XML Psychiatrist

Peter Scott

October 19, 2004

Contents

1 Introduction

Despite the ambivalent feelings of some people, XML is taking over the world. More and more, programs have to take XML files as input. However, this presents us with a problem: error checking. How can we be sure that the information in the XML document isn't junk? We could include a lot of error checking in our XML processing code, but that complicates things considerably and is prone to bugs. Ideally, we would like to separate out the sanity checking machinery. How should we do that?

We could use a Document Type Definition, or DTD. Unfortunately, DTDs have a verbose, clunky syntax, and they can't say anything about the information in the document. They can specify the structure, but they can't see past it. Also, DTDs don't support namespaces.

We could use an XML Schema. They support some content type checking, but they, too, have problems. They are written in XML, which is both a blessing and a curse. It's a blessing because XML is easy to parse; it's a curse because the format is so verbose that your fingers will ache with pain and you will get carpal tunnel syndrome if you type too much of it. Also, it lacks some more powerful and dangerous features, such as full use of Common Lisp in the validation process. Schemas and DTDs both share the disadvantage that they aren't part of Lisp, and are not designed to be used from Lisp processes.

I wanted to do some sanity checking on XML documents from a Common Lisp program I'm writing. I didn't want to use DTDs or Schemas for the reasons above. I wanted to be able to write the specifications for XML structures in a lispy syntax. I wanted enough power to grow. So, I wrote xml-psychiatrist.

1.1 Features

xml-psychiatrist has a number of interesting features:

- Basic structural validation
- Regular expressions for matching things
- Type checking using the Common Lisp type system
- Namespace support (still relatively experimental, since I have never found much use for XML namespaces and therefore have never used them much).
- The full power of Common Lisp. You can define your own functions for validating data (see 3), or you can even extend the framework itself (see 4).

1.2 Installation

Get it with asdf-install:

```
(require :asdf)            ; Or however you load ASDF on your Lisp
(require :asdf-install)
(asdf-install:install :xml-psychiatrist)
```

Alternately, you could download it from the xml-psychologist web site

2 Basic usage

At its simplest, xml-psychiatrist just takes a nested XML-like structure and tests to make sure that the actual XML matches.

I was writing a program which takes transcripts of interviews and processes them. The interview transcripts look like this, `test.xml`:

```
<transcript firstname="Rodney" lastname="Sporklenburg">

<section id="childhood" name="My Childhood">
<topic id="infancy" name="Infants drool too much">
<para>That's the truth: infants regard drooling as almost as much fun as
pounding the remote control on the table until the batteries fall
```

```
out. It's icky.</para>
</topic>

<topic id="earlychildhood" name="Early Childhood">
<para>I thought that airplanes were birds. That creeps me out. And
cartoons creeped me out back then.</para>
<para>It occurs to me just how small tiny children are. It's always
a little scary to think that these kids could probably kill each other
with that little bouncy ball.</para>
<para>Or maybe I'm just paranoid.</para>
</topic>
</section>
</transcript>
```

Here is some sample code from the program, `test.lisp`:

```
(require 'asdf)
(asdf:operate 'asdf:load-op :xml-psychiatrist)
(use-package :xmls)
(use-package :xmls-utilities)    ; Comes with xml-psychiatrist
(use-package :xml-psychiatrist)

(defun sanity-check-transcript-xml (transcript)
  (multiple-value-bind (is-sane-p error-message)
      (toplevel-match (tag "transcript" ((attr "firstname")
                                         (attr "lastname"))
                           (tag+ "section" ((attr "id")
                                            (attr "name"))
                                 (tag+ "topic" ((attr "id")
                                                (attr "name"))
                                       (match-anything))))
                      transcript)
    (unless is-sane-p
      (error error-message))))
```

The `:xmls-utilities` package comes with xml-psychiatrist, and is a small collection of useful functions for dealing with the xmls XML parser. For more information about xmls, see the xmls web site.

The function `sanity-check-transcript-xml` takes an XML parse tree from `parse-xml-file` and validates it. The XML specification here is this part:

```
(tag "transcript" ((attr "firstname")
   (attr "lastname"))
```

```
     (tag+ "section" ((attr "id")
      (attr "name"))
   (tag+ "topic" ((attr "id")
  (attr "name"))
 (match-anything))))
```

It matches a tag, “transcript”, with attributes “firstname” and “lastname”, which has one or more child tags “section”, which have attributes “id” and “name”, and one or more child tags “topic”, etc. At the very base of the XML structure, we need to be able to match parsed character data (just ordinary strings) and arbitrary tags for formatting purposes, so we use the `match-anything` matcher.

Load the `test.lisp` file shown above and go to the REPL:

```
* (sanity-check-transcript-xml (parse-xml-file "test.xml"))

NIL
```

Not very exciting, is it? That’s because the XML validation was sucessful and the document was shown to be sane. Naturally, the XML validator just did its checking quietly and without signalling an error.

The function `parse-xml-file` is in the `:xmls-utilities` package. It takes a filename and returns the parsed representation of an XML file.

Let’s take a closer look at the machinery:

```
* (tag "transcript" ((attr "firstname")
   (attr "lastname"))
     (tag+ "section" ((attr "id")
      (attr "name"))
   (tag+ "topic" ((attr "id")
  (attr "name"))
 (match-anything))))

#<XML-PSYCHIATRIST::MATCHER {9B96129}>
```

The tag macro and the attr function both return objects called matchers. A matcher looks at a node in the XML parse tree and tries to match it based on certain criteria. For example, the matcher `(attr "firstname")` matches an attribute named “firstname”. We can add further constraints. The matcher `(attr "age" :type 'integer)` matches an attribute named “age” which, when read by the Lisp reader, has the type `'integer`. Let’s try it out:

```
* (toplevel-match (attr "age" :type 'integer) '("age" "24"))

0
NIL
* (toplevel-match (attr "age" :type 'integer) '("age" "blue"))

NIL
"Matcher #<MATCHER {9B5C889}> did not match"
```

The `toplevel-match` function takes as arguments a matcher and a node in xmls parse-tree format. It returns two results, `matched-p` and `error-message`. `matched-p` is whether or not the matcher matched, and if it isn't nil, that indicates that the matcher did indeed match. If it is nil, then `error-message` becomes important. `error-message` will contain a string explaining why the matcher didn't match.

In the example above, the matcher matched the first node because it was an "age" attribute whose value was an integer. However, the matcher did not match the second node, because "blue" is not an integer. The Common Lisp type system allows for even more impressive feats:

```
* (toplevel-match (attr "age" :type '(integer 0 170)) '("age" "12"))

0
NIL
* (toplevel-match (attr "age" :type '(integer 0 170)) '("age" "500"))

NIL
"Matcher #<MATCHER {9B5C889}> did not match"
```

In this example, we constrain the type of the "age" attribute to be `'(integer 0 170)`, an integer between 0 and 170. This makes sense, because nobody can have a negative age, and it is very unlikely that anybody will live to be older than 170 years. For more information on the specifics of the Common Lisp type system, see the HyperSpec.

3 The standard matchers

There are a number of matchers predefined, and they should be enough for almost all cases. You have already seen the `tag` and `attr` matchers. In this section, we look at all the standard matchers.

3.1 The `tag` matcher

The `tag` matcher matches a single tag.

Syntax:

tag *name (attribute-matcher*) child-matcher** → *matcher*

Arguments and Values:

name—One of three things:

1. A regular expression that will match the name of the attribute. Note that "^" and "$" are added to the start and end of the regular expression string.
2. A cons of two regular expressions, both of which have "^" and "$" added as above, which match each element of the `(attribute-name . namespace)` cons used by xmls to represent namespaces.
3. A designator for a function. This is called on the attribute name (which may be a string or a cons, as described above) and returns a boolean representing whether or not the function matches the attribute name.

attribute-matcher—matchers that will match attributes, like the `attr` matcher (see 3.5).

child-matcher—matchers that will match child nodes, like tags or character data.

3.2 The `tag?` matcher

The `tag?` matcher is like the `tag` matcher except that it is optional. If a `tag?` is not matched, it's no big deal. If a `tag` is not matched, though, it generates an error.

3.3 The `tag*` matcher

The `tag*` matcher is like the `tag` matcher except that it matches zero or more tags, rather than one.

3.4 The `tag+` matcher

The `tag*` matcher is like the `tag` matcher except that it matches one or more tags, rather than one and only one.

3.5 The `attr` matcher

The `attr` matcher matches an attribute.

Syntax:

attr *name &key how-many possible-values type matches-regexp test-fn* → *matcher*

Arguments and Values:

name—One of three things:

1. A regular expression that will match the name of the attribute. Note that "^" and "$" are added to the start and end of the regular expression string.
2. A cons of two regular expressions, both of which have "^" and "$" added as above, which match each element of the `(attribute-name . namespace)` cons used by xmls to represent namespaces.
3. A designator for a function. This is called on the attribute name (which may be a string or a cons, as described above) and returns a boolean representing whether or not the function matches the attribute name.

how-many—a symbol indicating how many times the attribute can match children of its parent tag. One of:

1. `:one`—matches one
2. `:one-or-more`—matches one or more
3. `:zero-or-more`—matches zero or more
4. `:zero-or-one`—matches zero or one (an optional matcher, in other words)

possible-values—a list of strings that are all the possible values of the attribute.

type—a type specifier for the type of the attribute's value when it has been read by the Common Lisp reader.

matches-regexp—a regular expression which the attribute's value must match. Note that "^" and "$" are added to the start and end of the regular expression.

test-fn—a designator for a function that takes the attribute's value as an argument and returns a boolean indicating whether or not the value should be matched. This lets you use the full power of Common Lisp in your sanity checking.

matcher—the matcher object returned.

3.6 Character data—the `pcdata` matcher

The `pcdata` matcher matches ordinary text—just strings, the actual contents of tags. There are, of course, a number of constraints you can put on the values of the strings.

Syntax:

pcdata *&key matches-regexp how-many test-fn possible-values type* → *matcher*

Arguments and Values:

matches-regexp—a regular expression which the text must match. Note that "^" and "$" are **not** added to the start and end of the regular expression. This is inconsistant, I know. However, it is more conventient this way.

how-many—a symbol indicating how many times the matcher can match children of its parent tag. One of:

1. `:one`—matches one
2. `:one-or-more`—matches one or more
3. `:zero-or-more`—matches zero or more
4. `:zero-or-one`—matches zero or one (an optional matcher, in other words)

test-fn—a designator for a function that takes the text as an argument and returns a boolean indicating whether or not the text should be matched. This lets you use the full power of Common Lisp in your sanity checking.

possible-values—a list of strings that are all the possible values of the text.

type—a type specifier for the type of the pcdata's value when it has been read by the Common Lisp reader.

matcher—the matcher object returned.

3.7 The `match-anything` matcher

Sometimes you want to excuse some part of your XML document from checking for some reason. For these regions, you want to use the `match-anything` matcher.

Syntax:

match-anything *&key how-many* → *matcher*

Arguments and Values:

how-many—a symbol indicating how many times the matcher can match children of its parent tag. One of:

1. `:one`—matches one
2. `:one-or-more`—matches one or more
3. `:zero-or-more`—matches zero or more
4. `:zero-or-one`—matches zero or one (an optional matcher, in other words)

matcher—the matcher object returned.

4 The matcher API

It is possible to extend xml-psychiatrist to do things I never anticipated. I've tried to make sure that the standard matchers do everything that you could reasonably hope to do, but I can't be sure of that. Therefore, you can create new matchers if you so desire. This section documents how to make new matchers.

All matchers are subclasses of the `matcher` class (see 4.1). The main work is done in the `match` method:

match *matcher node* → *matched-p*

matcher—a matcher.

node—an xmls node.

matched-p—a boolean specifying whether or not `matcher` matched `node`.

You can get examples of matchers by looking at the source code.

4.1 The `matcher` class

All matchers are of the `matcher` class.

Slots:

how-many, initarg `:how-many`—a symbol indicating how many times the matcher can match children of its parent tag. One of:

1. `:one`—matches one
2. `:one-or-more`—matches one or more
3. `:zero-or-more`—matches zero or more
4. `:zero-or-one`—matches zero or one (an optional matcher, in other words)

name, initarg `:name`—a string or other data type specifying the name of the matcher, used in error reporting and debugging.

The base matcher class is not much good for real use. For actual matching, you must subclass it.

4.2 The `fun-matcher` class

To make it simpler in some cases to write matchers, there is a matcher class called `fun-matcher` which only requires a function which it calls on a node and which returns a boolean.

Slots:

fun, initarg `:fun`—a function designator for the matcher function, which is called on an xmls node and returns a boolean to indicate whether or not the node is matched.

how-many, initarg `:how-many`—a symbol indicating how many times the matcher can match children of its parent tag. One of:

1. `:one`—matches one
2. `:one-or-more`—matches one or more
3. `:zero-or-more`—matches zero or more
4. `:zero-or-one`—matches zero or one (an optional matcher, in other words)

name, initarg `:name`—a string or other data type specifying the name of the matcher, used in error reporting and debugging.

Example:

```
(make-instance 'fun-matcher
               :fun #'(lambda (node)
                        (string= node "foo"))
               :how-many :one-or-more
               :name "foo-string")
```

4.3 Error reporting

It is possible to stop all validation and report failure with an error message. This is done with the `xml-validation-error` function. This is dangerous, however, since it does not play well with optional matching, and it is very difficult to truly understand the ramifications of stopping with an error.

The `xml-validation-error` function takes a format control string and arguments, and stops validation with the error produced by applying format to the format string and arguments.

Syntax:

xml-validation-error *format-string &rest arguments* → nil

Arguments and Values:

format-string—a format control string. This should not include any newlines.

arguments—arguments to format.

4.3.1 Leaf and branch matchers

It is important to understand the difference between two general types of matchers, *branch matchers* and *leaf matchers*. Leaf matchers match things like attributes and strings, while branch matchers are more concerned with structure, such as tag matchers. In general, leaf matchers should never call the `xml-validation-error` function, and branch matchers should be very careful about calling it. There are no hard and fast rules, unfortunately.

When in doubt, don't call `xml-validation-error`. Chances are, things will be just fine if you don't.

5 Advanced examples

Now that we know all about how xml-psychiatrist works and the various options available, we can define some much more complicated examples.

5.1 Person records

Let's start with a type of XML document. Each XML file will define a person. It looks like this, `person.xml`:

```
<person age="17" gender="male">
    <name>
        <first>Peter</first>
        <last>Scott</last>
    </name>
    <tagline>Hey, Steve!</tagline>
</person>
```

In plain English, a person has `age` and `gender` attributes, a `name` tag (containing `first` and `last` fields each containing some alphabetic text with no spaces), and a `tagline` tag containing that person's favorite thing to say. Let's write a matcher for that:

```
(use-package :xml-psychiatrist)
(use-package :xmls-utilities)

(toplevel-match
 (tag "person" ((attr "age" :type '(integer 0 170))
               (attr "gender" :possible-values '("male" "female")))
      (tag "name" ()
           (tag "first" ()
                (pcdata :matches-regexp "^[a-zA-Z]+$"))
           (tag "last" ()
                (pcdata :matches-regexp "^[a-zA-Z]+$")))
      (tag? "tagline" ()
            (pcdata)))
 (parse-xml-file "person.xml"))
```

Look that over until you understand it. You can look up all the little bits; this is an example of how you put them together. Try messing up `person.xml` and seeing what happens. To get you started, let's try making a very old person:

```
<person age="65000000" gender="male">
    <name>
        <first>Peter</first>
        <last>Scott</last>
    </name>
    <tagline>Hey, Steve!</tagline>
</person>
```

When we validate this, it should choke when it sees the claim that this person is 65 million years old. Let's try running our validation code again:

```
NIL
"In \"person\" tag, the attribute \"age\" is not allowed here."
```

Sure enough, the validator caught the mistake. It's a bit on the stupid side, so it says that the attribute `age` is not allowed there rather than that the attribute age is way too big, but at least you get some idea of where the problem is.

5.2 Hosts files

Many programs require lists of hosts on a network, along with some information about what to do with them. Suppose that you were writing a network monitor like Nagios, and you needed to perform certain checks on hosts on a network. You might ping a host to make sure it's still up and running, you might connect to an HTTP server to check that it is running, you might try to open a POP3 connection with an email server, or you might try a number of other things. What would a list of hosts look like? Let's look at a possible XML format for it, `hosts.xml`:

```
<hostlist>
  <host name="Kestrel">
     <ip>12.34.65.212</ip>
     <description>email server</description>
     <checks>
        <check>ping</check>
        <check>pop3</check>
        <check>imap</check>
        <check>smtp</check>
     </checks>
  </host>

  <host name="Pokey">
```

```
    <ip>114.43.1.12</ip>
    <description>A desktop computer</description>
    <checks>
      <check>ping</check>
    </checks>
  </host>
</hostlist>
```

This defines two hosts, Kestrel and Pokey. Kestrel is an email server and should be checked extensively. Pokey is just a desktop computer and should just be pinged regularly to make sure it's running. Both have IP addresses specified.

How would we validate this? Let's start by looking at the format of the hosts file. A `hostlist` is made up of one or more `hosts`, each of which has a `name` attribute which is composed of alphanumeric characters and no spaces or special symbols, an `ip` child tag with an IP address, an optional `description` and a list of `checks`. We can write a basic validator like this:

```
(use-package :xml-psychiatrist)
(use-package :xmls-utilities)

(toplevel-match
 (tag "hostlist" ()
      (tag+ "host" ((attr "name" :matches-regexp "[a-zA-Z]+"))
            (tag "ip" ()
                 (pcdata :matches-regexp "^[0-9][0-9]?[0-9]?\.
[0-9][0-9]?[0-9]?\.[0-9][0-9]?
[0-9]?\.[0-9][0-9]?[0-9]?$"))
            (tag? "description" ()
                  (pcdata))
            (tag "checks" ()
                 (tag+ "check" ()
                       (pcdata)))))
 (parse-xml-file "hosts.xml"))
```

This handles most problems. If the XML file doesn't have any checks defined, it will tell you that. If the XML file misspells a tag, it will tell you. But there are a few small problems with this validator which could be exploited by a properly evil and intelligent person:

1. That impenetrable regular expression for matching IP addresses just makes sure that there are four numbers, each containing 1–3 digits, separated by dots. An IP address, though, consists of four numbers between 0 and 255

separated by dots. Perhaps we could match this with a regular expression, but it wouldn't be very pleasant.

2. The contents of the `check` tags should be checked to ensure that they are valid checks. Your network monitor should *not* have to puzzle over the meaning of `<check>mak shur evrythings wrking haha LoL!!!</check>`. This can be done simply by specifying `:possible-values` for the PC-DATA matcher in the `check` tags. If we wanted to, we could also check to make sure that none of the checks was repeated, but this isn't worth the trouble. Repeating a check is just telling you again to do it.

5.2.1 IP addresses

An IP address, as mentioned before, consists of four numbers between 0 and 255 separated by dots. If we don't want to use a regular expression for this, we should use some Lisp code. Specifically, we need a function which takes a string as an argument and returns a boolean based on whether or not the string is a valid IP address:

```
(defun ip-address-p (string)
  (multiple-value-bind (matched groups)
      (cl-ppcre:scan-to-strings
       "^([0-9]{0,3})\.([0-9]{0,3})\.([0-9]{0,3})\.([0-9]{0,3})$"
       string)
    (let ((numbers (mapcar #'read-from-string (coerce groups 'list))))
      (and matched
           (every #'(lambda (number)
                      (<= 0 number 255))
                  numbers)))))
```

This function makes sure that a basic regular expression matches and uses it to pull apart the string. It then checks that each of the four numbers are in the proper range. Let's test it:

```
* (ip-address-p "12.36.212.56")

T
* (ip-address-p "12.00000.212.56")

NIL
* (ip-address-p "12.512.212.56")

NIL
```

```
* (ip-address-p "12.512.212.hello")

NIL
```

It sure seems to work. Now that we have this, though, what do we do with it? We plug it into our validation code:

```
(toplevel-match
 (tag "hostlist" ()
      (tag+ "host" ((attr "name" :matches-regexp "[a-zA-Z]+"))
            (tag "ip" ()
                 (pcdata :test-fn #'ip-address-p))
            (tag? "description" ()
                  (pcdata))
            (tag "checks" ()
                 (tag+ "check" ()
                       (pcdata)))))
 (parse-xml-file "hosts.xml"))
```

Try messing up IP addresses in the `hosts.xml` file and running the new code. It now detects problems!

5.2.2 Restricting check names

We want to make sure that the contents of the `check` tags are all valid, so we first need to make a list of valid checks:

```
(defparameter *valid-checks* '("ping" "pop3" "imap" "smtp" "http")
  "List of valid check names")
```

Now that we've decided what all the valid check names should be, we can change our validation code to reflect this:

```
(toplevel-match
 (tag "hostlist" ()
      (tag+ "host" ((attr "name" :matches-regexp "[a-zA-Z]+"))
            (tag "ip" ()
                 (pcdata :test-fn #'ip-address-p))
            (tag? "description" ()
                  (pcdata))
            (tag "checks" ()
```

```
                  (tag+ "check" ()
                        (pcdata :possible-values *valid-checks*)))))
 (parse-xml-file "hosts.xml"))
```

Now if we try to enter something in a `check` tag other than “ping”, “pop3”, “imap”, “smtp”, or “http”, validation will fail.