference process. In *Proceedings of the 16th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval.*, pp. 191-202, 1993.

[Port80] Porter, M. An algorithm for suffix stripping. *Program*, 14(3), 130-137.

[Xuxx] Corpus-Based aStemming using Co-occurrence of Word Variants