Syntax highlighting for code blocks using ixml

Pieter Lamers (John Benjamins Publishing Company)
Nico Verwer (Rakensi)

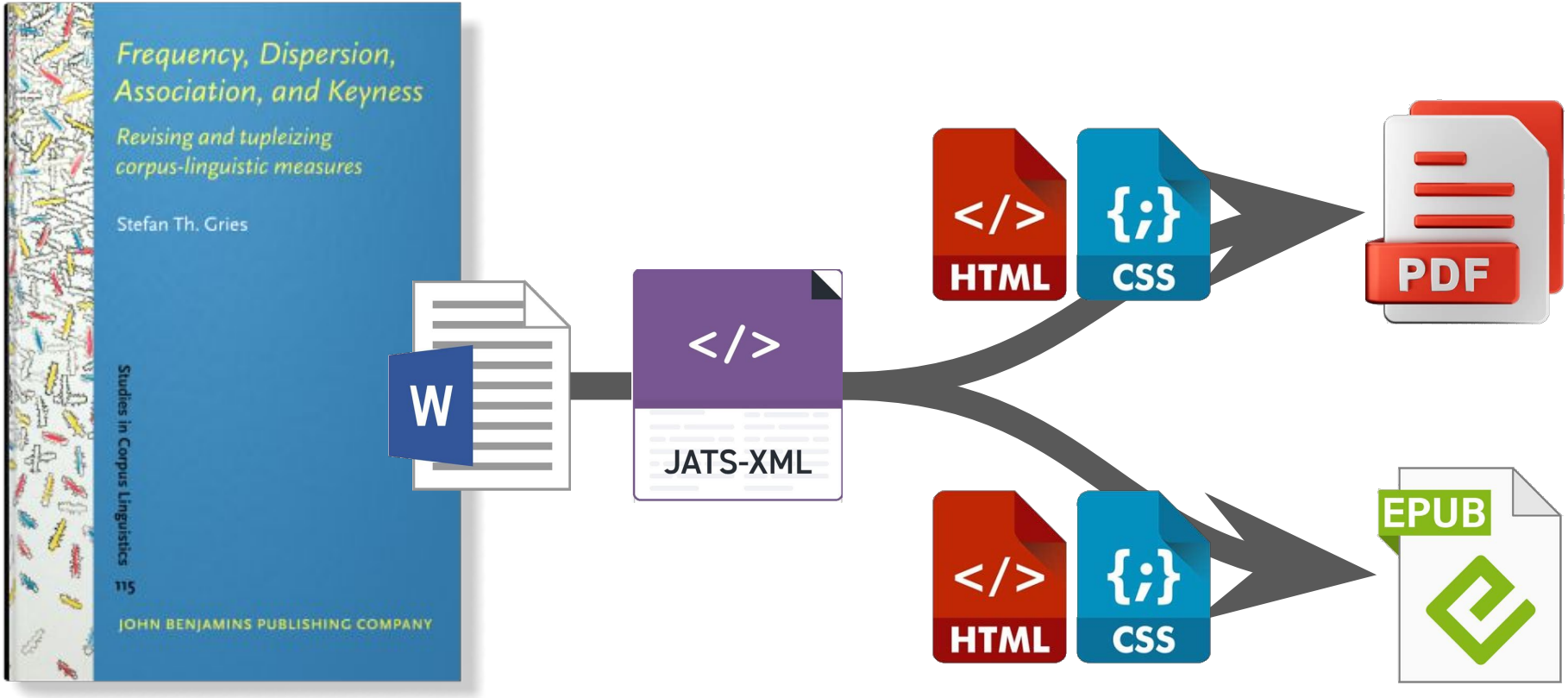
John Benjamins Publishing

Scholarly publisher
Books, Journals, Online resources
Humanities: Linguistics, Translation studies

XML-first workflow

- Word to JATS/BITS
- Linguistics examples, glosses, speech transcripts
- code

The book and the process



Code blocks in MS Word

RStudio/(Posit) installed. Finally, you should minimally have current versions of the following packages installed: `data.table`, `doParallel`, `edgeR`, `foreach`, `iterators`, `magrittr`, `mgcv`, `microbenchmark`, `parallel`, and `Rcpp`.

OK, we begin by setting up the workspace for everything that follows ...

```
rm(list=ls(all=TRUE))
sapply(c("data.table", "doParallel", "edgeR", "foreach", "KLD4C",
         "magrittr", "microbenchmark", "mgcv", "parallel", "Rcpp"),
   library, character.only=TRUE, logical.return=TRUE, quietly=TRUE)
gc()
```

... and go over two notational conventions I will sometimes use. The first is that I will sometimes write code in a way that may seem a little surprising at first, but there's a good reason for it. For example, I will sometimes write arithmetic operations like subtractions or divisions not in the canonical way but in a function-inspired way; here are some examples:

```
"+"(2, 3) # same as the 'normal' way of 2+3
```

Code blocks in JATS

<code> Code

A container element for technical content such as programming language code, pseudo-code, schemas, or a markup fragment.

Usage/Remarks

The <code> element may contain preformatted text, which may contain emphasis elements for syntax coloring, or it may contain an external link to a binary executable file.

No syntax coloring is present yet, in our case.

Best Attribute Practice

The various semantic attributes should be used to name the type of code, the language, the intended platform(s), etc. Executable code should always be marked as such using the `@executable` attribute. The `@specific-use` attribute may also be used to describe the rationale or uses for a code sample.

▼ Models and Context

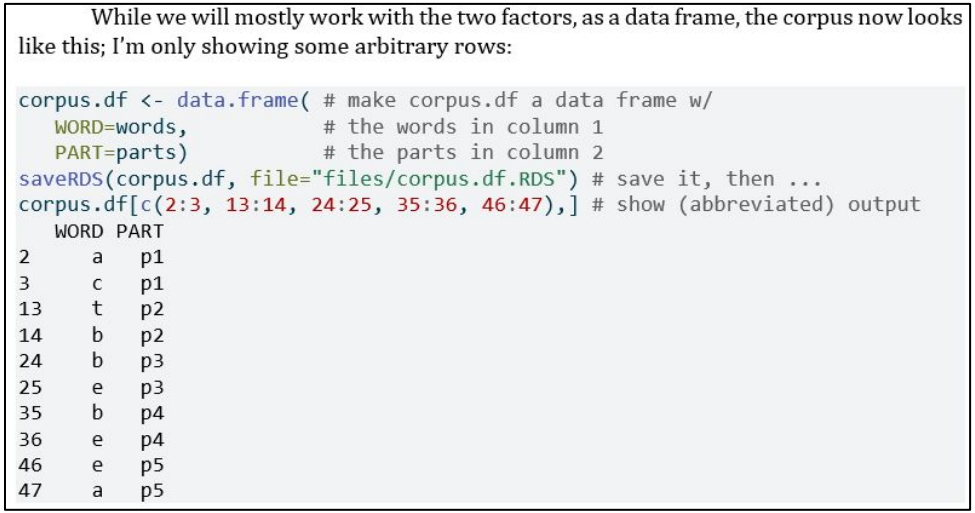
▼ Expanded Content Model

```
(#PCDATA | email | ext-link | uri | bold | fixed-case | italic | mono | 
overline | overline-start | overline-end | roman | sans-serif | sc | strike | 
underline | underline-start | underline-end | ruby | inline-graphic | inline-
media | private-char | abbrev | index-term | index-term-range-end | milestone-
end | milestone-start | named-content | styled-content | fn | target | xref | 
sub | sup)*
```

Named content is present for output and comments.

Example 1

```
<code language="r" language-version="4.3.2">
corpus.df &lt;- data.frame( # make corpus.df a data frame w/
   WORD=words,           # the words in column 1
   PART=parts)           # the parts in column 2
saveRDS(corpus.df, file="files/corpus.df.RDS") # save it, then ...
corpus.df[c(2:3, 13:14, 24:25, 35:36, 46:47),] # show (abbreviated) output
<named-content content-type="output">
    WORD PART
 2     a   p1
 3     c   p1
 13    t   p2
 14    b   p2
 24    b   p3
 25    e   p3
 ...
</named-content>
</code>
```



While we will mostly work with the two factors, as a data frame, the corpus now looks like this; I'm only showing some arbitrary rows:

```
corpus.df <- data.frame( # make corpus.df a data frame w/
   WORD=words,           # the words in column 1
   PART=parts)           # the parts in column 2
saveRDS(corpus.df, file="files/corpus.df.RDS") # save it, then ...
corpus.df[c(2:3, 13:14, 24:25, 35:36, 46:47),] # show (abbreviated) output
   WORD PART
2     a   p1
3     c   p1
13    t   p2
14    b   p2
24    b   p3
25    e   p3
35    b   p4
36    e   p4
46    e   p5
47    a   p5
```

Example 2

```
<code language="r" language-version="4.3.2" xml:space="preserve">
ICEGB.wout.annot &lt;- lapply( # make ICEGB.wout.annot the result of applying
   ICEGB.w.annot, gsub,        # to ICEGB.w.annot the function gsub, replacing
   pattern="(?x)  <named-content content-type="comment"># (set free-spacing)</named-content>
   ^.*\\{         <named-content content-type="comment"># from beginning to {</named-content>
   ([^}]+)        <named-content content-type="comment"># characters not }</named-content>
   \\}$           <named-content content-type="comment"># till the }</named-content>
   ", replacement="\\1", # by this, namely the word just remembered
   perl=TRUE)            # using Perl-compatible expressions
</code>
```

Not a valid R comment, inside a regular expression!

Solution 1: Regular Expressions

- Based on highlight.js. [https://github.com/highlightjs/highlight.js/blob/main/src/languages/r.js]
- Converted into XSLT, then use `<xsl:analyze-string>`.

```
<xsl:variable name="r:MODULE_IDENT_RE">(([a-zA-Z]|\.[._a-zA-Z])[._a-zA-Z0-9]*)(::)</xsl:variable>
<xsl:variable name="r:FUNCTION_CALL_RE">(([a-zA-Z]|\.[._a-zA-Z])[._a-zA-Z0-9]*)(\s*\()</xsl:variable>
<xsl:variable name="r:PARAMETER_RE">(([a-zA-Z]|\.[._a-zA-Z])[._a-zA-Z0-9]*)(\s*=\s*)</xsl:variable>
<xsl:variable name="r:NUMBER_TYPES_RE_1">0[xX][0-9a-fA-F]+\.[0-9a-fA-F]*[pP][+-]?\d+i?</xsl:variable>
...
<xsl:variable name="r:LITERAL_RE">NULL|NA|TRUE|FALSE|Inf|NaN|...</xsl:variable>
<xsl:variable name="r:STRING_RE">"([^\\"]|\\.)*"|'([^\\']|\\.)*'</xsl:variable>
<xsl:variable name="r:COMMENT_RE">#.*$</xsl:variable>
<xsl:variable name="r:MATCH_REs" select="($r:COMMENT_RE, $r:STRING_RE, ..., $r:IDENT_RE)"/>
<xsl:variable name="r:MATCH_RE" select="string-join($r:MATCH_REs, '|')"/>
```

Solution 1: evaluation

- It works in most cases.
- Regular expressions < context-free grammars.
  - That is not a big problem; we are not syntax checking.
- Some things cannot be parsed correctly.
  - There is an example in the demo.
- The code is not elegant.
  - Matching happens twice.

```
<xsl:analyze-string select="$unindented-code" regex="{$r:MATCH_RE}" flags="m">
  <xsl:matching-substring>
    <xsl:choose>
      <xsl:when test="matches(., '^('||$r:COMMENT_RE||')$')">
        <xsl:sequence select="r:styled-content('comment', .)"/>
      </xsl:when>
      <xsl:when test="matches(., '^('||$r:LITERAL_RE||')$')">
        <xsl:sequence select="r:styled-content('literal', .)"/>
```

Solution 2: IXML Parsing

- Based on a BNF grammar [https://github.com/ropensci/ozunconf19/issues/28]
- Converted into ixml: ~ 350 lines.
  - Found some small errors in the grammar. The real R compiler uses something else.

```
prog : _ , expr_or_assign ** prog_separator , _ .
```

and so on…

- Spacing is tricky, and easily increases ambiguity.
- The code is elegant.

```
declare variable $sc:grammar as xs:string := unparsed-text('xmldb:exist://db/apps/da2024/r.ixml');
declare variable $sc:parser := ixml:transparent-invisible-xml($sc:grammar, map{});
```

Problem: Ambiguity

R is inherently ambiguous.

The expression `x<-37` can be parsed in two ways:

- `x <- 37` # assignment
- `x < (-37)` # comparison

The R compiler solves this by first doing lexical analysis,
giving assignment priority, and then parse the sequence of lexical symbols.

The result of ixml parsing is one of the possible parsings.
It is not possible to give assignment priority over comparison.

Solving the assignment / comparison ambiguity

Add a pre-processing step turning “<-” into “←”.

```
<xsl:template match="code//text()">
  <xsl:sequence select="replace(., '&lt;-', '&#x2190;')"/>
</xsl:template>
```

Then parse.

Undo pre-processing in a post-processing step.

```
<xsl:template match="code//text()">
  <xsl:sequence select="replace(., '&#x2190;', '&lt;-')"/>
</xsl:template>
```

Problem: Embedded markup that is not R code

```
<code language="r" language-version="4.3.2" xml:space="preserve">
ICEGB.wout.annot &lt;- lapply( # make ICEGB.wout.annot the result of applying
  ICEGB.w.annot, gsub,         # to ICEGB.w.annot the function gsub, replacing
  pattern="(?x)   <named-content content-type="comment"># (set free-spacing)</named-content>
  ^.*\\{          <named-content content-type="comment"># everything from beginning to {</named-content>
  ([^}]+)         <named-content content-type="comment"># characters that are not }</named-content>
  \\}$            <named-content content-type="comment"># till the }</named-content>
  ", replacement="\\1", # by this, namely the word just remembered
  perl=TRUE)            # using Perl-compatible expressions
</code>
```

Ignore `<named-content>` elements inside `<code>`, keep the content.
But R does not allow comments in the middle of a regular expression.

Take away all <named-content> elements and their content.
But then we lose content that should be displayed.

Solving embedded markup

An ixml parser has text as its input, no embedded XML elements.

```
fn:invisible-xml(
    $grammar as (xs:string | element(ixml))? ,
    $options as map(*)?
) as function(xs:string) as document-node()
```

A *transparent invisible XML* parser accepts any item() as input, and ignores XML elements, but not their text content. See my presentation at XML Prague 2024.

```
ixml:transparent-invisible-xml(
    $grammar  as (xs:string | element(ixml))? ,
    $options  as map(*)?
) as function(item()) as document-node()
```

Tixml ignores markup but keeps text content

```
<code language="r" language-version="4.3.2" xml:space="preserve">
ICEGB.wout.annot &lt;- lapply( # make ICEGB.wout.annot the result of applying
  ICEGB.w.annot, gsub,         # to ICEGB.w.annot the function gsub, replacing
  pattern="(?x)   <named-content content-type="comment"># (set free-spacing)</named-content>
  ^.*\\{          <named-content content-type="comment"># everything from beginning to {</named-content>
  ([^}]+)         <named-content content-type="comment"># characters that are not }</named-content>
  \\}$            <named-content content-type="comment"># till the }</named-content>
  ", replacement="\\1", # by this, namely the word just remembered
  perl=TRUE)            # using Perl-compatible expressions
</code>
```

The parser will produce syntax errors.

Solution: more pre- / post-processing

```
<xsl:template match="named-content">
  <xsl:copy>
    <xsl:sequence select="@*"/>
    <xsl:attribute name="content" select="serialize(node())"/>
  </xsl:copy>
</xsl:template>
```

This ‘hides’ the mixed content, moving it inside an attribute.

Unhide:

```
<xsl:template match="named-content">
  <xsl:copy>
    <xsl:sequence select="@* except @content"/>
    <xsl:sequence select="parse-xml-fragment(string(@content))"/>
  </xsl:copy>
</xsl:template>
```

Advantages of (t)ixml over regexp

- Readability of code and grammar
  - Grammar separated from code
  - Non-terminals are meaningful.
- Accuracy
  - A context free grammar is more powerful than regular expressions.
  - There are errors in syntax highlighting using regular expressions.
- Syntactic detail
  - Regular expressions recognize 12 syntactic classes.
  - The ixml parser recognizes 87 non-terminals.

- Disadvantage: More work
  - Regular expressions are only needed for highlighted parts.

Demo

- eXist-db + Markup Blitz + eXist (t)ixml functions

Links:

- https://github.com/nverwer/DA2024-syntax-highlighting
- https://github.com/nverwer/exist-ixml-xar
- https://jats.nlm.nih.gov/archiving/tag-library/1.3/element/code.html

Future work

Use Treesitter for parsing.