Parsing Text With XSLT 3

Liam Quin
Delightful Computing

https://www.delightfulcomputing.com/

Logistics

- Slides are available in various formats and typefaces at
  - https://www.delightfulcomputing.com/talks/

Overview

- Not an introduction to parser theory;
- Ad-hoc parsing rather than strictly grammar based;
- Emphasis: features new in XSLT 3 that facilitate writing ad-hoc text parsers;
- Examples mostly come from working on Eddie 2.

Eddie 2 & Parsing

- Eddie 2 needs to read two DTDs and compare them in specific ways;
- It can also read your XSLT stylesheet to guess whether you have written all the templates you need;
- It also reads a config file (simple XML though).

Eddie 2 Report

role

role not in configuration file

Children differ

index-term, index-term-range-end, inline-media

(all these children are in the Eddie2 configuration already)

Or-groups with different children

```
<!ELEMENT role "(
        #PCDATA|email|ext-link|uri|inline-supplementary-material|related-article|
        related-object|hr|bold|fixed-case|italic|monospace|overline|overline-start|
        overline-end|roman|sans-serif|sc|strike|underline|underline-start|underline-
        end|ruby|alternatives|inline-graphic|inline-media|private-char|chem-struct|
        inline-formula|tex-math|mml:math|abbrev|index-term|index-term-range-end|
        milestone-end|milestone-start|named-content|styled-content|fn|target|xref|sub|
        sup|x
)*">
```

resource-id
resource-name
resource-wrap
✖response
role
✔roman
✖rp
✖rt
✖ruby
✔sans-serif
✔sc
✖season
✔sec
✖sec-meta
✔see
✔see-also
✔self-uri
✔series
✔series-text
✔series-title

Production DTD Example

```
<!ELEMENT app
    (title, index*, (%para.level;|fn)*, intro?, sect*)
>
```

- When writing XSLT, only the resulting list of elements usually matters, but the parameter entities can help understand the DTD.
- The DTD is a text file, so we might first think of …

Regex approach to ad-hoc parsing

- Use substitutions to turn input into something regular and then handle that instead
-
```
replace(
    "<!ELEMENT\s+(\i\c*)\s+(.*?)\s*>",
    "<e><n>$1</n><model>$2</model></e>”
)
```
- But how far should you go?

Avoid the Sledgehammer

- Any sufficiently powerful regular expression is indistinguishable from line noise.
- Use whitespace to format expressions (“x” flag);
- You can use intermediate variables;
- Beware that {...} marks an attribute value template in xsl:analyze-string. Use a variable.

Don’t do this

```
# first the case with parens, LAST (FIRST) and variations,
# all with the parens:
^
( # 1
    (?:ST\.\s+)?
    [A-ZÆÉŒ][A-ZÆÉŒ-]*[A-ZÆŒÉ]'?             # at least 2 letters at the start
    (?: # a multi-word cluster is allowed here:
        \s+
        (?: # optional St. or roman numeral or word
            [XVI]+\.? |
            ST\. |
            Y | # for a Spanish name, Alcala Y Henares
            du |
            (?:
                [LD]'\s*
            )?

            [A-ZÆŒ]+-?[A-ZÆŒ]+
        )
        '? # E' is usually (not always) used for É in the book
    )*
)
( # 2
    (?: # optional ,alternate, alternate, or alternate,
        ,?
        (?:\s+[A-ZÆŒÉ]+,?)*
        (?:
            (?:
                \s+or |
                \s+or\s+simply |
                \s+or\s+[dD]e |
                \s+ in \s+ [lL]atin |
                \s+or,\s+as\s+pronounced,
            )
            (?:\s+[A-ZÆŒÉ]+,?)+
            ,?
        )*
    )?
  \s+
)
[(](.*?)[)] # 3 - require the parens
( # 4 - sep
    ,?
    [.,]|
    \s+a|\s+an|\s+one|\s+was|\s+the|\s+were|\s+lived|\s+of|\s+called|
    [.,]?\s+[sS]ee|\s+is|\s+son|
    \s+surnamed|
    \s+D[AEU]\s[A-ZÆ]+,? # for vol 4 p. 110ff, and elsewhere
)
( # 5 - rest
   \s+
   .*
)
# [a-z][a-z] # require at least 2 lower case letters to avoid running header
```

Instead

- Make a little language and compile it into a regular expression, or use multiple smaller patterns;
- Match a little at a time; use maps to represent state;
- Use *fn:tokenize()* and match on sequences;
- Note: for HTML *class* attributes use *contains-token()* instead, to get case sensitivity & corner cases right.

The actual Eddie 2 DTD parser…

- Uses an array of maps to hold a state table;
- Each map has a string or regex to match the next token, a name for error reporting, and a function to handle the rest of the construct.
- Each construct (<!ELEMENT, <!ATTRIBUTE etc.) has its own syntax and its own function;
- The functions can safely use regexes.

Simple Grammars

- Sometimes you have a really simple grammar to match & simple replace() is readable, with intermediate variables;
- Eddie 2 can read your XSLT file and make sure you have a template for every changed element; the code parses XSLT match patterns to do this.

Match Patterns in XSLT 3

- An XSLT 3 match pattern is either a *predicate pattern* or a *match pattern*.
- A *predicate pattern* .[*test*] matches if the test is true, and can match anything.
  - For Eddie 2, use match="sock", not match=" . [name() eq 'sock'] "
- A *selection pattern* uses a subset of XPath 3
  - These are the regular XSLT match= templates we want to Eddie 2 to check for us.
  - The grammar for them is simple; let's take a quick look at a fragment of it:

Selection patterns

UnionExprP ::= IntersectExceptExprP (("union" | "|")   IntersectExceptExprP)*

IntersectExceptExprP ::= PathExprP (("intersect" | "except") PathExprP)*

PathExprP ::= RootedPath | ("/" RelativePathExprP?) | ("//" RelativePathExprP) | RelativePathExprP

RootedPath ::= (VarRefXP30 | FunctionCallP) PredicateListXP30 (("/" | "//") RelativePathExprP)?

RelativePathExprP ::= StepExprP (("/" | "//") StepExprP)*

StepExpr ::= PostfixExprP | AxisStepP

Matching selection patterns

```
<!--*  Remove XPath comments first, (: .... :) turning them into a space *-->
<xsl:variable name="without-comments" as="xs:string"
   select="replace($input, '[(][:].*?[:][)]', ' ')" />
<!--* Remove strings, so we can safely remove predicates later
     * without worrying about strings containing [ or ]
     *-->
<xsl:variable name="noquot_re" select=" '&quot;[^&quot;]*&quot;' " as="xs:string" />
<xsl:variable name="without-single-quote-strings" as="xs:string"
   select='replace($without-comments, $noquot_re, " ")' />
```

Commentary

- You could do this part in XSLT 2 just as well;
- Intermediate variables help me to understand what i did;
- The variables can also be printed with <xsl:message>$var={$var}</xsl:message> (XSLT 3)

  or examined in a debugger (e.g. Oxygen XML Developer™)

Returning a result

- The "parse" returns the original match attribute or an empty sequence, and a sequence of zero or more element names;
- An array is a good choice here, so i could add more information later, such as a mode attribute: [$attr, $elements, $mode ...]
- Could also use a map and give the items names.
- Note: arrays and maps preserve node identity and can contain aby sort of item, including function items.

Arrays & Maps vs Elements

- Arrays & Maps use less memory than elements
- Can preserve node identity and values inside them
- Fragile: poor type safety as="map(*)"
- Fussy: it's an error if you forget to type a variable or parameter or if you don't specify the return type of a function or template (could use schematron to mitigate this?)

Table Driven Parsing

- Maps can nest:

```
input-token: “<!ELEMENT”,
parse-table : {
    input-token: $XMLNAME,
    parse-table: {
        input-token: “#PCDATA”
```

Table Driven Parsing

- Maps can contain functions:

  input-token: “<!--”,

  handler: handle-comment#3
- Could also put the function inline,

  handler: function ($input as xs:string …) { … }

  but it’s easier to debug if it has a name.

A Tail of Two Recursions

- Recursive templates & functions can use a lot of memory unless the interpreter spots tail recursion and turns them loopy.
- The xsl:iterate instruction explicitly enforces tail recursion amenable code so it's *strictly loopy*.
- Parsing can make *deep* recursion.

Finite State Machines

- E.g. a separate set of tables to handle different sections in a book, with an input rule to move between them;
- This starts to get closer to a traditional parser, more computer-sciency;
- Remember who will *read* the stylesheet!

Use map:for-each()

- To map each key/value pair to a new value (possibly a map entry) use the *map:for-each()* function; or use keys() ! Function() {…}
- The XSLT 1 way would have been a recursive template; in XSLT 2, a recursive function (if XSLT 1 or 2 had maps, that is!)

Streaming and Parsing

- Streaming stylesheets can go reasonably quickly and use less memory;
- New XSLT instructions like *xsl:where-populated* are useful even outside streaming: a much more efficient way to make container elements only if they contain something (e.g. a list).

Higher Order Functions

- You can make a "visitor pattern" from functions, can have templates and functions that return functions, and can use functions as another way alongside fn:transform() to avoid modes;
- Passing a function as an argument to a function can be a good, clear way to encapsulate context (e.g. a *getToken()* function).

Skimming the Surface

- We've looked at new data types (maps, arrays), new operators, higher order functions (functions as values), streaming, templates that return functions, arrays, maps…
- XSLT 3 brings big and deep changes…
- You always need to keep in mind the *rhetorical nature* of what you write, and the expected audience;
- Ad-hoc parsing of text is often very appropriate, and XSLT 3 has lots of tools to help you.
- Oh, and Eddie 2? He's doing fine. Thanks for asking.

</talk>

Liam Quin

Delightful Computing

https://www.delightfulcomputing.com