RXSLT Grammar

Sam Wilmott
sam@wilmott.ca
www.wilmott.ca

August 3, 2006

Within and between RXSLT commands, comments can be entered, starting with "#" and continuing to the end of the line. Comments aren't allowed within XPath expressions.

RXSLT instructions (top-level, within template and character-level) are generally terminated by the end of the line (or a comment ending the line). If an instruction has "content", like "template", that content must be either:

1. prefixed by a colon, or
2. be placed on following lines, which must be indented more than the main instruction.

In the grammar, where an instruction has content, that content is marked by a colon (":") in the grammar.

Instructions at the same level are generally indented the same amount. Alternatively, if a non-top-level instruction is followed by a semicolon, then what follows it is seen as an instruction at the same level, no matter whether it appears on the same or a following line, and no matter how it is indented.

Line breaks are also allowed where the following material is "obviously" part of the preceding instruction: mandatory parts of the instruction, following a comma, continuation of a multi-part string literal, or within a parenthesized expression. The indentation convention is similar to that of Python, except that content need not be all indented the same amount, just indented more than the main instruction. Due to the indentation, there are no "end" instruction markers in the language, except for the optional "end" at the end of the program.

The parts of the grammar are annotated with the corresponding chapters of the XSLT 1.0 specification.

2. Stylesheet Structure

```
rxslt-spec = explict-spec | implicit-spec

explicit-spec = (xslt-version top-level-att* ":")?
                import* top-level+ "end"?

xslt-version = "xslt1.0" | "xslt1.1"

top-level-att =
    "xmlns" name? "=" string-literal
  | "extension-element-prefixes" "=" name-list
  | "exclude-result-prefixes" "=" name-list
  | "id" "=" string-literal

implicit-spec = (instruction | result-element)*

import = "import" string-literal
```

where string-literal is a URI

```
top-level = include
          | strip-space
          | preserve-space
          | output
          | key
          | decimal-format
          | attribute-set
          | variable
          | param
          | template
          | namespace-alias
```

```
include = "include" string-literal
```

where string-literal is a URI

```
documentation = "documentation"
                (documentation-element | documentation-comment |
                 documentation-pi | string-literal)*

documentation-element = <name ... </name>

documentation-comment = <!-- ... -->

documentation-pi = <? ... ?>
```

Documentation consists of one or more valid XML elements, comments, PIs or string literals that appear at the top level of an RXSLT program. String literals are short-hand for XML comments.

3. Data Model

```
strip-space = "strip-space" name-list
```

The list of names is the value of the <strip-space/> element's "elements" attribute.

```
preserve-space = "preserve-space" name-list
```

The list of names is the value of the <preserve-space/> element's "elements" attribute.

4. Expressions

```
expr = XPathExpr
```

An XPathExpr is as described by the "Expr" node in the XPath 1.0 grammar. Within an expr, line breaks and RXSLT comments are allowed, as described in the "Rethinking XSLT" article.

```
pattern = XPathExpr
```

The same restrictions apply to a "pattern" as to an "expr". In addition, XSLT imposes further restrictions on XPath expressions used as patterns.

If an expr consists only of a slash, it will sometimes need to be parenthesized or the next item placed on a following line, to disambiguate what the next token means. For example, in the following, without the parentheses, "use" would be recognized as part of the "match" XPath expression:

```
key top match / use @id   # Ambiguous.  Instead use
key top match (/) use @id # or
key top match /
        use @id
```

Fortunately this is only a problem in a few odd cases. But it's still something that needs some more thought before the language settles down completely.

5. Template Rules

```
group-mode = "mode" name
           | "no-mode"

template = "named-template" name
           ":" (simple-text | instruction | result-element | param)*
         | number? "template" pattern
           ":" (simple-text | instruction | result-element | param)*
```

where number is priority, the name is template name, and pattern is template match.

```
result-content = result-element | result-comment | result-pi

result-element = <name ... </name>

result-comment = <!-- ... -->

result-pi = <? ... ?>
```

i.e. a well-formed XML fragment with:

- XML-style escaping.
- All characters with XML meaning except for "{".
- Nested instructions and expressions grouped in "{" ... "}", in one context as defined by XSLT, and two others as defined by RXSLT.
- If "{" ... "}" occurs within an attribute value, its content is an "expr". This is as defined by XSLT.
- If "{" ... "}" occurs with in a start element, but not in an attribute value, then its content must be a comma-separated list of names, that is translated into a "used-attribute-sets" attribute specification.
- If "{" ... "}" occurs within element content, its content is one or more instances of "instruction". If an instruction appears on the same line as the "{", its indent level is considered to be -1, even if it is separated from the "{" by white space.
- In all three cases, doubled {{ represents a single "{" character. Likewise doubled "}}" represents a single "}" character.

A result-comment or result-pi is a valid one of either in XML form.

```
apply-templates = "apply-templates" ("with-nodes" expr)? ("with-mode" name)? ":"
                  (with-param | sort)*
```

The default for expr is node().

```
apply-imports = "apply-imports"
```

6. Named Templates

```
call-template = "call-template" name with-param*
```

7. Creating the Result Tree

```
namespace-alias = "stylesheet-prefix" string-literal
                | "result-prefix" string-literal
```

Together these two declarations make up the XSLT <namespace-alias/> element. The default for both values is "#default".

```
template-content = (simple-text | instruction | result-content)*

instruction =
    char-instruction
  | processing-instruction
  | comment
  | element
  | attribute

element = "element" avt ("in" avt)? use-directive?
          ":" template-content
```

The first name is the element name. The "in" name is namespace name.

```
attribute = "attribute" avt ("in" avt)? ":" char-text
```

The first name is the attribute name. The "in" name is namespace name.

```
attribute-set = "attribute-set" name use-directive? ":" attribute*
```

Where the first name is an attribute set name.

```
use-directive = "use" name-list
```

Used to specify the names of a "use-attribute-set" attribute.

```
text = "unescaped"? string-literal

processing-instruction = "processing-instruction" avt ":" char-text

comment = "comment" ":" char-text

copy = "copy" use-directive? ":" template-content

value-of = "unescaped"? "value-of" expr

copy-of = "copy-of" expr

avt = name
```

An "avt" can contain XPath expressions as per the XSLT 1.0 specification. If so, the name must be quoted, and the XPath parts of it surrounded in { ... }.

```
char-text = (simple-text | char-instruction)*

simple-text = string-part

name-list = name ("," name)*

name = XML-name | string-part+

string-literal = XML-name | string-part ("+" string-part)*

string-part = "'" (anything but a ') "'"
            | """ (anything but a ") """
```

Strings are like XML attribute values, except that you can split a string into parts separated by plusses and white space.

```
char-instruction =
    apply-templates
  | call-template
  | apply-imports
  | for-each
  | value-of
  | copy-of
  | number
  | choose
  | if
  | text
  | copy
  | variable
  | message
  | fallback

number = ("number" | "single-number" | "multiple-number" | "any-number")
         number-option*
```

"number" is equivalent to "single-number".

```
number-option =
    "count" "=" pattern
  | "from" "=" pattern
  | "value" "=" expr
  | "format" "=" avt              # default = 1
  | "lang" "=" avt
  | "letter-value" "=" avt
  | "grouping-separator" "=" avt
  | "grouping-size" "=" avt
```

8. Repetition

```
for-each = "for-each" expr ":"
           (simple-text | instruction | result-content | sort)*
```

9. Conditional Processing

```
if = "if" expr ":" template-content

choose = "choose" ":"
         ("when" expr ":" template-content)+
         ("otherwise" ":" template-content)?
```

10. Sorting

```
sort = "sort" ("using" expr)? sort-option*
```

The default for expr is "using .". Note that the XSLT 1.0 spec says "sort cannot occur after any other elements or any non-whitespace character".

```
sort-option =
    "lang" "=" avt
  | "data-type" "=" avt  # "text" or "number" or qname-but-..., default = "text"
  | "order" "=" avt      # "ascending" or "descending", default = "ascending"
  | "case-order" "=" avt # "upper-first" or "lower-first"
```

11. Variables and Parameters

```
variable = "variable" name "=" expr
         | "variable" name ":" template-content

param = "param" name "=" expr
      | "param" name ":" template-content

with-param = "with" name "=" expr
           | "with" name ":" template-content
```

12. Additional Functions

```
PrimaryExpr = "document" "(" object ("," node-set)? ")" # -> node-set
            | "current" "(" ")" # -> node-set
            | "unparsed-entity" "(" string ")" # -> string
            | "generate-id" "(" node-set? ")" # -> string
            | "system-property" "(" string ")" # -> object

key = "key" name "match" pattern "use" expr

decimal-format = "decimal-format" (name ":")? decimal-format-option*

decimal-format-option =
    "decimal-separator" "=" string-literal   # default = "."
  | "grouping-separator" "=" string-literal  # default = ","
  | "infinity" "=" string-literal            # default = "Infinity"
  | "minus-sign" "=" string-literal          # default = "-"
  | "NaN" "=" string-literal                 # default = "NaN"
  | "percent" "=" string-literal             # default = "%"
  | "per-mille" "=" string-literal           # default = "&#x2030;"
  | "zero-digit" "=" string-literal          # default = "0"
  | "digit" "=" string-literal               # default = "#"
  | "pattern-separator" "=" string-literal   # default = ";"
```

13. Messages

```
message = "message" ":" template-content
        | "terminate" ":" template-content
```

15. Fallback

```
fallback = "fallback" ":" template-content

PrimaryExpr = "element-available" "(" string ")"  # -> boolean
            | "function-available" "(" string ")" # -> boolean
```

These are expr grammar enhancements to grammar in "4. Expressions":

16. Output

```
output = "output" output-option*

output-option =
    "method" "=" ("xml" | "html" | "text" | name)
```

```
| "version" "=" name
| "encoding" "=" string-literal
| "xml-declaration" "=" ("yes" | "no")
| "standalone" "=" ("yes" | "no")
| "doctype-public" "=" string-literal
| "doctype-system" "=" string-literal
| "cdata-section-elements" "=" name-list
| "indent" "=" ("yes" | "no")
| "media-type" "=" string-literal
```