# The Pattern Datatype

**16 March 2018**

*Editorial note —* This is a **first public draft** of a standard defining a regular expression dialect for use in FHISO standards. This document is not endorsed by the FHISO membership, and may be updated, replaced or obsoleted by other documents at any time.

The public tsc-public@fhiso.org mailing list is the preferred place for comments, discussion and other feedback on this draft.

| | |
|---|---|
| Latest public version: | `https://fhiso.org/TR/patterns` |
| This version: | `https://fhiso.org/TR/patterns-20180316` |

*Editorial note —*

This document defines a "least-common denominator" regular expression dialect. An explicit goal is to have all *patterns* be trivially modifiable to work with as many mainstream regular expression engines and libraries as possible.

In particular, in interest of compatibility, the *pattern* in this document

- each defines a regular language (does not include back references).
- does not have named or general category classes (`[:alphanum:]`, `\pL`, etc.).
- does not define capturing groups, and thus does not need or support lazy quantifiers.
- does not have partial-string matching, and thus does not need to define if `(a|an)` and `(an|a)` match differently.

## 1 Introduction

### 1.1 Conventions used

Where this standard gives a specific technical meaning to a word or phrase, that word or phrase is formatted in bold text in its initial definition, and in italics when used elsewhere. The key words **MUST**, **MUST NOT**, **REQUIRED**, **SHALL**, **SHALL NOT**, **SHOULD**, **SHOULD NOT**, **RECOMMENDED**, **NOT RECOMMENDED**, **MAY** and **OPTIONAL** in this standard are to be interpreted as described in [RFC 2119].

An application is **conformant** with this standard if and only if it obeys all the requirements and prohibitions contained in this document, as indicated by use of the words MUST, MUST NOT, REQUIRED, SHALL and SHALL NOT, and the relevant parts of its normative references. Standards referencing this standard MUST NOT loosen any of the requirements and prohibitions made by this standard, nor place additional requirements or prohibitions on the constructs defined herein.

> *Note* — Derived standards are not allowed to add or remove requirements or prohibitions on the facilities defined herein so as to preserve interoperability between applications. Data generated by one *conformant* application must always be acceptable to another *conformant* application, regardless of what additional standards each may conform to.

If a *conformant* application encounters data that does not conform to this standard, it MAY issue a warning or error message, and MAY terminate processing of the document or data fragment.

This standard depends on FHISO's **Basic Concepts for Genealogical Standards** standard. To be *conformant* with this standard, an application MUST also be *conformant* with [Basic Concepts]. Concepts defined in that standard are used here without further definition.

> *Note* — In particular, precise meaning of *string*, *character*, *term*, *datatype*, *structured non-language-tagged datatype*, and *pattern* are given in [Basic Concepts].

Indented text in grey or coloured boxes does not form a normative part of this standard, and is labelled as either an example or a note.

> *Editorial note* — Editorial notes, such as this, are used to record outstanding issues, or points where there is not yet consensus; they will be resolved and removed for the final standard. Examples and notes will be retained in the standard.

The grammar given here uses the form of EBNF notation defined in §6 of [XML], except that no significance is attached to the capitalisation of grammar symbols. *Conforming* applications MUST NOT generate data not conforming to the syntax given here, but non-conforming syntax MAY be accepted and processed by a *conforming* application in an implementation-defined manner.

This standard uses *prefix notation* when discussing specific *terms*. The following *prefix* bindings are assumed in this standard:

| | |
|---|---|
| `types` | `https://terms.fhiso.org/types/` |

## 2 Pattern

As defined in [Basic Concepts], a *pattern* is a regular expression intended to provides a constraint on the *lexical space* of a *datatype*. This document defines both the `types:Pattern` *datatype* and the semantics of what it means for a *string* to *match* a *pattern*.

### 2.1 Matching and Languages

A **pattern** is an element of the `types:Pattern` *datatype*. Every *pattern* is said to **match** a (possibly-infinite) set of *strings*. The set of *strings* that a particular *pattern matches* is defined by the contents of the *pattern* itself, using the rules specified in this document.

*Editorial note* — This draft is inconsistent in its use of "match" vs "*match*". It is not clear which is preferable.

[Basic Concepts] currently refers to "match" as an undefined word, which is convenient because not all uses of "match" in that document refer to matching a pattern (others match EBNF productions, Accept headers, informal requirements, etc.) That flexibility is supported by not formally defining "match".

Conversely, much of this document presents a formal definition of what it means to *match* a *pattern* or its component parts. That formality suggests "*match*" should be a formally-defined word.

As an alternative, the entire document could be re-written to define the "language of" a *pattern* as many theoretical computer science text do, which would greatly reduce the occurrences of the word "*match*". This draft has taken the position that "language" it too important a word in other genealogical contexts for its formal definition herein to be wise.

*Note* — Those familiar with theoretic computer science might be used to referring to the set of *strings* matched by a *pattern* as the "language of" that *pattern*, and referring to the *pattern* itself as a "regular expression". Since these terms are not necessary to define the semantics of a *pattern*, and since "language" in particular has a second, more common, and very important meaning in genealogy, this document does not make normative use of those terms.

*Example* — Consider the *pattern* "`[ab][cd]?`". The set of *strings* that that *pattern matches* is {"`a`", "`ac`", "`ad`", "`b`", "`bc`", "`bd`"}. We say that "`[ab][cd]?`" *matches* the *string* "`a`" and does *not match* the string "`aa`".

## 2.2 Hierarchical Definition of Patterns

This section presents a set of definitions that fully define both the *lexical space* of the `types:Pattern` *datatype* and the set of *strings* matched by a given *pattern*. It does this by introducing and naming several intermediate concepts. These intermediate concepts are presented for expository purposed only and may be changed or removed from future versions of this standard. In particular, this documents' presentation of the following key words SHOULD NOT be directly referenced in other documents: *regular expression*, *branch*, *piece*, *quantifier*, *atom*, *normal character*, *metacharacter*, *banned character*, *escaped character*, *character class*, *positive character class*, *negative character class*, *character range*, *wildcard*.

> *Editorial note —* The list of key words defined in this document that SHOULD NOT appear in other documents is inelegant, but defining these terms is not a principle goal of this document either. However, definitions of patterns that do not name (at least most of) these parts are hard to follow.

### 2.2.1 Components that Match Strings

The following components define portions of *patterns* which match *strings*

#### 2.2.1.1 Regular Expression

A **regular expression** consists of one or more *branches*. Between each *branch* is a single U+007C | character.

```
regExp ::= branch ( '|' branch )*
```

The set of *strings matched* by a *regular expression* is the union of the sets of *strings matched* by its *branches*.

#### 2.2.1.2 Branch

A **branch** consists of one or more *pieces*. The *pieces* of a *branch* appear adjacent to one another with no intervening characters.

```
branch ::= piece piece*
```

A *branch matches* a *string* $s$ if and only if a prefix of $s$ matches the first *piece* of the *branch* and the remainder of $s$ matches the remaining *pieces*.

> *Editorial note —* While the above is definition of *branch matching* does not appear to be ambiguous, is it formally incomplete as it does not define what it means for a *string* to "match the remaining pieces". The following is more rigorous, but also more verbose:
>
> > A sequence of $n$ *pieces* (where $n \geq 2$) *matches* a *string* $s$ if and only if $s$ can be represented as a concatenation of $n$ *strings* $s_1$ $s_2$ ... $s_n$ such that $s_1$ *matches* the first *piece* in the sequence, $s_2$ matches the second *piece*, and so on with $s_n$ matching the last *piece*.
> >
> > A *branch matches* a *string* if either (a) the *branch* consists of only a single *piece* and that *piece* matches the *string*, or (b) the *branch* consists of a sequence of *pieces* and that sequence of *pieces matches* the *string*.
>
> Feedback is invited on which version is preferable.

#### 2.2.1.3 Piece

A **piece** consists of an *atom*, possibly followed by a **quantifier**.

```
piece ::= atom quantifier?
quantifier ::= [?*+] | ( '{' quantity '}' )`
quantity ::= quantRange | quantMin | QuantExact
quantRange ::= QuantExact ',' QuantExact
quantMin ::= QuantExact ','
QuantExact ::= 0 | [1-9] [0-9]*
```

The set of *strings matched* by a *piece* depends on the *quantifier* used. If S is an *atom* and *L*(S) is the set of strings matched by *S* then

| Piece | Set of strings matched |
|---|---|
| S | *L*(S) |
| S ? | the empty string, and all strings in *L*(S) |
| S * | all concatenations of zero or more strings in *L*(S) |
| S + | all concatenations of one or more strings in *L*(S) |
| S {n,m} | all concatenations of at least *n* and no more than *m* strings in *L*(S) |
| S {n} | all concatenations of exactly *n* strings in *L*(S) |
| S {n,} | all concatenations of at least *n* strings in *L*(S) |

When the above table refers to "strings in *L*(S)", the strings do not need to be distinct.

> *Example —* If *L*(S) is {"a", "b"} then *L*(S *) includes an infinite number of strings, including "", "a", "b", "aa", "ab", "ba", "bb", "aaa", etc.

> *Example —* If *L*(S) is {"a", "b"} then *L*(S {3}) contains 8 strings: {"aaa", "aab", "aba", "abb", "baa", "bab", "bba", "bbb"}.

> *Note —* The above omits {,n}, which some regex dialects allow as a shorthand for {0,n}.

> *Note —* The above omits the lazy quantifiers *?, +?, etc., which some dialects allow to select between derivations of a particular string.

> *Note —* The above prohibits {02,12} and other leading zeros in quantities.

#### 2.2.1.4 Atom

An **atom** is either a *normal character*, an *escaped character*, a *character class*, or a parenthesised *regular expression*.

```
atom ::= NormalChar | escapedChar | charClass | '(' regExp ')'
```

An *atom* that is a parenthesized *regular expression* matches the same set of *strings* as its *regular expression* (the parentheses do not directly contribute to the *match*).

An *atom* that is a *normal character* or *escaped character* matches any single-character string containing the character represented by the *normal character* or *escaped character*.

An *atom* that is a *character class* matches any single-character *string* containing a character within the *character class*.

### 2.2.2 Components that represent characters

The following components define portions of *patterns* which represent single *characters*

#### 2.2.2.1 Normal Character

A **normal character** is any *character* that is not a *metacharacter* or a *banned character*.

Each *normal character* represents itself.

The **metacharacters** are '`.`', '`\`', '`?`', '`*`', '`+`', '`{`', '`}`', '`(`', '`)`', '`|`', '`[`', and '`]`'.

The **banned characters** are '`^`', '`$`', '`&`', '`/`', and the escapable control characters U+0009, U+000A, and U+000D.

> *Note* — The above REQUIRES that *metacharacters* do not appear as *normal characters* unescaped. Some dialects are more permissive, allowing e.g. `}` to appear unescaped, but that MUST NOT be done in *patterns*.

> *Note* — The *banned characters* have special meaning in some regular expression dialects, and as such MUST NOT appear unescaped in any *pattern*.

> *Editorial note* — The set of *banned characters* was selected by a survey of several regular expression dialects, but may be incomplete. Community input on other characters that may have special meaning in some dialects is invited.

#### 2.2.2.2 Escaped Character

An **escaped character** is a U+005C `\` followed by a single character, which must be a *metacharacter*, a *class metacharacter*, a *banned character*, or one of U+0074 `t`, U+006E `n`, or U+0072 `r`.

An escaped *metacharacter*, *class metacharacter*, or *banned character* represents the *metacharacter*, a *class metacharacter*, or *banned character* itself.

An escaped U+0074 `\t` represents the *character* U+0009 (the horizontal tab).

An escaped U+006E `\n` represents the *character* U+000A (the line fed).

An escaped U+0072 `\r` represents the *character* U+000D (the carriage return).

> *Note* — Some dialects of regular expressions allow any character to be escaped without special meaning, but others do not or have additional special meanings for some characters (such as `\f`, `\A`, etc). For maximal compatibility, *patterns* MUST NOT escape characters other than those listed above.

> *Note* —
>
> Code-point escapes (e.g., `\x{2F2E}` for ⼮) are not provided in this specification because they are not supported in some common regular expression engines such as POSIX and XML. Instead, Unicode should be expressed in the same encoding used by the *strings* being checked for membership in a regular expression's language.
>
> If the chosen engine is byte- rather than code-point-oriented, care should be made that (a) quantifiers bind to characters, not bytes; and (b) character class ranges are correctly handled. Binding can be achieved by adding parentheses around each multi-byte character; how to achieve character class ranges is not known in general by the authors of this specification.

#### 2.2.2.3 Class Character

A **class character** is either an *escaped character* or a single character that is neither a *class metacharacter* nor a *banned character*.

The **class metacharacters** are '.', '\', '-', '|', '[', and ']'.

> *Example* — Because *banned characters* are not permitted unescaped as *class characters*, "`[A-^]`" is not a *pattern* (as it includes the *banned character* '^') even though it is accepted by some regular expression engines.

> *Editorial note* — *class character* might be clearer if we add a definition for *class normal character* (like we do a *normal character*)

### 2.2.3 Components that define sets of characters

The following components define sets of individual characters.

#### 2.2.3.1 Character Class

A **character class** is either a *positive character class*, a *negative character class*, or a *wildcard*.

```
charClass ::= posCharClass | negCharClass | wildcard
```

#### 2.2.3.2 Positive Character Class

A **positive character class** is a set of one or more *character ranges* within brackets.

```
posCharClass ::= '[' charRange+ ']'
```

A *positive character class* defines the union of the sets defined by its *character ranges*.

> *Note* — The ranges do not need to be mutually exclusive nor presented in any particular order.

#### 2.2.3.3 Character Range

A **character range** is either a single *class character* or two *class characters* separated by a U+002D `-`.

```
charRange ::= classChar | classChar '-' classChar
```

If a character range has two *class characters* the second MUST NOT have a numerically lesser code point than the first.

A single-*class character character range* defines the singleton set containing only the character that its *class character* represents. A two-character *character range* defines the set of all *characters* with code points that are both

- numerically greater than or equal to the code point of the *character* that the first *class character* represents, and
- numerically less than or equal to the code point of the *character* that the second *class character* represents.

#### 2.2.3.4 Negative Character Class

A **negative character class** is a set of one or more *character ranges*, preceded by U+005E `^`, within brackets.

```
negCharClass ::= '[^' charRange+ ']'
```

A *negative character class* defines the set of all *characters* that are not within the union of the sets defined by its *character ranges*.

#### 2.2.3.5 Wildcard

A **wildcard** is represented as U+002E `.`.

```
wildcard ::= '.'
```

The *wildcard* defines the set of all *characters*.

> *Note* — The above definition includes new line characters in `.`. When using an engine that does not do so, replace all `.` with something else, such as `(.|[\r\n])`, `(.|\s)`, or `[\s\S]`. Which one works depends on the engine in question.

## 2.3 The `types:Pattern` datatype

FHISO uses the `types:Pattern` *datatype* to represent *patterns*. It MUST NOT be used for *pattern*-like regular expression variants that do not conform to this standard's definition of a *pattern*.

> *Example* — In ECMAScript, "`/^.[.]$/`" is a regular expression that matches a two-character string ending with a period. Because this is not a valid *pattern*, it does not have the `types:Pattern` *datatype*.

The *lexical space* of this *datatype* is the space of all *strings* that match the `regExp` production in §2.2.1.1.

> *Example* — Thus the *string* "`([A-Z][a-z]+ )*`" is within the *lexical space* of this *datatype*, but "`^\x{FFEF}.*$`" is not, despite being a valid regular expression in some engines.

This is a *structured non-language-tagged datatype* which has the following properties:

**Datatype definition**

| | |
|---|---|
| Name | `https://terms.fhiso.org/types/Pattern` |
| Type | `http://www.w3.org/2000/01/rdf-schema#Datatype` |
| Pattern | `.*` |
| Supertype | *No non-trivial supertypes* |
| Abstract | `false` |

## 3 Dialect Guide

> *Note* — This entire section is non-normative.

> *Editorial note* — This section is incomplete, and mostly added to as a sanity-check to see if engines I use can handle these regexs. It should probably either be removed or completed.

Following are some suggestions for making regular expressions in the above dialect work with various regular expression engines.

**C++11 std::regex**
: Use the `ECMAScript` variety and `regex_match` (not `regex_search`). Replace non-escaped `.` with `(.|\s)`.

**C++ boost::regex**
: Use the `ECMAScript` variety. How to ensure full match not known to the author of this document.

**ECMAScript**
: Surround expression with `^(…)$`. Replace non-escaped `.` with `(.|\s)`.

**Java** Surround expression with `^(?s…)$`.

**.NET** Use the `RegexOptions.Multiline` option or replace non-escaped `.` with `(.|\n)`. Surround expression with `^(…)$`.

**Perl** Use `m/^(…)$/s`.

**PCRE**
: Use the `PCRE_UTF8` option. Surround expression with `^(…)$`.

**PCRE2**
: Use the `PCRE2_UTF` and `PCRE2_DOTALL` options. Surround expression with `^(…)$`.

**PHP** Surround expression with `/^(…)$/us` with the `preg_…` functions.

**POSIX**
: Requires extensive modifications. Basic mode required escaping metacharacters. Things that do not require escaping may forbid escaping and require pre-processing to strip `\s`.

**Python**
: Use the `re.DOTALL` option. In Python 3.4 and later, use the `fullmatch` function; otherwise use `match` and surround the expression with `(…)$`.

**Ruby**
: Surround expression with `/\A(…)\Z$/m`.

**XML** Replace non-escaped `.` with `[\s\S]`.

# 4 References

## 4.1 Normative references

**[Basic Concepts]**

FHISO (Family History Information Standards Organisation). *Basic Concepts for Genealogical Standards*. First public draft. (See https://fhiso.org/TR/basic-concepts.)

**[RFC 2119]**

IETF (Internet Engineering Task Force). *RFC 2119: Key words for use in RFCs to Indicate Requirement Levels.* Scott Bradner, eds., 1997. (See https://tools.ietf.org/html/rfc2119.)

**[XML]**

W3C (World Wide Web Consortium). *Extensible Markup Language (XML) 1.1*, 2nd edition. Tim Bray, Jean Paoli, C. M. Sperberg-McQueen, Eve Maler, François Yergeau, and John Cowan eds., 2006. W3C Recommendation. (See https://www.w3.org/TR/xml11/.)

---