**Open DICOM*web* Project**

**Validation Profiles**

Table of Contents

[1 Syntax 2](#_Toc460766387)

[1.1 ABNF List Extension: # Rule 2](#_Toc460766388)

[2 Expressions 2](#_Toc460766389)

[3 Value Expressions 3](#_Toc460766390)

[3.1 Variables 3](#_Toc460766391)

[3.2 Numbers 3](#_Toc460766392)

[3.2.1 Integers 3](#_Toc460766393)

[3.2.2 Decimal Numbers 4](#_Toc460766394)

[3.2.3 Numbers 4](#_Toc460766395)

[3.3 Strings 4](#_Toc460766396)

[3.3.1 Character Escapes 5](#_Toc460766397)

[3.3.2 Code Point Escapes 5](#_Toc460766398)

[3.3.3 String Syntax 6](#_Toc460766399)

[3.4 Arrays 6](#_Toc460766400)

[3.5 Functions 6](#_Toc460766401)

[4 Profile Expressions 6](#_Toc460766402)

[4.1 Environment 6](#_Toc460766403)

[4.2 Rules 7](#_Toc460766404)

[4.3 Private Groups 8](#_Toc460766405)

[4.4 Profile Definition 8](#_Toc460766406)

[5 Appendix A 9](#_Toc460766407)

Profiles are defined by a JSON Object. The JSON Object is composed of identifier/expression pairs, where the identifier is delimited by double quotes.

# Syntax

The Profile language is defined by a grammar, which is written in using ABNF, which is defined in [RFC5234]. This grammar uses many of the basic syntax rules defined in [Appendix B1], which are reproduced for the reader's convenience in Appendix A of this document. Basic rules are all uppercase. They typically resolve to terminal values in one evaluation step.

The basic rules have been extended with the following rules, which are used throughout this grammar.

BACKSPACE = %x08 ; backspace

VTAB = %x0B ; vertical tab

FORMFEED = %x0C ; form feed

NEWLINE = LF / CR LF ; depends on the operating system

OWS = \*WSP ; optional whitespace

DQ = DQUOTE ; short form of double quote

GROUP = "0x" 4HEXDIG ; a DICOM group

TAG = "0x" 8HEXDIG ; a DICOM Data Element Tag

{ = "{" ; syntactic sugar

} = "}" ; ...

## ABNF List Extension: # Rule

A **#** syntax rule is an extension to the ABNF rules of [RFC5234]. It is used to improve readability. The **#** rule is similar to **\*** and is used to define comma-delimited lists of syntactic expressions or sexp.

The full form is:

<a>\*<b>sexp

where <a> and <b> are optional integer values, indicating at least <a> and at most <b> occurrences of the sexp. Each sexp, except the last, is separated from the next sexp with a comma (','). If zero or one sexp are present, then no commas are generated.

Default values are a = 0 and b = infinity so that #<sexp> allows any number, including zero; 1\*<sexp> requires at least one; 3\*3<sexp> allows exactly 3; and 1\*2<sexp> allows one or two.

*Note*: This is different from the list extension rule defined in [RFC7030], which requires at least one sexp.

# Expressions

There are two types of expressions: value expressions and profile expressions. Value expressions (vexp) compute values that are stored in data elements. There are five types of value expressions:

vexp = variable / number / string / array / f-call

When evaluated, a value expression computes a literal value, typically a string.

Profile expressions are used to define different aspects of a Profile. There are three types of profile expressions:

pexp = environment / private-group / rules

Value expressions can be used in Profile expressions, but not vice versa.

# Value Expressions

There are five different types of value expressions: 1) variable references, 2) literal numbers,

3) interpolated strings, 4) arrays, and 5) function calls. The following subsections defined the syntax and meaning of these expressions.

The values that can be produced by evaluating a value expression are: a number, a string, a list of numbers, or a list of strings.

## Variables

Variable expressions define an identifier that is associated with a value expression. Variables are defined in an environment (see Section x.y), which is a JSON object that contains identifier/vexp pairs.

An *Identifier* is a string that has at least one (initial) character. The initial character may be an alphabetic or '\_' character. Subsequent characters may be alphabetic, numeric or '\_'. Identifiers are case insensitive. So, "abc", "aBc", and "ABC" represent the same identifier.

Figure X contains the syntax for Identifiers.

**Figure X: Identifier Syntax**

initial = ALPHA / "\_"

subsequent = initial / DIGIT

identifier = initial \*subsequent

The details of defining variables in an environment are discussed in Section 4.X.

## Numbers

While JSON only allows a single type of number, the Profiler provides additional number types that correspond to the various DICOM Value Representations (VR).

### Integers

The general type of integers is integer. Integers are sub-divided into signed and unsigned integers. Unsigned integers are always non-negative and have no sign. Unsigned integers have the prefix uint and trailing digits that denote the maximum number of bits in a value of that type. There are 3 specific types of uint:

uint8, uint16, uint32

Signed integers have an optional sign. They have the prefix int and trailing digits that denote the maximum number of bits in a value of that type. There are 2 specific types of int:

int16, int32

### Decimal Numbers

The general type of if non-integer numbers is float. There are three specific subtypes of decimal numbers:

decimal, float32, float64

The decimal type is a string containing a decimal number in scientific notation. The float32 and float64 types are [IEEE 754](https://en.wikipedia.org/wiki/IEEE_floating_point) binary encodings of floating point numbers.

### Numbers

Figure X contains the syntax for numbers.

**Figure X: Number Syntax**

uint = 1\*DIGIT

uint8 = uint ; 8-bit unsigned integer

uint16 = uint ; 16-bit unsigned integer

uint32 = uint ; 32-bit unsigned integer

SIGN = "-" / "+"

integer = [SIGN] uint

int16 = integer ; 16-bit signed integer

int32 = integer ; 32-bit signed integer

EXPMARK = "E" / "e"

exponent = EXPMARK [SIGN] uint

decimal = [SIGN] uint ["."]

decimal =/ [SIGN] \*DIGIT "." uint [exponent]

decimal =/ [SIGN] uint ["." \*DIGIT] exponent

float32 = decimal ; 32-bit floating point number

float64 = decimal ; 64-bit floating point number

The types with trailing digits specify the maximum number of bits in their representation. If there are no trailing digits, the type is general.

Before storing decimal values, the Profiler will convert VRs of DS into the following canonical string format:

sign uint "." uint "E" sign uint

The SIGN, ".", "E" (canonical exponent marker) and uint shall be present. uints must have at least one digit, which may be zero.

## Strings

In general, the Profiler supports UTF-8 strings. However, all DICOM VRs do not support UTF-8 strings, so the Profiler will ensure that only valid strings are stored in elements.

The Profiler supports embedding expressions in strings, aka *Interpolated Strings*. When the string is evaluated each embedded expression is evaluated and the resulting values are converted (coerced) into strings and replace the embedded expression the enclosing string.

An unescaped $ character in a string signifies the beginning of an interpolated expression. The $ sign may be followed by either:

* a single identifier: $id
* a expression delimited by curly braces: ${vexp}

The form $id is equivalent to ${id}. An interpolated string "s1${vexp}s2" is equivalent to the concatenation of the strings "s1", exp.toString () and "s2". Where exp.toString() is the result of evaluating vexp and converting it into a string.

A string is said to be raw or fully evaluated when it no longer contains any escape sequences or interpolation expressions.

*Note*: *The expression inside the interpolation could itself include strings, which could again be interpolated recursively; however, the Profiler does not support recursive interpolation.*

### Character Escapes

Strings can be composed of any UTF-8 visible code point except **"**, **$**, and **\**, or NEWLINE. These characters must be escaped using the **\** character. There are also character escapes defined for certain control characters.

The following character escape sequences are defined:

**Figure X: Character Escapes**

\" = DQ

\$ = "$"

\\ = "\"

\b = BACKSPACE

\f = FORMFEED

\n = NEWLINE

\r = CR

\t = TAB

\v = VTAB

### Code Point Escapes

There are also code point escapes, which allow code point values to be embedded in strings. These are expressed using the following escape sequences.

**Figure X: Code Point Escapes**

\x 2HEXDIG

\u (4HEXDIG / {1\*6HEXDIG})

Strings may contain any visible UTF-8 code points except **"**, $, **\**.

### String Syntax

Figure X contains the syntax for strings.

**Figure X: String Syntax**

string = DQ (\*codepoint/iexp) DQ

codepoint = UTF8 / escape

iexp = "$"(identifier/{vexp})

codepoint = UTF8 / escape

utf8 = %x32-33 / %x35 / %x37-91 / %x93-7E / %x0100-FFFF

escape = "\" (esc-char / esc-code)

esc-char = "\" (DQ / "\" / "/" / "b" / "f" / "n" / "r" / "t" / "$")

esc-code = "\" (("x" 2HEXDIG) / ("u" 4HEXDIG) / ("u" {1\*6HEXDIG}))

TODO: Is the UTF8 rule sufficient?

## Arrays

Arrays have the following syntax:

array = "[" #vexp "]"

where #vexp denotes a comma separated list of zero or more value expressions. Arrays are used to compute the values of multi-valued elements. Once it has been evaluated, all values in an array must have the same type.

## Functions

### Global Lists

There are two global lists. One specifies elements that MUST be kept in the Dataset. The other specifies elements that should be removed from the Dataset.

An element may be on one list or the other but not both.

#### Keep

keep: [1#keyword]

A list of elements, specified by keyword, that should NOT be removed from the Study.

#### Remove

remove: [1#keyword]

A list of elements, specified by keyword, that should be removed from the Study.

#### Require

require: [1#keyword]

A list of elements, specified by keyword, that must be present in the Study.

### Global Functions

#### Private (*move to Profile*)

private(String creator, Map<vr, values>) map

The private function adds a set of private tags to a Dataset. The creator is a String that identifies the private group.

### Value Functions

The functions in this section change the contents of the Dataset; they

do not change the values of elements contained in the Dataset.

#### Age

target: PatientAge age()

The age function removes the PatientBirthData (0010,1010), and converts its value to the corresponding age relative to the current date and adds a PatientAge (0010,1010) to the Dataset with that value.

The target *must* be the PatientAge element.

For example:

PatientAge: age(PatientBirthDate)

Issue: Can this act on any other element than PatientAge?

#### Round

PatientAge: round({Keyword source: this, int size: 10})

The round function is used to create age groups. It rounds an age into a group or bin. The center of the first group is always at zero.

If the source has a VR of DA or DT, the value of the source is converted to an age relative to the current date. It then rounds the age into one of the groups, where each group contains size years.

For example, if the patient's age is 57, then:

PatientAge: round(10)

will equal 60.

Issue: what are valid targets? Isn't the source always PatientAge?

#### Always

target: always(var e)

The always function evaluates the expression, and if the target is present in the Dataset replaces its value with the value of the expression; otherwise, a target Element with that value is added to the Dataset.

Issue: This should have a better name.

### General Functions that Act on Elements

#### Add

target: add(List newValue)

The add function adds replaces the current value of the target with the value in the newValue argument. If target is not present in the Dataset, it is added to the Dataset with newValue as its value. newValue must be compatible with the VR of the target.

#### Contents

contents({ Keyword source: this,

RegExp regex: "",

String replacement: "",

String default: ""})

The contents function returns the value of the source element. If the source is not present it returns the default value.

TODO: finish.

#### encrypt({Keyword source: this, String key: "")

#### hash({Keyword source: this, uint maxChars, uint maxWords})

Issue: does this only work on strings?

#### process(Sequence sq)

Runs the de-identifier on the sequence.

Issue: Shouldn't this be the default action.

#### require({Keyword source: this})

If the target does not exist, it is added to the Dataset with the source's value. If the source does not exist, the value is empty.

If the target exists, this function does nothing.

### General Functions that Produce String Values

#### Append

append(String exp0)

or

append([ #e ])

The append function appends the value of its argument to the target's value. The target *must* be a multi-valued function, i.e. its VM *must* be greater than 1.

#### Blank

blank(int n)

The blank function returns a string containing n space characters. n must be a non-negative integer. If n = 0 the empty string ("") is returned.

The blank function can only be used with Elements with VRs that are strings.

Issue: Is blank really necessary since it can be denoted by a string of space characters, e.g." ".

#### Hash

hash({Keyword source: this})

Issue: how are the various hash functions different?

The hash function computes the hash of the source's value. If source is not present the target becomes the source. If the source has not value this function does nothing

The current hash algorithm is SHA256.

Note: this also performs the function of @hashptid.

#### Integer

integer(Keyword source: this, String type, uint width)

Issue: what is type?

#### Truncate

truncate({Keyword source: this, int n: 0})

The truncate function truncates the source's values.

If n is zero, the value is "". If n is positive the value is the first n characters of the source. If n is negative the value is the last n characters of the source.

### Date and Times Functions

#### Today

DA target: today({char separator: ""})

The values of the Date and Time functions are either strings or arrays of strings.

The today function returns the current date as a string. It has one argument, which specifies a separator character. The value returned by the today function has the following format:

yyyy\_mm\_dd

where yyyy is the current year, mm is the current month, and dd is the current day. \_ is a separator character that defaults to the empty string ("").

For example, if today is January 31, 1993, then:

today("-")

would return the following string:

"1993-01-31"

#### Date

date({ Keyword source: this,

int year,

int month,

int day,

separator: "", default:""})

All arguments are optional:

if source is omitted, the target becomes the source element.

If any of year, month, or day are missing the corresponding value in the source element is used.

If the source value is not present in the dataset or has no value, the default is used as the source.

If the target is not present in the Dataset, it is inserted into the Dataset.

The date function returns the date specified by its arguments as a string in the following format:

yyyy\_mm\_dd

where year, month, day are integers corresponding to the year, month, and date respectively, and separator is an optional argument that specifies the separator character ('\_'), if any.

The value of the date function can be used with any Data Elements that have VR that are string (SH, LO) or text (ST, LT, UT). It may also be used with Data Elements that have a VR of DA, in which case any separator`` value is ignored.

#### Increment Date

dateInc(Keyword source: this, Int days)

Issue: is element necessary?

The dateAdd function has one argument that is an integer. It specifies the number of days to be added to each of the date values contained in source, which must have a value representation of DA. dataAdd returns a string containing the resulting dates separated the current date plus days as a string. The value returned by the today function has the following format:

yyyy\_mm\_dd

where yyyy is the current year, mm is the current month, and dd is the current day. \_ is a separator character that defaults to the empty string ("").

#### Now

target: now(char separator = "")

The now function returns the current 24-hour time as a string. It has one argument, which specifies a separator character. The value returned by the now function has the following format:

hh\_mm\_ss

where hh is the current hour, mm is the current minute, and ss is the current second. \_ denotes a separator character that defaults to the empty string (""), which is the DICOM separator.

**\*\*Fix\*\***:

For example, if today is January 31, 1993, then:

date("-")

would return the following string:

"1993-01-31"

### Functions that Act on UIDs

#### Hash

hash({ Keyword source: this,

String root: "2.25.",

String suffix: ""})

The hash function creates a hash of the source Element, which must have a VR of UI.

If source is not present, then target must have a VR of UI

If target has a VR of UI the suffix is ignored. In general, it is best not to supply the root argument.

The current hash algorithm is SHA256.

Issue: the hash must be a valid UI – so 2.25.<hash>

### Functions that Act on the Elements with VR of PN

#### Initials

target: initials(Keyword source: this)

The VR of target must be PN.

Issue: Is this really needed?

### ### Functions that Act on the PatientID (0010,0010)

#### Lookup

lookup(Table table, action)

Returns the String associated with the Patient ID in the Trial.patientLookupTable.

What about actions: remove, keep, empty, abort; otherwise quarantine.

Issue: What other tables might be defined?

### Conditional Functions

#### If Exists

ifExists({ Keyword source: this, true: t, false: f})

If the source exists the value of the true expression is stored in target; otherwise, the value of the false expression is stored in target.

#### If Empty

ifEmpty({Keyword source: this, true: t, false: f})

If source is not present in the Dataset, or has no value or has a blank value (a value that consists of only space characters), then the then the value of the true expression is stored in target; otherwise, the value of the false expression is stored in target.

#### If Equals

ifEquals({Keyword source: this, String s, true: t, false: f})

If the source value is exactly equal to s, then the then the value of true expression is stored in target; otherwise, the value of false expression is stored in target.

#### If Contains

ifContains(Keyword source: this,String s, true: t, false: f)

If the source value contains s, then the then the value of true expression is stored in target; otherwise, the value of false expression is stored in target.

#### If Matches

ifMatches(Keyword source: this, String s, true: t, false: f)

If source value matches s, then the then the value of true expression is stored in target; otherwise, the value of false expression is stored in target.

pattern may be a String or a Regular Expression. Regular expressions use the JavaScript syntax.

#### Is Root Dataset

isRootDataset(true: t, false: f)

If the current Dataset is the Root Dataset for the Instance then true; otherwise, false.

Note: This is a new name for Select().

### Miscellaneous

#### Abort

TODO: What should this do?

#### Quarantine

quarantine()

Sends the study to the Trials Quarantine URL

TODO: add details

# Profile Expressions

There are five types of profile expressions: Environments, Private Groups, Profiles, Trials, and Sites.

## Environment

An environment is a JSON object that contains a set of variable definitions, which are also known as identifier/vexp pairs. The variable name is an identifier, and it value is expressed by the vexp. In the environment definition the identifier must be delimited by double quotes; however, in value expressions identifiers are not delimited by double quotes. In order to distinguish between these two forms of identifier the latter is referred to as a quoted-id. For example, in the following environment:

{

"TrialId": "98765",

"TrailName": "Some Trial Name with an Id=$TrialId"

}

Two variables are defined the first has the identifier TrialId and the second has the identifier TrialName. The value of TrialId is the string "98765", and the value of TrialName is the string "Some Trial Name with an Id=98765". This is because the value of TrialName is an interpolated string containing the variable TrialId. When the string is evaluated the value of TrialId, which is "98765", replaces the $TrialId expression in the string.

Each variable identifier in an environment must be unique. The variables defined in an environment are used in evaluating Profile Rules.

Environments have the following syntax:

env = {#var}

var = OWS quoted-id OWS ":" OWS vexp OWS

quoted-id = DQ identifier DQ

*Note*: DQ denotes a literal double quote character **"**, and OWS is optional whitespace.

Profiles, Trials, and Sites contain environments for defining and referencing variables.

## Rules

Each Profile defines a set of Rules using a JSON object. Each Rule has 1) a target Data Element keyword, and 2) a value expression. When a rule is evaluated, first the value expression is evaluated to determine its value. Then, if the value, satisfies the VR and VM of the target element, it is stored in that element.

Each target in a Profile must be unique and must refer to one of the DICOM Data Element keywords defined in PS3.6, Table 1-1.

TODO: Should we allow tags as well as keywords to specify the target?

The Profile Rules are defined using the following syntax:

rules = {#rules}

rules = OWS target OWS ":" 1\*(OWS vexp)

target = DQ keyword DQ

vexp = identifier / number / string /list / fcall

fcall = identifier "(" #arg ")"

arg = vexp

The rules in a Profile are evaluated in the tag order of the target elements. Each rule is fully evaluated before the of following rule is evaluated.

Each Rule is evaluated in left to right order as follows:

1. Variables are resolved against the current environment and are replaced by their values. If the value of a variable is a non-raw string, then the string is evaluated against the environment until it is fully resolved and the value is the fully resolved string. If the string cannot be fully resolved the Profiler will throw an error.
2. Non-raw strings are evaluated against the current environment until they are fully resolved.
3. Expressions in Lists are evaluated in left to right order until they are fully resolved.
4. Function arguments are evaluated in left to right order until they are resolved. The evaluation order for optional arguments is the order in which they are defined by their function.
5. If the function defines an argument that is a source element, and it is not present, then the value of the target element, if any, is used as the value of the source. If the target element does not exist or has no value. Its value is "" (empty string) for string VRs, and null for all other VRs.
6. Functions are evaluated once their arguments are all evaluated.
7. If the resulting value is a sequence of strings possibly separated by optional whitespace (OWS). The strings are concatenated together and the resulting string becomes the value of the rule.
8. If the resulting values are separated by commas, the value of the rule is the list of values.
9. Finally, the resulting value is stored in the target element if it exists. If it does not exist, the target may be created if the first function specifies that non-present elements shall be created.

Once all the rules have been evaluated and applied to their targets. The application of the Profile is complete.

## Private Groups

Figure X contains the syntax rules for Private Groups.

**Figure X: Private Group Syntax**

private = {#group}

group = creator ":" {#element}

creator = DQ identifier DQ

element = offset ":" "[" vr, values "]"

offset = "0x" 4HEX

vr = 2ALPHA ; a DICOM VR

values = "[" #value "]" ; an array of one or more vales

value = string / number

## Profile Definition

Profiles are specified in JSON. Figure X, is a template for a JSON Profile specification.

**Figure X: Validation Profile in JSON**

{

"Name": <identifier>,

"Version": <version>,

"LastModified": <date-time>

"Validated": <date-time>,

"Path": <url>,

"Quarantine": <url>,

"DeIdOptions": [],

"Comments": {

"lineNo": "Comment line"

},

"GlobalRules": {

"keepGroup18": bool,

"keepGroup20": bool,

"keepGroup28": bool,

"keepCurves": bool,

"keepOverlays": bool,

"removeCurves": bool,

"removeOverlays": bool,

"removeSafePrivate":

},

"Parameters": <parameters>,

"Rules": {

#(target : rule)

},

"Errors": {

"lineNo": "Error message"

}

"SourceLines": [\*<string>],

}

# Error Handling

TODO:

# Appendix A

The grammar defined in this document uses the following core rules from

**[**RFC5234**]**.

ALPHA = %x41-5A / %x61-7A

~~BIT= "0" / "1"~~

~~CHAR = %x01-7F~~

CR = %x0D ; carriage return

CRLF = CR LF ; Internet standard newline

CTL = %x00-1F / %x7F ; control characters

DIGIT = %x30-39

DQUOTE = %x22 ; The double quote character

HTAB = %x09 ; horizontal tab

LF = %x0A ; linefeed

~~LWSP = \*(WSP / CRLF WSP)~~

~~OCTET = %x00-FF ; 8 bits of data~~

SP = %x20

~~VCHAR = %x21-7E ; visible (printing) characters~~

WSP = SP / HTAB ; white space

**[**RFC5234**]: https://tools.ietf.org/html/rfc5234**

**[**Appendix B1**]: https://tools.ietf.org/html/rfc5234#appendix-B.1**

**[**RFC7030**]: https://tools.ietf.org/html/rfc7030**