Finite automata for line breaking and segmentation

To: UTC/PAG
From: Robert Bastian, Robin Leroy
Date: 2026-07-27

Contents

Proposed changes to UAX #14 2
9.3 Implementation using a Deterministic Finite Automaton 2
9.3.1 Structure 3
9.3.2 Pseudocode 4
9.3.3 Tailorings 6
9.3.4 Complex context-dependent break determination 7
Proposed changes to UAX #29 8
6.3 State Machines 8
Rationale / Q&A 9
Why do we need this in the Standard? Could it be published as a UTN? 9
Why not publish line.txt? 10
Aside: the RBBI rule syntax 11
Why include support for the CSS tailorings in the DFA? 12
Do we need the lookaheads? 12
Acknowledgements 13
References 13

Summary

Like the Unicode Bidirectional Algorithm, the line breaking algorithm is performance-critical: text cannot be displayed before it has run. Whereas the rules of the UBA are very stable[1], those of the line breaking algorithm change regularly to improve the support of typographical rules for various writing systems and languages. This means that implementers need to change their line breaking implementation every year, handling additional complexity without losing performance.

The algorithm as specified is extremely slow: it involves running up to 80 regex matches for every position in the text. UAX #14 used to provide a fast implementation in the form of a space-aware pair table. This

[1] Excluding clarifications, the last change to the rules was in Unicode Version 8.0.0 (2015) to address some edge cases, and the last major change was in Unicode Version 6.3.0 (2013).

approach ceased to be compatible with the increasing complexity of the rules, and the pair table was removed in Unicode Version 10.0 (2017). No replacement was provided.

In the nine years since, with the exception of ICU (ICU4C and ICU4J), attempts at performant implementations of line breaking have suffered from issues of correctness and maintainability; even ICU4X, published by the Unicode Consortium, has routinely included an implementation of the line breaking algorithm that was correct but a year or two out of date.

Since Unicode Version 11.0 (2018), UAX #29 has had the following text:

> The conversion into a regular expression is very straightforward for grapheme cluster boundaries. It is not as easy to convert the word and sentence boundaries, nor the more complex line boundaries [UAX14]. However, it is possible to also convert their rules into fast, deterministic finite-state machines that will recognize all the same boundaries that the rules do. The implementation of text segmentation in the ICU library follows that strategy.

Replicating ICU's conversion of these complex rules into state machines was, however, left as an exercise to the reader.

Indeed, for the past 23 years, ICU has been using deterministic finite automata (DFAs) to determine break opportunities and segmentation boundaries. We propose adding a description of the DFAs to UAX #14, and publishing the data underlying these DFAs in the data files associated with UAXes #14 and #29.

We define a tailoring mechanism based on remapping of characters to different DFA symbols (similar to tailoring property assignments in the UAX #14 formulation), instead of requiring changes to the state machine itself (similar to changing the rules in the UAX #14 formulation). This mechanism can be used by CLDR for language-specific tailorings, and for CSS tailorings. Indeed, several tailorings of line breaking are required by CSS and therefore implemented by major browsers. Their implementation using the tailoring mechanism significantly reduces the required data size compared to needing different DFAs for the tailored rules, and reduces the overall maintenance burden (only one line breaking DFA needs to be maintained, instead of nine as had ICU so far).

Proposed changes to UAX #14

In Unicode Standard Annex #14, Unicode Line Breaking Algorithm, add a section as follows.

9.3 Implementation using a Deterministic Finite Automaton

Given that it is often necessary to apply the line breaking algorithm before text can be first displayed, an implementation that follows the definition of the algorithm in *Section 6, Line Breaking Algorithm*, would be unacceptably slow: at each position in the text, a regular expression match on the context before and after needs to be run for each rule until one rule matches. Further, unless one changes all subsequent rules to incorporate LB10 as described in *Section 9.1, Combining Marks in Regular Expression-Based Implementations*, the text buffer used to determine the applicability of the rules must also be modified from the original string.

This, together with the associated data files, defines a deterministic finite automaton (DFA) augmented with lookaheads which determines line break opportunities in a single pass through the text.

The same basic implementation can be used for the segmentation algorithms defined in Unicode Standard Annex #29, "Unicode Text Segmentation" [UAX29].

Review Note: During this public review period, the data files for state machines implementing the line breaking and segmentation algorithms are available at:
https://unicode.org/review/pri555/17.0.0/ for Version 17.0 of these algorithms;
https://unicode.org/review/pri555/18.0.0/ for Version 18.0 of these algorithms.
The files defining the untailored line breaking state machine are LineBreakSymbols.txt, LineBreakStates.txt, and LineBreakTransitions.txt.
If this proposal is accepted, files for subsequent versions of the Standard would likely be published at https://unicode.org/Public/<version>/ucd/auxiliary/.

9.3.1 Structure

The DFA operates on *symbols*, which are defined as sets of code points and strings. The symbols define a partition of the code space: each code point is in exactly one symbol. The symbols are distinct from the line break classes: classes that behave identically are merged into one symbol, and classes that contain characters whose behaviour differs because of other properties are split. The file LineBreakSymbols.txt defines the symbols. The symbols are defined by UnicodeSet expressions. They may contain multi-character strings; these are used for the end of text marker (see *Section 9.3.2, Pseudocode*) and in the definition of tailorings (see *Section 9.3.3.1, Shadow Symbols*).

> **Example:** The following line from LineBreakSymbols.txt defines a symbol with the name "BK|NL|eot" containing those characters with the property Line_Break=Mandatory_Break, those with Line_Break=Next_Line, and the string "eot":
>
> ```
> BK|NL|eot ; [[\p{Line_Break=Mandatory_Break}] [\p{Line_Break=Next_Line}] [[{eot}]]] ;
> ```

The names of the symbols are generally mnemonic of their composition in terms of line break classes and other properties, but should be treated as arbitrary opaque strings by implementations, and would likely be replaced by indices in any optimized implementation. In particular, the names of symbols are not default identifiers.

The DFA has *states*, as well as *lookaheads*. Each state has a name, and each lookahead has a name. Each state also has:

- an `Accepting` value. This may be Yes, No, or the name of a lookahead.
- optionally, a `Lookahead` value, which is the name of a lookahead
- a `Break_Type` value. For line breaking, the `Break_Type` field may be the empty string (denoting a break opportunity) or the string "Mandatory" (denoting a mandatory break).

The states and their properties (including lookaheads) are defined in the file LineBreakStates.txt.

As for the symbols, the names of states and lookaheads are mnemonics of the sequence of symbols encountered to get to the state or the lookahead, but should be treated as opaque strings, and are not default identifiers. In particular, lookahead names contain slashes; these indicate the position of the break in the sequence: a lookahead X Y / Z matches when the DFA encounters the sequence of symbols X Y Z, and indicates a break after X Y. The sole exception is the state with the name START: this name identifies the start state.

> **Examples:** The following line from LineBreakStates.txt defines a state with the name "BK", an `Accepting` value of Yes, no `Lookahead` value, and a `Break_Type` value of Mandatory.
>
> ```
> BK ; Yes ;  ; Mandatory
> ```
>
> The following line defines a state with the name "SP IS", an `Accepting` value of No, a `Lookahead` value of "ZW / CM", and no `Break_Type` value:

```
SP IS ; No ; ZW / CM ;
```

The following line defines a state with the name "SP IS", an `Accepting` value of No, a `Lookahead` value which is the lookahead name "ZW / CM", and no `Break_Type` value:

```
SP IS ; No ; ZW / CM ;
```

The following line defines a state with the name "ZW CM", an Accepting value which is the lookahead name "ZW / CM", and no Break_Type value:

```
ZW CM ; ZW / CM ;  ;
```

Finally, for each state s and each symbol σ, there may be a *transition from s on σ*, `Transition (`s`, `σ`)`, which is a state. The transitions are defined in LineBreakTransitions.txt.

Example: The following line from LineBreakTransitions.txt defines a transition from state "BK" on symbol "LF". The value of `Transition (`"CR"`, `"LF"`)` is state "BK":

```
CR ; LF ; BK
```

In practice, the transitions would be stored in an array indexed by states and symbols, with a special value (not used as a state index) indicating a lack of transition.

9.3.2 Pseudocode

The line breaking algorithm is then implemented as shown on the next page. Note that the DFA moves from one break to the next, in other words, it recognizes the break-to-break language: this means that when applied to a segmentation algorithm, it recognizes the segments (grapheme clusters, words and inter-word segments, or sentences).

```
1.  -- Adds a Text_Position to Breaks if there is a break before the code point at that position in
2.  -- Document, associating it with a Break_Type.
3.  -- For a break at the end of text, the position past the end of Document is added to Breaks.
4.  procedure Get_Breaks(
5.     Document : in Text
6.     Breaks   : out map from Text_Position to Break_Type)
7.  is
8.    Last_Break      : Text_Position := Document .First;
9.  begin
10.   Find_All_Breaks:
11.   while Last_Break ≠ the position past the end of Document loop
12.     Run_DFA:
13.     declare
14.       S                       : State := START;
15.       Symbol_Ahead            : Symbol;
16.       Last_Accepting_Position : Text_Position;
17.       Last_Break_Type         : Break_Type;
18.       Lookahead_Positions     : map from Lookahead to Text_Position
19.                               := empty map;
20.       I                       : Text_Position := Last_Break;
21.     begin
22.       Find_Next_Break:
23.       loop
24.         if I = the position past the end of Document then
25.           Symbol_Ahead := The symbol containing the string "eot";
26.         else
27.           Symbol_Ahead := The symbol containing the code point at position I in Document;
28.           Advance I by one code point in Document;
29.         end if;
30.         if Transition (S, Symbol_Ahead) exists then
31.           S := Transition (S, Symbol_Ahead);
32.         else
33.           Last_Break := Last_Accepting_Position;
34.           exit loop Find_Next_Break;
35.         end if;
36.
37.         if S.Accepting = Yes then
38.           Last_Accepting_Position := I;
39.           Last_Break_Type         := S.Break_Type;
40.         elsif S.Accepting is a Lookahead and
41.               Lookahead_Positions (S.Accepting) exists then
42.           Last_Break      := Lookahead_Positions (S.Accepting);
43.           Last_Break_Type := S.Break_Type;
44.           exit loop Find_Next_Break;
45.         end if;
46.
47.         if S.Lookahead exists then
48.           Lookahead_Positions (S.Lookahead) := I;
49.         end if;
50.       end loop Find_Next_Break;
51.
52.       Breaks (Last_Break) := Last_Break_Type;
53.     end Run_DFA;
54.   end loop Find_All_Breaks;
55. end Get_Breaks;
```

9.3.3 Tailorings

9.3.3.1 Character Mappings

Many line break and segmentation tailorings can be expressed as "treat character X like character Y". For instance, a sentence break tailoring catering to Greek could treat U+003B SEMICOLON and its canonical equivalent U+037E GREEK QUESTION MARK like U+003F QUESTION MARK. Such tailorings are described in machine-readable form by sequences of lines of the form *source* ; *target*, where *source* is an arbitrary UnicodeSet expression, and *target* is a UnicodeSet expression defining a set with a single element.

In this syntax, the tailoring mapping the semicolon and its canonical equivalent to the question mark can be described with the single line

```
[\N{SEMICOLON} \N{GREEK QUESTION MARK}] ; [?]
```

Tailorings that map characters to other characters are easily accommodated in the DFA implementation: they define a change to the partition of the code space into symbols. For instance, when the CSS line-break property is set to 'normal', characters of line break class CJ are treated as if they had line break class ID; this can be expressed as a character mapping, where U+3042 あ HIRAGANA LETTER A has Line_Break=ID:

```
\p{lb=CJ} ; [あ]
```

Such tailorings change the assignments of line break classes or other properties, but not the rules in *Section 6, Line Breaking Algorithm*.

9.3.3.2 Shadow Symbols

In general, tailorings that require changes to the rules require changing the DFA, often resulting in very different states and transitions. The CSS specification describes some line break tailorings that many implementers need to support, some of which correspond to changes in the rules. The DFA defined by the data files associated with this annex supports these modified rules using *shadow symbols*. These are symbols that contain no code points; these symbols are never encountered by DFA in an untailored implementation. The shadow symbols contain strings which can be used to refer to them in the target of a tailoring.

For example, when the CSS line-break property is set to 'loose', breaks between characters of line break class IN are allowed; this corresponds to changing rule LB22 from × IN to [^IN] × IN. The DFA instead operates as if there were an additional line break class LOOSE_IN, and as if were two separate rules, × IN and [^LOOSE_IN] × LOOSE_IN, where the line break class LOOSE_IN is empty; the CSS 'loose' tailoring of characters of line break class IN then consists in changing the assignment of these characters to line break class LOOSE_IN. The shadow symbol corresponding to the empty class LOOSE_IN is defined as follows:

```
LOOSE_IN ; [{LOOSE_IN}]
```

This symbol contains no code points; it contains the magic string "LOOSE_IN". The following tailoring treats all characters with Line_Break class IN as if they were in the symbol LOOSE_IN.

```
\p{lb=IN} ; [{LOOSE_IN}]
```

9.3.3.3 CSS tailorings

Tailorings files implementing the CSS line-break "normal" and "loose" options are provided.

Review Note: During this public review period, tailorings for Chinese and Japanese versions of all three CSS line-break values ("strict", "normal", and "loose") are also provided.
The files are available at https://unicode.org/review/pri555/17.0.0/ for Version 17.0 of the line breaking algorithm and https://unicode.org/review/pri555/18.0.0/ for Version 18.0. The file names are given by the following table:

	Languages: zh, ja	Other languages
line-break: strict	LineBreakTailoring_cj.txt	(not tailored)
line-break: normal	LineBreakTailoring_normal_cj.txt	LineBreakTailoring_normal.txt
line-break: loose	LineBreakTailoring_loose_cj.txt	LineBreakTailoring_loose.txt

Note that in this table, each _cj file is a subset of the non-cj counterpart to its right, and each file is a subset of the file below it. This may be formalized as an inheritance relationship in a future version of this proposal to reduce duplication and improve maintainability.

If this proposal is accepted, files for subsequent versions of the Standard would likely be published at https://unicode.org/Public/<version>/ucd/auxiliary/ for the language-independent tailorings. The language-specific tailorings would probably be handled by CLDR.

9.3.4 Complex context-dependent break determination

The symbols and transitions given in LineBreakSymbols.txt and LineBreakStates.txt incorporate the default from rule LB1, treating characters with class SA as if they had class AL or CM depending on their general category. However, implementations that support dictionary-based line breaking of scripts with Line_Break class Complex_Context (SA) need to detect characters that require dictionary-based handling. In order to facilitate that, the symbols for characters of class SA are distinct from the symbols for characters of classes AL and CM, and they are marked by the presence of a third field in LineBreakSymbols.txt, which is a symbol name; that symbol is the *non-dictionary equivalent symbol*. A symbol which has a non-dictionary equivalent symbol is a *dictionary symbol*.

> **Example:** The following line in LineBreakSymbols.txt defines the symbol "SA_Mn|SA_Mc" as the set of characters with Line_Break=Complex_Context and General_Category either Nonspacing_Mark or Spacing_Mark. It has a non-dictionary equivalent symbol, which is the symbol "CM" (see rule LB1). "SA_Mn|SA_Mc" is therefore a dictionary symbol.
>
> ```
> SA_Mn|SA_Mc ; [[[\p{lb=SA}&\p{gc=Mn}]] [[\p{lb=SA}&\p{gc=Mc}]]] ; CM
> ```

An implementation that supports dictionary-based breaking can trigger complex context-dependent line breaking when it encounters a dictionary symbol. An implementation that does not support dictionary-based breaking, or that supports it only on a subset of dictionary symbols, can move unsupported characters from a dictionary symbol to its non-dictionary equivalent symbol; the behaviour of the state machine is then retained under that mapping.

> **Note:** In some cases, the fallback rule-based behaviour of dictionary symbols is different from that of any non-dictionary symbol. In those cases, the non-dictionary equivalent symbol is the dictionary symbol itself.

Proposed changes to UAX #29

Amend Section 6.3, *State Machines*, as follows:

6.3 State Machines

The rules for grapheme clusters can be easily converted into a regular expression, as in *Table 1b, Combining Character Sequences and Grapheme Clusters*. It must be evaluated starting at a known boundary (such as the start of the text), and it will determine the next boundary position. The resulting regular expression can also be used to generate fast, deterministic finite-state machines that will recognize all the same boundaries that the rules do.

For more information on Unicode Regular Expressions, see Unicode Technical Standard #18, "Unicode Regular Expressions" [UTS18].

The conversion into a regular expression is very straightforward for grapheme cluster boundaries. It is not as easy to convert the word and sentence boundaries, nor the more complex line boundaries [UAX14]. However, it is possible to also convert their rules into fast, deterministic finite-state machines that will recognize all the same boundaries that the rules do. ~~The implementation of text segmentation in the ICU library follows that strategy.~~ Such state machines are defined by the following files, in the format described in Section 9.3, *Implementation using a Deterministic Finite Automaton*, of [UAX14]:

Boundaries	**Files**
Extended grapheme cluster	GraphemeClusterBreakSymbols.txt, GraphemeClusterBreakStates.txt, GraphemeClusterBreakTransitions.txt
Word	WordBreakSymbols.txt, WordBreakStates.txt, WordBreakTransitions.txt
Sentence	SentenceBreakSymbols.txt, SentenceBreakStates.txt, SentenceBreakTransitions.txt

The break types of the word break state machine characterize the preceding segment: if the type is "Letter", the preceding segment is word-like—such as a word consisting of letters—; if the type is "Number", the preceding segment is number-like—such as a number consisting of digits. If there is no type, the preceding segment is an interword segment, such as spaces and punctuation. This classification can be used to extract the words from a piece of text.

The break types of the sentence break state machine characterize the mode of sentence termination: if the break type is "Nonterminated", no sentence terminal punctuation was found, and instead the paragraph ended without a final full stop. If a sentence terminal punctuation was found, there is no break type.

The grapheme cluster breaking state machine does not make use of break types.

These implementations can be used with tailorings expressed using the mechanisms defined in Section 9.3.3, *Tailorings*, of [UAX14].

> Review Note: During this public review period, a tailoring of sentence breaking for Greek is provided as SentenceBreakTailoring_el.txt. It will likely be moved to CLDR in a future revision.

~~For more information on Unicode Regular Expressions, see Unicode Technical Standard #18, "Unicode Regular Expressions" [UTS18].~~

Rationale / Q&A

Why do we need this in the Standard? Could it be published as a UTN?

When an early draft of this proposal was first brought to the Properties & Algorithms Group, a concern was raised that it would create an additional burden on the UTC as part of the release, for something that is not a necessary part of the release; and that optimization tricks are normally out of scope for the standard. A UTN was suggested instead.

There is precedent for publishing an optimized implementation as part of the standard here: from 1999 to 2017, UAX #14 included an efficient implementation of line breaking in the form of a space-aware pair table (Section 7). There is also precedent for removing such an implementation if it becomes unmaintainable: Section 7 was eventually removed[2].

When it comes to maintenance burden, it is worth noting that changes made to the Unicode line breaking algorithm need to be implemented efficiently by Unicode Consortium projects (ICU and ICU4X). For the past twenty-four years, this has required ICU maintainers (Andy Heninger for eighteen years, now Robin Leroy) to produce tables of the form being proposed here: this burden already exists at the Consortium level. Importantly, when implementability issues are encountered when producing these tables, these come back to the post-beta UTC with requests for last-minute changes[3]: the ability to produce these tables is already a constraint placed on the UTC, but because it is assessed by a different TC during beta review, it creates churn for the UTC at an inopportune time.

Most importantly, the lack of implementation guidance for the line breaking algorithm clearly hampers adoption. Some experiments with Notepad on Windows 11 show that this editor does not implement LB21a (Unicode 6.1, 2012), Unicode 8.0 changes to LB22, etc.: this implementation has not tracked most changes to the Unicode line breaking algorithm for the past fifteen years. As the algorithm becomes more complex to handle more advanced typography[4], implementations other than ICU find themselves unable to follow these changes. In fact, we are unaware of any conformant performant open-source implementation of the line breaking algorithm outside the Consortium since Unicode Version 13.0 (2020)[5]. Even within the Consortium, ICU4X's attempt at an implementation independent from ICU's led to serious maintainability issues:

- major user-visible[6] conformance issues were found in the Unicode 15.0 implementation, which was made conformant[7] only at the end of November 2023: two months after the Unicode 15.1 release, and fourteen months after the Unicode 15.0 release;
- the implementation was updated[8] to 15.1 only in September 2024, a year after the Unicode 15.1 release;

[2] 147-A79.

[3] See, e.g., 176-C27, 180-C18.

[4] Recently: aksara breaking for Brahmic scripts in 15.1, better heuristics for simplified Chinese quotation marks in 16.0, improved handling of Hebrew hyphens in 17.0, improved handling of French quotation marks in 15.1 and dashes in 18.0.

[5] A state-machine based implementation in Rust, unicode-linebreak, was created for Unicode Version 12.1; its 13.0 update was incomplete, failing to take the changes to LB30 into account—and since this discrepancy is not documented, the implementation is not conformant. No attempt has been made at updating this implementation to Unicode Version 15.1 and later.

[6] Including a line break in the middle of 🏳️‍🌈.

[7] By https://github.com/unicode-org/icu4x/pull/4389.

[8] By https://github.com/unicode-org/icu4x/pull/5218.

— the 16.0 update never happened;
— the 17.0 update[9] is still in progress as of July 2026.

A standard that no-one adopts is a failed standard; if UAX #14 is worth maintaining, it is worth maintaining what is needed for people to actually implement it.

Why not publish line.txt?

The DFA is generated from a source file in written in a domain-specific language based on lex [LS79], describing the break-to-break language in terms of Unicode-aware regular expressions; the original version of that source file, in the ICU code base, is called line.txt; the tool that produces the DFA is the *RBBI*[10] *table builder*[11]. We propose publishing the DFA, and a specification for how to find breaks using the DFA, instead of line.txt and a specification for the RBBI table builder. There are three reasons for that:

1. The RBBI table builder is vastly more complex than the break iterator itself; requiring implementers to implement a custom DFA generator would be an unnecessary burden.
2. The RBBI table builder changes more frequently than the actual logic of the break iterator. Indeed, it underwent some nontrivial changes during the preparation of this proposal, which resulted in more compact tables, but did not change the logic of break iteration, which has last changed more than twenty years ago.
   It is likely that further changes will be made to the RBBI table builder and line.txt rule syntax in order to improve the maintainability of the rules (see the aside below), whereas there is no foreseeable reason to change, in future versions, the logic of break iteration, nor the syntax of the artefacts proposed in this document.
3. The DFA is, in a sense, "more canonical" than the line.txt rules. Specifically, in the absence of lookaheads and of use of the eot symbol (for instance, for the grapheme cluster breaking and word breaking rules), there is, up to isomorphism, a unique minimal DFA recognizing a given regular language, which can be derived from any regular expression that recognizes that language, see [ASU86, pp. 135 *sqq.*] or [Moo56]. However, constructing a minimal or even approximately minimal regular expression for a given regular language is PSPACE-hard, see [JR93] and [GS07]; in practice, line.txt is a somewhat arbitrary[12] description of the language.
   In the presence of the eot symbol (which means that two symbol sequences, with and without eot, can correspond to the same string), and of lookaheads (which mean the DFA does more than recognizing a regular language), the uniqueness theorem does not apply; nevertheless, it is clear that line.txt reflects the preferences of its maintainer a lot more than the resulting DFA.
   Regular expressions and grammars are valuable when they provide insights into the structure of the language they define; this is for instance the case of the regular grammar for grapheme clusters given in UAX #29. However, there is no known meaningful internal structure to strings that are unbreakable under the line breaking algorithm.

[9] First failed attempt: https://github.com/unicode-org/icu4x/pull/7823; current attempt: https://github.com/unicode-org/icu4x/pull/8041.

[10] Rule-Based Break Iterator.

[11] See https://github.com/unicode-org/icu/blob/main/icu4c/source/common/rbbitblb.cpp and https://github.com/unicode-org/icu/blob/main/icu4j/main/core/src/main/java/com/ibm/icu/text/RBBITableBuilder.java.

[12] Because of the chaining mechanism, line.txt really contains sliced-up regular expressions, and the choice of slicing is also somewhat arbitrary. Some rules, such as LB28a and LB25, are expressed as large regular expressions, whereas others are broken up in smaller chunks corresponding to the individual UAX #14 sub-rules.

Aside: the RBBI rule syntax

The RBBI table builder is, effectively, a Unicode-aware version of the lex scanner generator [LS79], implemented as described in [ASU86], with one twist.

As in lex, regular expressions are used to describe "tokens" (in this context, strings that are unbroken), and lookaheads (using the syntax "`/`") can be used to recognize tokens based on context ahead. However, the `!!chain` feature is unique: if strings $s_1 s_2 \ldots s_k$ and $s_k s_{k+1} \ldots s_n$ are both tokens, their *chaining* on s_k, *i.e.*, $s_1 s_2 \ldots s_k s_{k+1} \ldots s_n$, becomes a token too. This makes it straightforward to describe rules of the form X × Y (keep X and Y together). They correspond to token definitions `X  Y;`, and the chaining takes care of building up longer sequences from such kept-together pairs. The rules with one-sided context, e.g., × Z, require some care, as preceding break rules in the algorithm, e.g., T ÷, mean that they are not expressible as `. Z;` (keep anything and Z together), but bookkeeping of sets of unbroken characters after a given break rule, *e.g.*, `[^T] Z;`, make these tractable. The "treat as" rule LB9 is implemented as described in UAX #14, hence the swarm of `$CM*` in the examples below, where we have deemphasized them to make the regular expressions easier to read.

Rules that involve longer context are often straightforward, and sometimes (though not always) clearer than in UAX #14, as long as that context is unbreakable; for instance, the rule LB28a for orthographic syllables is a single regular expression matching a syllable,

```
($AP $CM*)?
($AS | $AK | [◌] ) ($CM* $VI $CM* ($AK | [◌] ))*
( $CM* $VI | (($CM* ($AS | $AK | [◌] ))? $CM* $VF) )?;
```

whereas it consists of four separate keep-together rules in UAX #14:

AP × (AK | [◌] | AS)
(AK | [◌] | AS) × (VF | VI)
(AK | [◌] | AS) VI × (AK | [◌])
(AK | [◌] | AS) × (AK | [◌] | AS) VF

The rule LB25 for numbers is even more striking; it is a single regular expression for numbers, which is mentioned in the standard:

```
(($PR | $PO) $CM*)?(($OP | $HY) $CM*)? ($IS $CM*)?
$NU ($CM* ($NU | $SY | $IS))*
($CM* ($CL | $CP))? ($CM* ($PR | $PO))?;
```

whereas UAX #14 has fifteen sub-rules (actually more, as they must be combined with some earlier rules).

NU (SY | IS)* CL × PO
NU (SY | IS)* CP × PO
NU (SY | IS)* CL × PR
NU (SY | IS)* CP × PR
NU (SY | IS)* × PO
NU (SY | IS)* × PR
PO × OP NU
PO × OP IS NU
PO × NU
PR × OP NU
PR × OP IS NU
PR × NU
HY × NU

IS × NU
NU (SY | IS)* × NU

There are cases where rules that require extended context are difficult to handle in the RBBI language, most notably where LB15b interacts with LB15a, requiring these particularly messy regular expressions that bear little resemblance to LB15a and LB15b and do not match anything particularly meaningful:

```
$LB8NonBreaks [\p{Pf} & $QU] $CM* ([\p{Pi} & $QU] $CM* $SP*)+ .;
$LB8NonBreaks [\p{Pf} & $QU] $CM* ([\p{Pi} & $QU] $CM* $SP*)+ $SP $CM+ $AL_FOLLOW?;
$CAN_CM $CM*  [\p{Pf} & $QU] $CM* ([\p{Pi} & $QU] $CM* $SP*)+ .;
$CAN_CM $CM*  [\p{Pf} & $QU] $CM* ([\p{Pi} & $QU] $CM* $SP*)+ $SP $CM+ $AL_FOLLOW?;
^$CM+  [\p{Pf} & $QU] $CM* ([\p{Pi} & $QU] $CM* $SP*)+ .;
^$CM+  [\p{Pf} & $QU] $CM* ([\p{Pi} & $QU] $CM* $SP*)+ $SP $CM+ $AL_FOLLOW?;
```

This is one area where improvements to the RBBI table builder and rule syntax would be welcome: if the chaining could be made to work on multi-character overlaps, *i.e.*, to make $s_1 s_2 \ldots s_k \ldots s_{k+l} \ldots s_n$ a token if $s_1 s_2 \ldots s_k \ldots s_{k+l}$ and $s_k \ldots s_{k+l} \ldots s_n$ are both tokens for some $l \geq 0$, the rules would be more straightforward to maintain with no impact on the actual execution of the DFA once generated.

Why include support for the CSS tailorings in the DFA?

Browsers are one of the major implementers of the line breaking algorithm; they need to implement the CSS tailoring. A DFA that did not support those would be difficult to amend according to the tailorings; in practice, this would require constructing a whole new DFA, which, as we have seen, is something with which implementers struggle. It is even trickier to construct a unified DFA which supports all CSS options, and in the absence of such a unified DFA, implementers would end up carrying several subtly different tailored DFAs, resulting in a large memory footprint (this has been the case of Chrome via ICU for many years, with now nine copies of the line breaking DFA). In contrast, the DFA is not made substantially larger by the addition of the shadow symbols, and an implementation that does not require the tailorings can straightforwardly drop these symbols anyway.

The line breaking algorithm as specified by UAX #14 has long provided support for CSS tailoring; line breaking class CJ, added in Unicode Version 6.1, exists solely to support the CSS line-break tailorings; in an untailored implementation, it is treated as NS by LB1.

Do we need the lookaheads?

Contrary to the grapheme cluster, word, and sentence breaking state machines, the line breaking state machine uses lookaheads.

As of this writing, lookaheads are used in the implementation of rules LB8a, LB15c, LB19 and LB19a, and LB30a. It is not clear whether the lookaheads need to be involved in the implementation of all of these rules. In the case of LB19 and LB19a, which are specified as keep-together rules, the use of a lookahead is primarily a workaround to avoid the need for humongous regular expressions in the RBBI rule syntax to describe overlapping contexts of rules for quotation marks.

However, lookaheads *are* required to implement LB15c (SP ÷ IS NU). Indeed, there is a break in the sequence SP IS ÷ ID, so that after encountering a sequence <SPACE, FULL STOP>, the automaton can return either the position after the space if a character with lb=NU follows, as in <SPACE ÷ FULL STOP × DIGIT FIVE>, or the position after the full stop if a character with lb=ID follows, as in <SPACE × FULL STOP ÷ CJK UNIFIED IDEOGRAPH-55B5>. Since these positions cannot both be stored as the last accepting position, the automaton must return something other than the last accepting position in one of these cases.

It might be possible to amend the line breaking rules to avoid lookaheads (after all, SP IS ÷ ID is a somewhat degenerate case). Avoiding lookaheads would mean that we would be solving a theoretically simpler problem

(recognizing a regular language), which would mean that we would know that the DFA is minimal, and it would generally be satisfying. However, since the framework in which the rules are specified in UAX #14 clearly allows for rules that require lookaheads, since lookaheads have in practice been used by ICU for the past twenty years to implement the rules, and since lookaheads are a basic functionality of recognizers such as those generated by lex, it seems more future proof to specify this capability than to try to get rid of it.

Likewise, we allow for multiple lookaheads, designated by names in the data files, even though the current line break state machine only uses one, because ICU has used multiple lookahead slots for many years, and because rules that require multiple lookahead slots are easily specified[13].

Acknowledgements

Robert Bastian wrote an implementation based on the data files and algorithm described here; he provided important feedback on them and helped identify and address inefficiencies in earlier versions of the state machines. Rich Gillam wrote the first DFA-based implementation in ICU in 1999. Andy Heninger wrote the RBBI table builder in ICU in 2002; he maintained the rules from which the DFAs are generated until 2019. Robin Leroy maintains the rules since 2023, made changes to the RBBI table builder to produce tables with fewer states, symbols, and lookahead slots, wrote the tooling that produces the new data files, and authored the bulk of this document.

References

[ASU86] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullmann, *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 1986.
ISBN: 0-201-10088-6.

[GS07] Gregor Gramlich and Georg Schnitger. “Minimizing nfa’s and regular expressions”. In: *Journal of Computer and System Sciences* 73.6 (September 2007), pp. 908–923.
DOI: 10.1016/j.jcss.2006.11.002

[JR93] Tao Jiang and B. Ravikumar. “Minimal NFA Problems are Hard”. In: *SIAM Journal on Computing* 22.6 (1993), pp. 1117–1141.
DOI: 10.1137/0222067

[LS79] Mike E. Lesk and Eric E. Schmidt “Lex – A Lexical Analyzer Generator”. In: *Unix time sharing system: Unix programmer’s manual (7th Edition)*. Vol. 2B. Bell Telephone Laboratories, Incorporated, Murray Hill, New Jersey, 1979.

[Moo56] Edward F. Moore, “Gedanken-experiments on sequential machines”. In *Automata studies*, pp. 129–153. Ann. of Math. Stud., no. 34. Princeton University Press, Princeton, New Jersey, 1956.

[13] Consider rules

[x] ÷ [y] [z] [1]
[x] [y] ÷ [z] [2]
[x] [y] [z] ÷ [3]
ALL × ALL

After encountering “xyz”, the automaton must remember three different positions so it can return the correct one depending on the fourth character.