Overview of Prototype HQMF code:

# Goals

The purpose of this prototype was to try to compile HQMF descriptions of measures into pseudocode similar to the manually generated popHealth stage 1 code. A library that implements QDM functions and operators is assumed. Variable names were manually mapped from IDs to popHealth-like names for each measure, although this could easily be fixed.

# “Architecture” of Solution

The term is used loosely, since this is a throw away prototype.

**Stage 1** (lib/parsehqmf.rb) parses the XML file using Nokogiri, and builds a much simpler, but information content equivalent, JSON file (hierarchy of hashes). In addition to the original XML file, parsehqmf.rb looks for a ‘map’ file that maps IDs to ph\_id (popHealth ID) values. Here is an example entry:

"D484D744-BB90-45C9-8935-6CFF152DFCE1": {

"title": "Procedure performed: bilateral mastectomy",

"template": "2.16.840.1.113883.3.560.1.6",

"ph\_id": "measure.procedure\_performed\_bilateral\_mastectomy"

},

In addition, parsehqmf.rb can take advantage of a map of known data template IDs (lib/hqmf\_templates.json) that I derived by scanning the 113 NQF-published e-measures from 2/2011. This file maps the templateIDs used for data specification to an associated description (found in the comments of the HQMF text), and enumerates all of the concepts that were associated with that template. Currently the information in this file is only used for debug printing.

Parsehqmf.rb builds a hash of all IDs that can be referenced in the text from the content of the measure attributes, the data specifications, and the section IDs (the population section ID is referenced in the denominator section). In the example HQMF files form 2/2011, there are sometimes incorrect IDs, which I fixed manually. Most are distinct. This hash includes the ID as key, and the description (from the map file) and the templateID.

The logic is processed section (IPP, DENOM, etc) by section. The logic is found in an abominable mess of “sourceOf” tags, the code recursively calls the process\_clause method to consume these. The only method of significance that is invoked handles relative timing blocks, there is special handling (inline) for repetitions (COUNT).

When data references are encountered, they are resolved by lookup into the hash of IDs, and the bulk of the (highly redundant) body is ignored. This by itself reduces the clutter dramatically.

The output of parsehqmf is a Json file representing the information content of the XML file. For measure 0043, the JSON file (itself fairly verbose, with all of the hash tags) is 7x smaller than the original XML.

**Stage 2** (lib/generateCode.rb) loads the JSON produced by parsehqmf.rb, and proceeds to successively rewrite the parse tree of each section. The steps are:

1. removed redundant nodes with typeCode COMP. I have no idea why chains of these are emitted, but they are simply clutter. Eliminating them makes for more canonical structure of the tree.
2. Iteratively rewrite the tree, and emitting leaf nodes when ready

The tree rewrites depend on knowing the distance of each node from a leaf node. Since the underlying data structure (hash of hashes) doesn’t help with this, the code maintains this by traversing the tree periodically for this purpose. In addition, the rewrites depend on the ‘type’ of a node. These are the types used:  
 1) value: a node that has had code generated already (see below)

2) date\_interval: a period with a start and end (e.g., the measurement period)

3) ignore: this is used for the Initial Patient Population, when it is referenced in the denominator…

4) date\_single: a single date (e.g., the end of the measurement period)

5) event\_list: any data element in the data spec of the measure (e.g., active diagnosis of diabetes).

There are 8 patterns for code generation. This covered “successful” code generation from 4 measures. I imagine the approach could be stretched to handle all of the ratio measures by adding another 6-10 patterns. The patterns are applied in a brute force manner, applying each pattern to the nodes within the tree for which it is appropriate. A more intelligent approach would have been to traverse the tree and choose for each node an appropriate transform…sigh. Most of the patters are quite similar (#3-#8), but I didn’t bother to try to factor the code, but simply cut and pasted since that was expedient. The only code that is factored is the analysis of the TIMING blocks.

The patterns are:

1. mergeDateTimeInverval:

- dateSingle TIMING dateInterval (e.g., birthdate 64 years before measurement period)

1. mergeValueSets

- If a node has N children and N or N-1 of them are eventLists ORed together, this implies an OR/Union of their events. A new eventList is created that merges all of the ORed event lists, simplifying the tree.

1. mergeEventListDuringInterval:

an event list with a child DURING an interval

1. mergeEventListDuringIntervalFlat:

- eventList and a DURING that are siblings

1. mergeEventListTimingInterval

- Similar to #3, except the operator is a TIMING operator, not DURING

1. mergeEventListDuringEventList

- similar to #3, except the child is an event list

1. mergeEventListTimingEventList

- similar to #6 except TIMING, not DURING

1. MergeEventListTimingEventListFlat

* similar to #4 except TIMING, not DURING

When a pattern ‘fires’, a node is marked as ‘emitted’, and assigned a name and a value. A string “<name> = <value>” is emitted to the output stream, and all subsequent references to that node (from the nodes parents) will refer to the node by name. So, for example, pattern 2 would result in a node with name “m1” (the names are m followed by a number that increments) and the value would be “normalize(<list of eventLists>)”. All subsequent references to the merged event list would be by name – m1.

If no patterns can fire, and there remains nodes that haven’t been ‘emitted’ in the tree, then something bad has happened, usually a pattern that is either missing, or needs to be extended to handle a slightly different structure.