A Framework for Complex Tokenisation and its Application to Newspaper Text

Robert Dale

Language Technology Group
Microsoft Research Institute
School of MPCE
Macquarie University
New South Wales 2109 Australia

Robert.Dale@mq.edu.au

Abstract

A word is more than a sequence of characters between two spaces. This fact has generally been ignored in research on natural language processing; but recognising the complexity of what it is to be a word is of crucial importance if we are to add sophisticated natural language processing techniques to existing document processing applications to make them more language-sensitive.

This paper describes a framework for the tokenisation of text that tries to address this problem by providing a parameterisable approach to the tokenisation task, so that NLP *components can be provided with a richer analysis of real texts. We demonstrate the ideas with application to the wide variety of word forms that appear in newspaper text.*

Keywords: tokenisation, intelligent text processing, natural language processing

1 Introduction

Typical text processing tools such as word processors and text formatting tools embody very simple notions of what constitutes a word: generally, in such systems, a word is any sequence of characters bounded by spaces, or sometimes by other punctuation marks. These characterisations of wordhood are useful and appropriate where the kinds of operations that are to be performed on words are simple: for example, deleting a word, moving the cursor forwards or backwards a word, or deciding whether there is sufficient space to place a word on the current output line.

As the systems we build try to do more sophisticated things with text, these simple characterisations begin to break down. In particular, as we try to make use of techniques and ideas from research in natural language processing, we need to develop more sophisticated notions of what constitutes a word, and we need to recognise that words have internal structure that can be usefully manipulated.

We say that a system embodies LANGUAGE SENSITIVITY if it views text as more than just a sequence of characters, and takes on board linguistically motivated characterisations of the data: so, individual characters are combined into words; words are combined into sentences, perhaps with some intermediate levels of structure to indicate syntactic constituency; and sentences are combined together into paragraphs. Many current applications possess what may appear to be language sensitivity, but in general this is an illusion: it is usually the case that simple heuristics substitute for a deeper understanding.

An obvious place to look for the kinds of information and generalisations that we need for language sensitivity is in the area of natural language processing: this broad term covers technologies concerned with morphology, syntax, semantics and pragmatics, all key notions in dealing with text more intelligently. Unfortunately, much of the work in these areas is far from broad application; and more importantly from the point of view of this paper, there is a missing link that still needs to be developed. We mentioned above that words have internal structure that can be usefully manipulated. For much work in linguistics, this structure is characterised in terms of morphology: for example, the word *churches* has a base form *church* and a plural ending *es*. A great deal of work has been done in this area, and much of it is useful in the development of intelligent text processing systems; however, even before we begin to examine a word's morphology, we need to recognise that not all words are so simple, and that real texts are not as neat and tidy and well-behaved as those discussed in linguistics textbooks or used as examples in laboratory prototypes of

Proceedings of the Second Australian Document Computing Symposium, Melbourne, Australia, April 5, 1997.

natural language processing systems. Quite apart from important issues such as breadth of coverage, problems raise themselves much earlier.

This is an important realisation, and one that has only recently been accepted as an issue for work in natural language processing. It has been provoked by the increasing amount of NLP research that tries to use large corpora of real texts as data; this work makes it hard to ignore the realities of text. As a result, in the last few years we have begun to see research from a natural language processing perspective that tries to say something about the characteristics of real written language: notable work in this area is Nunberg's [1990] linguistically-motivated analysis of punctuation, Grefenstette's [1994] work on tokenisation, and Palmer and Hearst's [1994] work on sentence segmentation.

The work described in this paper is in the same spirit. The topic the paper addresses is one of the first problems that has to be faced in building a bridge between text processing and natural language processing: what is a word? If we are to combine text processing techniques and natural language processing techniques for maximum effectiveness, it is precisely here that the crucial interface lies, and so it is important that we develop as robust a model of what constitutes a word as possible. The goal of the research described here is to develop an easily customisable tokeniser that can handle arbitrary text files as input, producing whatever output a client natural language processing system prefers to see.

The work described here derives from some experimental systems we have developed over the last few years; see in particular [Dale 1990; Matheson and Dale 1993; Dale and Douglas 1996]. On the basis of this research we have become convinced that there is, perhaps not surprisingly, no one answer to what should count as a word; it all depends, of course, on what task is being carried out. What we require, then, is a truly flexible approach where we can experiment with and develop different notions of wordhood. This paper describes the framework we are developing to explore these questions.

The paper is structured as follows. In Section 2, we sketch our overall approach to the problem of tokenisation. In Sections 3 and 4, we go beyond the simple notion that a word is a sequence of characters bounded by spaces and describe a framework for what we call 'universal tokenisation'; and in Sections 5 and 6 we go on to exemplify this framework in the context of an analysis of a small amount of newspaper text. Section 8 provides some concluding remarks.

2 Our Approach to Tokenisation

We begin by taking the view that a text is made up of what we will call tokens, and that tokens can be of two types, which we call word tokens and punct (for punctuation) tokens. A word token is used to represent, naturally, a word, along with any punctuation which is properly part of that word (LEXICAL PUNCTUATION). A punct token is used to represent any contiguous sequence of (non-lexical) punctuation characters in the text.

Unfortunately, the ASCII character set does not divide straightforwardly into those characters which form words and those which form puncts: some characters can belong to either, depending on the context. There are a number of these ambiguous characters, the full stop or period being the most common, since it can appear as a sentence terminator (in which case it is part of a punct) or as punctuation within an abbreviation (in which case it is part of a word). So, in example (1) below, the character string *rhino(s)* would be represented by a single word token, as would the string *eventually*; the open parenthesis immediately before the first *e* in eventually is not part of a word token, but is part of a punct token, consisting of a space and an open parenthesis and falling between the two word tokens.

(1) The rhino(s) (eventually) ate the cake.

A number of heuristics can be used in order to decide how to tokenise a text which contains ambiguous characters; when higher level knowledge is available from lexical sources and syntactic context, this can be used to disambiguate cases where there is doubt.

To enable higher-level linguistic processing to be applied to individual tokens or sequences of tokens, it proves useful to represent each token as an object that maintains information derived from the analysis of the word or punctuation sequence it corresponds to. For a given token, this structure might contain the following information:

- information regarding the syntactic category of the token, and its root form if this is different from the token itself, along with syntactic features such as number;
- information relating to the semantic type of the token, derived from a lexicon if one is available: in the context of style-checking, for example, it might be useful to know whether the word is the name of a month, or a unit of measure;
- information about the typographic form of the token: in particular, the casing of the word is

Type	Example
mixed-case-word	PhD
alphanumeric	DEC10
abbreviation	i.e.
	B.B.C.
	Ph.D.
ordinal-number	23rd
date	23rd December 1992
	23-12-92
os-pathname	/home/user3/fred
real-number	23.4
measurement	23.4 kg
latex-object	\documentstyle{article}
...	...

Figure 1: Some complex tokens

of significance, and other features such as the typeface used are important in the context of text processing.

So, for example, the first word in the sentence *Is this the best solution?* would be analysed as having the root *be*, with the syntactic features of present tense and singular number, and the typographic feature of capitalised casing.

This much is straightforward. It turns out, however, that real tokens can be quite complicated objects, with considerable internal structure beyond the morphological structure that standard natural language processing techniques can identify. Figure 1 shows some examples of the kinds of tokens we have to consider if we are to reliably process real text. Note that we have included here some tokens, such as the first example of a date token, that consist of more than one word: these are textual entities which correspond to the Text Encoding Initiative's notion of a CRYSTAL, and which for some text processing purposes are best viewed as single tokens. There is clearly a hazy line between tokenhood in this sense and the notion of syntactic constituent that we find in the linguistics literature.

3 The Framework

To be able to process texts that contain tokens like those we have just described, it is important that we take on board the complexities of the data. After much experimentation, our current view is that tokenisation is best performed by a process that uses two separate stages.

The overall architecture is shown in Figure 2; here, everything inside the dotted line is part of the tokeniser. The basic idea is that a stream of characters is read into the tokeniser from some external source, and then successively processed through the different internal components to produce some stream of higher level objects, which we will call TOKENS, that can be used by some client—this could be a parser, or some component of an information retrieval system, for example.

The system is broken down into the modules shown to provide customisability at a number of different levels where the ability to customise seems like a useful thing to have. The individual components have the following functionalities.

The Bundler: The Bundler is the simplest and least intelligent part of the tokeniser. It knows only how to map the characters in the character set used into a predefined set of character classes; each character is in only one class. The character$\longrightarrow$character class mapping is defined by a control file, and so is easily changed; the Bundler uses this information to segment the input stream into BUNDLES, which we'll also call SIMPLE TOKENS. A bundle or simple token is simply a sequence of one or more characters from the same character class: so, for example, given appropriate definitions in the control file, any sequence of alphabetic characters might constitute a bundle, and each space character might constitute a bundle.

The Compounder: The Compounder takes as input the simple tokens provided by the Bundler and decides whether any of these simple tokens need to be collected together into what we will call COMPLEX TOKENS. Exactly what counts as a complex token is determined by the Compounder's control file: for example, an simple alphabetic token followed by a simple numeric token might be put together to form a complex token. There are also what we might think of as multi-word complex tokens, or after the TEI, CRYSTALS. These are sequences of words and symbols, such as dates and names, that are constructed in a regular way but which are typically not catered for by a conventional natural language parser. The job of the Compounder is to build complex tokens corresponding to these crystals, packaging them up and annotating them in such a way that the client parser need not be concerned with their internal details but can still make use of them.

The Interface: If we want our tokeniser to be generally useful to a wide range of clients, then we cannot assume too much about the nature of the input expected by these clients. The job of the Interface is to convert from the tokeniser's internal structures into the form of input expected by a particular client; again this trans-

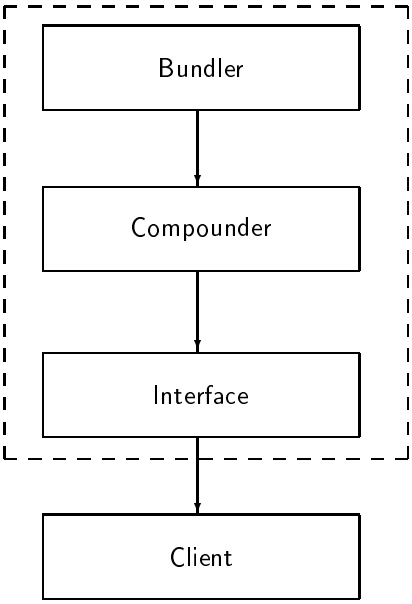

Figure 2: The overall architecture of the tokeniser

formation is carried out using rules specified in a control file. Thus we might expect different control files to allow the generation of lists of Lisp symbols, lists of Prolog atoms, or complex feature structures, as required. This aspect of the tokeniser's behaviour will not be discussed further in the present paper; we will focus instead on the internal aspects of tokenisation.

4 The Individual Components

4.1 The Bundler

In what follows we will assume that the input characters are ASCII, although there is no reason why other character sets should not be used.

4.1.1 Specifying Bundling Rules

The means of character class specification should make it as easy as possible to specify the mapping from each character in the character set to its class. Each character class corresponds to a token TYPE. The specification also needs to have some way of saying whether a token of a particular type contains only one character or many: for example, any sequence of numeric characters constitutes a numeric token, but a open-paren token consists of a single open-parenthesis: if we find two open-parentheses in a row, we need to be able to specify that they should be separate tokens if that is the most useful way to view them.

Character	Class	Iterable?
A	upper-alpha	Yes
⋮	⋮	⋮
a	lower-alpha	Yes
⋮	⋮	⋮
0	numeric	Yes
⋮	⋮	⋮
␣	spaces	Yes
%	percent	No
⋮	⋮	⋮

Figure 3: A character class table

We can specify the character to character class mapping by means of a table like that in Figure 3; this is effectively a short-hand notation for a set of rules of the following form:

(2) a. lower-alpha $\longrightarrow$ [a-Z]$^{+}$
b. open-paren $\longrightarrow$ (

Here, the left hand side of the rule is a specification of the name of the character class (and therefore token type) in question; the right hand side of the rule is a regular expression specification of the contents of tokens of this type. These rules can then be compiled into an appropriate finite-state machine; such a compilation procedure would also carry out appropriate error checking and ensure, for example, that no character has been assigned to two different classes.[1]

One possible complete character$\longrightarrow$character class mapping is shown in Figure 4.[2] Note that these bundling rules have the following consequences:

1. Any word which begins with an initial capital letter followed by a sequence of lower-case letters will be viewed as a sequence of two simple tokens.

2. All non-alphanumeric characters except spaces, line feeds, tabs and hyphens are viewed as single-character tokens.

The important thing to note is that these consequences are consequences of the particular bundling rules we have specified; a different set of

[1] In our most recent work we have been using Yacc and Lex to perform these processes, although in the general case rather more flexible mechanisms are required.

[2] To explain the notational conventions employed here: square brackets are used to indicate ranges; symbols inside angle brackets name characters that are difficult to show in their regular form; and the use of a subscripted '+' indicates that one or more instances of the preceding character specification are required.

lower-alpha	⟶ [a-z]$^+$
upper-alpha	⟶ [A-Z]$^+$
numeric	⟶ [0-9]$^+$
spaces	⟶ ␣$^+$
line-feeds	⟶ ⟨LineFeed⟩$^+$
tabs	⟶ ⟨Tab⟩$^+$
hyphens	⟶ -$^+$
open-paren	⟶ (
close-paren	⟶)
open-square	⟶ [
close-square	⟶]
open-curly	⟶ [
close-curly	⟶]
open-angle	⟶ <
close-angle	⟶ >
slash	⟶ /
backslash	⟶ \
full-stop	⟶ .
comma	⟶ ,
colon	⟶ :
semi-colon	⟶ ;
exclamation-mark	⟶ !
question-mark	⟶ ?
dollar	⟶ $
tilde	⟶ ~
open-quote	⟶ ‘
close-quote	⟶ ’
double-quote	⟶ "
ampersand	⟶ &
at-sign	⟶ @
percent	⟶ %
caret	⟶ ^
asterisk	⟶ *
underscore	⟶ _
plus	⟶ +
equals	⟶ =
pipe	⟶ |

Figure 4: A sample set of bundling rules

bundling rules can be used, without affecting the overall approach to tokenisation.

4.1.2 Output Tokens

The tokens generated by the Bundler can be represented by feature structures like those shown below:

(3) a.
$$\begin{bmatrix} \text{token: simple} \\ \text{contents: } \texttt{ntidisestablishmentarianism} \\ \text{type: lower-alpha} \end{bmatrix}$$

b.
$$\begin{bmatrix} \text{token: simple} \\ \text{contents: (} \\ \text{type: open-paren} \end{bmatrix}$$

Any simple token must have three fields: a token field with the value simple, a contents field that contains a string consisting of the characters that make up the token, and a type field that specifies the character class of the token.

4.2 The Compounder

4.2.1 Overview

The Compounder's job is to put together tokens to make complex tokens: a typical instance is where the bundler has decided that, in the string *23rd*, the *23* forms one simple token and the *rd* forms another. The Compounder's rules will specify that these two tokens can be combined to make a complex token. So, from the tokens in examples (4a) and (4b) the Compounder might build the token in example (5).

(4) a.
$$\begin{bmatrix} \text{token: simple} \\ \text{contents: 23} \\ \text{type: numeric} \end{bmatrix}$$

b.
$$\begin{bmatrix} \text{token: simple} \\ \text{contents: } \texttt{rd} \\ \text{type: lower-alpha} \end{bmatrix}$$

(5)
$$\begin{bmatrix} \text{token: complex} \\ \text{parts: } \begin{bmatrix} 1: & \begin{bmatrix} \text{token: simple} \\ \text{contents: 23} \\ \text{type: numeric} \end{bmatrix} \\ 2: & \begin{bmatrix} \text{token: simple} \\ \text{contents: } \texttt{rd} \\ \text{type: lower-alpha} \end{bmatrix} \end{bmatrix} \\ \text{type: ordinal-number} \end{bmatrix}$$

Whereas no ambiguity was possible in the Bundler's ouptut, we now get the possibility that there may be different sequences of complex tokens: for example, a full stop might be combined with the preceding simple token to make some kind of abbreviation, but might equally be considered to be a sentence-terminating token. This requires the Compounder to make an intelligent decision on the basis of whatever sources of knowledge it has access to. In different tokenisation experiments we have used different solutions to this problem. In both the *BibEdit* [Matheson and Dale 1993] and *Editor's Assistant* [Dale and Douglas 1996] work, our tokeniser would examine the contents of surrounding tokens in order to make a decision: for example, on encountering a simple token that corresponds to a full stop, the compounding stage may check to see if the immediately preceding token is potentially part of an abbreviation, or whether the immediately following token could be the first word in a new sentence. A more linguistically sophisticated system might try to make use of whatever syntactic and semantic knowledge is available, but realistically such sources of information are beyond the capabilities of current systems. Another strategy, and probably the best in the longer term, is to take the view that it is *not* the job of the Compounder to decide what the correct answer is: instead, multiple parses should be produced. Also, it is possible that a number of tokens may be combined in the same way but given multiple possible interpretations: so, for example, the complex token above might also be considered to be a computer name.

Notice also that the rules used by the Compounder are potentially quite complex: ideally, in the example above, the rule needs to have some way of distinguishing ordinal-number tokens from other kinds of alphanumeric tokens. So, for example, one rule for alphanumerics might look like the following:

(6) X0 $\longrightarrow$ X1 X2
⟨X0 type⟩ = alphanumeric
⟨X1 type⟩ = alphabetic
⟨X2 type⟩ = numeric

Of course, this is not as expressive as we would like it to be: using this limited formalism for expressing compounding rules would mean that we'd need a very large number of rules to cover all the possibilities.[3]

A rule for ordinal-numbers might look like the following:

(7) X0 $\longrightarrow$ X1 X2
⟨X0 type⟩ = ordinal-number
⟨X1 type⟩ = alphabetic
⟨X2 type⟩ = numeric
⟨X2 contents⟩ isa ordinal-ending

Here we have added the notion of contents-field type checking by means of a new operator, isa; this has the same net effect as performing lexical lookup. Ultimately, it becomes necessary to develop a type hierarchy, where types are more or less specific: so, for example, we might have alphabetic, alphanumeric and punctuated as sub-types of compound tokens, and os-pathname, hyphenated-word and date as subtypes of punctuated tokens.

5 Applying Tokenisation to Newspaper Text

We have described above some fairly complex machinery for breaking a text into tokens. A valid question to ask is whether this complexity is really required. This section looks at the results of an analysis of newspaper text (in particular, one issue of *The Guardian*) to identify the real variety of tokens that we have to consider. After this analysis, we provide a grammar for compounding that covers this data.

A significant proportion of the tokens found in newspaper text consist of the simple alphabetic forms; but there are a number of more complex token types too, enumerated below.

5.1 Hyphenated Compounds

A hyphenated compound is a token consisting of smaller tokens connected together by a hyphen. Identifying this structure can be important for carrying out any intelligent processing that is required.

Figures 5 and 6 show two categories of hyphenated compounds found in the analysed text; in each case we have reproduced the entire sets detected in order to demonstrate the wide range of semantic constructs that appear, although we do not at the moment have much to say about specific semantic analyses that would be appropriate; the point is that, if we do want to perform intelligent processing of these 'words', then we do need to have available some analysis of their internal structure.

5.2 Apostrophed Words

Apostrophed words are relatively straightforward, but once more we have to be able to decompose them appropriately to carry out appropriate processing tasks.

Possessives: *royal's*, *year's*, *Citizen's*, *BBC2's*, *MPs'* and *nurses'*; a very large proportion

[3] The formalism used here is very deliberately based on that used in the unification-based grammar formalisms popular in the natural language processing community; see, for example, Sheiber [1986].

three-day
two-thirds
ex-inmates
ex-millionaire
short-sighted
short-termism
well-received
sister-in-law
editor-in-chief
milk-and-water
million-year-old
three-year-old
Anglo-American
Caiger-Smith
Churchill-Coleman
Hindu-Muslim
Coca-Cola
Johnny-Come-Latelys
Serb-controlled
Italian-style
Biblical-style
Co-operation
B-Specials
UN-sponsored
US-led
al-Hariri
anti-Barre
counter-IRA
outer-London
pro-MPLA

Figure 5: Alphabetic hyphenated-compounds

of these are possessives of proper names. A case to watch out for is the apostrophed abbreviation, as in *Inc.'s*.

Contractions: *'cos*, *aren't*, *couldn't*, *didn't*, *Everywhere's*, *He's*, *I'm*, *It'll*, *It's*, *we'll*, and *We've*.

5.3 Number Compounds

Numbers are more than simple sequences of digits. To cater for this fact, it is useful to have a notion of NUMBER COMPOUND. As it happens, in the analysed text these are more common than simple numerics. The different subtypes we have identified are as follows:

- comma-punctuated numerics as in *250,000*
- decimals as in *3.1* and *61.67*
- currency amounts, as in *$400*, *£289*, *£10,000*, *£1.8*, *£23.7*, *$12.20*, and *$116.5*.

11-14
5-2
200-400
1994-95
12-year
15-day
24-hour
100-strong
80-yard
72-year-old
18-21-year-olds
10,000-word
33,000-strong
12th-century
pre-20th
Start-2
AK-47s
MiG-23
£26-27
£100,000-plus
£1,874-a-week

Figure 6: Other hyphenated-compounds

5.4 Less Common Compounds

There are a number of other less common kinds of compound tokens. We have found the following categories to be in evidence:

alphanumerics: True simple alphanumerics—tokens consisting only of alphabetic characters and digits—are quite rare. Apart from the predictable *17th*, *61st* and *194th*, we also get *1960s*, *DM150*, *M15*, *28min*, *01sec*, *5ft*, and *5ins*.

mixed-case: Again, these are much less common than one might have thought. Examples: *MPs*, *CDs*, *McAvennie*, *USAir*, and *DoE*.

slashed-compounds: These are very rare: *Heath/Walker*, *AIRMIC/broker*, and *1993/94*.

full-stopped-tokens: These are very rare: *A.*, *Inc.*, and *C.J.*.

6 A Grammar for Compound Tokens in Newspaper Text

In the previous section we have outlined the various kinds of tokens found in our analysis of newspaper text. In this section we present a grammar for compound tokenisation that covers this data. Different compounding rules are likely to be required for other genres of text. It should also be noted that the rules were developed only to handle body text: advertisements and sports results might produce some additional token types.

We will make the assumption that a text consists of a sequence of tokens, and we will require that a text be an alternating sequence of wordand punct tokens. This means our top level rule for the decomposition of a text is as follows:

(8) text $\longrightarrow$ {punct} word (punct word)* {punct}

6.1 A Grammar for Word Tokens

1. Words can be simple or complex, in the sense that they may consist of only one bundle, or they may consist of a number of bundles. When a word consists of a number of bundles, the alphabetic and numeric bundles may be separated by punctuation characters; where this is the case, each punctuation character may appear only once in any given complex word, with the exception of hyphens, which can appear multiple times. To deal with this, we define word tokens at the top level as follows:

   (9) word $\longrightarrow$ nohyphen-word | hyphenated-compound

2. A nohyphen-word is either simple or compound:

   (10) nohyphen-word $\longrightarrow$ simple-word

   (11) nohyphen-word $\longrightarrow$ compound-word

3. Words which are simple-words are those which contain only alphabetic or numeric characters:

   (12) simple-word $\longrightarrow$ lower-alpha | upper-alpha | mixed-case | numeric | alphanumeric

   Of these types, only lower-alpha, upper-alpha, and numeric are provided as primitives by the BUNDLER rules specified earlier. We also therefore require the following rules:

   (13) mixed-case $\longrightarrow$ (upper-alpha | lower-alpha) (upper-alpha | lower-alpha)$^+$

   (14) alphanumeric $\longrightarrow$ numeric (lower-case | upper-case)$^+$

   (15) alphanumeric $\longrightarrow$ (lower-case | upper-case)$^+$ numeric

   It would be useful to have a rule for initcap-rest-lowers tokens, but this requires more sophistication that the current rule specifications permit (it requires reference to the *length* of the bundles).

4. Words which are compound-words are of various kinds.

   (16) compound-word $\longrightarrow$ number-compound

   (17) compound-word $\longrightarrow$ apostrophed-word

5. Apart from their appearance with alphabetic characters in alphanumeric tokens, numbers often appear in conjunction with other characters. We capture all the interesting cases with the following rules:

   (18) number-compound $\longrightarrow$ {currency-symbol} numeric {comma numeric} {full-stop numeric}

   (19) currency-symbol $\longrightarrow$ dollar

   Note that currency amounts such as *140DM* will be identified as alphanumeric tokens.

6. apostrophed-words are handled by the following rules:

   (20) apostrophed-word $\longrightarrow$ compound-word close-quote {compound-word}

   (21) apostrophed-word $\longrightarrow$ close-quote compound-word

7. Finally, hyphenated-compounds are captured by the following rule:

   (22) hyphenated-compound $\longrightarrow$ nohyphen-word (hyphens nohyphen-word)$^+$

   This gives a flat structure for multiply-hyphenated compounds, and leaves to some subsequent processing the question of whether some hierarchy should be introduced within this structure.

The complete grammar for word tokens is collected together in Figure 7.

7 A Grammar for Punctuation Tokens

Punctuation tokens are relatively straightforward. The complete set of punct tokens we permit is defined by the grammar in Figure 8. We identify three general types of punct tokens:

spacing: these are tokens whose primary purpose is to separate words; they are typically made up of combinations of space characters, but we also allow the possibility for a spacing token to consist simply of a sequence of hyphens.

constituent-delimiter: these are complex tokens consisting of the punctuation characters which terminate clauses and phrases, along with their associated spacing tokens.

word	$\longrightarrow$	nohyphen-word | hyphenated-compound
nohyphen-word	$\longrightarrow$	simple-word
nohyphen-word	$\longrightarrow$	compound-word
simple-word	$\longrightarrow$	lower-alpha | upper-alpha | mixed-case | numeric | alphanumeric
mixed-case	$\longrightarrow$	(upper-alpha | lower-alpha) (upper-alpha | lower-alpha)$^+$
alphanumeric	$\longrightarrow$	numeric (lower-case | upper-case)$^+$
alphanumeric	$\longrightarrow$	(lower-case | upper-case)$^+$ numeric
compound-word	$\longrightarrow$	number-compound
compound-word	$\longrightarrow$	apostrophed-word
number-compound	$\longrightarrow$	{currency-symbol} numeric {comma numeric} {full-stop numeric}
currency-symbol	$\longrightarrow$	dollar
apostrophed-word	$\longrightarrow$	compound-word close-quote {compound-word}
apostrophed-word	$\longrightarrow$	close-quote compound-word
hyphenated-compound	$\longrightarrow$	nohyphen-word (hyphens nohyphen-word)$^+$

Figure 7: The grammar for word tokens

compound-punct: these are tokens made up of parentheses along with some combination of spacing and constituent-delimiter tokens. Note that we don't consider constituent-delimiter tokens to be compound-punct tokens in this sense.

Some points to note about the grammar in Figure 8:

1. It does not include punctuation tokens that contain square, curly, or angle brackets: these are omitted for simplicity, but would be treated in the same way as the open-paren and close-parens.

2. The treatment of open-parens allows the possibility that the open-paren may not be preceded by a space; this is required in order to deal with text-initial open-parens, but it has the consequence that word-internal open-parens will be labelled as non-word-internal; also, cases where the space is missing by accident will not cause the parser to fail. The same comments apply to the treatment of close-parens, except that in this case it is the following space that may not be present.

3. For simplicity, we don't specify any rules that cover backslash, tilde, at-sign, underscore, plus, equals, pipe, caret, or asterisk; these characters do not appear in the text we are using as our example.

8 Conclusions

We have described a general architecture for the parameterisable tokenisation of free-form texts, and provided details of the grammars required for one particular text type we have analysed.

The key features of the approach described here are as follows:

- The approach is completely parameterisable, so that different notions of what it is to be a word can be adopted in different contexts.
- Tokenisation is separated into two distinct steps, referred to here as BUNDLING and COMPOUNDING.
- This separation allows us to specify bundling as a deterministic, finite-state process that is cheap to implement; any more context-sensitive or intelligent processing is localised in the compounding stage.

The results of such a process are then available for further processing by more sophisticated tools, whether these be document-processing based or natural language processing-based. For example, in some recent work carried out in conjunction with the University of Sydney, we have attached a freely available morphological analysis module to a tokeniser based on the principles described here; the results of tokenisation are then further extended with morphological information.

The result is a considerably more sophisticated notion of what it is to be a word, a step we believe to be of paramount importance in bridging the divide between text processing and natural language processing.

Acknowledgements

The ideas expressed here have benefitted from discussions with Shona Douglas, Jason Johnston, Chris Manning, and Colin Matheson; all errors remain the author's own.

punct	⟶	spacing |
		constituent-delimiter |
		compound-punct
spacing	⟶	space |
		line-feeds {space} |
		tabs |
		line-feeds {tabs} |
		hyphens |
		space hyphens space
constituent-delimiter	⟶	(full-stop | comma | exclamation-mark | question-mark |
		colon | semi-colon) spacing
compound-punct	⟶	{spacing} (open-paren | open-quote)
compound-punct	⟶	(close-paren | close-quote) {(spacing | constituent-delimiter)}

Figure 8: Rules for Punct tokens

References

R Dale (1990) A Rule-based approach to Computer-Assisted Copy Editing. In *Computer Assisted Language Learning*, 2, pp59–67.

R Dale and S Douglas (1996) Two Investigations into Intelligent Text Processing. Pages 123–145 in *The New Writing Environment*, edited by Mike Sharples and Thea van der Geest. Springer, London.

G Grefenstette (1994) *Explorations in Automatic Thesaurus Discovery.* Kluwer Academic Publishers, Dordrecht.

C A Matheson and R Dale (1993) BibEdit: A Knowledge-Based Copy Editing Tool for Bibliographic Information. In E S Atwell (ed), *Knowledge at Work in Universities: Proceedings of the Second Annual Conference of the Higher Education Funding Councils' Knowledge Based Systems Initiative.* Cambridge, December 1993.

G Nunberg (1990) *The Linguistics of Punctuation.* CSLI/University of Chicago Press.

D Palmer and M Hearst (1994) Adaptive sentence boundary disambiguation. In *Proceedings of 4th ACL Conference for Applied Natural Language Processing*, Stuttgart, October 1994.

S M Shieber (1986) *An Introduction to Unification-based Approaches to Grammar.* The University of Chicago Press, Chicago, Illinois.