The FHIR Specification includes a mapping language. The mapping language has a concrete syntax, defined and described in this page, and an abstract syntax, which is found in the [StructureMap](https://www.hl7.org/fhir/structuremap.html) resource. See also the [Tutorial](https://www.hl7.org/fhir/mapping-tutorial.html).

The mapping language describes how one set of Directed Acyclic Graphs (an instance) is transformed to another set of directed acyclic graphs. It is not necessary for the instances to have formal declarations and/or be strongly typed - just that they have named children that themselves have properties. On the other hand, when the instances are strongly typed - specifically, when they have formal definitions that are represented as [Structure Definitions](https://www.hl7.org/fhir/structuredefinition.html), the mapping language can use additional type related features.

The mapping language addresses two very different kinds of transformations:

* Structural changes between the source and target structures
* Differences in content and formats in string (and related) primitives contained within the structures

A map has 6 parts:

* Metadata
* Embedded [ConceptMaps](https://www.hl7.org/fhir/conceptmap.html) to translate between different code systems
* References to the structures involved in the mapping
* Imports: additional Maps used by this map
* A series of groups of transformations, each with a list of input variables
* The of transformation rules in each group

7.7.0.1 Executing Maps

Maps are executed by a mapping engine, which is created at execution time by a host software application. The mapping engine takes one or more inputs of data instances (directed acyclic graphs) and a map, and produces a set of outputs as specified by the map. The map is written to be aware of the structure of the input and output data instances, and contains the rules of how to convert between the two formats.

 The mapping engine can inspect the input data, to query an element in the instance for its children, to find its primitive value, and (optionally) its type; tThe mapping language can call on the engine for this information. The mapping language also requires that the engine has application support for the following operations:

* ValueSet validation operation
* Translation operation
* Lookup another tree of data
* Create an instance tree
* Return the correct string format to refer to a tree (input or output)

These functions constitute a Mapping Support API that makes maps portable between different systems

Generally, it is assumed the invocation of the engine follows some pattern like this:

* The host application creates the mapping engine, and passes it a handle to the standard services
* The host application opens the StructureMap resource, and asks the engine to prepare it (load, check, cache-up etc.)
* The host application asks the engine to execute and provides a set of resources, that are suitable for the the map
* Any created output will be created via the standard API

Some host applications may be able to determine how to combine maps and inputs on the fly based on their metadata, and require minimal configuration, while others may require manual arrangements in order to manage the map execution process.

7.7.0.2 General Syntax Notes

Mapping files are always plain text in Unicode. Whitespace is any Unicode whitespace, and the particular whitespace used is not significant, except that Unicode end of line characters terminate a comment. Comments are started by the characters "//" and can be found anywhere.

The abstract model includes documentation for each item. The canonical text representation is for each item to be on its own line, with documentation at the end of the line as a comment.

All names defined by the map language - group, rule and variable names - must be valid [ids](https://www.hl7.org/fhir/datatypes.html#id) (1-64 characters, upper and lowercase letters, numbers, dashes and dots). To avoid parsing ambiguities however, they cannot start with a character, cannot be one of the keywords used in the language (see section Reserved Keywords below) and cannot contain a dot or dash, unless the names can be escaped. Escaping can be done by surrounding them by backticks or double quotes. For example:

 src document4 "not-found" "section4.5" `group`

7.7.0.3 Metadata

The first part of the mapping syntax establishes the name of the mapping. For example:

 map "http://hl7.org/fhir/StructureMap/CodeSystem3to4" = "R3 to R4 Conversions for CodeSystem"

The letters "map" are the first non-whitespace non-comment characters in the source. This is followed by the canonical URL that identifies the map uniquely, and then a human readable name for the map, quoted in strings.

*todo: add additional metadata? Yes, maybe in comments like javadoc or C# xmldoc?*

7.7.0.4 Structure Definition References

The next optional section of the map references the set of structure definitions that are used or produced by this map. For example:

 uses "http://hl7.org/fhir/3.0/StructureDefinition/CodeSystem" alias CodeSystemR3 as source // documentation

 uses "http://hl7.org/fhir/StructureDefinition/CodeSystem" as target // documentation

This section lists one or more structure definitions that the map makes use of, and indicates for each structure definition, how it is used. It may also provide an alias - a name used for the type inside the mapping language - this may be necessary when transforming from source to target where both source and target use overlapping type names (not unusual). If no alias is given, the name for the type will default to the name given in the StructureDefinition (StructureDefinition.name).

Any kind of structure definition may be referenced, including data types, resources, constraints on those, and logical models.

There are 4 modes in which a structure definition may be used:

* **source**: One of more instances of this type are passed to the mapping engine when the mapping is executed, and serve as the source from which mapping is performed
* **queried**: The map may ask the (via the API) for some instances of this type. For further discussion, see [below](https://www.hl7.org/fhir/mapping-language.html#api)
* **target**: One or more instances of this type are passed in, and will be populated from the source material
* **produced**: The map may ask (via the API) for some instances of this type to be created. For further discussion, see [below](https://www.hl7.org/fhir/mapping-language.html#api)

The simplest case, which is common, is where a single structure is converted to another single structure. in this case, the map specifies one target, and one source. Such maps are easy to use automatically - the host application has content in one format, creates an empty instance of the target, and asks the mapping engine to convert.

However, many mappings are not so simple. For instance, converting from a single CDA document to FHIR typically creates a set of resources. In this case, there is a single target - a [Bundle](https://www.hl7.org/fhir/bundle.html), but it is also useful to specify a set of other structure definitions for resources that may be created as part of the bundle. Alternatively, converting from one source model to another might involve looking up other information in other instances of data.

It's also possible for a map not to specify any structure definition dependencies. A map that doesn't indicate any structure definitions can still be used, but the type features of the map language can't be used, and such maps typically require special development to integrate the execution of the map into an application.

7.7.0.5 Map Imports

This next optional section references additional maps that are used by this map. For example:

 imports "http://hl7.org/fhir/StructureMap/\*3to4" // documentation

Maps can be broken up into several files, each containing a coherent set of groups. For example, when writing mappings for CDA to FHIR, one might have one file to map the main document, and another file containing the mappings for the datatypes (e.g. CD to CodeableConcept). How imported maps are actually used is discussed below.

The url in the import statement may contain a "\*" as a wildcard character (as shown above) to include any matching maps that are available to the mapping engine.

7.7.0.6 Groups

Each Mapping source contains one or more groups, each containing one or more mapping rules. Each group declares a set of input and output variables that are shared by the rules. The in- and output variables define exactly which instances are passed to the mapping, and provides names by which they may be passed when invoking the map:

group [group-name] (inputs) (extends [other-group]) (<<stereotype>>) { // documentation

 .. rules ..

}

For example:

 group CodeSystem(source src : CodeSystemR3, target tgt : CodeSystem) extends DomainResource <<type+>>

 {

 // documentation

 .. rules ..

 }

Each group has a name, which is how the mapping is invoked. The first group is special, in that this is the group invoked if no name is provided (e.g. starting the mapping by a host application).

The inputs of a group are also referred to (below) as its input parameters, or just as parameters; or as input variables, and are a comma separated list where each items has the format:

 [mode] [name]( : [type])

Each input to the group has a name. All input variables have a mode, which may be one of source or target (see above). Inputs may have a type, but are not required to. There must be at least two input variables (source and target) - else there's nothing to map, except for the special case of the first group that may only have a single input. Groups may have additional input or output inputs, where that's necessary.

Groups may extend other groups, which means that the rules in the other group also apply (typically, this is used with specializing classes in an OO context). When a group extends another group, it SHALL have the same input parameters (by mode, name, and type if specified) though their order may differ, and it MAY have additional parameters.

The stereotypes <<types>> or <<type+>> can be added to the end of the group declaration to indicate that this group provides a set of mappings that are intended to be used as the default way to map from source to target. For more information, see the section on "Default mapping groups" below.

Default mapping groups SHALL have two parameters, a source, and a target, in that order, and both SHALL have specified types for the inputs.

7.7.0.7 Transform Rules

The main portion of a map consists of a set of transform rules that describe how source content is transformed into target content. The full format for a rule looks like this:

src\_context.field as new\_variable where condition -> tgt\_context.field = create([type]) as new\_variable then [details] "name";

For example:

 src.value : code as vs0 -> tgt.value = create("code") as vt0 then code(vs0, vt0) "valueCode";

Each rule has three main parts:

* **Source Content**: The initial part of the rule before the '->'. Specifies one or more elements from the source that contribute to the mapping
* **Target Transform**: Starts after the '->' and consists of zero or more specifications of how to use the source content to create the target content
* **Dependent Rules**: Optional part starting with 'then'. Specifies a group or set of child rules to apply within the scope of the rule.

Each rule may be assigned a name, though this is usually inferred by the parser and not specified directly. The name is used in trace logs (a record generated by the conversion engine recording the transform process). Names must be unique within the context of the map. Typically, the name is trivial and can be safely and usefully generated by the engine processing the map, so this is often left out.

The three main parts are described in more detail in the following sections.

7.7.0.7.1 Source Content

The source content is formed by one or more source statements, which can be assigned a variable name and then be used when specifying target content, or re-used in subsequent transforms and dependent rules. Multiple source statement are separated by a comma:

 [source], [source] -> ...

Each [source] contains the following items:

context.element { : type {min..max}} {default [value]} { list-option } as variable where [FHIRPath] check [FHIRPath]

For example:

 src.value : integer 0..\* default 10 first as vs0

 where value >= 10 check value <= 100

 log value

* **context.element**: The fist part of the source statement consists of two parts: the **context** and the **element**.

The context is an identifier which is either declared as a source for the map or as a source parameter or any named variable within the group in which this rule is nested.

The element is an (optional) name of a child element of the context. If the name is not provided, the source is the context. If it is provided, the rule will apply once for each element on the context that matches this name.

* **type**: A type may be specified. If a type is specified, only elements that have the specified type are selected as part of the source
* **min..max**: Specified cardinality (may use '\*' for the upper bound). If a cardinality is specified and the cardinality of the element in the source is not within the specified cardinality, the mapping engine raises an error instead of completing the transformation.
* **default**: A default value, used if there is no source value found. If there's a default value on an item that can repeat, it will only be used once. Default values only apply to primitive types, and the value is a FHIRPath statement (usually a literal)
* **list-option**: by default, the rule will apply once for each occurrence of the element in the source. The list option can override this to specify different behaviour. Valid values are 'first', 'last', 'not\_first', 'not\_last' and 'only\_one'
* **as variable**: if an element is specified, a variable must be assigned. This variable name may be used in the target statement, and may be re-used in other dependent rules (see below). There is one special case described below (simple form) where the variable name is provided explicitly
* **where [condition]**: a FHIRPath expression that is evaluated to boolean on the context. If the expression returns false, the source element has no match
* **check [condition]**: a FHIRPath expression that is evaluated to boolean on the context. If the expression returns false, the mapping engine raises an error. Note: it is usual to have either a where or a check clause, but possible to have both. If both are provided, the where clause is evaluated first, and the check only applies if the where expression is true

If there are multiple source statements, the rule applies for the permutation of the source elements from each source statement. E.g. if there are 2 source statements, each with 2 matching elements, the rule applies 4 times, one for each combination. Typically, if there is more than one source statement, only one of the elements would repeat.

Once the source content is evaluated, the engine performing the evaluation has a list of elements assigned to variables. For each time the rule is applied, each of the variables contains a single value. These variables are now mapped into the target structures in the target transformation.

7.7.0.7.1.1 Log Statement

Each source can include a log statement:

 log [expression]

Where expression is a FHIRPath statement. e.g.

 log 'not handled yet'

Puts a plain string in the log file. Alternatively, the log statement can contain FHIRPath:

 log src.field

Log statements are often used to note that some particular source element is not yet mapped.

7.7.0.7.2 Target Transform

Each rule specifies zero or more target transformation statements, which specify how source content is used to create target content. These target statements can also be assigned to variables that can be used in subsequent transform rules. If no targets are specified (no ->), no transformation is done and there are no created targets, just newly defined source variables, which can then be used in subsequent dependent rules. Multiple target statements are separated by a comma, like this:

... -> [target], [target] then...

Each [target] contains the following items:

 context.element = transform\_code(parameters...) as variable {list\_modes}

For example:

 context.element = copy(parameter, ...) as vt1 first

* **context.element**: The first part of the target statement consists of two parts: the **context** and the **element**.

The context is an identifier which is either declared as a target for the map, a taret parameter or any named variable within the group (including the variables from the source content) in which this rule is nested.

The element is the name of a child element that is valid in the context. The created value will be placed into the named element

* **transform(parameters)**: details how the content that is created is transformed from the source data using a specific transformation function. See below for the list of available functions. If no transform is provided, then the element is autocreated. It is an error if auto-created elements are primitive types, or have more than one possible type
* **as variable**: a variable name may be assigned, which allows the created item to be re-used in the dependent rules
* **list\_modes**: control how elements that repeat are managed when the transform rule is evaluated. Values can be 'first', 'share', 'last' and 'collate'. TODO: What do these do?

Transform statements may just contain an invocation of a transform function. In this case, a variable must be defined, and the created value is available in the variable for use in subsequent transformations.

Each time the rule is applied, the engine determines the value from the transforms, considers the list mode, if required and creates that specified content in the target instance. Within a target transform, the target statements are processed in order, so that a transform statement may refer to a variable defined by a prior transform statement.

The following list specifies that transforms that can be specified. Each transform takes one or more parameters:

|  |  |  |
| --- | --- | --- |
| **Name** | **parameters** | **Documentation** |
| create | type | use the standard API to create a new instance of data. Where structure definitions have been provided, the type parameter must be a string which is a known type of a root element. Where they haven't, the application must know the name somehow |
| copy | source | simply copy the source to the target as is (only allowed when the types in source and target match- typically for primitive types). In the concrete syntax, this is simply represented as the source variable, e.g. src.a = tgt.b |
| truncate | source, length | source must be some stringy type that has some meaningful length property |
| escape | source, format1, format2 | Change the internal escaping of a string element. Note: this is not often needed, as mostly the escaping is done on the base format |
| cast | source, type? | cast source from one type to another. target type can be left as implicit if there is one and only one target type known |
| append | source... | source is element or string - just append them all together |
| translate | source, map\_uri, output | use the [translate operation](https://www.hl7.org/fhir/conceptmap-operation-translate.html). The source is some type of code or coded datatype, and the source and map\_uri are passed to the translate operation. The output determines what value from the translate operation is used for the result of the operation (code, system, display, Coding, or CodeableConcept) |
| reference | source | return a string that references the provided tree properly |
| dateOp | ?? | Perform a date operation. Parameters to be documented |
| uuid | n/a | Generate a random UUID (in lowercase). No Parameters |
| pointer | resource | Return the appropriate string to put in a Reference that refers to the resource provided as a parameter |
| evaluate | resource | Execute the supplied FHIRPath expression and use the value returned by that. The 2nd parameter - FHIRPath expression - is evaluated in the context of the first parameter, and the result used as the value. In the concrete syntax, there is a short hand for this operation, by supplying () around the parameter. In this case, there is no context for the FHIRPath expression, and it must start with a reference to one of the defined variables |
| cc | (text) or (system. Code[, display]) | Create a CodeableConcept from the parameters provided |
| c | system. Code[, display] | Create a Coding from the parameters provided |
| qty | (text) or (value, unit, [system, code]) | Create a quantity. Parameters = (text) or (value, unit, [system, code]) where text =s the natural representation e.g. [comparator]value[space]unit |
| id | system, value[, type] | Create an identifier. where type is a code from the identifier type value set |
| cp | (value) or (system, value) | Create a contact detail. If no system is provided, the system should be inferred from the content of the value |

TODO: explain how optional parameters work with transforms (append only?), document list mode

7.7.0.7.3 Dependent Rules

Once the source elements are evaluated, and any specified targets created, the engine has a set of variables that represent source and target contexts in which further mapping may occur. The set of variables includes those provided to the group that contains the rule, and those created by the application of the rule. For some created elements that are primitive types, that's the end of the road - there's nothing more to do with them. But if either or both the source and target types are complex, there are usually additional mapping rules that need to apply to the newly created variables.

Dependent rules specify what additional rules are evaluated when the rule is complete:

 .. then {

 .. other rules...

 }

When a rule contains other rules, the variables from the containing rules are all available to the contained rules. Alternatively, a rule can nominate another group of rules from the same or an imported mapping. Each rule or group is listed by name, and then a set of parameters are provided.

 .. then rule(param, param)

The parameters provided must match the parameters required by the dependent rule, in order. In addition, the mode of the variable must match - inputs that are targets must be target variables. Note, though, that target variables can be treated as source for a group.

Groups are resolved by name by looking through all the groups in all the available maps referenced by the uses (see above) statements. The name must be unique within the scope of these maps.

7.7.0.8 Simple Form

If no dependent rules are specified, and if the is only one source and target, and they both specify a variable, the rule can be written in an abbreviated form:

 src.element -> tgt.element;

This is implicitly the same as

 src.element as vvs -> tgt.element = create('type') as vvt then defaultMappingGroup(vvs, vvt)

Where the name of the type given as a parameter to 'create' and the group invoked by the 'then' are determined by the context of src.element and tgt.element and the selected default mapping group, as documented in the next section. Note that default mapping groups are only invoked when no dependent rules or explicit group invocations are specified.

7.7.0.9 Default mapping groups

It is not necessary to explicitly invoke groups for each mapping. Instead groups can be declared to be the "default" mapping for a given source and target type. Groups acting as defaults have either <<types>> or <<type+>> in their declaration.

Groups marked with types are used by default when the engine encounters a mapping with a source and target type where the types match the source and target type of the group. Of course, there can be only one such group for each combination of source and target type for the engine to unambiguously determine which default group to invoke.

In addition to the above use, groups may be marked with type+. They will act like a default mapping group, just like types, but additionally they will be invoked when the target type is not fixed, i.e. when mapping to an element with a choice type. This means that a type+ group will be used as the default as long as the source type of the instance to map matches the source type of the group. Even so, the target will then always be taken to be the target type of the group.

7.7.0.10 Formal Grammar

The formal grammar for the mapping language, specified using ANTLR, can be found [here](https://www.hl7.org/fhir/mapping.g4).

Note that this grammar uses FHIRPath as an embedded syntax. Full details on FHIRPath and its grammar can be found [here ](http://hl7.org/fhirpath).

7.7.0.11 Mapping Support API

todo

7.7.0.12 Reserved keywords

This is the list of reserved keywords, which cannot be used as identifiers and names for variables, unless escaped.

map

uses

as

alias

imports

group

extends

default

where

check

log

then

true

false

types

type

first

not\_first

last

not\_last

only\_one

share

collate

source

target

queried

produced

conceptMap

prefix

®© HL7.org 2011+. FHIR Release 4 (Technical Correction #1) (v4.0.1) generated on Fri, Nov 1, 2019 09:37+1100. [QA Page](https://www.hl7.org/fhir/qa.html)
Links: [Search ](http://hl7.org/fhir/search.cfm) | [Version History](https://www.hl7.org/fhir/history.html) | [Table of Contents](https://www.hl7.org/fhir/toc.html) | [Cre](https://www.hl7.org/fhir/credits.html)