6

A Method for Tokenizing Text

RONALD M. KAPLAN

6.1 Introduction

The stream of characters in a natural language text must be broken up into distinct meaningful units (or tokens) before any language processing beyond the character level can be performed. If languages were perfectly punctuated, this would be a trivial thing to do: a simple program could separate the text into word and punctuation tokens simply by breaking it up at white-space and punctuation marks. But real languages are not perfectly punctuated, and the situation is always more complicated. Even in a well (but not perfectly) punctuated language like English, there are cases where the correct tokenization cannot be determined simply by knowing the classification of individual characters, and even cases where several distinct tokenizations are possible. For example, the English string *chap*. can be taken as either an abbreviation for the word *chapter* or as the word *chap* appearing at the end of a sentence, and *Jan*. can be regarded either as an abbreviation for *January* or as a sentence-final proper name. The period should be part of the word-token in the first cases but taken as a separate token of the string in the second. As another example, white-space is a fairly reliable indicator of an English token boundary, but there are some multi-component words in English that include white-space as internal characters (e.g. *to and fro*, *jack rabbit*, *General Motors*, *a priori*).

These difficulties for English are relatively limited and text-processing applications often either ignore them (e.g., simply forget about abbreviations and multi-component words—there are many more difficult problems to worry about) or treat them with special-purpose machinery. But this is a much bigger problem for other languages (e.g. Chinese text is very poorly

Inquiries into Words, Constraints and Contexts.
Antti Arppe et al. (Eds.)

punctuated; the Greek on the Rosetta stone has no spaces at all) and they require a more general solution.

This paper describes just such a general solution to the problem. We characterize the tokenizing patterns of a language in terms of a regular relation T that maps any tokenized-text, a string of characters with token-boundaries explicitly marked, into the concrete string of characters and punctuations marks that would be an expression in proper typography for the given sequence of tokens. For example, it maps the tokenized-text

(1) Happily TB , TB he TB saw TB the TB jack rabbit TB in TB Jan. TB .

into the actual text

(2) Happily, he saw the jack rabbit in Jan.

where TB is the explicit token-boundary marker. However, (1) is not the only tokenized-text that can be expressed concretely as (2). Alternative sources include texts in which there is a token-boundary between *jack* and *rabbit* (3a) and where the last word is taken as a proper name (3b).

(3) a. Happily TB , TB he TB saw TB the TB jack TB rabbit TB in TB Jan.
b. Happily TB , TB he TB saw TB the TB jack rabbit TB in TB Jan TB.

If *T* maps these (and perhaps other) alternative sources into the concrete text in (2), then its inverse T^{-1} will map the text in (2) back to the different explicitly tokenized sources in (1) and (3).

The correct tokenizing patterns for a language are thus defined by a regular relation or finite-state transducer, formal devices that have the power to characterize the complexity and ambiguity of punctuation conventions across the languages of the world. We describe a particular algorithm for applying such a transducer to a given text. This algorithm is a variant of the general composition algorithm for finite-state transducers, but it is specialized to the particular properties of text streams: they are usually quite long but they can be represented by finite-state machines with a single (acyclic) path. The algorithm uses this fact to provide efficient management of temporary storage and to provide guaranteed output for individual substrings of the text as rapidly as possible. The output is guaranteed in the sense that no data later in the text will cause the tokenization of a previous substring to be retracted as a possible tokenization—unless the text as a whole is ill-formed. Thus a client can safely invest its own computational resources for higher-order applications (text indexing, question-answering, machine translation…) without fear of wasting effort on incremental tokenizations that have no valid future.

6.2 Tokenizing Relations

A tokenizing relation can be defined for a particular language by a set of rules that denote regular relations (Kaplan and Kay, 1994), by a regular expression over pairs, or by the state-transition diagram of a finite-state transducer. For example, the following ordered set of obligatory context-sensitive rewriting rules defines a regular relation that gives a first-approximation account of English punctuation conventions:

(4) a. period TB $\rightarrow \varepsilon$ / __ period
b. TB $\rightarrow \varepsilon$ / __ right-punctuation
c. TB $\rightarrow \varepsilon$ / left-punctuation __
d. TB $\rightarrow$ space
e. space $\rightarrow$ white-space^{+}

The first rule deletes an abbreviatory (word-internal) period when it comes before a sentence-final period, as needed in the mapping of (1) to (2) above. The second rule causes token-boundaries to be deleted in front of punctuation marks that normally appear at the ends of words (e.g. closing quotes, commas) while the third deletes token boundaries after opening punctuation marks (e.g. open quotes). The fourth converts any remaing token-boundaries to space characters, and the fifth expands those spaces, as well as the internal spaces of multi-component tokens, to arbitrary sequences of white-space characters (space, carriage return). Other rules could be added to deal, for example, with optional end-of-line hyphenation, contractions, and sentence-initial capitalization. These rules are written in the generative direction, which may seem counterintuitive since the problem at hand is a recognition problem. But we have found that, as a general principle, such mappings are easier to characterize as reductions of more structured (i.e. explicitly tokenized) to less structured representations, with recognizers obtained by applying the relations in reverse. Grammars of this type can be compiled into finite-state transducers using the methods described by Kaplan and Kay (1994). Systems of two-level transducers (Koskenniemi, 1983) or two-level rules (Kaplan and Kay, 1994) can also be used to define regular tokenizing relations.

6.3 The General Idea

With such a relation in hand, the problem of finding the set of all admissible tokenizations of a text is simply an instance of the general problem of recognizing a text with respect to a transducer. If the text is interpreted as a (single-string) regular language, the solution to this problem is computed by the recognition operator Rec, defined as

(5) $\mathrm{Rec}(T, \mathit{text}) \stackrel{\mathrm{def}}{=} \mathrm{Dom}(T \circ \mathrm{Id}(\mathit{text}))$

This operator constructs the identity relation that takes the given text into itself, and composes that with the tokenizing relation. This has the effect of restricting the general tokenizer T to the subrelation that only has the given text as its output side. The domain of that restricted relation is exactly the set of tokenized strings that T would express as the given text (see Kaplan and Kay, 1994, for a discussion of the domain and identity relations and other relevant concepts). The result is a regular set (perhaps an infinite set if T is not linear bounded) that can be represented as a finite-state automaton. This formula is equivalent to Sproat's (1995) characterization of the tokenizing problem, modulo a transposition of the relational coordinates.

Formula (5) defines the computation we want to perform, and it is made up of operations (identity, composition, domain-extraction) that are easy to implement given a finite-state transducer presentation of T and a finite-state machine representation of the text. But the normal implementations of these operations would be quite impractical when applied to a long text. They tend to build intermediate data structures that are linear in the length of the text, and they would produce no results at all until the entire text has been processed. The standard algorithms may be acceptable if we are willing to preprocess a long text to break it up into sentence-size chunks and then operate on those one at a time. This is the arrangement contemplated by Sproat (1995), for example, but it requires additional heuristic machinery and may not deal gracefully with sentence-boundary ambiguities.

We describe here a method of evaluating this formula that is general and uniform and applies with practical efficiency to texts of arbitrary length. The method has four desirable properties:

1. It produces output incrementally, and does so whenever it reaches a point in the text where all local tokenization ambiguities have been resolved (so-called "pinch-points").
2. The temporary storage it requires is bounded by the maximum distance between pinch-points. Storage can be recycled in the computation between pinch-points when any path of ambiguity dies out.
3. It never causes previously produced output to be retracted, unless the text as a whole has no proper tokenization (is globally ill-formed).
4. It combines nicely with higher-level sources of lexical information, so that dictionary constraints on tokenization (e.g. a list of known abbreviations) can be taken into account without modifying the run-time algorithm. It can also be combined with syntactic constraints defined in grammatical formalisms that are closed under composition with regular relations. These include context-free grammars and Lexical Functional

Grammars (Kaplan and Bresnan, 1982).[1]

In a well-punctuated language, pinch-points come fairly close together (e.g. almost (but not always) at every punctuation mark). Thus, this algorithm is practical even for relatively simple languages like English, and it obviates the need to develop special-purpose code to take advantage of its limited tokenization patterns (or to provide less than accurate approximations). In a poorly punctuated language the pinch-points will not be as close together, so that reliable output will be provided less frequently and the amount of internal storage will be larger. But these effects are directly proportional to the inherent complexity of the language—it is extremely unlikely that there is a simpler way of handling the facts of the matter.

6.4 A Practical Method

We start from the general implementation of the composition, domain, and identity operations in the formula

$$\mathrm{Dom}(T \circ \mathrm{Id}(text))$$

This sort of formula is usually evaluated by operating on the finite-state transducer accepting T and the finite-state machine accepting the text. The composition algorithm is normally a quadratic operation bounded by the product of the number of states of its two operand transducers (in this case representing T and $\mathrm{Id}(text)$. It would produce an output transducer with states corresponding to pairs of states of T and $\mathrm{Id}(text)$, with the transitions from each of the pair-states encoding what the next moves are, given that that the particular pair of states in the T and $\mathrm{Id}(text)$ fsts has been reached. An indexing structure must be maintained, so that the data-structure representing a given pair-state can be obtained if that same pair of states is reached on alternative paths through T and $\mathrm{Id}(text)$. In this event, the forward moves previously determined for that pair-state are valid on the new path, and the alternatives can be merged together. This can happen when the composition of disjunctive and infinite relations (with cyclic fsts) is computed. Indeed, if the merger is not performed, then the composition computation may not terminate on cyclic input machines. This general algorithm is impractical, or at least unattractive, for tokenizing an arbitrary text because it would require an amount of indexing storage proportional to the length of the text.

[1] LFG's nonbranching dominance (off-line parsability) condition must be generalized slightly to preserve decidability of the membership problem in the case that the tokenizing relation T is not linear-bounded. A dominance chain must be regarded as nonbranching if a re-encountered category is paired not just with the same position of a single string but with the same suffix language of a tokenized regular set (as denoted by a state of the finite-state machine that represents the tokenization result).

Our tokenizing method circumvents these difficulties by finding correlations between segments of the text and segments of the tokenizing relation. We let T stand interchangeably for the regular relation or a transducer that represents it, as appropriate. We suppose that s is the start-state of T and, without loss of generality, that f is its single final state. If q and r are states of T, we define ${}_qT_r$ to be the set of all pairs of strings for which there is an accepting path in T that starts at state q and ends at state r. We also interpret the text itself as a finite-state automaton with numbered junctures between characters playing the role of its states. Thus ${}_i text_j$ is the substring of the text between positions i and j and the entire text of n characters is denoted as ${}_0 text_n$. We now say that

(6) A text-position k is a *pinch-point for* $\mathrm{Rec}(T, text)$ *with pinch-state* p iff

$$\mathrm{Rec}(t, text) = \mathrm{Rec}({}_sT_p, {}_0 text_k) \cdot \mathrm{Rec}({}_pT_f, {}_k text_n).$$

The raised dot denotes the usual concatenation operator for formal languages: the concatenation $L_1 \cdot L_2$ is the language $\{xy \mid x \in L_1, y \in L_2\}$. Thus, if k is a pinch-point and p is its pinch-state, any tokenization t of the text can be partitioned into two strings t_1 and t_2 such that t_1 is a tokenization of the subtext before position k provided by a path in T up to state p and t_2 is a tokenization for the rest of the text provided by a path in T leading from state p. If such a k-p pair can be identified, then the strings in the regular language $\mathrm{Rec}({}_sT_p, {}_0 text_k)$ can be supplied as incremental output to a client application, and the legal continuations of all those strings are completely determined by k and p and can be computed by evaluating the suffix tokenization-expression $\mathrm{Rec}({}_pT_f, {}_k text_n)$. The condition in (6) can be applied iteratively, so that a sequence of incremental outputs can be provided one after the other.

We see that the computation of $\mathrm{Rec}(T, text)$ can be suspended whenever a pinch-point is reached and that all tokenizations for the subtext to that point can be delivered as output. Moreover, the detailed pair-state indexing structures required by the composition operator can be discarded at such a point, and only two pieces of information, the position k and the pinch-state p, are needed to continue tokenizing the rest of the text. The challenge, of course, is to identify pinch-points and pinch-states at the earliest positions of the text; that is what our method for tokenizing text is organized to do.

We observe that the text itself is a linear string and so $\mathrm{Id}(text)$ is represented by a single-path transducer. This means that a composition path that has advanced beyond a particular text state/position will never return to that same text-position. Thus, the indexing structures necessary to find previous pair-states and carry out future mergers do not have to be maintained across

the whole text. In particular, if we carry out the state-traversal computation of the composition algorithm in a breadth-first manner, then we can recycle the storage for all the indexing structures involving a given text-position as soon as we have examined all possible next-moves at that position.

The next obvious observation is that we can simulate the identity map on the text by letting each character stand for its own image on the other tape of the (now fictional) identity transducer. We can implement the breadth-first composition in this special case by simply maintaining a list of configurations at a current character position. These configurations represent all the states in the transducer T that were paired up with that text position, and they record for eventual output the domain characters that were produced along the way.

In pursuing the future of a given configuration, we compare the transitions leaving its transducer state with the next character in the text. We build new configurations at the next-character position for those transitions that match the next text character. If a transition has an ε (empty string) so that it does not advance through the text, the new configuration is added to the end of the list of current-position configurations. However, before we add a new configuration to one of the lists, we check to see whether a configuration with the same transducer-state is already present. If such a configuration does exist, we add alternatives to its output record instead of adding a separate new configuration. This has the same effect as the merger step in the general composition algorithm: it collapses alternative paths and insures termination even if the T transducer is cyclic.

Thus, we maintain two lists of configurations: one for the current text position, one for the next. When we have exhaustively processed all the current position's configurations, we can recycle them to a free-list, move forward to the next position, and start processing the configurations there.

The output of this computation is given by the domain side of the T transitions that carry us across the text. We construct incrementally the transitions of the finite-state machine that represents this output by associating an output-state with each of the configurations. The transitions of this state are labeled with the domain transitions of matching arcs of T. A subtlety, however, is that these transitions point backwards, towards the output state of the configuration currently providing for the match against the text. We also maintain on each output-state a reference count of the number of (backwards) transitions that point to it. This enables us to implement a reference-counting garbage collector that collects the storage for failed paths as soon as possible. A path fails if there are no arcs at the T state of a configuration that match the current text-character. In that case, the (backward) transitions at the configuration's output state are collected, and the reference counts of the states they point to are decremented. If those counts go to zero, then those states and their backward-pointing arcs can also be reclaimed, recursively.

This arrangement will collect all unneeded output storage as soon as possible, provided that T itself is linear-bounded. If T is linear-bounded, then $\mathrm{Dom}(T \circ \mathrm{Id}(text))$ will be finite and there will be no loops in the output finite-state machine. Otherwise, there can be infinitely many tokenized strings, represented by a cyclic fsm, and there can be self-justifying loops that will not be incrementally collected when a cyclic path cannot later be extended. In that case, the loop-structures on failed paths will not be reclaimed until a pinch-point is reached, the current output fsm is provided back to the client, and all current-output structures are freed.

We note as an aside that a non-linear-bounded tokenizing relation producing infinitely many results is not entirely implausible. It is a convention of English text that appositives and nonrestrictive relative clauses are both set off by commas, and the appearance of commas is thus an important signal for the higher-level syntactic analysis of a text. Logically then, a nonrestrictive relative ending in an appositive should be followed by two commas, but a sequence of commas is always reduced to one by the rules of English typography. This reduction can be expressed by adding the following rewriting rule to the specification of T:

(7) comma TB $\rightarrow \varepsilon$ / __ comma

The resulting tokenizer will introduce an arbitrary number of tokenization commas in front of every comma that appears in the actual text, leaving it to a syntactic analyzer to pick the ones that satisfy grammatical constraints. This may not be helpful for unsophisticated uses of the output, but it can enable substantial simplifications of grammars for applications requiring a full syntactic or semantic analysis.

To finish describing our tokenizing method, we observe that there is an easy way of recognizing that a pinch-point has been reached. This is a point at which all open paths have come together at a single state of T, so that there is only one way of proceeding further ahead. This is marked by the fact that there is only one item on the list of current configurations. Several items are on the list if there are current ambiguities, one item remains when those ambiguities have a determinate future, and the list becomes empty only when the text itself is ill-formed. When there is only a single open configuration, the output trailing behind that configuration can be provided to the client, with the guarantee that the current output will not be retracted unless the text as a whole is ill-formed (this can happen only if the range of T is not Σ^*; it is decidable in advance whether T has this unattractive property). Thus, it is safe to provide output whenever a singleton configuration-list is reached. It may be desirable not to produce output in every such situation, but instead let the output accumulate internally until a token boundary is also reached. This means that the client will receive outputs only in chunks of meaningful

units. In a well-punctuated language, the output will come essentially at every word boundary. The pinch-state needed for tokenizing the rest of the text is the state of T stored in the single configuration.

The output can be produced by reversing the arcs of the output-fsm that has been threaded through the configuration, and then providing the result as an fsm structure to the client. Or the output-fsm can be copied to some other graph data structure as determined by the client, for example, the initial chart of a chart-parser for higher-level syntactic analysis.

6.5 Higher-level Lexical Constraints

The tokenizing relation T can be modified in various ways to incorporate higher-level constraints on tokenization and thus to eliminate inappropriate tokenizations at an early stage of processing. If these additional constraints can be expressed as regular relations or regular languages, they can be composed into the original formula to provide a more restrictive tokenization mapping. For example, suppose that the lexical relation L is regular and maps sequences of word stem-and-tag-strings separated by token boundaries into the corresponding inflected forms of the words (see, for example, Karttunen *et al.*, 1992). Then

(8) $\mathrm{Dom}(L \circ T \circ \mathrm{Id}(text))$

defines the morphological analysis of the tokenized text. Composition is an associative operation, so we can rewrite this as

(9) $\mathrm{Dom}((L \circ T) \circ \mathrm{Id}(text))$

Since L and T are independent of the text, the composition $L \circ T$ can be evaluated at compile-time to produce a new transducer LT that maps tokenized-stems to surface text. Applying our algorithm to LT will produce the tokenized morphological analysis of the text.

If a full lexical or morphological relation is not available or not desired, it may still be advantageous to augment the tokenizing relation with a small amount of additional information. This can be used to control the rampant ambiguity that comes from letting every space be construed as expressing either a token boundary or the internal space of a multi-component word, and from letting every period be interpreted as marking either a sentence boundary or an abbreviation. Suppose that M is a list of all the multi-component words that are to be allowed, that A is a list of known abbreviations, and that X is the regular set of all strings not containing either space or period. Define the regular language XAM to be the union of X, A, and M. The strings in this language do not contain spaces or periods unless they are in M or A. We construct $[(XAM\ \mathrm{TB})^* XAM]$, allowing for the elements of XAM to be packed together with token boundary separators. The relation

(10) $\mathrm{Id}([(XAM\ \mathrm{TB})^* XAM]) \circ T$

can then be used as a tokenizing relation that produces space/period ambiguities only for the strings contained in M or A.

6.6 The Best Tokenization

Even with higher-level lexical constraints there may be many more alternative tokenizations than are desired, and many of them may be quite improbable. Sproat (1995) suggests a different approach to selecting the most appropriate outputs. He proposes to characterize the tokenization relation as a weighted regular relation, where each mapping is associated with a weight, perhaps a probability, that measures its degree of acceptability. The idea is to provide the output produced along the best, most highly weighted path through the transducer. It is easy to accommodate this proposal within our pinch-point framework, since the best-path operator BP distributes over concatenation (11) and thus distributes over our pinch-points (12):

(11) $\mathrm{BP}(L_1 \cdot L_2) = \mathrm{BP}(L_1) \cdot \mathrm{BP}(L_2)$

(12) $\mathrm{BP}(\mathrm{Rec}(t, text)) = \mathrm{BP}(\mathrm{Rec}({}_sT_p, {}_0text_k)) \cdot \mathrm{BP}(\mathrm{Rec}({}_pT_f, {}_ktext_n))$

We simply provide at each pinch-point the output corresponding to the best path rather than the full set of tokenizations. This will produce the best tokenization of the entire text.

References

Kaplan, R. and Bresnan, J. 1982. Lexical Functional Grammar: A formal system for grammatical representation. In J. Bresnan (ed.), *The mental representation of grammatical relations.* Cambridge: MIT Press, 173-281.

Kaplan R. and Kay, M. 1994. Regular models of phonological rule systems. *Computational Linguistics*, 20(3), 331-378.

Karttunen, L., Kaplan, R., and Zaenen, A. 1992. Two-level morphology with composition. In *COLING'92*, Nantes, France, 141-148

Koskenniemi, K. 1983. *Two-level morphology: A general computational model for word-form recognition and production.* Publication 11, University of Helsinki, Department of General Linguistics, Helsinki.

Sproat, R. 1995. A finite-state architecture for tokenization and grapheme-to-phoneme conversion for multilingual text analysis. In S.Armstrong and E. Tzoukermann (eds.), *Proceedings of the EACL SIGDAT Workshop*, Dublin, Ireland, 65-72.