HL7 Standard:
Clinical Quality Language Specification,
Release 1 - US Realm

Draft Standard for Trial Use

May 2015

Publication of this draft standard for trial use and comment has been approved by Health Level Seven International (HL7). This draft standard is not an accredited American National Standard. The comment period for use of this draft standard shall end 24 months from the date of publication. Suggestions for revision should be submitted at http://www.hl7.org/dstucomments/index.cfm.

Following this 24 month evaluation period, this draft standard, revised as necessary, will be submitted to a normative ballot in preparation for approval by ANSI as an American National Standard. Implementations of this draft standard shall be viable throughout the normative ballot process and for up to six months after publication of the relevant normative standard.

IMPORTANT NOTES:

HL7 licenses its standards and select IP free of charge. **If you did not acquire a free license from HL7 for this document,** you are not authorized to access or make any use of it. To obtain a free license, please visit http://www.HL7.org/implement/standards/index.cfm.

If you are the individual that obtained the license for this HL7 Standard, specification or other freely licensed work (in each and every instance "Specified Material"), the following describes the permitted uses of the Material.

A. HL7 INDIVIDUAL, STUDENT AND HEALTH PROFESSIONAL MEMBERS, who register and agree to the terms of HL7's license, are authorized, without additional charge, to read, and to use Specified Material to develop and sell products and services that implement, but do not directly incorporate, the Specified Material in whole or in part without paying license fees to HL7.

INDIVIDUAL, STUDENT AND HEALTH PROFESSIONAL MEMBERS wishing to incorporate additional items of Special Material in whole or part, into products and services, or to enjoy additional authorizations granted to HL7 ORGANIZATIONAL MEMBERS as noted below, must become ORGANIZATIONAL MEMBERS of HL7.

B. HL7 ORGANIZATION MEMBERS, who register and agree to the terms of HL7's License, are authorized, without additional charge, on a perpetual (except as provided for in the full license terms governing the Material), non-exclusive and worldwide basis, the right to (a) download, copy (for internal purposes only) and share this Material with your employees and consultants for study purposes, and (b) utilize the Material for the purpose of developing, making, having made, using, marketing, importing, offering to sell or license, and selling or licensing, and to otherwise distribute, Compliant Products, in all cases subject to the conditions set forth in this Agreement and any relevant patent and other intellectual property rights of third parties (which may include members of HL7). No other license, sublicense, or other rights of any kind are granted under this Agreement.

C. NON-MEMBERS, who register and agree to the terms of HL7's IP policy for Specified Material, are authorized, without additional charge, to read and use the Specified Material for evaluating whether to implement, or in implementing, the Specified Material, and to use Specified Material to develop and sell products and services that implement, but do not directly incorporate, the Specified Material in whole or in part.

NON-MEMBERS wishing to incorporate additional items of Specified Material in whole or part, into products and services, or to enjoy the additional authorizations granted to HL7 ORGANIZATIONAL MEMBERS, as noted above, must become ORGANIZATIONAL MEMBERS of HL7.

Please see http://www.HL7.org/legal/ippolicy.cfm for the full license terms governing the Material.

Ownership. Licensee agrees and acknowledges that **HL7 owns** all right, title, and interest, in and to the Trademark. Licensee shall **take no action contrary to, or inconsistent with**, the foregoing.

Licensee agrees and acknowledges that HL7 may not own all right, title, and interest, in and to the Materials and that the Materials may contain and/or reference intellectual property owned by third parties ("Third Party IP"). Acceptance of these License Terms does not grant Licensee any rights with respect to Third Party IP. Licensee alone is responsible for identifying and obtaining any necessary licenses or authorizations to utilize Third Party IP in connection with the Materials or otherwise. Any actions, claims or suits brought by a third party resulting from a breach of any Third Party IP right by the Licensee remains the Licensee's liability.

Following is a non-exhaustive list of third-party terminologies that may require a separate license:

Terminology	Owner/Contact
Current Procedures Terminology (CPT) code set	American Medical Association http://www.ama-assn.org/ama/pub/physician-resources/solutions-managing-your-practice/coding-billing-insurance/cpt/cpt-products-services/licensing.page?
SNOMED CT	International Healthcare Terminology Standards Developing Organization (IHTSDO) http://www.ihtsdo.org/snomed-ct/get-snomed-ct or info@ihtsdo.org
Logical Observation Identifiers Names & Codes (LOINC)	Regenstrief Institute
International Classification of Diseases (ICD) codes	World Health Organization (WHO)

<u>Identifying Information for Specification:</u>

Specification Name and Release Number: HL7 Standard: Clinical Quality Language Specification, Release 1

Realm: US

Ballot Level: Draft Standard for Trial Use (DSTU)

Ballot Cycle: January 2015

Specification Date: November 2014

Co-Chair (CDS):	Guilherme Del Fiol, MD, PhD University of Utah Health Care guilherme.delfiol@utah.edu
Co-Chair (CDS):	Robert Jenders, MD, MS jenders@ucla.edu
Co-Chair (CDS):	Kensaku Kawamoto, MD, PhD University of Utah Health Care kensaku.kawamoto@utah.edu
Co-Chair (CDS):	Howard Strasberg Walters Kluwer Health howard.strasberg@wolterskluwer.com
Co-Chair (CQI):	Patricia Craig The Joint Commission pcraig@jointcommission.org
Co-Chair (CQI):	Floyd Eisenberg iParsimony LLC FEisenberg@iParsimony.com
Co-Chair (CQI):	Crystal Kallem RHIA, CPHQ Lantana Consulting Group Crystal.Kallem@Lantanagroup.com
Co-Chair (CQI):	Chris Millet cmillet@qualityforum.org
Co-Chair (CQI):	Walter Suarez, MD, MPH Kaiser Permanente walter.q.suarez@kp.org
Co-Chair (ITS):	Paul Knapp Knapp Consulting Inc. pknapp@pknapp.com
Co-Chair (ITS):	Dale Nelson Lantana Consulting Group dale.nelson@squaretrends.com
Co-Chair (ITS):	Andy Stechishin HL7 Canada andy.stechishin@gmail.com
Co-Author:	Bryn Rhodes Veracity Solutions bryn@veracitysolutions.com
Co-Author:	Chris Moesel The MITRE Corporation cmoesel@mitre.org
Co-Author:	Marc Hadley The MITRE Corporation mhadley@mitre.org
Co-Author:	Mark Kramer The MITRE Corporation mkramer@mitre.org
Co-Author:	Robert Dingwell The MITRE Corporation bobd@mitre.org
Co-Editor:	Aziz Boxwala, PhD Meliorix aziz.boxwala@meliorix.com
Co-Editor:	Kanwarpreet Sethi Lantana Consulting Group kp.sethi@lantanagroup.com

Acknowledgments

The authors wish to recognize the S&I Framework Clinical Quality Framework Initiative Work Group and the HL7 Clinical Decision Support, Clinical Quality Improvement, and Implementable Technology Specifications Work Groups for their contributions to this document.

Copyrights

EXECUTIVE SUMMARY

In support of the United States' national objectives for healthcare reform, the Office of the National Coordinator for Health Information Technology (ONC) Standards and Interoperability (S&I) Framework has sponsored the development of harmonized interoperability specifications. These specifications are designed to support national health initiatives and healthcare priorities, including Meaningful Use, the Nationwide Health Information Network, and the ongoing mission to improve population health.

The nation is reaching a critical mass of electronic health record systems (EHRs) that comply with data and vocabulary standards. Providers seeking to meaningfully use EHRs face a variety of challenging tasks. Those tasks include assessing needs, selecting and negotiating with a system vendor or reseller, implementing project management, and instituting workflow changes to improve clinical performance, control costs, and ultimately, improve outcomes. Additionally, many providers face the challenge of integration and interoperation with disparate systems. Many institutions use their own proprietary vocabularies and data models. Though this may offer some internal flexibility, it comes with a high, often hidden, long term maintenance cost.

In support of this wide deployment of EHRs, there is an opportunity to implement a learning health system that includes clinical quality measurement and improvement aspects and provides a broad range of benefits that can contribute towards improved health of individuals and the population as a whole (refer to "Digital Infrastructure for the Learning Health System: The Foundation for Continuous Improvement in Health and Health Care: Workshop Series Summary").

The S&I Framework Clinical Quality Framework Initiative (CQF) is developing a foundational specification, reusing much of the work currently done in health quality standardization, to enable the structuring and encoding of quality content for use as "knowledge artifacts." These artifacts can be used in support of many areas of the healthcare system, including quality and utilization measurement, disease outbreak detection, comparative effectiveness analysis, evaluation of drug treatment efficacy, monitoring health trends, and other public health, research, and information sharing across the continuum of care. Although the scope of this project focuses on quality knowledge and decision support, potential uses for CQL are not limited to these areas. For example, the CQL grammar can be used to express formal information extraction and transformation rules for converting and deriving data as it is moved from one representation or use to another.

One key benefit of this proposed approach is the definition of a "lingua franca" for the exchange of quality knowledge and artifacts. Rather than having an unscalable network of point-to-point communication channels, each with its own set of transformations, different organizations will only need to transform their content to a CQF-compatible format to communicate effectively with any other point in the network of providers that comprises today's healthcare system. If the models and vocabularies are rich enough, some quality vendors may opt to use CQF as an internal specification in the future.

This specification is developed in support of the CQF Artifact Sharing Use Case and is intended to assist implementers in the development of clinical quality knowledge artifacts for both the decision support and quality measurement domains. The approach adopted in this specification is

HL7 Standard: Clinical Quality Language, Release 1 - US Realm

designed to be flexible and reusable, and to provide a baseline for health quality vendors and implementers of systems that create and use knowledge artifacts to improve the health of individuals and the population as a whole.

Table of Contents

1 INTRODUCTION

The Clinical Quality Language Specification defines a representation for the expression of clinical knowledge that can be used within both the Clinical Decision Support (CDS) and Clinical Quality Measurement (CQM) domains. Although several standards exist for the expression of clinical quality logic, these standards are not widely adopted and present various barriers to point-to-point sharing of clinical knowledge artifacts such as lack of tooling, complexity of implementation, or insufficient expressivity.

Rather than attempt to address these shortcomings in one of the existing standards, this specification provides a solution to enable shared understanding of clinical knowledge by defining a syntax-independent, canonical representation of logic that can be used to express the knowledge in any given artifact, and point-to-point sharing by defining a serialization for that representation.

The canonical representation, the Expression Logical Model (ELM), is informed conceptually by the requirements of the clinical quality domains of measurement and improvement, and technically by compiler design best practices. The resulting canonical representation provides a basis for sharing logic in a way that is at once verifiable, computable, and can serve as the input to language processing applications such as translation, tooling, or even execution engines.

In addition, this specification introduces a high-level, domain-specific language, Clinical Quality Language (CQL), focused on clinical quality and targeted at measure and decision support artifact authors. This high-level syntax can then be rendered in the canonical representation provided by ELM.

1.1 Background

Clinical Decision Support and Clinical Quality Measurement are closely related, share many common requirements, and both support improving healthcare quality. However, the standards used for the electronic representation of CDS and CQM artifacts have not been developed in consideration of each other, and the domains use different approaches to the representation of patient data and computable expression logic. The first step in enabling a harmonization of these approaches is clearly identifying the various components involved in the specification of quality artifacts, and then establishing as a principle the notion that they should be treated independently. Broadly, the components of an artifact involve specifying:

- Metadata – Information about the artifact such as its identifier and version, what health topics it covers, supporting evidence, related artifacts, etc.

- Clinical Quality Information – The structure and content of the clinical data involved in the artifact

- Expression Logic – The actual knowledge and reasoning being communicated by the artifact

Considering each of these components separately, the next step involves identifying the relationship of the current specifications to each component, as shown in the following table:

	Model Type	Quality Information	Computable Expression Logic	Metadata
Clinical Decision Support (CDS)	Physical and logical	Virtual Medical Record (vMR)	CDS Knowledge Artifact Specification	CDS Knowledge Artifact Specification/Decision Support Service
Electronic Clinical Quality Measurement (eCQM)	Physical	Quality Reporting Document Architecture (QRDA)	Health Quality Measure Format (HQMF)	Health Quality Measure Format (HQMF)
	Logical	Quality Data Model (QDM)	Quality Data Model (QDM)	

<div align="center">TABLE 1–A</div>

The discrepancy shown here between standards used in the different domains introduces burdens on both vendors and providers in electronic healthcare quality domains, including:

- Inability to share logic between CDS and CQM artifacts, even though large portions of the logic involved represent the same conceptual knowledge

- Duplicated effort in the interpretation, integration, and execution of CDS and CQM artifacts

- Duplicated effort in the mapping of clinical information from vendor and provider systems to the different CDS and CQM artifacts

Using the framework of metadata, data model, and expression logic, the following diagram depicts the overall target specification areas involved in clinical quality artifact representation:
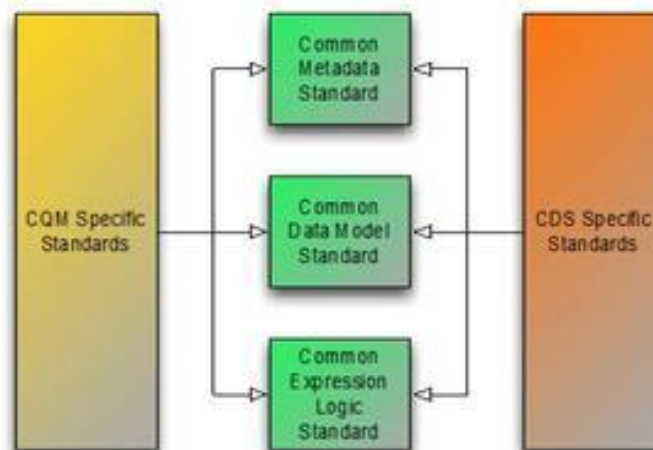
<div align="center">FIGURE 1–A</div>

Following this overall structure, this specification focuses on the common representation of expression logic that CQM and CDS-specific artifact standards can then reference. Separate

specifications address metadata and data model. Note, however, that the QUICK specification, and the Quality Improvement Core (QICore) Profiles are being developed concurrently with this specification to ensure that the two specifications interoperate effectively.

In addition, this specification is designed to be data model independent, meaning that CQL and ELM have no explicit dependencies on any aspect of any particular data model. Rather, the specification allows for any data model to be used, so long as a suitable description of that data model is supplied. Chapter 7 of this specification discusses how that description is supplied, and what facilities an implementation must support in order to enable complete data model independence of CQL and ELM.

1.2   Clinical Quality Framework Initiative

The S&I Framework is an approach adopted by ONC's Office of Standards & Interoperability to fulfill its charge of enabling harmonized interoperability specifications to support national health outcomes and healthcare priorities. The S&I Framework is a collaborative community of participants from the public and private sectors who are focused on providing the tools, services, and guidance to facilitate the functional exchange of health information. More information about the S&I Framework can be found here: http://siframework.org/

The S&I Framework uses a set of integrated functions, processes, and tools that enable execution of specific value-creating initiatives. Each S&I initiative focuses on a single, narrowly scoped, broadly applicable challenge.

The Clinical Quality Framework (CQF) is an S&I initiative focused on identifying, defining, and harmonizing standards and specifications that promote integration and reuse between Clinical Decision Support (CDS) and Clinical Quality Measurement (CQM). Additional information about the CQF initiative, including a project charter, can be found here: http://wiki.siframework.org/Clinical+Quality+Framework+Charter+and+Members

Stakeholder input and subject matter expert (SME) guidance has led to the development of several CQF use cases defining the functional aspects of clinical quality measurement and improvement. These use cases are described in detail here: http://wiki.siframework.org/Clinical+Quality+Framework+Use+Cases

1.3   Approach

As discussed in Section 1.1, one key principle underlying the current harmonization efforts is the separation of responsibilities within an artifact into *metadata*, *clinical information*, and *expression logic*. Focusing on the expression logic component and identifying the requirements common to both quality measurement and decision support, the Clinical Decision Support HL7 Work Group produced a harmonized conceptual requirements document: *HL7 Domain Analysis Model: Harmonization of Health Quality Artifact Reasoning and Expression Logic*. These requirements form the basis for the reasoning capabilities that this specification provides.

Building on those conceptual requirements, this specification defines the logical and physical layers necessary to achieve the goal of a unified specification for expression logic for use by both the clinical quality and decision support domains.

Broadly, this specification can be viewed from three perspectives:

- Author – The author perspective is concerned with clearly and correctly communicating and interpreting the semantics defined at the conceptual level, from a human perspective.
- Logical – The logical perspective is concerned with representing the semantics of expressions in the simplest complete way.
- Physical – The physical perspective is concerned with clearly and correctly communicating or interpreting the semantics defined at the logical level, from a machine perspective.

In other words, the logical level of the specification can be thought of as a complete bi-directional mapping between the author and physical levels. The various components involved in the specification are then concerned with ensuring that semantics can be clearly communicated through each of these levels.

1.3.1 Author Perspective

At the highest level, the author perspective is concerned with the human-readable description of clinical quality logic. This level is represented within this specification as a high-level syntax called Clinical Quality Language (CQL). CQL is a domain-specific language for clinical quality and is intended to be usable by clinical domain experts to both author and read clinical knowledge.

The author perspective is informed conceptually by the Quality Data Model (QDM), the current conceptual representation of electronic clinical quality measures. This heritage is intended to provide familiarity and continuity for authors coming from the quality space.

1.3.2 Logical Perspective

The logical perspective of the specification is concerned with complete and accurate representation of the semantics involved in the expression of quality logic, independent of the syntax in which that logic is rendered.

For the logical layer, this specification defines a Unified Modeling Language (UML) model called the Expression Logical Model (ELM) that defines a canonical representation of expression logic. This approach is intended to simplify implementation and machine processing by focusing on the content of an expression, rather than the syntax used to render it. The approach is based on and motivated by the concept of an Abstract Syntax Tree from traditional compiler implementation. The following diagram depicts the steps performed by a traditional compiler:
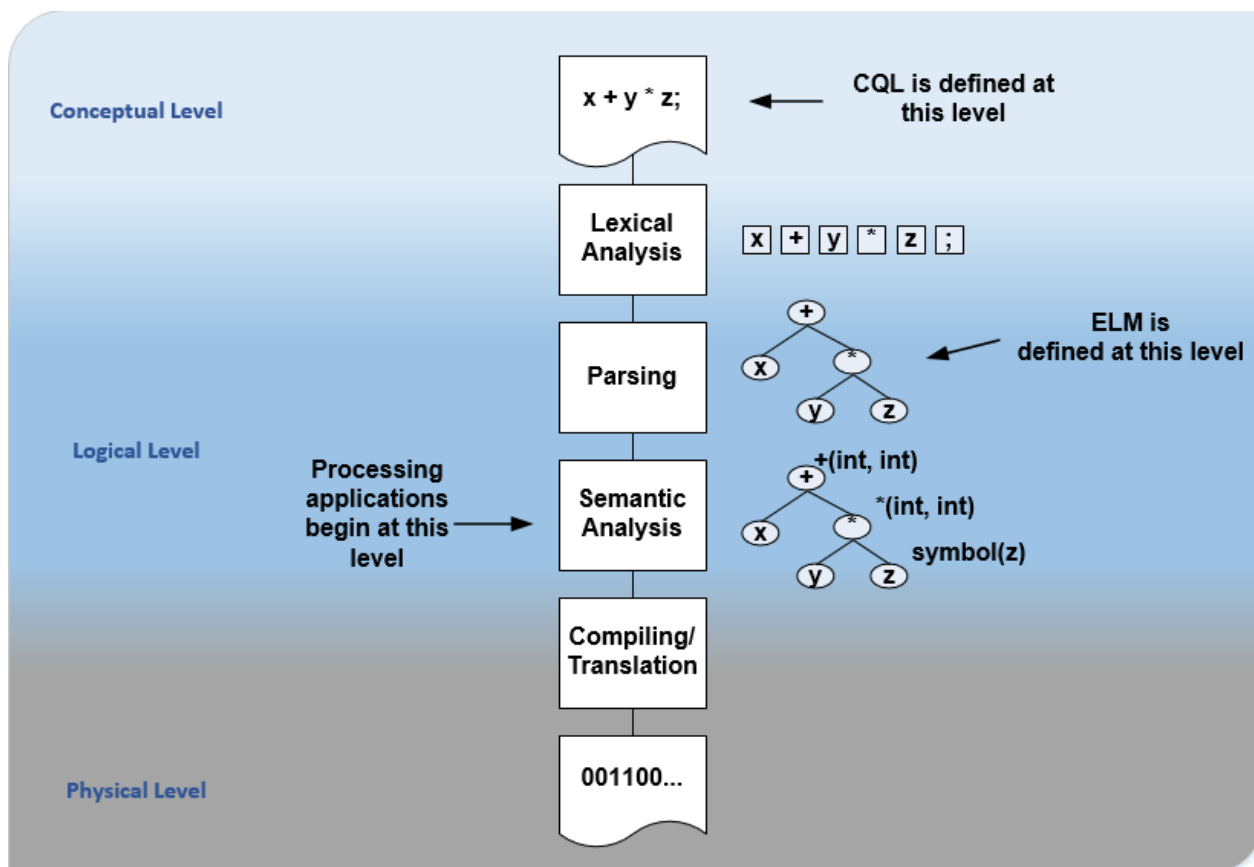
FIGURE 1−B

As shown here, the ELM representation is defined as an Abstract Syntax Tree, eliminating the need for lexical analysis and parsing steps, and allowing implementations to concentrate on the core representation of the logic.

In addition, this approach avoids potential ambiguity that must be resolved with operator precedence and/or the use of parentheses in traditional expression languages.

The result is a dramatic reduction in the complexity of processing quality artifacts, whether that processing involves translation to another format, evaluation of the logic, or building a user-interface for authoring or visual representation of the artifact.

The logical perspective is informed conceptually by the HL7 Version 3 Standard: Clinical Decision Support Knowledge Artifact Specification, Release 1.2 (CDS KAS), the current representation of clinical decision support artifacts. This heritage is intended to provide familiarity and continuity for authors and consumers in the decision support space.

1.3.3 Physical Perspective

The physical perspective is concerned with the implementation and communication aspects of the logical model—specifically, with how the canonical representation of expression logic is shared between producers and consumers. This specification defines an XML schema representation of the ELM for this purpose, describes the intended semantics of CQL, and discusses various implementation approaches.

1.4   Audience

The audience for this specification includes stakeholders and interested parties from a broad range of health quality applications, including health IT vendors, quality agencies, quality artifact authors and consumers, and any party interested in producing or consuming health quality artifacts.

The specification is written with the following major roles in mind:

Role	Description
Author	A clinical domain expert or clinical artifact author intending to use the Clinical Quality Language specification to author or understand quality artifacts
Developer	A developer interested in building more complex clinical quality artifacts as well as shared libraries for use by authors
Integrator	A health IT professional interested in integrating quality artifacts based on the Clinical Quality Language specification into a health quality system
Implementer	A systems analyst, architect, or developer interested in building language processing applications for artifacts based on the Clinical Quality Language specification, such as translators, interpreters, tooling, etc.

<div align="center">TABLE 1–B</div>

Note that even the material in Chapter 2 is somewhat technical in nature, and that Authors will benefit from some familiarity with and/or training in basic computer language and database language topics.

In general, each of these roles will benefit from focusing on different aspects of the specification. In particular, the Author role will be primarily interested in Chapter 2, the Language Guide for the high-level CQL syntax; the Developer role will be primarily interested in Chapters 2 & 3; the Integrator role will be primarily interested in Chapter 4, the formal description of the logical model; and the Implementer role will be primarily interested in Chapters 5, 6, and 7, which discuss the intended execution semantics, translation semantics, and physical representation, respectively.

1.5   Scope of the Specification

The Clinical Quality Language specification includes the following components:

- CQL Grammar – An ANTLR4 grammar file formally defining the syntax for the high-level authoring language described by this specification

- Expression Logical Model – A UML model that specifies a canonical representation for expression logic

- ELM XML Schemas – XML schemata defining a physical representation for the serialization and sharing of expression logic specified in the ELM

Note that syntax highlighting is used throughout the specification to make the examples easier to read. However, the highlighting is for example use only and is not intended to be a normative aspect of the specification.

1.6   Alignment to CQF Artifact Sharing Use Case

The specific requirements implemented within this specification focus on the structure, semantics, and encoding of expression logic representation within quality artifacts. These requirements are directly tied to the Clinical Quality Framework Artifact Sharing Use Case. Full material on this Use Case can be found here:

http://wiki.siframework.org/Clinical+Quality+Framework+Use+Cases

In particular, this specification enables the sharing use case by defining a high-level syntax suitable for authors, a logical-level representation suitable for language processing applications, and a mechanism for translation between them. The following diagram depicts how these specifications will be used in the sharing use case:

Clinical Quality Language (CQL) — Authors use CQL to produce libraries containing human-readable yet precise logic.

Expression Logical Model (ELM) — ELM XML documents contain machine-friendly rendering of the CQL logic. This is the intended mechanism for distribution of libraries.

Native   Java Script   Drools   SQL — Implementation environments will either directly execute the ELM, or perform translation from ELM to their target environment language.

FIGURE 1–C

1.6.1 Use Case Assumptions and Conditions

It is important for implementers to clearly understand the underlying environmental assumptions, defined in Section 5 of the CQF Use Case document referenced in the previous section, to ensure that these assumptions align to the implementation environment in which content will be exchanged using a knowledge artifact. Failure to meet any of these assumptions could impact implementation of the knowledge artifact.

1.7   Organization of this Specification

The organization of this specification follows the outline of the perspectives discussed in the Approach section—conceptual, logical, and physical. Below is a listing of the chapters with a short summary of the content of each.

Chapter 2 – Author's Guide provides a high-level discussion of the Clinical Quality Language syntax. This discussion is a self-contained introduction to the language targeted at clinical quality authors.

Chapter 3 – Developer's Guide provides a more in-depth look at the Clinical Quality Language targeted at developers familiar with typical development languages such as Java, C#, and SQL.

Chapter 4 – Logical Specification provides a complete description of the elements that can be used to represent quality logic. Note that Chapters 2 and 3 describe the same functional capabilities of the language, and that anything that can be expressed in one mechanism can be equivalently expressed in the other.

Chapter 5 – Language Semantics describes the intended semantics of the language, covering topics such as data layer integration and expected run-time behavior.

Chapter 6 – Translation Semantics describes the mapping between CQL and ELM, as well as outlines for how to perform translation from CQL to ELM, and vice versa.

Chapter 7 – Physical Representation is reference documentation for the XML schema used to persist ELM.

Appendix A – CQL Syntax Formal Specification discusses the ANTLR4 grammar for the Clinical Quality Language.

Appendix B – CQL Reference provides a complete reference for the types and operators available in CQL, and is intended to be used by authors and developers alike.

Appendix C – Reference Implementations provides information about where to find reference implementations for a CQL-ELM translator, a CQL Execution Framework for JavaScript, and other related tooling.

Appendix D – References

Appendix E – Acronyms

Appendix F – Glossary

2 AUTHOR'S GUIDE

This chapter introduces the high-level syntax for the Clinical Quality Language focused on measure and decision support authors. This syntax provides a human-readable, yet precise mechanism for expressing logic in both the measurement and improvement domains of clinical quality.

The syntax, or structure, of CQL is built from several basic elements called *tokens*. These tokens are *symbols*, such as + and *, *keywords*, such as `define` and `from`, *literals*, such as `5` and `'active'`, and *identifiers*, such as `Person` and `"Inpatient Encounters"`.

Statements of CQL are built up by combining these basic elements, separated by *whitespace* (spaces, tabs, and returns), to produce language elements. The most basic of these language elements is an *expression*, which is any statement of CQL that returns a value.

Expressions are built by combining *terms*, such as literals and identifiers, using *operators*, either symbolic operators, such as + and -, operator phrases such as `and` and `exists`, or named operators called *functions*, such as `First()` and `AgeInYears()`.

At the highest level, CQL is organized around the concept of a *library*, which can be thought of as a container for artifact logic. Libraries contain *declarations* which specify the items the library contains. The most important of these declarations is the *named expression*, which is the basic unit of logic definition in CQL.

In the sections that follow, the various constructs introduced above will be discussed in more detail, beginning with the kinds of declarations that can be made in a CQL library, and then moving through the various ways that clinical information is referenced and queried within CQL, an overview of the operators available in CQL, and ending with a detailed walkthrough of authoring specific quality artifacts using a running example.

Note that throughout the discussion, readers may wish to refer to Appendix B – CQL Reference for more detailed discussion of particular concepts.

2.1   Declarations

All the constructs that can be expressed within CQL are packaged in a container called a *library*. Libraries provide a convenient unit for the definition, versioning, and distribution of logic. For simplicity, libraries in CQL correspond directly with a single file.

Libraries in CQL provide the overall packaging for CQL definitions. Each library allows a set of declarations to provide information about the library as well as to define constructs that will be available within the library.

Libraries can contain any or all of the following constructs:

Construct	Description
library	Header information for the library, including the name and version, if any.
using	Data model information, specifying that the library may access types from the referenced data model.
include	Referenced library information, specifying that the library may access constructs

	defined in the referenced library.
codesystem	Codesystem information, specifying that logic within the library may reference the specified codesystem by the given name.
valueset	Valueset information, specifying that logic within the library may reference the specified valueset by the given name.
parameter	Parameter information, specifying that the library expects parameters to be supplied by the evaluating environment.
context	Patient/population context, specifying the overall context for the statements that follow.
define	The basic unit of logic within a library, a define statement introduces a named expression that can be referenced within the library, or by other libraries.
function	Libraries may also contain function definitions. These are most often used as part of shared libraries.

<div align="center">TABLE 2-A</div>

The following sections discuss these constructs in more detail.

2.1.1 Library

The `library` declaration specifies both the name of the library and an optional version for the library. The library name is used as an identifier to reference the library from other CQL libraries, as well as eCQM and CDS artifacts. A library can have at most one library declaration.

The following example illustrates the library declaration:

```
library CMS153_CQM version '2'
```

The above declaration names the library with the identifier `CMS153_CQM` and specifies the version `'2'`.

2.1.2 Data Models

A CQL library can reference zero or more data models with `using` declarations. These data models define the structures that can be used within retrieve expressions in the library.

For more information on how these data models are used, see the Retrieve section.

The following example illustrates the using declaration:

```
using QUICK
```

The above declaration specifies that the QUICK model will be used as the data model within the library.

If necessary, a version specifier can be provided to indicate which version of the data model should be used.

2.1.3 Libraries

A CQL library can reference zero or more other CQL libraries with `include` declarations. Components defined within these included libraries can then be referenced within the library by using the locally assigned name for the library.

For more information on libraries, refer to the Using Libraries to Share Logic section.

The following example illustrates an include declaration:

```
include CMS153_Common version '2' called Common
```

Components defined in the `CMS153_Common` library, version 2, can now be referenced using the assigned name of `Common`. For example:

```
define SexuallyActive:
  exists (Common.ConditionsIndicatingSexualActivity)
  or exists (Common.LaboratoryTestsIndicatingSexualActivity)
```

This expression references `ConditionsIndicatingSexualActivity` and `LaboratoryTestsIndicatingSexualActivity` defined in the `CMS153_Common` library using the local alias `Common`.

The syntax used to reference these expressions is a *qualified identifier* consisting of two parts. The qualifier, `Common`, and the identifier, `ConditionsIndicatingSexualActivity`, separated by a dot (.).

2.1.4 Valuesets

A CQL library may contain zero or more named valuesets using the `valueset` declaration. A valueset declaration specifies a local identifier that represents a valueset and can be used anywhere within the library that a valueset is expected.

The following example illustrates a valueset declaration:

```
valueset "Female Administrative Sex": '2.16.840.1.113883.3.560.100.2'
```

This definition establishes the local identifier `"Female Administrative Sex"` as a reference to the external identifier for the valueset, an OID in this case: `'2.16.840.1.113883.3.560.100.2'`. The external identifier need not be an OID, it may be a URL, or any other identification system. CQL does not interpret the external id, it only specifies that the external identifier be a string that can be used to uniquely identify the valueset within the implementation environment.

This valueset definition can then be used within the library wherever a valueset can be used:

```
define InDemographic: Patient.gender in "Female Administrative Sex"
```

The above examples define the `InDemographic` expression as true for patients whose `Gender` is a code in the valueset identified by `"Female Administrative Sex"`.

Note that the name of the valueset uses double quotes, in contrast to the string representation of the object identifier (OID) for the valueset, which uses single quotes. Single quotes are used to build arbitrary strings in CQL; double quotes are used to represent names of constructs such as valuesets and expression definitions.

Note also that the local identifier for a valueset is user-defined and not required to match the actual name of the valueset identified within the external valueset repository. Good practice would dictate that the names should at least be conceptually similar, but CQL makes no prescription either way.

For more information about valuesets as a value within CQL, refer to the Clinical Values section.

2.1.5 Parameters

A CQL library can define zero or more parameters. Each parameter is defined with the elements listed in the following table:

Element	Description
Name	A unique identifier for the parameter within the library
Type	The type of the parameter – Note that the type is only required if no default value is provided. Otherwise, the type of the parameter is determined based on the default value.
Default Value	An optional default value for the parameter

<center>TABLE 2–B</center>

The parameters defined in a library may be referenced by name in any expression within the library. When expressions in a CQL library are evaluated, the values for parameters are provided by the environment. For example, a library that defines criteria for a quality measure may define a parameter to represent the measurement period:

```
parameter MeasurementPeriod
  default Interval[@2013-01-01T00:00:00.0, @2014-01-01T00:00:00.0)
```

Note the syntax for the default here is called an *interval selector* and will be discussed in more detail in the section on 2.4.5 Interval Values.

This parameter definition can now be referenced anywhere within the CQL library:

```
define InDemographic:
  AgeInYearsAt(start of MeasurementPeriod) >= 16
    and AgeInYearsAt(start of MeasurementPeriod) < 24
```

The above example defines the InDemographic expression as patients whose age at the start of the MeasurementPeriod was at least 16 and less than 24.

The default value for a parameter is optional, but if no default is provided, the parameter must include a type specifier:

```
parameter MeasurementPeriod Interval<DateTime>
```

If a parameter definition does not indicate a default value, the parameter is considered *required*, meaning that a value must be supplied by the evaluation environment, typically as part of the evaluation request.

2.1.6 Context

The context declaration defines the overall context for statements within the language. CQL supports two contexts:

Context	Description
Patient	The Patient context specifies that expressions should be interpreted with reference to a single patient.
Population	The Population context specifies that expressions should be interpreted with reference to the entire population of patients.

TABLE 2-C
The following example illustrates the use of the Patient context:

```
context Patient

define InDemographic:
  AgeInYearsAt(start of MeasurementPeriod) >= 16
    and AgeInYearsAt(start of MeasurementPeriod) < 24
    and Patient.gender in "Female Administrative Sex"
```

Because the context has been established as Patient, the expression has access to patient-specific concepts such as the `AgeInYearsAt()` operator and the `Patient.gender` attribute. Note that the attributes available in the Patient context are defined by the data model in use.

A library may contain zero or more context statements, with each context statement establishing the context for subsequent statements in the library. When no context is specified, the default context is `Patient`.

Effectively, the statement `context Patient` defines an expression named Patient that returns the patient data for the current patient, as well as restricts the information that will be returned from a retrieve to a single patient, as opposed to all patients. For more information on context, refer to the Retrieve Context discussion below.

2.1.7 Statements

A CQL Library can contain zero or more `define` statements describing named expressions that can be referenced either from other expressions within the same library or by containing quality and decision support artifacts.

The following example illustrates a simple `define` statement:

```
define InpatientEncounters:
  [Encounter: "Inpatient"] E
    where E.length <= 120 days
      and E.period ends during MeasurementPeriod
```

This example defines the `InpatientEncounters` expression as Encounter events whose code is in the `"Inpatient"` valueset, whose `length` is less than or equal to 120 days, and whose `period` ended (i.e. patient was discharged) during `MeasurementPeriod`.

Note that the use of terms like `Encounter`, `length`, and `period`, as well as which code attribute is used to compare with the valueset, are defined by the data model being used within the library; they are not defined by CQL.

For more information on the use of define statements, refer to the Using Define Statements section.

2.2   Retrieve

The *retrieve* declaration is the central construct for accessing clinical information within CQL. The result of a retrieve is always a list of some type of clinical data, based on the type described by the retrieve and the context (Patient or Population) in which the retrieve is evaluated.

The retrieve in CQL has two main parts: first, the *type* part, which identifies the type of data that is to be retrieved; and second, the *filter* part, which optionally provides filtering information based on specific types of filters common to most clinical data.

Note that the retrieve only introduces data into an expression; operations for further filtering, shaping, computation, and sorting will be discussed in later sections.

2.2.1 Clinical Statement Structure

The retrieve expression is a reflection of the idea that clinical data in general can be viewed as clinical statements of some type as defined by the model. The type of the clinical statement determines the structure of the data that is returned by the retrieve, as well as the semantics of the data involved.

The type may be a general category, such as a Condition, Procedure, or Encounter, or a more specific instance such as an ImagingProcedure, or a LaboratoryTest. The data model defines the available types that may be referenced by a retrieve.

In the simplest case, a retrieve specifies only the type of data to be retrieved. For example:

```
[Encounter]
```

Assuming the default context of `Patient`, this example retrieves all Encounter statements for a patient.

2.2.2 Filtering with Valuesets

In addition to describing the type of clinical statements, the retrieve expression allows the results to be filtered using a valueset. The use of codes within clinical data is ubiquitous, and most clinical statements have at least one code-valued attribute. In addition, there is typically a "primary" code-valued attribute for each type of clinical statement. This primary code is used to drive the valueset filter. For example:

```
[Condition: "Acute Pharyngitis"]
```

This example requests only those Conditions whose primary code attribute is a value from the valueset identified by `"Acute Pharyngitis"`. The attribute used as the primary code attribute is defined by the data model being used.

In addition, the retrieve expression allows the filtering attribute name to be specified:

```
[Condition: severity in "Acute Severity"]
```

This requests clinical statements that assert the presence of a condition with a severity in the `"Acute Severity"` valueset.

2.2.3 Retrieve Context

Within the Patient context, the results of any given retrieve will always be scoped to a single patient, as determined by the environment. For example, in a quality measure evaluation environment, the Patient context may be the current patient being considered. In a clinical decision support environment, the Patient context would be the patient for which guidance is being sought.

By contrast, within the Population context, the results of any given retrieve will not be limited to a single Patient. For example:

```
[Condition: "Acute Pharyngitis"] C where C.onsetDateTime during MeasurementPeriod
```

When evaluated within the Patient context, the above example returns `"Acute Pharyngitis"` conditions during `MeasurementPeriod` for the current patient only. In the Population context, this example returns `"Acute Pharyngitis"` conditions during `MeasurementPeriod` for all patients.

Because context is associated with each declaration, it is possible for expressions defined in the Patient context to reference expressions defined in the Population context and vice versa.

In a Population context, a reference to a Patient context expression results in the execution of that expression for each patient in the population, and the implementation environment combines the results. If the result type of the Patient context expression is not a list, the result of accessing it from a Population context will be a list with elements of the type of the Patient context expression. If the result type of the Patient context is a list, the result will be a list of the same type, but with the results of the evaluation for each patient in the population combined into a single list.

In a Patient context, a reference to a Population context expression results in the evaluation of the Population context expression, and the result type is unaffected.

2.3   Queries

Beyond the retrieve expression, CQL provides a *query* construct that allows the results of retrieves to be further filtered, shaped, and extended to enable the expression of arbitrary clinical logic that can be used in quality and decision support artifacts.

Although similar to a retrieve in that a query will typically result in a list of patient information, a query is a more general construct than a retrieve. Retrieves are by design restricted to a particular set of criteria that are commonly used when referencing clinical information, and specifically constructed to allow implementations to easily build data access layers suitable for use with CQL. For more information on the design of the retrieve construct, refer to Clinical Data Retrieval in Quality Artifacts.

The query construct has a *primary source* and four main *clauses* that each allow for different types of operations to be performed:

Clause	Operation
Relationship (with/without)	Allows relationships between the primary source and other clinical information to be used to filter the result.
Where	The where clause allows conditions to be expressed that filter the result to only the information that meets the condition.
Return	The return clause allows the result set to be shaped as needed, removing elements, or including new calculated values.
Sort	The sort clause allows the result set to be ordered according to any criteria as needed.

TABLE 2–D

Each of these clauses will be discussed in more detail in the following sections.

A query construct begins by introducing an *alias* for the primary source:

```
[Encounter: "Inpatient"] E
```

The primary source for this query is `[Encounter: "Inpatient"]`, and the alias is `E`. Subsequent clauses in the query must reference elements of this source by using this name.

Note that although the alias in this example is a single-letter abbreviation, `E`, it could also be a longer abbreviation:

```
[Encounter: "Inpatient"] Enc
```

2.3.1 Filtering

The `where` clause allows the results of the query to be filtered by a condition that is evaluated for each element of the query being filtered. If the condition evaluates to true for the element being tested, that element is included in the result. Otherwise, the element is excluded from the resulting list.

For example:

```
[Encounter: "Inpatient"] E
  where duration in days of E.period >= 120
```

The alias `E` is used to access the `period` attribute of each encounter in the primary source. The filter condition tests whether the duration of that range is at least 120 days.

The condition of a `where` clause is allowed to contain any arbitrary combination of operations of CQL, so long as the overall result of the condition is boolean-valued. For example, the following where clause includes multiple conditions on different attributes of the source:

```
[CommunicationRequest] C
  where C.mode = 'ordered'
    and C.sender.role = 'nurse'
    and C.recipient.role = 'doctor'
    and C.indication in "Fever"
```

Note that because CQL uses three-valued logic, the result of evaluating any given boolean-valued condition may be *unknown* (`null`). For example, if the list of inpatient encounters from the first example contains some elements whose `period` attribute is `null`, the result of the condition for that element will not be `false`, but `null`, indicating that it is not known whether or not the duration of the encounter was at least 120 days. For the purposes of evaluating a filter, only elements where the condition evaluates to `true` are included in the result, effectively treating the unknown results as `false`.

2.3.2 Shaping

The `return` clause of a CQL query allows the results of the query to be shaped. In most cases, the results of a query will be of the same type as the primary source of the query. However, some

HL7 Standard: Clinical Quality Language, Release 1 - US Realm
May 2015

scenarios involve the need to extract only specific attributes or to perform computations on the data involved in each element. The return clause enables this type of query.

For example:

```
[Encounter: "Inpatient"] E
  return Tuple { id: E.identifier, lengthOfStay: duration in days of E.period }
```

This example returns a list of tuples (structured values), one for each inpatient encounter performed, where each tuple consists of the `id` of the encounter as well as a `lengthOfStay` attribute, whose value is calculated by taking the duration of the `period` for the encounter. Tuples are discussed in detail in later sections.

2.3.3 Sorting

CQL queries can sort results in any order using the `sort by` clause. For example:

```
[Encounter: "Inpatient"] E sort by start of period
```

This example returns inpatient encounters, sorted by `period`.

Calculated values can also be used to determine the sort, `ascending` (`asc`) or `descending` (`desc`), as in:

```
 [Encounter: "Inpatient"] E
  return Tuple { id: E.identifier, lengthOfStay: duration in days of E.period }
  sort by lengthOfStay desc
```

Note that the properties that can be specified within the sort clause are determined by the result type of the query. In the above example, `lengthOfStay` can be referenced because it is introduced in the return clause.

If no ascending or descending specifier is provided, the order is ascending.

If no sort clause is provided, the order of the result is undefined and may vary by implementation.

The sort clause may include multiple attributes, each with their own sort order:

```
[Encounter: "Inpatient"] E sort by start of period desc, identifier asc
```

Sorting is perfomed in the order in which the attributes are defined in the sort clause, so this example sorts by `period` descending, then by `identifier` ascending. A query may only contain a single sort clause, and it must always appear last in the query.

2.3.4 Relationships

In addition to filtering by conditions, some scenarios need to be able to filter based on relationships to other sources. The CQL `with` and `without` clauses provide this capability. For example:

```
[Encounter: "Ambulatory/ED Visit"] E
  with [Condition: "Acute Pharyngitis"] P
    such that P.onsetDateTime during E.period
      and P.abatementDateTime after end of E.period
```

This query returns `"Ambulatory/ED Visit"` encounters performed where the patient also has a condition of `"Acute Pharyngitis"` that overlaps after the `period` of the encounter.

The `without` clause returns only those elements from the primary source that do not have a specific relationship to another source. For example:

```
[Encounter: "Ambulatory/ED Visit"] E
  without [Condition: "Acute Pharyngitis"] P
    such that P.onsetDateTime during E.period
      and P.abatementDateTime after end of E.period
```

This query is the same as the previous example, except that only encounters that *do not* have overlapping conditions of `"Acute Pharyngitis"` are returned. In other words, if the *such that* condition evaluates to true (if the Encounter has an overlapping Condition of Acute Pharyngitis in this case), then that Encounter is not included in the result.

A given query may include any number of with and without clauses in any order, but they must all come before any where, return, or sort clauses.

Note that the `such that` condition of `with` and `without` clauses need not be based on timing relationships, it may contain any arbitrary expression, so long as the overall result is boolean-valued. For example:

```
[MedicationDispense: "Warfarin"] D
  with [MedicationPrescription: "Warfarin"] P
    such that P.status = 'active'
      and P.identifier = D.authorizingPrescription.identifier
```

This example retrieves all dispense records for active prescriptions of Warfarin.

2.3.5 Full Query

The clauses described in the previous section must appear in the correct order to specify a valid CQL query. The general order of clauses is:

primary-source alias

> *with-or-without-clauses*
>
> *where-clause*
>
> *return-clause*
>
> *sort-clause*

A query must contain an aliased primary source, but this is the only required clause.

A query may contain zero or more with or without clauses, but they must all appear before any where, return, or sort clauses.

A query may contain at most one where clause, and it must appear after any with or without clauses, and before any return or sort clauses.

A query may contain at most one return clause, and it must appear after any with or without or where clauses, and before any sort clause.

A query may contain at most one sort clause, and it must be the last clause in the query.

For example:

```
[Encounter: "Inpatient"] E
  with [Condition: "Acute Pharyngitis"] P
    such that P.onsetDateTime during E.period
      and P.abatementDateTime after end of E.period
  where duration in days of E.period >= 120
  return Tuple { id: E.id, lengthOfStay: duration in days of E.period }
  sort by lengthOfStay desc
```

This query returns all `"Inpatient"` encounter events that have an overlapping condition of `"Acute Pharyngitis"` and a duration of at least 120 days. For each such event, the result will include the `id` of the event and the duration in days, and the results will be ordered by that duration descending.

Note that the query construct in CQL supports other clauses that are not discussed here. For more information on these, refer to Multi-Source Queries and Non-Retrieve Queries.

2.4 Values

CQL supports several categories of values:

- Simple values, such as strings, numbers, and dates

- Clinical values, such as quantities and valuesets

- Structured values (called tuples), such as Medications, Encounters, and Patients

- Lists, which can contain any number of elements of the same type

- Intervals, which define ranges of ordered values, such as numbers or dates

The result of evaluating any expression in CQL is a value of some type. For example, the expression `5` results in the value `5` of type `Integer`. CQL is a *strongly-typed* language, meaning that every value is of some type, and that every operation expects arguments of a particular type.

As a result, any given expression of CQL can be verified as meaningful, at least in terms of the operations performed. For example, consider the following expression:

```
6 + 6
```

The expression involves the addition of values of type `Integer`, and so is a meaningful expression of CQL. By contrast:

```
6 + 'active'
```

This expression involves the addition of a value of type `Integer`, `6`, to a value of type `String`, `'active'`. This expression is meaningless since CQL does not define addition for values of type `Integer` and `String`.

However, there are cases where an expression is meaningful, even if the types do not match exactly. For example, consider the following addition:

```
6 + 6.0
```

This expression involves the addition of a value of type `Integer`, and a value of type `Decimal`. This is meaningful, but in order to infer the correct result type, the `Integer` value will be implicitly converted to a value of type `Decimal`, and the `Decimal` addition operator will be used, resulting in a value of type `Decimal`.

May 2015

To ensure there can never be a loss of information, this implicit conversion will only happen from `Integer` to `Decimal`, never from `Decimal` to `Integer`.

In the sections that follow, the various categories of values that can be represented in CQL will be considered in more detail.

2.4.1 Simple Values

CQL supports several types of simple values:

Value	Examples
Boolean	`true, false`
Integer	`16, -28`
Decimal	`100.015`
String	`'pending', 'active', 'complete'`
DateTime	`@2014-01-25,` `@2014-01-25T14:30:14.559`
Time	`@T12:00:00.0Z` `@T14:30:14.559-07:00`

<p align="center">TABLE 2–E</p>

2.4.1.1 Boolean

The Boolean type in CQL supports the logical values `true` and `false`. These values are most often encountered as the result of Comparison Operators, and can be combined with other boolean-valued expressions using Logical Operators.

2.4.1.2 Integer

The Integer type in CQL supports the representation of whole numbers, positive and negative. CQL supports a full set of Arithmetic Operators for performing computations involving whole numbers.

In addition, any operation involving Decimals can be used with values of type Integer because Integer values can always be implicitly converted to Decimal values.

2.4.1.3 Decimal

The Decimal type in CQL supports the representation of real numbers, positive and negative. As with Integer values, CQL supports a full set of Arithmetic Operators for performing computations involving real numbers.

2.4.1.4 String

String values within CQL are represented using single-quotes:

```
'active'
```

Note that if the value to be represented contains a single-quote, two single-quotes within the string can be used to represent it in CQL:

```
'patient''s condition is normal'
```

2.4.1.5 DateTime and Time

CQL supports the representation of both `DateTime` and `Time` values.

`DateTime` values are used to represent an instant along the timeline, known to at least the year precision, and potentially to the millisecond precision. `DateTime` values are specified using an at-symbol (@) followed by an ISO-8601 textual representation of the `DateTime` value:

```
@2014-01-25
@2014-01-25T14:30:14.559
```

`Time` values are used to represent a time of day, independent of the date. `Time` value must be known to at least the hour precision, and potentially to the millisecond precision. `Time` values are specified using at-symbol (@) followed by an ISO-8601 textual representation of the `Time` value:

```
@T12:00:00.0Z
@T14:30:14.559-07:00
```

Note that the `Time` value literal format is identical to the time value portion of the `DateTime` literal format.

For both `DateTime` and `Time` values, timezone may be specified as either UTC time (`Z`), or as a timezone offset. If no timezone is specified, the timezone of the evaluation request timestamp is used.

FOR MORE INFORMATION ON THE USE OF DATE/TIME VALUES WITHIN CQL, REFER TO THE TABLE 2–J Date/Time Operators section.

Specifically, because `DateTime` and `Time` values may be specified to varying levels of precisions, operations such as comparison and duration calculation may result in null, rather than the true or false that would result from the same operation involving fully specified values. For a discussion of the effect of imprecision on date/time operations, refer to the Comparing Dates and Times section.

2.4.2 Clinical Values

In addition to simple values, CQL supports some types of values that are specific to the clinical quality domain. For example, CQL supports *codes*, *concepts*, *quantities*, and *valuesets*.

2.4.2.1 Quantities

A quantity is a number with an associated unit. For example:

```
6 'gm/cm3'
80 'mm[Hg]'
3 months
```

CQL supports the following built-in units for time granularities:

```
years
months
days
hours
minutes
seconds
milliseconds
```

In addition, CQL supports any valid Unified Code for Units of Measure (UCUM) unit code using the string representation of the UCUM code immediately following the numeric value, as shown in the first example in this section. UCUM codes can be specified in the case-sensitive (c/s) or case-insenstive form (c/i).

For a discussion of the operations available for quantities, see the Quantity Operators section.

2.4.2.2 Code

The use of codes to specify meaning within clinical data is ubiquitous. CQL therefore supports a top-level construct for dealing with codes using a structure called `Code` that is consistent with the way terminologies are typically represented.

The `Code` type has the following elements:

Name	Type	Description
code	String	The identifier for the code.
display	String	A description of the code.
system	String	The identifier of the code system.
version	String	The version of the code system.

<div align="center">TABLE 2–F</div>

In addition, CQL provides a Code literal that can be used to reference an existing code from a specific code system:

```
Code '66071002' from "SNOMED-CT:2014" display 'Type B viral hepatitis'
```

The example specifies the code `'66071002'` from the previously defined `"SNOMED-CT:2014"` codesystem, which specifies both the `system` and `version` of the resulting code.

Note that the `display` clause is optional. The above example references the code `'66071002'` from the `"SNOMED-CT:2014"` code system.

2.4.2.3 Concept

Within clinical information, multiple terminologies can often be used to code for the same concept. As such, CQL defines a top-level construct called `Concept` that allows for multiple codes to be specified.

The Concept type has the following elements:

Name	Type	Description
codes	List<Code>	The list of equivalent codes representing the concept.
display	String	A description of the concept.

<div align="center">TABLE 2–G</div>

Note that the semantics of `Concept` are such that the codes within a given concept should all be semantically equivalent at the code level, but CQL itself will make no attempt to ensure that is the case. Concepts should never be used as a surrogate for proper valueset definition.

The following example illustrates the use of a `Concept` literal:

```
Concept
{
  Code '66071002' from "SNOMED-CT:2014",
  Code 'B18.1' from "ICD-9-CM:2014"
} display 'Type B viral hepatitis'
```

This example constructs a `Concept` with display `'Type B viral hepatitis'` and primary code of `'66071002'`.

2.4.2.4 Valuesets

As a value, a valueset is simply a list of `Code` values. However, CQL allows valuesets to be used without reference to the codes involved by declaring valuesets as a special type of value within the language.

The following example illustrates some typical valueset declarations:

```
valueset "Acute Pharyngitis": '2.16.840.1.113883.3.464.1003.102.12.1011'
valueset "Acute Tonsillitis": '2.16.840.1.113883.3.464.1003.102.12.1012'
valueset "Ambulatory/ED Visit": '2.16.840.1.113883.3.464.1003.101.12.1061'
```

Each valueset declaration defines a local identifier that can be used to reference the valueset within the library, as well as the global identifier for the valueset, typically an OID or URI.

These valueset identifiers can then be used throughout the library. For example:

```
define Pharyngitis: [Condition: "Acute Pharyngitis"]
```

This example defines `Pharyngitis` as any Condition where the code is in the `"Acute Pharyngitis"` valueset.

Whenever a valueset reference is actually evaluated, the resulting *expansion set*, or list of codes, depends on the *binding* specified by the valueset declaration. By default, all valueset bindings are *dynamic*, meaning that the expansion set should be constructed using the most current published version of the valueset.

However, CQL also allows for *static* bindings which allow two components to be set:

1. Version – The version of the valueset to be referenced, specified as a string.

2. Code Systems – A list of code systems referenced by the valueset definition.

If the binding specifies a valueset version, then the expansion set must be derived from that specific version. This does not restrict the code system versions to be used, therefore more than one expansion set is possible.

If any code systems are specified, they indicate which version of the particular code system should be used when constructing the expansion set. Note that if the external valueset definition explicitly states that a particular version of a code system should be used, then it is an error if the code system version specified in the CQL static binding does not match the code system version specified in the external valueset definition. To create a reliable static binding where only one value set expansion set is possible, both the value set version AND the code system versions should be specified.

The following example illustrates the use of static binding based only on the version of the valueset:

```
valueset "Diabetes": '2.16.840.1.113883.3.464.1003.103.12.1001' version '20140501'
```

The next example illustrates a static binding based on both the version of the valueset, as well as the versions of the code systems within the valueset:

```
codesystem "SNOMED-CT:2013-09": '2.16.840.1.113883.6.96' version '2031-09'
codesystem "ICD-9-CM:2014": '2.16.840.1.113883.6.103' version '2014'
codesystem "ICD-10-CM:2014": '2.16.840.1.113883.6.90' version '2014'
valueset "Diabetes": '2.16.840.1.113883.3.464.1003.103.12.1001' version '20140501'
  codesystems ( "SNOMED-CT:2013-09", "ICD-9-CM:2014", "ICD-10-CM:2014" )
```

See the Terminology Operators section for more information on the use of valuesets within CQL.

2.4.2.5 Codesystems

In addition to their use as part of valueset definitions, codesystem definitions can be referenced directly within an expression, just like valueset definitions.

See the Terminology Operators section for more information on the use of codesystems within CQL.

2.4.3 Structured Values (Tuples)

Structured values, or *tuples*, are values that contain named elements, each having a value of some type. Clinical information such as a Medication, a Condition, or an Encounter is represented using tuples.

For example, the following expression retrieves the first Condition with a code in the "Acute Pharyngitis" valueset for a patient:

```
define FirstPharyngitis:
  First([Condition: "Acute Pharyngitis"] C sort by C.onsetDateTime desc)
```

The values of the elements of a tuple can be accessed using a dot qualifier (.) followed by the name of the element:

```
define PharyngitisOnSetDateTime: FirstPharyngitis.onsetDateTime
```

If the result of the expression being accessed is null, the result of the element access is also null. For example, in the above, if FirstPharyngitis is null, the result of accessing the onsetDateTime element is also null.

Tuples can also be constructed directly using a tuple selector:

```
define Info: Tuple { Name: 'Patrick', DOB: @2014-01-01 }
```

If the tuple is of a specific type, the name of the type can be used instead of the Tuple keyword:

```
define PatientExpression: Patient { Name: 'Patrick', DOB: @2014-01-01 }
```

If the name of the type is specified, the tuple selector may only contain elements that are defined on the type, and the expressions for each element must evaluate to a value of the defined type for the element.

Note that tuples can contain other tuples, as well as lists:

```
define Info:
  Tuple
  {
    Name: 'Patrick',
    DOB: @2014-01-01,
    Address: Tuple { Line1: '41 Spinning Ave', City: 'Dayton', State: 'OH' },
    Phones: { Tuple { Number: '202-413-1234', Use: 'Home' } }
  }
```

Accordingly, element access can nest as deeply as necessary:

```
Info.Address.City
```

This accesses the `City` element of the `Address` element of `Info`. Lists can be traversed within element accessors using the list indexer (`[]`):

```
Info.Phones[1].Number
```

This accesses the `Number` element of the first element of the `Phones` list within `Info`.

2.4.4 List Values

CQL supports the representation of lists of any type of value (including other lists), but all the elements within a given list must be of the same type.

Lists can be constructed directly, as in:

```
{ 1, 2, 3, 4, 5 }
```

But more commonly, lists of tuples are the result of retrieve expressions. For example:

```
[Condition: code in "Acute Pharyngitis"]
```

This expression results in a list of tuples, where each tuple's elements are determined by the data model in use.

Lists in CQL use one-based indexes, meaning that the first element in a list has index 1, as opposed to 0. For example, given the list of integers:

```
{ 6, 7, 8, 9, 10 }
```

The first element is 6 and has index 1, the second element is 7 and has index 2, and so on.

Note that in general, clinical data may be expected to contain various types of collections such as sets, bags, lists, and arrays. For simplicity, CQL deals with all collections using the same collection type, the *list*, and provides operations to enable dealing with different collection types. For example, a set is a list where each element is unique, and any given list can be converted to a set using the `distinct` operator.

For a description of the distinct operator, as well as other operations that can be performed with lists, refer to the List Operators section.

2.4.5 Interval Values

CQL supports the representation of intervals, or ranges, of values of various types. In particular, intervals of date/time and ranges of integers and reals.

Intervals in CQL are represented by specifying the low and high points of the interval and whether the boundary is inclusive (meaning the boundary point is part of the interval) or exclusive (meaning the boundary point is excluded from the interval). Following standard mathematics notation, inclusive (closed) boundaries are indicated with square brackets, and exclusive (open) boundaries are indicated with parentheses. For example:

```
Interval[3, 5)
```

This expression results in an interval that contains the integers 3 and 4, but not 5.

```
Interval[3.0, 5.0)
```

This expression results in an interval that contains all the real numbers >= 3.0 and < 5.0.

Intervals can be constructed based on any type that supports unique and ordered comparison. For example:

```
Interval[@2014-01-01T00:00:00.0, @2015-01-01T00:00:00.0)
```

This expression results in an interval that begins at midnight on January 1, 2014, and ends just before midnight on January 1, 2015.

2.5   Operations

In addition to retrieving clinical information about a patient or population, the expression of clinical knowledge artifacts often involves the use of various operations such as comparison, logical operations such as and and or, computation, and so on. To ensure that the language can effectively express a broad range of knowledge artifacts, CQL includes a comprehensive set of operations. In general, these operations are all *expressions* in that they can be evaluated to return a value of some type, and the type of that return value can be determined by examining the types of values and operations involved in the expression.

This means that for each operation, CQL defines the number and type of each input (*argument*) to the operation and the type of the result, given the types of each argument.

The following sections define the operations that can be used within CQL, divided into semantically related categories.

2.5.1 Comparison Operators

The most basic operation in CQL involves comparison of two values. This is accomplished with the built-in comparison operators:

Operator	Name	Description
=	Equality	Returns true if the arguments are the same value
<>	Inequality	Returns true if the arguments are not the same value
>	Greater than	Returns true if the left argument is greater than the right argument
<	Less than	Returns true if the left argument is less than the right argument
>=	Greater than or equal	Returns true if the left argument is greater than or equal to the right argument
<=	Less than or	Returns true if the left argument is less than or equal to the right

Operator	Name	Description
	equal	argument
between		Returns true if the first argument is greater than or equal to the second argument, and less than or equal to the third argument

<div align="center">TABLE 2–H</div>

In general, the equality and inequality operators can be used on any type of value within CQL, but both arguments must be the same type. For example, the following equality comparison is legal, and returns true:

```
5 = 5
```

However, the following equality comparison is invalid because numbers and strings cannot be meaningfully compared:

```
5 = 'completed'
```

For date/time values, equality is defined to account for the possibility that the date/time values involved are specified to varying levels of precision. For a complete discussion of this behavior, refer to Comparing Dates and Times.

For structured values, equality returns true if the values being compared are the same type (meaning they have the same types of elements) and the values for each element are the same value. For example, the following comparison returns true:

```
Tuple { id: 'ABC-001', name: 'John Smith' } = Tuple { id: 'ABC-001', name: 'John Smith' }
```

For lists, equality returns true if the lists contain the same elements in the same order. For example, the following lists are equal:

```
{ 1, 2, 3, 4, 5 } = { 1, 2, 3, 4, 5 }
```

And the following lists are not equal:

```
{ 1, 2, 3, 4, 5 } <> { 5, 4, 3, 2, 1 }
```

Note that in the above example, if the second list was sorted ascending prior to the comparison, the result would be true.

For intervals, equality returns true if the intervals use the same point type and cover the same range. For example:

```
[1..5] = [1..6)
```

This returns true because the intervals cover the same set of points, 1 through 5.

The relative comparison operators (>, >=, <, <=) can be used on types of values that have a natural ordering such as numbers, strings, and dates.

The between operator is shorthand for comparison of an expression against an upper and lower bound. For example:

```
4 between 2 and 8
```

This expression is equivalent to:

```
4 >= 2 and 4 <= 8
```

May 2015

For all the comparison operators, the result type of the operation is Boolean. Note that because CQL uses three-valued logic, if either or both of the arguments is `null`, the result of the comparison is `null`.

2.5.2 Logical Operators

Combining the results of comparisons and other boolean-valued expressions is essential and is performed in CQL using the following logical operations:

Operator	Description
and	Logical conjunction
or	Logical disjunction
xor	Exclusive logical disjunction
not	Logical negation

<div align="center">TABLE 2–I</div>

The following examples illustrate some common uses of logical operators:

```
AgeInYears() >= 18 and AgeInYears() < 24
INRResult > 5 or DischargedOnOverlapTherapy
```

Note that all these operators are defined using three-valued logic, which is defined specifically to ensure that certain well-established relationships that hold in standard boolean (two-valued) logic also hold. The complete semantics for each operator are described in the Logical Operators section of Appendix B – CQL Reference.

2.5.3 Arithmetic Operators

The expression of clinical logic often involves numeric computation, and CQL provides a complete set of arithmetic operations for expressing computational logic. In general, these operators have the standard semantics for arithmetic operators, with the general caveat that unless otherwise stated in the documentation for a specific operation, if any argument to an operation is `null`, the result is `null`.

The following table lists the arithmetic operations available in CQL:

Operator	Name	Description
+	addition	Performs numeric addition of its arguments
-	subtraction	Performs numeric subtraction of its arguments
*****	multiply	Performs numeric multiplication of its arguments
/	divide	Performs numeric division of its arguments
div	truncated divide	Performs integer division of its arguments
mod	modulo	Computes the remainder of the integer division of its arguments
Ceiling		Returns the first integer greater than or equal to its argument
Floor		Returns the first integer less than or equal to its argument
Truncate		Returns the integer component of its argument
Abs		Returns the absolute value of its argument

HL7 Standard: Clinical Quality Language, Release 1 - US Realm
May 2015

-	negate	Returns the negative value of its argument
Round		Returns the nearest numeric value to its argument, optionally specified to a number of decimal places for rounding
Ln	natural logarithm	Computes the natural logarithm of its argument
Log	logarithm	Computes the logarithm of its first argument, using the second argument as the base
^	exponentiation	Raises the first argument to the power given by the second argument

<div align="center">TABLE 2–J</div>

2.5.4 Date/Time Operators

Operations on date and time data are an essential component of expressing clinical knowledge, and CQL provides a complete set of date/time operators. These operators broadly fall into five categories:

- Construction – Building or selecting specific date/time values
- Comparison – Comparing date/time values
- Extraction – Extracting specific components from date/time values
- Arithmetic – Performing date/time arithmetic
- Duration – Computing durations between date/time values

2.5.4.1 Constructing Date/Time Values

In addition to the literals described in the DateTime and Time section, the `DateTime` and `Time` operators allow for the construction of specific date/time values based on the values for their components. For example:

```
DateTime(2014, 7, 5)
DateTime(2014, 7, 5, 4, 0, 0, 0, -7)
```

The first example constructs the DateTime July 5, 2014. The second example constructs a DateTime of July 5, 2014, 04:00:00.0 UTC-07:00 (Mountain Standard Time).

The `DateTime` operator takes the following arguments:

Name	Type	Description
Year	Integer	The year component of the datetime
Month	Integer	The month component of the datetime
Day	Integer	The day component of the datetime
Hour	Integer	The hour component of the datetime
Minute	Integer	The minute component of the datetime
Second	Integer	The second component of the datetime
Millisecond	Integer	The millisecond component of the datetime
Timezone Offset	Decimal	The timezone offset component of the datetime (in hours)

TABLE 2-K

At least one component other than timezone offset must be provided, and for any particular component that is provided, all the components of broader precision must be provided. For example:

```
DateTime(2014)
DateTime(2014, 7)
DateTime(2014, 7, 11)
DateTime(null, null, 11) // invalid
```

The first three expressions above are valid, constructing dates with a specified precision of years, months, and days, respectively. However, the fourth expression is invalid, because it attempts to create a date with a day but no year or month component.

The only component that is ever defaulted is the timezone component. If no timezone component is supplied, the timezone component is defaulted to the timezone of the timestamp associated with the evaluation request.

The `Time` operator takes the following arguments:

Name	Type	Description
Hour	`Integer`	The hour component of the datetime
Minute	`Integer`	The minute component of the datetime
Second	`Integer`	The second component of the datetime
Millisecond	`Integer`	The millisecond component of the datetime
Timezone Offset	`Decimal`	The timezone offset component of the datetime

TABLE 2-L

As with the DateTime operator, at least the first component must be supplied, and for any particular component that is provided, all components of broader precision must be provided. If timezone is not supplied, it will be defaulted to the timezone of the timestamp associated with the evaluation request.

In addition to the ability to construct specific dates and times using components, CQL supports three operators for retrieving the current date and time:

Operator	Description
Now	Returns the date and time of the start timestamp associated with the evaluation request
Today	Returns the date (with no time components) of the start timestamp associated with the evaluation request
TimeOfDay	Returns the time-of-day of the start timestamp associated with the evaluation request

TABLE 2-M

The current date and time operators are defined based on the timestamp of the evaluation request for two reasons:

1. The operations will always return the same value during any given evaluation request, ensuring that the result of an expression containing Now() or Today() will always return the same result within the same evaluation (determinism).

2. The operations are based on the timestamp associated with the evaluation request, allowing the evaluation to be performed with the same time zone information as the data delivered with the evaluation request.

By defining the date construction operators in this way, most clinical logic can safely ignore timezone information, and the logic will be evaluated with the expected semantics. However, if timezone information is relevant to a particular calculation, it can still be accessed as a component of each datetime value.

In addition, all operations on dates and times are defined to take timezone information into account, ensuring that datetime operations perform correctly and consistently.

In addition to date and time values, CQL supports the construction of time durations using the name of the precision as the unit for a quantity. For example:

```
3 months
1 year
5 minutes
```

Valid time duration units are:

```
year
years
month
months
day
days
hour
hours
minute
minutes
second
seconds
millisecond
milliseconds
```

Note that CQL supports both plural and singular duration units to allow for the most natural expression but that no attempt is made to enforce singular or plural usage.

Note also that the UCUM time-period units can be used when expressing duration quantities.

2.5.4.2 Comparing Dates and Times

CQL supports comparison of date/time values using the expected comparison operators. Note however, that when date/time values are not specified completely, the result may be `null`, depending on whether there is enough information to make an accurate determination. In general, CQL treats date/time values that are only known to some specific precision as an uncertainty over the range at the first unspecified precision. For example:

```
DateTime(2014)
```

This value can be read as "some date within the year 2014", because only the year component is known. Applying these semantics yields the intuitively correct result when comparing date/time values with varying levels of precision.

```
DateTime(2012) < DateTime(2014, 2, 15)
```

This example returns `true` because even though the month and day of the first date are unknown, the year, 2012, is known to be less than the year of the second date, 2014. By contrast:

```
DateTime(2015) < DateTime(2014, 2, 15)
```

The result in this example is `false` because the year, 2015, is not less than the year of the second date. And finally:

```
DateTime(2014) < DateTime(2014, 2, 15)
```

The result in this example is `null` because the first date could be any date within the year 2014, so it could be less than the second date, but it could be greater.

Note that when determining equality, these semantics imply that if either date/time has unspecified components, the result of the comparison will be unknown. However, it is often the case that comparisons should only be carried to a specific level of precision. To enable this, CQL provides precision-based versions of the comparison operators:

Operator	Precision-based Operator
=	same as
<	before
>	after
<=	same or before
>=	same or after

<div align="center">TABLE 2-N</div>

If no precision is specified, these operators are equivalent to the symbolic comparison operators, implying comparison precision to the millisecond. However, each operator allows a precision specifier to be used. For example:

```
DateTime(2014) same year as DateTime(2014, 7, 11)
DateTime(2014, 7) same month as DateTime(2014, 7, 11)
DateTime(2014, 7, 11) same day as DateTime(2014, 7, 11, 14, 0, 0)
```

Each of these expressions returns `true` because the date/time values are equal at the specified level of precision and above. For example, `same month as` means the same year *and* the same month.

For relative comparisons involving equality, the `same as` operator is suffixed with `before` or `after`:

```
DateTime(2015) same year or after DateTime(2014, 7, 11)
DateTime(2014, 4) same month or before DateTime(2014, 7, 11)
DateTime(2014, 7, 15) same day or after DateTime(2014, 7, 11, 14, 0, 0)
```

Each of these expressions also returns `true`. And finally, for the relative inequalities (< and >):

```
DateTime(2015) after year of DateTime(2014, 7, 11)
DateTime(2014, 4) before month of DateTime(2014, 7, 11)
DateTime(2014, 7, 15) after day of DateTime(2014, 7, 11, 14, 0, 0)
```

Each of these expressions also returns `true`.

Note that these operators may still return `null` if the date/time values involved have unspecified components at or above the specified comparison precision.

2.5.4.3 Extracting Date and Time Components

Given a date/time value, CQL supports extraction of any of the components. For example:

```
date from X
year from X
minute from X
```

These examples extract the date from X, the year from X, and the minute from X. The following table lists the valid extraction components and their resulting types:

Component	Description	Result Type
date from X	Extracts the date of its argument (with no time components)	`DateTime`
time from X	Extracts the time of its argument	`Time`
year from X	Extracts the year component its argument	`Integer`
month from X	Extracts the month component of its argument	`Integer`
day from X	Extracts the day component of its argument	`Integer`
hour from X	Extracts the hour component of its argument	`Integer`
minute from X	Extracts the minute component of its argument	`Integer`
second from X	Extracts the second component of its argument	`Integer`
millisecond from X	Extracts the millisecond component of its argument	`Integer`
timezone from X	Extracts the timezone offset component of its argument	`Decimal`

TABLE 2–O

Note that if X is `null`, the result is `null`. If a date/time value does not have a particular component specified, extracting that component will result in `null`. Note also that if the timezone component for a particular date/time value was not provided as part of the constructor, because the value is defaulted to the timezone of the evaluation request, the result of extracting the timezone component will be the default timezone, not null.

2.5.4.4 Date/Time Arithmetic

By using quantities of time durations, CQL supports the ability to perform calendar arithmetic with the expected semantics for durations with variable numbers of days such as months and years. The arithmetic addition and subtraction symbols (+ and -) are used for this purpose. For example:

```
Today() - 1 year
```

The above expression computes the date one year before today, taking into account variable length years and months. Any valid time duration can be added to or subtracted from any datetime value.

Note that as with the numeric arithmetic operators, if either or both arguments are `null`, the result of the operation is `null`.

The operation is performed by attempting to derive the highest granularity precision first, working down successive granularities to the granularity of the time-valued quantity. For example, consider the following addition:

```
DateTime(2014) + 24 months
```

This example results in the value `DateTime(2016)` even though the date/time value is not specified to the level of precision of the time-valued quantity.

2.5.4.5 Computing Durations and Differences

In addition to constructing durations, CQL supports the ability to compute duration and difference between two datetimes. For duration, the calculation is performed based on the calendar duration for the precision. For difference, the calculation is performed by counting the number of boundaries of the specific precision crossed between the two dates.

```
months between X and Y
```

This example calculates the number of months between its arguments. For variable length precisions (months and years), the operation uses the calendar length of the precision to determine the number of periods.

For example, the following expression returns 2:

```
months between @2014-01-01 and @2014-03-01
```

This is because there are two whole calendar months between the two dates. Fractional months are not included in the result. This means that this expression also returns 2:

```
months between @2014-01-01 and @2014-03-15
```

For difference, the calculation is concerned with the number of boundaries crossed:

```
difference in months between X and Y
```

The above example calculates the number of month boundaries crossed between X and Y.

To illustrate the difference between the two calculations, consider the following examples:

```
duration in months between @2014-01-31 and @2014-02-01
difference in months between @2014-01-31 and @2014-02-01
```

The first example returns 0 because there is less than one calendar month between the two dates. The second example, however, returns 1, because a month boundary was crossed between the two dates.

The following duration units are valid for the duration and difference operators:

```
years
months
days
hours
minutes
seconds
milliseconds
```

If the first argument is after the second, the result will be negative.

In addition, if either date/time value involved is not specified to the level of precision for the duration or difference being calculated, the result will be an *uncertainty* covering the range of possible values for the duration. Subsequent comparisons using this uncertainty may result in `null` rather than `true` or `false`. For a detailed discussion of the behavior of uncertainties, refer to the Uncertainty section.

If either or both arguments are `null`, the result is `null`.

2.5.5 Timing and Interval Operators

Clinical information often contains not only date/time information as timestamps (points in time), but intervals of time, such as the effective time for an encounter or condition. Moreover, clinical logic involving this information often requires the ability to relate this temporal information. For example, a clinical quality measure might look for "patients with an inpatient encounter during which a condition started". CQL provides an exhaustive set of operators for describing these types of temporal relationships between clinical information.

These interval operations can be broadly categorized as follows:

- General – Construction, extraction, and membership operators
- Comparison – Comparison of two intervals
- Timing – Describing the relationship between two intervals using boundaries
- Computation – Using existing intervals to compute new ones

2.5.5.1 Operating on Intervals

General interval operators in CQL provide basic operations for dealing with interval values, including construction, extraction, and membership.

Interval values can be constructed using the *interval selector*, as discussed in Interval Values above.

Membership testing for intervals can be done using the `in` and `contains` operators. For example:

```
Interval[3, 5) contains 4
4 in Interval[3, 5)
```

These two expressions are equivalent (inverse of each other) and both return `true`.

The boundary point for an interval can be determined using the `start of` and `end of` operators:

```
start of Interval[3, 5)
end of Interval[3, 5)
```

The first expression above returns 3, while the second expression returns 4.

Note that the starting and ending point of an interval may be `null`, the meaning of which depends on whether the interval is closed (inclusive) or open (exclusive). If a boundary point is `null` and the boundary is exclusive, the boundary is considered unknown and operations involving that point will return `null`. For example:

```
Interval[3, null) contains 5
```

This expression results in `null`. However, if the point is `null` and the interval boundary is inclusive, the boundary is interpreted as the beginning or ending of the range of the point type. For example:

```
Interval[3, null] contains 5
```

This expression returns `true` because the `null` ending boundary is inclusive and is therefore interpreted as extending to the end of the range of possible values for the point type of the interval.

For numeric intervals, CQL defines a `width` operator, which returns one more point than the ending boundary minus the starting boundary:

```
width of Interval[3, 5)
width of Interval[3, 5]
```

The first expression returns 2 (ending boundary of 4, minus the starting boundary of 3, plus 1), while the second expression returns 3 (ending boundary of 5, minus the starting boundary of 3, plus 1). In other words, the `width` operator returns the number of points that are included in the interval.

For date/time intervals, CQL defines a `duration in` operator as well as a `difference in` operator, both of which are defined in the same way as the date/time duration and difference operators, respectively. For example:

```
duration in days of X
```

is equivalent to:

```
days between start of X and end of X
```

This returns the number of whole days between the starting and ending dates of the interval x.

2.5.5.2 Comparing Intervals

CQL supports comparison of two interval values using a complete set of operations. The following table describes these operators with a diagram showing the relationship between two intervals that is characterized by each operation:

Operator/Inverse	Diagram	Interpretation
X same as Y Y same as X	X Y	start of X = start of Y and end of X = end of Y
X before Y Y after X	X Y	end of X < start of Y
X meets before Y Y meets after X	X Y	successor of end of X = start of Y
X overlaps before Y Y overlaps after X	X Y	start of X <= start of Y and start of Y <= end of X
X begins Y Y begun by X	X Y	start of X = start of Y and end of X <= end of Y
X included in (during) Y Y includes X	X Y	start of X >= start of Y and end of X <= end of Y
X ends Y Y ended by X	X Y	start of X >= start of Y and end of X = end of Y

TABLE 2–P

Each of these operators returns `true` if the intervals x and y are in the given relationship to each other. If either or both arguments are `null`, the result is `null`. Otherwise, the result is `false`.

In addition, CQL allows `meets` and `overlaps` to be invoked without the `before` or `after` suffix, indicating that either relationship should return `true`. In other words, x `meets` y is equivalent to x `meets before` y `or` x `meets after` y, and similarly for the `overlaps` operator.

Note that to use these operators, the intervals must be of the same point type. For example, it is invalid to compare an interval of date/times with an interval of numbers.

2.5.5.3 Timing Relationships

In addition to the interval comparison operators described above, CQL allows various timing relationships to be expressed by directly accessing the start and end boundaries of the intervals involved. For example:

```
X starts before start Y
```

This expression returns `true` if the start of X is before the start of Y.

In addition, timing phrases allow the use of time durations to offset the relationship. For example:

```
X starts 3 days before start Y
```

This returns `true` if the start of X is equal to three days before the start of Y. Timing phrases can also include `or less` and `or more` to determine how the time duration is interpreted. For example:

```
X starts 3 days or less before start Y
X starts 3 days or more before start Y
```

HL7 Standard: Clinical Quality Language, Release 1 - US Realm
May 2015

The first expression returns `true` if the start of X is within the interval beginning three days before the start of Y and ending just before the start of Y. The second expression returns `true` if the start of X is less than or equal to three days before the start of Y.

Timing phrases also allow the use of `within` to establish a range for comparison:

```
X starts within 3 days of start Y
```

This expression returns `true` if the start of X is in the interval beginning three days before the start of Y and ending 3 days after the start of Y.

In addition, if either comparand is a date/time, rather than an interval, it can be used in any of the timing phrases without the boundary access modifiers:

```
dateTimeX within 3 days of dateTimeY
```

In other words, the timing phrases in general compare two quantities, either of which may be an date/time interval or date/time point value, and the boundary access modifiers can be added to a given timing phrase to access the boundary of an interval.

The following table describes the operators that can be used to construct timing phrases:

Operator	Beginning Boundary (starts/ends)	Ending Boundary (start/end)	Duration Offset	Or Less/ Or More
same as	yes	yes	no	yes
before	yes	yes	yes	yes
after	yes	yes	yes	yes
within...of	yes	yes	required	no
during	yes	no	no	no
includes	no	yes	no	no

TABLE 2-Q

A yes in the Beginning Boundary column indicates that the operator can be preceded by `starts` or `ends` if the left comparand is an interval. A yes in the Ending Boundary column indicates that the timing phrase can be succeeded by a `start` or `end` if the right comparand is an interval. A yes in the duration offset column indicates that the timing phrase may include a duration offset, and a yes in the Or Less/OrMore column indicates that the timing phrase may include an `or less`/`or more` modifier (`or before`/`or after` for the `same as` operator).

In addition, to support more English-like phrasing of timing operations, the keyword `occurs` may appear anywhere that `starts` or `ends` can appear in the timing phrase. For example:

```
X occurs within 3 days of start Y
```

The `occurs` keyword is both optional and ignored by CQL. It is only provided to enable more natural phrasing.

2.5.5.4 Computing Intervals

CQL provides several operators that can be used to combine existing intervals into new intervals. For example:

```
Interval[1, 3] union Interval[3, 6]
```

May 2015

This expression returns the interval `[1, 6]`. Note that interval `union` is only defined if the arguments overlap or meet.

Interval `intersect` results in the overlapping portion of two intervals:

```
Interval[1, 4] intersect Interval[3, 6]
```

This expression results in the interval `[3, 4]`.

Interval `except` computes the difference between two intervals. For example:

```
Interval[1, 4] except Interval[3, 6]
```

This expression results in the interval `[1, 2]`. Note that `except` is only defined for cases that result in a well-formed interval. For example, if either argument properly includes the other and does not start or end it, the result of subtracting one interval from the other would be two intervals, and the result is thus not defined and the result will be `null`.

The following diagrams depict the `union`, `intersect`, and `except` operators for intervals:

union   intersect   except
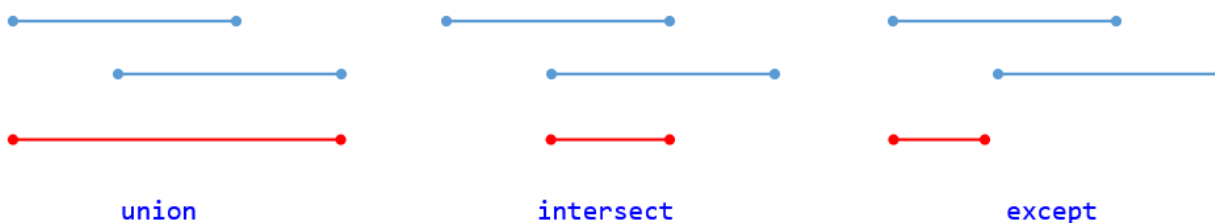
FIGURE 2–A

2.5.5.5 Date/Time Intervals

Because CQL supports date/time values with varying levels of precision, intervals of date/times can potentially involve imprecise date/time values. To ensure well-defined intervals and consistent semantics, date/time intervals are always considered to contain the full set of values contained by the boundaries of the interval. For example, the following interval expression contains all the instants of time, to the millisecond precision, beginning at midnight on January 1st, 2014, and ending at midnight on January 1st, 2015:

```
interval[DateTime(2014, 1, 1, 0, 0, 0, 0), DateTime(2015, 1, 1, 0, 0, 0, 0)]
```

However, if the boundaries of the interval are specified to a lower precision, the interval is interpreted as beginning at some time within the most specified precision, and ending at some time within the most specified precision. For example, the following interval expression contains all the instants of time, to the millisecond precision, beginning sometime in the year 2014, and ending sometime in the year 2015:

```
interval[DateTime(2014), DateTime(2015)]
```

When calculating the duration of the interval, this imprecision will in general result in an *uncertainty*, just as it does when calculating the duration between two imprecise date/time values.

In addition, the boundaries may even be specified to different levels of precision. For example, the following interval expression contains all the instants of time, to the millisecond precision, beginning sometime in the year 2014, and ending sometime on January 1st, 2015:

```
interval[DateTime(2014), DateTime(2015, 1, 1)]
```

2.5.6 List Operators

Clinical information is almost always stored, collected, and presented in terms of lists of information. As a result, the expression of clinical knowledge almost always involves dealing with lists of information in some way. The query construct already discussed provides a powerful mechanism for dealing with lists, but CQL also provides a comprehensive set of operations for dealing with lists in other ways. These operations can be broadly categorized into three groups:

- General List Operations – Operations for dealing with lists in general, such as constructing lists, accessing elements, and determining the number of elements
- Comparisons – Operations for comparing one list to another
- Computation – Operations for constructing new lists based on existing ones

2.5.6.1 Operating on Lists

Although the most common source of lists in CQL is the retrieve expression, lists can also be constructed directly using the *list selector* discussed in List Values.

The elements of a list can be accessed using the *indexer* ([]) operator. For example:

```
X[1]
```

This expression accesses the first element of the list X.

The index of an element e in a list X can be obtained using the IndexOf operator. For example:

```
IndexOf({'a', 'b', 'c' }, 'b') // returns 2
```

If the element is not found in the list, IndexOf returns 0.

In addition, the number of elements in a list can be determined using the Count operator. For example:

```
Count({ 1, 2, 3, 4, 5 })
```

This expression returns the value 5.

Membership in lists can be determined using the in operator and its inverse, contains:

```
{ 1, 2, 3, 4, 5 } contains 4
4 in { 1, 2, 3, 4, 5 }
```

The exists operator can be used to test whether a list contains any elements:

```
exists ( { 1, 2, 3, 4, 5 } )
exists ( { } )
```

The first expression returns true, while the second expression returns false. This is most often used in queries to determine whether a query returns any results.

The First and Last operators can be used to retrieve the first and last elements of a list. For example:

HL7 Standard: Clinical Quality Language, Release 1 - US Realm
May 2015

```
First({ 1, 2, 3, 4, 5 })
Last({ 1, 2, 3, 4, 5 })
First({})
Last({})
```

In the above examples, the first expression returns 1, and the second expression returns 5. The last two expressions both return `null` since there is no first or last element of an empty list. Note that the `First` and `Last` operators refer to the position of an element in the list, not the temporal relationship between elements. In order to extract the *earliest* or *latest* elements of a list, the list would first need to be sorted appropriately.

In addition, to provide consistent and intuitive semantics when dealing with lists, whenever an operation needs to determine whether or not a given list contains an element (including list operations discussed later such as `intersect`, `except`, and `distinct`), CQL uses the notion of *matching*, rather than pure equality. In general, two results match if they are the same value, or if they are both `null`. In this way, missing information can be dealt with uniformly in the context of lists, without changing the 3-valued logic semantics for dealing with missing information directly.

2.5.6.2 Comparing Lists

In addition to list equality, already discussed in Comparison Operators, lists can be compared using the following operators:

Operator	Description
X includes **Y**	Returns true if every element in list Y is also in list X, using matching semantics
X properly includes **Y**	Returns true if every element in list Y is also in list X and list X has more elements than list Y
X included in **Y**	Returns true if every element in list X is also in list Y, using matching semantics
X properly included in **Y**	Returns true if every element in list X is also in list Y, and list Y has more elements than list X

TABLE 2–R

```
{ 1, 2, 3, 4, 5 } includes { 5, 2, 3 }
{ 5, 2, 3 } included in { 1, 2, 3, 4, 5 }
{ 1, 2, 3, 4, 5 } includes { 4, 5, 6 }
{ 4, 5, 6 } included in { 1, 2, 3, 4, 5 }
```

In the above examples, the first two expressions are `true`, but the last two expressions are `false`.

2.5.6.3 Computing Lists

CQL provides several operators for computing new lists from existing ones.

To eliminate duplicates from a list, use the `distinct` operator:

```
distinct { 1, 1, 2, 2, 3, 4, 5 }
```

This example returns:

```
{ 1, 2, 3, 4, 5 }
```

HL7 Standard: Clinical Quality Language, Release 1 - US Realm
May 2015

Note that the distinct operator uses the notion of matching (`matches`) to detect duplicates. Because matching is defined for all types, this means that distinct can be used on lists with elements of any type. In particular, duplicates can be eliminated from lists of tuples, and the operation will use tuple matching (i.e. tuples are equal if they have the same type and the same values (or no value) for each element of the same name).

To combine all the elements from multiple lists, use the `union` operator:

```
{ 1, 2, 3 } union { 3, 4, 5 }
```

This example returns:

```
{ 1, 2, 3, 3, 4, 5 }
```

Note that duplicates are retained in the result of a `union`; a separate `distinct` must be used if duplicates must be eliminated.

To compute only the common elements from multiple lists, use the `intersect` operator:

```
{ 1, 2, 3 } intersect { 3, 4, 5 }
```

This example returns:

```
{ 3 }
```

To remove the elements in one list from another list, use the `except` operator:

```
{ 1, 2, 3 } except { 3, 4, 5 }
```

This example returns:

```
{ 1, 2 }
```

The following diagrams depict the `union`, `intersect`, and `except` operators:
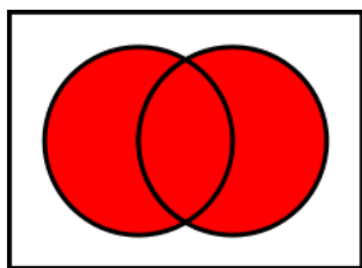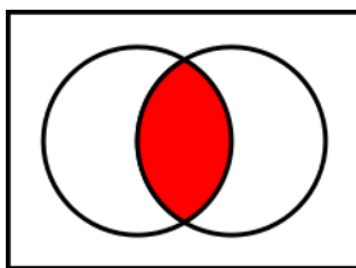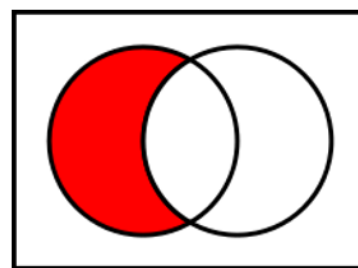
union   intersect   except

FIGURE 2–B

As with the `distinct` operator, the `intersect`, and `except` operators use the matching operator to determine when two elements are the same.

Because lists may contain lists, CQL provides an `expand` operation that can flatten lists of lists:

```
expand { { 1, 2, 3 },  { 3, 4, 5 } }
```

This example returns:

```
{ 1, 2, 3, 3, 4, 5 }
```

Note that as with the `union` operator, duplicate elements are retained in the result.

Note also that expand only flattens one level, it is not recursive.

Although the examples in this section primarily use lists of integers, these operators work on lists with elements of any type.

2.5.6.4 Lists of Intervals

Most list operators in CQL operate on lists of any type, but for lists of intervals, CQL supports a `collapse` operator that determines the list of *unique* intervals from a given list of intervals. Consider the following intervals:
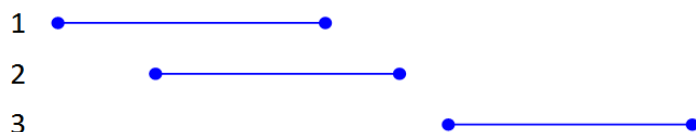
If we want to determine the total duration *covered* by these intervals, we cannot simply use the `distinct` operator, because each of these intervals is different. Yet two of them overlap, so they cover part of the same range. We also can't simply perform an aggregate `union` of the intervals because some of them don't overlap, so there isn't a single interval that covers the entire range.

The solution is the `collapse` operator which returns the set of intervals that *completely cover* the ranges covered by the inputs:
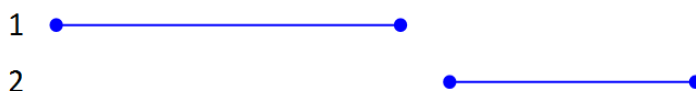
Now, when we take the `Sum` of the durations of the intervals, we are guaranteed not to overcount any particular point in the ranges that may have been included in multiple intervals in the original set.

2.5.7 Aggregate Operators

Summaries and statistical calculations are a critical aspect of being able to represent clinical knowledge, especially in the quality measurement domain. Thus, CQL includes a comprehensive set of aggregate operators.

Aggregate operators are defined to work on lists of values. For example, the `Count` operator works on any list:

```
Count([Encounter])
```

This expression returns the number of `Encounter` events.

The `Sum` operator, however, works only on lists of numbers:

```
Sum({ 1, 2, 3, 4, 5 })
```

This example results in the sum `15`. To sum the results of a list of `Observation` lab values, for example, a query is used to extract the values to be summed:

```
Sum([Observation] R return R.result)
```

In general, `null`s encountered during aggregation are ignored, and with the exception of `Count`, the result of the invocation of an aggregate on an empty list is `null`. `Count` is defined to return `0` for an empty list.

The following table lists the aggregate operators available in CQL:

Operator	Description
Count	Returns the number of elements in its argument
Sum	Returns the numeric sum of the elements in the list
Min	Returns the minimum value of any element in the list
Max	Returns the maximum value of any element in the list
Avg	Returns the numeric average (mean) of all elements in the list
Median	Returns the statistical median of all elements in the list
Mode	Returns the most frequently occurring value in the list
StdDev	Returns the sample standard deviation (square root of the sample variance) of the elements in the list
PopStdDev	Returns the population standard deviation (square root of the population variance) of the elements in the list
Variance	Returns the sample variance (average distance of the data elements from the sample mean, corrected for bias by using N-1 as the denominator in the mean calculation, rather than N) of the elements in the list
PopVariance	Returns the population variance (average distance of the data elements from the population mean) of the elements in the list

TABLE 2–S

2.5.8 Clinical Operators

CQL supports several operators for use with the various clinical types in the language.

2.5.8.1 Quantity Operators

All quantities in CQL have *unit* and *value* components, which can be accessed in the same way as properties. For example:

```
define IsTall: height.units = 'm' and height.value > 2
```

However, because CQL supports operations on quantities directly, this expression could be simplified to:

```
define IsTall: height > 2 'm'
```

This formulation also has the advantage of allowing for the case that the actual value of `height` is expressed in inches.

CQL supports the standard comparison operators (= <> < <= > >=) and the standard arithmetic operators (+ - * /) for quantities. In addition, aggregate operators that utilize these basic comparisons and computations are also supported, such as Min, Max, Sum, etc.

Note that complete support for unit conversion for all valid UCUM units would be ideal, but practical CQL implementations will likely provide support for a subset of units for commonly used clinical dimensions. At a minimum, however, a CQL implementation must respect units and throw an error if it is not capable of normalizing the quantities involved in a given expression to a common unit.

2.5.8.2 Terminology Operators

In addition to providing first-class *valueset* and *codesystem* constructs, CQL provides operators for retrieving and testing membership in valuesets and codesystems:

```
valueset "Acute Pharyngitis": '2.16.840.1.113883.3.464.1003.102.12.1011'
define InPharyngitis: X in "Acute Pharyngitis"
```

These statements define the InPharyngitis expression as true if the Code-valued expression X is in the "Acute Pharyngitis" valueset. Note that valueset membership is based strictly on the definition of matching (i.e. two codes are the same if they have the same values for the code, system, and version elements). CQL explicitly forbids the notion of *terminological equivalence* among codes being used in this context.

Note that this operator can be invoked with a code argument of type String, Code, and Concept. When invoked with a Concept, the result is true if any Code in the Concept is a member of the given valueset.

A common terminological operation involves determining whether a given concept is *implied*, or *subsumed* by another. This operation is generally referred to as *subsumption* and although useful, is deliberately omitted from this specification. The reason for this omission is that subsumption is generally a very complex operation, with different terminology systems providing different mechanisms for defining and interpreting such relationships. As a result, specifying how that occurs is beyond the scope of CQL at this time. This is not to say that a specific library of subsumption operators could not be provided and broadly adopted and used, only that the CQL specification does not attempt to dictate the semantics of that operation.

2.5.8.3 Patient Operators

To support determination of patient age consistently throughout quality logic, CQL defines several age-related operators:

Operator	Description
AgeInYearsAt(X)	Determines the age of the patient in years as of the date X
AgeInYears()	Determines the age of the patient in years as of today. Equivalent to AgeInYearsAt(Today())
AgeInMonthsAt(X)	Determines the age of the patient in months as of the date X
AgeInMonths()	Determines the age of the patient in months as of today. Equivalent to AgeInMonthsAt(Today())
AgeInDaysAt(X)	Determines the age of the patient in days as of the date X

AgeInDays()	Determines the age of the patient in days as of today. Equivalent to AgeInDaysAt(Today())
AgeInHoursAt(X)	Determines the age of the patient in hours as of the date/time X
AgeInHours()	Determines the age of the patient in hours as of now. Equivalent to AgeInHoursAt(Now())
CalculateAgeInYearsAt(D, X)	Determines the age of a person with birthdate D in years as of the date X
CalculateAgeInYears(D)	Determines the age of a person with birthdate D in years as of today. Equivalent to CalculateAgeInYearsAt(D, Today())
CalculateAgeInMonthsAt(D, X)	Determines the age of a person with birthdate D in months as of the date X
CalculateAgeInMonths(D)	Determines the age of a person with birthdate D in months as of today. Equivalent to CalculateAgeInMonthsAt(D, Today())
CalculateAgeInDaysAt(D, X)	Determines the age of a person with birthdate D in days as of the date X
CalculateAgeInDays(D)	Determines the age of a person with birthdate D in days as of today. Equivalent to CalculateAgeInDaysAt(D, Today())
CalculateAgeInHoursAt(D, X)	Determines the age of a person with birthdate D in hours as of the datetime X
CalculateAgeInHours(D)	Determines the age of a person with birthdate D in hours as of now. Equivalent to CalculateAgeInHoursAt(D, Now())

<div align="center">**TABLE 2–T**</div>

These operators calculate age using calendar duration.

2.6 Authoring Artifact Logic

This section provides a walkthrough of the process of developing shareable artifact logic using CQL. The walkthrough is based on the development of the logic for a simplified Chlamydia Screening quality measure and its associated decision support rule.

Although the examples in this guide focus on populations of patients, CQL can also be used to express non-patient-based artifacts such as episode-of-care measures, or organizational measures such as number of staff in a facility. For examples of these types of measures, see the Examples included with this specification.

2.6.1 Running Example

The running example for this walkthrough is a simplification of CMS153, version 2, Chlamydia Screening for Women. The original QDM for this measure was simplified by including only references to the following QDM data elements:

- Patient characteristics of Birthdate and Sex
- Diagnosis, Active
- Laboratory Test, Order
- Laboratory Test, Result

This results in the following QDM:

- **Initial Patient Population =**
  - AND: "Patient Characteristic Birthdate: birth date" >= 16 year(s) starts before start of "Measurement Period"
  - AND: "Patient Characteristic Birthdate: birth date" < 24 year(s) starts before start of "Measurement Period"
  - AND: "Patient Characteristic Sex: Female"
  - AND:
    - OR: "Diagnosis, Active: Other Female Reproductive Conditions" overlaps with "Measurement Period"
    - OR: "Diagnosis, Active: Genital Herpes" overlaps with "Measurement Period"
    - OR: "Diagnosis, Active: Gonococcal Infections and Venereal Diseases" overlaps with "Measurement Period"
    - OR: "Diagnosis, Active: Inflammatory Diseases of Female Reproductive Organs" overlaps with "Measurement Period"
    - OR: " Diagnosis, Active: Chlamydia" overlaps with "Measurement Period"
    - OR: "Diagnosis, Active: HIV" overlaps with "Measurement Period"
    - OR: "Diagnosis, Active: Syphilis" overlaps with "Measurement Period"
    - OR: "Diagnosis, Active: Complications of Pregnancy, Childbirth and the Puerperium" overlaps with "Measurement Period"
    - OR:
      - OR: "Laboratory Test, Order: Pregnancy Test"
      - OR: "Laboratory Test, Order: Pap Test"
      - OR: "Laboratory Test, Order: Lab Tests During Pregnancy"
      - OR: "Laboratory Test, Order: Lab Tests for Sexually Transmitted Infections"
      - during "Measurement Period"
- **Denominator =**
  - AND: "Initial Patient Population"
- **Denominator Exclusions =**
  - None
- **Numerator =**
  - AND: "Laboratory Test, Result: Chlamydia Screening (result)" during "Measurement Period"
- **Denominator Exceptions =**
  - None

Note that these simplifications result in a measure that is not clinically relevant, and the result of this walkthrough is in no way intended to be used in a production scenario. The walkthrough is intended only to demonstrate how CQL can be used to construct shareable clinical logic.

As an aside, one of the simplifications made to the QDM presented above is the removal of the notion of *occurrencing*. Readers familiar with that concept as defined in QDM should be aware that CQL by design does not include this notion. CQL queries are expressive enough that the correlation accomplished by occurrencing in QDM is not required in CQL.

The following table lists the QDM data elements involved and their mappings to the QUICK data structures:

QDM Data Element	QUICK Equivalent
Patient Characteristic Birthdate	Patient.birthDate
Patient Characteristic Sex	Patient.gender
Diagnosis, Active	Condition
Laboratory Test, Order	DiagnosticOrder

| Laboratory Test, Result | DiagnosticReport |

TABLE 2–U

Note that the specific mapping to the QUICK data structures is beyond the scope of this walkthrough; it is only provided here to demonstrate the link back to the original QDM.

Note also that the use of the QDM as a starting point was deliberately chosen to provide familiarity and is not a general requirement for building CQL. Artifact development could also begin directly from clinical guidelines expressed in other formats or directly from relevant clinical domain expertise. Using the QDM provides a familiar and relatively precise way to establish the starting requirements.

2.6.2 Clinical Quality Measure Logic

For clinical quality measures, the CQL library simply provides a repository for definitions of the populations involved. CQL is intended to support both CQM and CDS applications, so it does not contain quality measure specific constructs. Rather, the containing artifact definition, such as an HQMF document, would reference the appropriate criteria expression by name within the CQL document.

With that in mind, a CQL library intended to represent the logic for a CQM must expose at least the population definitions needed for the measure. In this case, we have criteria definitions for:

- Initial Patient Population
- Denominator
- Numerator

Looking at the Initial Patient Population, we have the demographic criteria:

- Patient is at least 16 years old and less than 24 years old at the start of the measurement period.
- Patient is female.

For the age criteria, CQL defines an `AgeInYearsAt` operator that returns the age of the patient as of a given date/time. Using this operator, and assuming a measurement period of the year 2013, we can express the patient age criteria as:

```
AgeInYearsAt(@2013-01-01) >= 16 and AgeInYearsAt(@2013-01-01) < 24
```

In order to use the `AgeInYearsAt` operator, we must be in the `Patient` context:

```
context Patient
```

In addition, to allow this criteria to be referenced both within the CQL library by other expressions, as well as potentially externally, we need to assign an identifier:

```
define InInitialPopulation:
  AgeInYearsAt(@2013-01-01) >= 16 and AgeInYearsAt(@2013-01-01) < 24
```

Because the quality measure is defined over a measurement period, and many, if not all, of the criteria we build will have some relationship to this measurement period, it is useful to define the measurement period directly:

```
define MeasurementPeriod: Interval[
  @2013-01-01T00:00:00.0,
  @2014-01-01T00:00:00.0
)
```

This establishes MeasurementPeriod as the interval beginning precisely at midnight on January 1st, 2013, and ending immediately before midnight on January 1st, 2014. We can now use this in the age criteria:

```
define InInitialPopulation:
  AgeInYearsAt(start of MeasurementPeriod) >= 16
    and AgeInYearsAt(start of MeasurementPeriod) < 24
```

Even more useful would be to define MeasurementPeriod as a *parameter* that can be provided when the quality measure is evaluated. This allows us to use the same logic to evaluate the quality measure for different years. So instead of using a define statement, we have:

```
parameter MeasurementPeriod default Interval[
  @2013-01-01T00:00:00.0,
  @2014-01-01T00:00:00.0
)
```

The InInitialPopulation expression remains the same, but it now accesses the value of the parameter instead of the define statement.

Since we are in the Patient context and have access to the attributes of the Patient (as defined by the data model in use), the gender criteria can be expressed as follows:

```
Patient.gender in "Female Administrative Sex"
```

This criteria requires that the gender attribute of a Patient be a code that is in the valueset identified by "Female Administrative Sex". Of course, this requires the valueset definition:

```
valueset "Female Administrative Sex": '2.16.840.1.113883.3.560.100.2'
```

Putting it all together, we now have:

```
library CMS153_CQM version '2'

using QUICK

parameter MeasurementPeriod default Interval[
  @2013-01-01T00:00:00.0,
  @2014-01-01T00:00:00.0
)

valueset "Female Administrative Sex": '2.16.840.1.113883.3.560.100.2'

context Patient

define InInitialPopulation:
  AgeInYearsAt(start of MeasurementPeriod) >= 16
```

HL7 Standard: Clinical Quality Language, Release 1 - US Realm
May 2015

```
    and AgeInYearsAt(start of MeasurementPeriod) < 24
    and Patient.gender in "Female Administrative Sex"
```

The next step is to capture the rest of the initial population criteria, beginning with this QDM statement:

"Diagnosis, Active: Other Female Reproductive Conditions" overlaps with "Measurement Period"

This criteria has three main components:

- The type of clinical statement involved
- The valueset involved
- The relationship to the measurement period

Using the mapping to QUICK, the equivalent retrieve in CQL is:

```
[Condition: "Other Female Reproductive Conditions"] C
  where Interval[C.onsetDateTime, C.abatementDateTime] overlaps MeasurementPeriod
```

This query retrieves all Condition events for the patient with a code in the "Other Female Reproductive Conditions" valueset that overlap the measurement period. Note that in order to use the overlaps operator, we had to construct an interval from the onsetDateTime and abatementDateTime elements. If the model had an interval-valued "effective time" element, we could have used that directly, rather than having to construct an interval.

The result of the query is a list of conditions. However, this isn't quite what the QDM statement is actually saying. In QDM, the statement can be read loosely as "include patients in the initial patient population that have at least one active diagnosis from the Other Female Reproductive Conditions valueset." To match this in CQL, what we really need to ask is whether the equivalent retrieve above returns any results, which is accomplished with the exists operator:

```
exists ([Condition: "Other Female Reproductive Conditions"] C
  where Interval[C.onsetDateTime, C.abatementDateTime] overlaps MeasurementPeriod)
```

Incorporating the next QDM statement:

OR: "Diagnosis, Active: Genital Herpes" overlaps with "Measurement Period"

We have:

```
exists ([Condition: "Other Female Reproductive Conditions"] C
  where Interval[C.onsetDateTime, C.abatementDateTime] overlaps MeasurementPeriod)
or exists ([Condition: "Genital Herpes"] C
  where Interval[C.onsetDateTime, C.abatementDateTime] overlaps MeasurementPeriod)
```

Which we can repeat for each Diagnosis, Active statement. Note here that even though we are using the same alias, C, for each query, they do not clash because they are only declared within their respective queries (or *scopes*).

Next, we get to the Laboratory Test statements:

- OR: "Laboratory Test, Order: Pregnancy Test"
- OR: "Laboratory Test, Order: Pap Test"
- OR: "Laboratory Test, Order: Lab Tests During Pregnancy"
- OR: "Laboratory Test, Order: Lab Tests for Sexually Transmitted Infections"

- during "Measurement Period"

We use the same approach. The equivalent retrieve for the first criteria is:

```
exists ([DiagnosticOrder: "Pregnancy Test"] O
  where Last(O.event E where E.status = 'completed' sort by E.date).date
    during MeasurementPeriod)
```

This query is retrieving pregnancy tests that were completed within the measurement period. Because diagnostic orders do not have a top-level completion date, the date must be retrieved with a nested query on the events associated with the diagnostic orders. The nested query returns the set of completed events ordered by their completion date, the `Last` invocation returns the most recent of those events, and the `.date` accessor retrieves the value of the `date` element of that event.

And finally, translating the rest of the statements allows us to express the entire initial population as:

```
define InInitialPopulation:
  AgeInYearsAt(start of MeasurementPeriod) >= 16
    and AgeInYearsAt(start of MeasurementPeriod) < 24
    and Patient.gender in "Female Administrative Sex"
    and
    (
      exists ([Condition: "Other Female Reproductive Conditions"] C
        where Interval[C.onsetDateTime, C.abatementDateTime] overlaps MeasurementPeriod)
      or exists ([Condition: "Genital Herpes"] C
        where Interval[C.onsetDateTime, C.abatementDateTime] overlaps MeasurementPeriod)
      or exists ([Condition: "Genococcal Infections and Venereal Diseases"] C
        where Interval[C.onsetDateTime, C.abatementDateTime] overlaps MeasurementPeriod)
      ...
      or exists ([DiagnosticOrder: "Pregnancy Test"] O
        where Last(O.event E where E.status = 'completed' sort by E.date).date
          during MeasurementPeriod)
      ...
    )
```

2.6.3 Using Define Statements

Because CQL allows any number of `define` statements with any identifiers, we can structure the logic of the measure to communicate more meaning to readers of the logic. For example, if we look at the description of the quality measure:

> *Percentage of women 16-24 years of age who were identified as sexually active and who had at least one test for chlamydia during the measurement period.*

it becomes clear that the intent of the Diagnosis, Active and Laboratory Test, Order QDM criteria is to attempt to determine whether or not the patient is sexually active. Of course, we're dealing with a simplified measure and so much of the nuance of the original measure is lost; the intent here is not to determine whether this is in fact a good way in practice to determine whether or not a patient is sexually active, but rather to show how CQL can be used to help communicate aspects of the meaning of quality logic that would otherwise be lost or obscured.

With this in mind, rather than expressing the entire initial patient population as a single `define`, we can break it up into several more understandable and more meaningful expressions:

```
define InDemographic:
  AgeInYearsAt(start of MeasurementPeriod) >= 16
    and AgeInYearsAt(start of MeasurementPeriod) < 24
    and Patient.gender in "Female Administrative Sex"

define SexuallyActive:
  exists ([Condition: "Other Female Reproductive Conditions"] C
    where Interval[C.onsetDateTime, C.abatementDateTime] overlaps MeasurementPeriod)
  or exists ([Condition: "Genital Herpes"] C
    where Interval[C.onsetDateTime, C.abatementDateTime] overlaps MeasurementPeriod)
  or exists ([Condition: "Genococcal Infections and Venereal Diseases"] C
    where Interval[C.onsetDateTime, C.abatementDateTime] overlaps MeasurementPeriod)
  ...
  or exists ([DiagnosticOrder: "Pregnancy Test"] O
    where Last(O.event E where E.status = 'completed' sort by E.date).date
      during MeasurementPeriod)
  ...

define InInitialPopulation:
  InDemographic and SexuallyActive
```

Restructuring the logic in this way not only simplifies the expressions involved and makes them more understandable, but it clearly communicates the intent of each group of criteria.

Note that the `InInitialPopulation` expression is returning a boolean value indicating whether or not the patient should be included in the initial population.

The next population to define is the denominator, which in our simplified expression of the measure is the same as the numerator. Because the intent of the CQL library for a quality measure is only to define the logic involved in defining the populations, it is assumed that the larger context (such as an HQMF artifact definition) is providing the overall structure, including the meaning of the various populations involved. As such, each population definition with the CQL library should include only those aspects that are unique to that population.

For example, the actual criteria for the denominator is that the patient is in the initial patient population. But because that notion is already implied by the definition of a population measure (that the denominator is a subset of the initial population), the CQL for the denominator should simply be:

```
define InDenominator: true
```

This approach to defining the criteria is more flexible from the perspective of actually evaluating a quality measure, but it may be somewhat confusing when looking at the CQL in isolation.

Note that the approach to defining population criteria will actually be established by the CQF-Based HQMF Implementation Guide. We follow this approach here just for simplicity.

Following this approach then, we express the numerator as:

```
define InNumerator:
  exists ([DiagnosticReport: "Chlamydia Screening"] R
    where R.diagnosticDateTime during MeasurementPeriod and R.result is not null)
```

Note that the `R.result is not null` condition is required because the original QDM statement involves a test for the presence of an attribute:

"Laboratory Test, Result: Chlamydia Screening (result)" during "Measurement Period"

The `(result)` syntax indicates that the item should only be included if there is some value present for the `result` attribute. The equivalent expression in CQL is the null test.

Finally, putting it all together, we have a complete, albeit simplified, definition of the logic involved in defining the population criteria for a measure:

```
library CMS153_CQM version '2'

using QUICK

parameter MeasurementPeriod default Interval[
  @2013-01-01T00:00:00.0,
  @2014-01-01T00:00:00.0
)

valueset "Female Administrative Sex": '2.16.840.1.113883.3.560.100.2'
...

context Patient

define InDemographic:
  AgeInYearsAt(start of MeasurementPeriod) >= 16
    and AgeInYearsAt(start of MeasurementPeriod) < 24
    and Patient.gender in "Female Administrative Sex"

define SexuallyActive:
  exists ([Condition: "Other Female Reproductive Conditions"] C
    where Interval[C.onsetDateTime, C.abatementDateTime] overlaps MeasurementPeriod)
  or exists ([Condition: "Genital Herpes"] C
    where Interval[C.onsetDateTime, C.abatementDateTime] overlaps MeasurementPeriod)
  or exists ([Condition: "Genococcal Infections and Venereal Diseases"] C
    where Interval[C.onsetDateTime, C.abatementDateTime] overlaps MeasurementPeriod)
  ...
  or exists ([DiagnosticOrder: "Pregnancy Test"] O
    where Last(O.event E where E.status = 'completed').date
      during MeasurementPeriod)
  ...

define InInitialPopulation:
  InDemographic and SexuallyActive

define InDenominator: true

define InNumerator:
  exists ([DiagnosticReport: "Chlamydia Screening"] R
    where R.diagnosticDateTime during MeasurementPeriod and R.result is not null)
```

2.6.4 Clinical Decision Support Logic

Using the same simplified measure expression as a basis, we will now build a complementary clinical decision support rule that can provide guidance at the point-of-care. In general, when choosing what decision support artifacts will be most complementary to a given quality measure,

several factors must be considered including EHR and practitioner workflows, data availability, the potential impacts of the guidance, and many others.

Though these are all important considerations and should not be ignored, they are beyond the scope of this document, and for the purposes of this walkthrough, we will assume that a point-of-care decision support intervention has been selected as the most appropriate artifact.

When building a point-of-care intervention based on a quality measure, several specific factors must be considered.

First, quality measures typically contain logic designed to identify a specific setting in which a particular aspect of health quality is to be measured. This usually involves identifying various types of encounters. By contrast, a point-of-care decision support artifact is typically written to be evaluated in a specific context, so the criteria around establishing the setting can typically be ignored. For the simplified measure we are dealing with, the encounter setting criteria were removed as part of the simplification.

Second, quality measures are designed to measure quality within a specific timeframe, whereas point-of-care measures don't necessarily have those same restrictions. For example, the diagnoses in the current example are restricted to the measurement period. There may be some clinically relevant limit on the amount of time that should be used to search for diagnoses, but it does not necessarily align with the measurement period. For the purposes of this walkthrough, we will make the simplifying assumption that any past history of the relevant diagnoses is a potential indicator of sexual activity.

Third, quality measures are written retrospectively, that is, they are always dealing with events that occurred in the past. By contrast, decision support artifacts usually involve prospective, as well as retrospective data. As such, different types of clinical events may be involved, such as planned or proposed events.

Fourth, quality measures, especially proportion measures, typically express the numerator criteria as a positive result, whereas the complementary logic for a decision support rule would be looking for the absence of the criteria. For example, the criteria for membership in the numerator of the measure we are using is that the patient has had a Chlamydia screening within the measurement period. For the point-of-care intervention, that logic becomes a test for patients that have *not* had a Chlamydia screening.

And finally, although present in some quality measures, many do not include criteria to determine whether or not there is some practitioner- or patient-provided reason for not taking some course of action. This is often due to the lack of a standardized mechanism for describing this criteria and is usually handled on a measure-by-measure basis as part of actually evaluating measures. Regardless of the reason, because a point-of-care intervention has the potential to interrupt a practitioner workflow, the ability to determine whether or not a course of action being proposed has already been considered and rejected is critical.

With these factors in mind, and using the CQL for the measure we have already built, deriving a point-of-care intervention is fairly straightforward.

To begin with, we are using the same data model, QUICK, the same valueset declarations, and the same context:

```
library CMS153_CDS version '2'

using QUICK

valueset "Female Administrative Sex": '2.16.840.1.113883.3.560.100.2'
...

context Patient
```

Note that we are not using the `MeasurementPeriod` parameter. There are other potential uses for parameters within the point-of-care intervention (for example, to specify a threshold for how far back to look for a Chlamydia screening), but we are ignoring those aspects for the purposes of this walkthrough.

For the `InDemographic` criteria, we are then simply concerned with female patients between the ages of 16 and 24, so we change the criteria to use the `AgeInYears`, rather than the `AgeInYearsAt` operator, to determine the patient's age as of today:

```
define InDemographic:
  AgeInYears() >= 16 and AgeInYears() < 24
    and Patient.gender in "Female Administrative Sex"
```

Similarly for the `SexuallyActive` criteria, we remove the relationship to the measurement period:

```
define SexuallyActive:
  exists ([Condition: "Other Female Reproductive Conditions"])
  or exists ([Condition: "Genital Herpes"])
  or exists ([Condition: "Genococcal Infections and Venereal Diseases"])
  ...
  or exists ([DiagnosticOrder: "Pregnancy Test"])
  ...
```

For the numerator, we need to invert the logic, so that we are looking for patients that have not had a Chlamydia screening, and rather than the measurement period, we are looking for the test within the last year:

```
not exists ([DiagnosticReport: "Chlamydia Screening"] R
  where R.diagnosticDateTime during Interval[Today() - 1 years, Today()]
    and R.result is not null)
```

In addition, we need a test to ensure that the patient does not have a planned Chlamydia screening:

```
not exists ([ProcedureRequest: "Chlamydia Screening"] R
  where R.timingDateTime same day or after Today())
```

And to ensure that there is not a reason for not performing a Chlamydia screening:

```
not exists ([Observation: "Reason for not performing Chlamydia Screening"])
```

We combine those into a `NoScreening` criteria:

```
define NoScreening:
  not exists ([DiagnosticReport: "Chlamydia Screening"] R
    where R.diagnosticDateTime during Interval[Today() - 1 years, Today()]
      and R.result is not null)
  and not exists ([ProcedureRequest: "Chlamydia Screening"] R
```

```
    where R.timingDateTime same day or after Today())
  and
    not exists ([Observation: "Reason for not performing Chlamydia Screening"])
```

And finally, we provide an overall condition that indicates whether or not this intervention should be triggered:

```
define NeedScreening: InDemographic and SexuallyActive and NoScreening
```

Now, this library can be referenced by a CDS knowledge artifact, and the condition can reference the `NeedScreening` expression, which loosely reads: the patient needs screening if they are in the appropriate demographic, have indicators of sexual activity, and do not have screening.

In addition, this library should include the proposal aspect of the intervention. In general, the overall artifact definition (such as a CDS KAS artifact) would define what actions should be performed when the condition is satisfied. Portions of that action definition may reference other expressions within a CQL library, just as the HQMF for a quality measure may reference multiple expressions within CQL to identify the various populations in the measure. In this case, the intervention may construct a proposal for a Chlamydia Screening:

```
define ChlamydiaScreeningRequest: ProcedureRequest {
  type: Code '442487003' from "SNOMED-CT"
      display ' Screening for Chlamydia trachomatis (procedure)'
  status: 'proposed'
  // values for other elements of the request...
}
```

The containing artifact would then use this expression as the target of an action, evaluating the expression if the condition of the decision support rule is met, and returning the result as the proposal to the calling environment.

2.6.5 Using Libraries to Share Logic

The previous examples of building a quality measure and a decision support artifact have so far demonstrated the ability to describe the logic involved using the same underlying data model, as well as the same expression language. Now we can take that one step further and look at the use of CQL libraries to actually express the artifacts using the same logic, rather than just the same data model and language.

We start by identifying the aspects that are identical between the two:

1. SexuallyActive criteria, without the timing relationship
2. ChlamydiaScreening test, without the timing relationship

With these in mind, we can create a new library, , that will contain the common elements:

```
library CMS153_Common version '2'

using QUICK

valueset "Female Administrative Sex": '2.16.840.1.113883.3.560.100.2'
...

context Patient
```

```
define ConditionsIndicatingSexualActivity:
  [Condition: "Other Female Reproductive Conditions"]
    union [Condition: "Genital Herpes"]
    union ...

define LaboratoryTestsIndicatingSexualActivity:
  [DiagnosticOrder: "Pregnancy Test"]
    union [DiagnosticOrder: "Pap"]
    union ...

define ResultsPresentForChlamydiaScreening:
  [DiagnosticReport: "Chlamydia Screening"] R where R.result is not null
```

Using this library, we can then rewrite the CQM to reference the common elements from the library:

```
library CMS153_CQM version '2'

using QUICK

include CMS153_Common version '2' called Common

parameter MeasurementPeriod default Interval[
  @2013-01-01T00:00:00.0,
  @2014-01-01T00:00:00.0
)

context Patient

define InDemographic:
  AgeInYearsAt(start of MeasurementPeriod) >= 16
    and AgeInYearsAt(start of MeasurementPeriod) < 24
    and Patient.gender in Common."Female Administrative Sex"

define SexuallyActive:
  exists (Common.ConditionsIndicatingSexualActivity C
    where Interval[C.onsetDateTime, C.abatementDateTime] overlaps MeasurementPeriod)
  or exists (Common.LaboratoryTestsIndicatingSexualActivity R
    where R.diagnosticDateTime during MeasurementPeriod)

define InInitialPopulation:
  InDemographic and SexuallyActive

define InDenominator:
  true

define InNumerator:
  exists (Common.ResultsPresentForChlamydiaScreening S
    where S.diagnosticDateTime during MeasurementPeriod)
```

And similarly for the CDS artifact:

```
library CMS153_CDS version '2'

using QUICK
```

```
include CMS153_Common version '2' called Common

valueset "Reason for not performing Chlamydia Screening": 'TBD'

context Patient

define InDemographic:
  AgeInYears() >= 16 and AgeInYears() < 24
    and Patient.gender in Common."Female Administrative Sex"

define SexuallyActive:
  exists (Common.ConditionsIndicatingSexualActivity)
  or exists (Common.LaboratoryTestsIndicatingSexualActivity)

define NoScreening:
  not exists (Common.ResultsPresentForChlamydiaScreening S
    where S.diagnosticDateTime during Interval[Today() - 1 years, Today()])
  and not exists ([ProcedureRequest: Common."Chlamydia Screening"] R
    where R.timingDateTime same day or after Today()

define NeedScreening: InDemographic and SexuallyActive and NoScreening
```

In addition to sharing between quality measures and clinical decision support artifacts, the use of common libraries will allow the same logic to be shared by multiple quality measures or decision support artifacts. For example, a set of artifacts for measurement and improvement of the treatment of diabetes could all use a common library that provides base definitions for determining when a patient is part of the population of interest.

3 DEVELOPER'S GUIDE

This chapter complements the Author's Guide by providing more in-depth discussion of language elements, semantics, more complex query scenarios, and more advanced topics such as typing and function definition. Readers are expected to be familiar with the content of the Author's Guide in the discussions that follow.

3.1 Lexical Elements

CQL is intended to be an authoring language. As such, the syntax is designed to be intuitive and clear, both when writing and reading the language. Care has been taken to ensure that the language contains a simple and clear core set of language elements, and that they all interact in a consistent and predictable manner.

As with any traditional computer language, CQL uses typical lexical elements such as whitespace, keywords, symbols, comments, and so on.

CQL defines the following basic lexical elements:

Element	Description
Whitespace	Whitespace defines the separation between all tokens in the language
Comment	Comments are ignored by the language, allowing for descriptive text
Literal	Literals allow basic values to be represented within the language
Symbol	Symbols such as +, -, *, and /
Keyword	Grammar-recognized keywords such as define and where
Identifier	User-defined identifiers

TABLE 3-A

Every valid CQL document can be broken down into a set of tokens, each of which is one of these basic lexical elements. The following sections describe each of these elements in more detail.

3.1.1 Whitespace

CQL defines *tab*, *space*, and *return* as *whitespace*, meaning they are only used to separate other tokens within the language. Any number of whitespace characters can appear, and the language does not use whitespace for anything other than delimiting tokens.

3.1.2 Comments

CQL defines two styles of comments, *single-line*, and *multi-line*. A single-line comment consists of two forward slashes, followed by any text up to the end of the line:

```
define Foo: 1 + 1 // This is a single-line comment
```

To begin a multi-line comment, the typical forward slash-asterisk token is used. The comment is closed with an asterisk-forward slash, and everything enclosed is ignored:

```
/*
This is a multi-line comment
Any text enclosed within is ignored
*/
```

Note that nested multi-line comments are not supported.

3.1.3 Literals

Literals provide for the representation of basic values within CQL. The following types of literals are supported:

Literal	Description
Null	The null literal (`null`)
Boolean	The boolean literals (`true` and `false`)
Integer	Sequences of digits in the range 0..2^32-1
Decimal	Sequences of digits with a decimal point, in the range 0.0.. 10^28 –10^-8
String	Strings of any character enclosed within single-ticks (`'`)
DateTime	The at-symbol (@) followed by an ISO-8601 compliant representation of a date/time
Time	The at-symbol (@) followed by an ISO-8601 compliant representation of a time
Quantity	An integer or decimal literal followed by a datetime precision specifier, or a UCUM unit specifier

<p style="text-align:center">TABLE 3–B</p>

3.1.4 Symbols

Symbols provide structure to the grammar and allow symbolic invocation of common operators such as addition. CQL defines the following symbols:

Symbol	Description
`:`	Definition operator, typically read as "defined as"
`()`	Parentheses for delimiting groups
`[]`	Brackets for indexing into lists and strings, as well as delimiting the retrieve expression
`{}`	Braces for delimiting lists, tuples, and function bodies
`<>`	Angle-brackets for delimiting generic types within type specifiers
`.`	Period for qualifiers and accessors
`,`	Comma for delimiting items in a syntactic list
`= <> <= < > >=`	Comparison operators for comparing values
`+ - * / ^`	Arithmetic operators for performing calculations

<p style="text-align:center">TABLE 3–C</p>

3.1.5 Keywords

Keywords are words that are recognized by the parser and used to build the various language constructs. CQL defines the following keywords:

after	display	meets	seconds
all	distinct	millisecond	start
and	div	milliseconds	starts
as	duration	minimum	sort
asc	during	minute	successor
ascending	else	minutes	such that
before	end	mod	then
between	ends	month	time
by	except	months	timezone
called	exists	not	to
case	expand	null	true
cast	false	occurs	Tuple
Code	from	of	union
codesystem	function	or	using
codesystems	hour	or after	valueset
collapse	hours	or before	version
Concept	if	or less	where
contains	in	or more	when
context	include	overlaps	width
convert	includes	parameter	with
date	included in	predecessor	within
day	intersect	private	without
days	Interval	properly	xor
default	is	public	year
define	library	return	years
desc	List	same	
descending	matches	singleton	
difference	maximum	second	

In general, keywords within CQL are also considered *reserved* words, meaning that it is illegal to use them as identifiers. If necessary, identifiers that clash with a reserved word can be double-quoted.

3.1.6 Identifiers

Identifiers are used to name various elements within the language. There are two types of identifiers in CQL, simple, and quoted.

A simple identifier is any alphabetical character or an underscore, followed by any number of alpha-numeric characters or underscores. For example, the following are all valid simple identifiers:

```
Foo
Foo1
_Foo
foo
FOO
```

Note also that these are all unique identifiers. By convention, simple identifiers in CQL should not begin with underscores, and should be Pascal-cased (meaning the first letter of every word within the identifier is capitalized), rather than using underscores.

In particular, the use of identifiers that differ only in case should be avoided.

A quoted identifier is any sequence of characters enclosed in double-quotes ():

HL7 Standard: Clinical Quality Language, Release 1 - US Realm
May 2015

```
"Encounter, Performed"
"Diagnosis, Active"
```

The use of double-quotes allows identifiers to contain spaces, commas, and other characters that would not be allowed within simple identifiers. This allows identifiers within CQL to be much more descriptive and readable.

To specify a quoted-identifier that includes a double-quote (`"`), two double-quotes (`""`) may be used:

```
"Encounter ""Inpatient"""
```

Note that double-quoted identifiers are still case-sensitive, and as with simple identifiers, the use of double-quoted identifiers that differ only in case should be avoided.

3.1.7 Operator Precedence

CQL uses standard in-fix operator notation for expressing computational logic. As a result, CQL also adopts the expected operator precedence to ensure consistent and predictable behavior of expressions written using CQL. The following table lists the order of operator precedence in CQL from highest to lowest:

Category	Operators
Primary	`. [] ()`
Conversion Phrase	`convert..to`
Unary Arithmetic	unary `+/-`
Extractor	`start`/`end`/`duration`/`width`/`successor`/`predecessor of` *component*/`singleton from`
Exponentiation	`^`
Multiplicative	`* /` `div mod`
Additive	`+ -`
Conditional	`if..then..else` `case..else..end`
Unary List	`distinct collapse expand`
Unary Test	`is null`/`true`/`false`
Type Operators	`is as cast..as`
Unary Logical	`not exists`

Between	`between`
	 precision `between`
	 `difference in` *precision* `between`

Comparison	`<= < > >=`

Timing Phrase	`same as`
	 `includes`
	 `during`
	 `before`/`after`
	 `within`

Interval Operators	`meets overlaps starts ends`
Equality	`= <> matches`

HL7 Standard: Clinical Quality Language, Release 1 - US Realm
May 2015

Membership	in contains
Conjunction	and
Disjunction	or xor
Binary List	union intersect except

<div align="center">

TABLE 3–D

</div>

As with any typical computer language, parentheses can always be used to force order-of-operations if the defined operator precedence results in the incorrect evaluation of a given expression.

When multiple operators appear in a single category, precedence is determined by the order of appearance in the expression, left to right.

3.1.8 Case-Sensitivity

To encourage consistency and reduce potential confusion, CQL is a case-sensitive language. This means that case is considered when matching keywords and identifiers in the language. For example, the following CQL is invalid:

```
Define Foo: 1 + 1
```

The declaration is illegal because the parser will not recognize Define as a keyword.

3.2   Libraries

Libraries provide the basic unit of code organization for CQL. Each CQL file contains a single library, and may include any number of libraries by reference, subject to the following constraints:

- The local identifier for a library must be unique within the artifact.
- Circular library references are not allowed.
- Library references are not transitive.

Because the identifier for a library is just an identifier, it may be either a simple identifier, or a quoted-identifier, which may actually be a URL, an OID, or any other identifier system. It is up to the implementation and environment what interpretation, if any, is given to the identifier of a library.

Libraries may also be declared with a specific version. When referencing a library, the reference may include a version specifier. If the reference includes a version specifier, the library with that version specifier must be used. If the reference does not include a version specifier, it is up to the implementation environment to provide the most appropriate version of the referenced library.

It is an error to reference a specific version of a library if the library does not have a version specifier, or if there is no library with the referenced version.

3.2.1 Access Modifiers

Each component of a library may have an access modifier applied, either `public` or `private`. If no access modifier is applied, the component is considered public. Only public components of a library may be accessed by referencing libraries. Private components can only be accessed within the library itself.

3.2.2 Identifier Resolution

For identifiers, if a library name is not provided, the identifier must refer to a locally defined component. If a library name is provided, it must be the local identifier for the library, and that library must contain the identifier being referenced.

3.2.3 Function Resolution

For functions, if a library name is not provided, the invocation must refer to a locally defined function, or a CQL system function. Function resolution proceeds by attempting to match the *signature* of the invocation, i.e. the number and type of each argument, to a defined signature for the function. Because the CQL type system supports subtyping, generics, and implicit conversion and casting, it is possible for an invocation signature to match multiple defined signatures. In these cases, the *least converting* signature is chosen, meaning the signature with the fewest required conversions. If multiple signatures have the same number of required conversions, an ambiguous resolution error is thrown, and the author must provide an explicit cast or conversion to resolve the ambiguity.

If a library name is provided, only that library will be searched for a resolution.

3.3 Data Models

CQL allows any number of data models to be included in a given library, subject to the following constraints:

- The data model identifier must be unique, both among data models, as well as libraries.
- Data model references are not included from referenced libraries. To reference the data types in a data model, an appropriate local using declaration must be specified.

As with library references, data model references may include a version specifier. If a version is specified, then the environment must ensure that the version specifier matches the version of the data model supplied. If no data model matching the requested version is present, an error is raised.

3.3.1 Alternate Data Models

Although the examples in this specification generally use the QUICK model (part of the Clinical Quality Framework), CQL itself does not require or depend on a specific data model. For example, the following sample is taken from the CMS146v2_using_QDM.cql file in the Examples section of the specification:

```
["Encounter, Performed": "Ambulatory/ED Visit"] E
  with ["Diagnosis, Active": "Acute Pharyngitis"] P such that
```

```
interval[P."start datetime", P."stop datetime")
        overlaps after interval[E."start datetime", E."stop datetime")
```

In this example, QDM is used as the data model. Note the use of quoted attribute identifiers to allow for the spaces in the names of QDM attributes.

3.3.2 Multiple Data Models

Because CQL allows multiple `using` declarations, the possibility exists for clashes within retrieve expressions. For example, a library that used both QUICK and vMR may clash on the name `Encounter`. In general, the resolution process for class names within CQL proceeds as follows:

- If the class name has no qualifier, then each model used in the current library is searched for an exact match.
    - o If an exact match is found in more than one model, the reference is considered ambiguous and an error is thrown that the class reference is ambiguous among the matches found.
    - o If an exact match is found in only one model, that model and type is used.
    - o If no match is found in any model, an error is thrown that the referenced name cannot be resolved.
- If the class name has a qualifier, then the qualifier specifies the model to be searched, and only that model is used to attempt a resolution.
    - o If the qualifier specifies the name of a model that cannot be found in the current library, an error is thrown that the referenced model cannot be found.
    - o If an exact match is found in the referenced model, that class is used.
    - o If no exact match is found, an error is thrown that the qualified class name cannot be resolved.

3.4 Types

CQL is a statically typed language, meaning that it is possible to infer the type of any given expression, and for any given operator invocation, the type of the arguments must match the types of the operands. To provide complete support for the type system, CQL supports several constructs for dealing with types including *type specifiers*, as well as *conversion*, *casting*, and *type-testing* operators.

CQL uses a single-inheritance type system, meaning that each type is derived from at most one type. Given a type T and a type T' derived from type T, the following statements are true:

- The type T is a *supertype* of type T'.
- The type T' is a *subtype* of type T.
- A value of type T' may appear anywhere a value of type T is expected.

3.4.1 System-Defined Types

CQL defines several base types that provide the elements for constructing other types, as well as for defining the operations available within the language.

The maximal supertype is System.Any. All other types derive from System.Any, meaning that any value is of some type, and also ultimately of type System.Any.

All the system-defined types derive directly from System.Any. The primitive types and their ranges are summarized here:

Type	Range	Step Size
Boolean	false..true	N/A
Integer	-2^31..2^31 – 1	1
DateTime	@0001-01-01T00:00:00.0..@9999-12-31T23:59:59.999	1 millisecond
Decimal	-10^28 – 10^-8..10^28 – 10^-8	10^-8
String	All strings of length 2^31-1 or less.	N/A
Time	@T00:00:00.0..@T23:59:59.999	1 millisecond

TABLE 3–E

In addition, CQL defines several structured types to facilitate representation and manipulation of clinical information:

Type	Description
Code	Represents a clinical terminology code, including the code identifier, system, version, and display.
Concept	Represents a single concept as a list of equivalent Codes.
Quantity	Represents a quantity with a dimension, specified in UCUM units.

TABLE 3–F

For more information about these types, refer to the CQL Reference section on Types.

3.4.2 Specifying Types

In various constructs, the type of a value must be specified. For example, when defining the type of a parameter, or when testing a value to determine whether it is of a specific type. CQL provides the *type specifier* for this purpose. There are four categories of type-specifiers, corresponding to the four categories of types supported by CQL:

- Named Types
- Tuple Types
- Interval Types
- List Types

The *named type specifier* is simply the name of the type. For example:

```
parameter Threshold Integer
```

This example declares a parameter named Threshold of type Integer.

The *tuple type specifier* allows the names and types of the elements of the type to be specified. For example:

```
parameter Demographics Tuple { address String, city String, zip String }
```

The *interval type specifier* allows the point-type of the interval to be specified:

```
parameter Range Interval<Integer>
```

And finally, the *list type specifier* allows the element-type of a list to be specified:

```
parameter Points List<Integer>
```

3.4.3 Type Testing

CQL supports the ability to test whether or not a value is of a given type. For example:

```
5 is Integer
```

returns `true` because `5` is an `Integer`.

In general, the *is* relationship determines whether or not a given type is derived from another type. Given a type T and a type T' derived from type T, the following definitions hold:

- Identity – T is T

- Subtype – T' is T

For interval types, given a point type P, and a point type P' derived from type P, interval type Interval<P'> is a subtype of interval type Interval<P>.

For list types, given an element type E, and an element type E' derived from type E, list type List<E'> is a subtype of list type List<E>.

For tuple types, given a tuple type T with elements E_1, E_2, ...E_n, names N_1, N_2, ...N_n, and types T_1, T_2, ...T_n, respectively, a tuple type T' with elements E'_1, E'_2, ...E'_n, names N'_1, N'_2, ...N'_n, and types T'_1, T'_2, ...T'_n, type T' is a subtype of type T if and only if:

- The number of elements in each type is the same: $|E| = |E'|$

- For each element in T, there is one element in T' with the same name, and the type of the matching element in T' is a subtype of the type of the element in T.

For structured types, the supertype is specified as part of the definition of the type. Subtypes inherit all the elements of the supertype and may define additional elements that are only present on the derived type.

3.4.4 Conversion

Conversion is the operation of turning a value from one type into another. For example, converting a number to a string, or vice-versa. CQL supports explicit conversion operators, as well as implicit conversion for some specific types.

3.4.4.1 Explicit Conversion

The explicit `convert` can be used to convert a value from one type to another. For example, to conver the string representation of a date/time to a `DateTime` value:

```
convert '2014-01-01T12:00:00.0-06:00' to DateTime
```

If the conversion cannot be performed, a run-time error will be raised. For example:

```
convert 'Foo' to Integer
```

will result in an error. The convert syntax is equivalent to invoking one of the defined explicit conversion operators:

Operator	Description
ToBoolean(String)	Converts the string representation of a boolean value to a Boolean value
ToInteger(String)	Converts the string representation of an integer value to an Integer value using the format (+|-)d*
ToDecimal(Integer)	Converts an Integer value to an equivalent Decimal value
ToDecimal(String)	Converts the string representation of a decimal value to a Decimal value using the format (+|-)d*.d*
ToQuantity(String)	Converts the string representation of a quantity value to a Quantity value using the format (+|-)d*.d*'units'
ToDateTime(String)	Converts the string representation of a date/time value to a DateTime value using ISO-8601 format: YYYY-MM-DDThh:mm:ss.fff(+|-)hh:mm
ToTime(String)	Converts the string representation of a time value to a Time value using ISO-8601 format: Thh:mm:ss.fff(+|-)hh:mm
ToString(Boolean)	Converts a Boolean value to its string representation (true|false)
ToString(Integer)	Converts an Integer value to its string representation
ToString(Decimal)	Converts a Decimal value to its string representation
ToString(Quantity)	Converts a Quantity value to its string representation
ToString(DateTime)	Converts a DateTime value to its string representation
ToString(Time)	Converts a Time value to its string representation
ToConcept(Code)	Converts a Code value to a Concept with the given Code as its primary and only Code. If the Code has a display value, the Concept will have the same display value.
ToConcept(List<Code>)	Converts a list of Code values to a Concept with the first Code in the list as the primary Code. If the primary Code has a display value, the Concept will have the same display value.

TABLE 3–G

FOR A COMPLETE DESCRIPTION OF THESE CONVERSION OPERATORS, REFER TO THE TABLE 9–D

Type Operators section in the CQL Reference.

3.4.4.2 Implicit Conversions

In addition to the explicit conversion operators discussed above, CQL supports the implicit conversions for specific types to enable expressions to be built more easily. The following table lists the explicit and implicit conversions supported in CQL:

From\To	Boolean	Integer	Decimal	Quantity	String	Datetime	Time	Code	Concept
Boolean	N/A	-	-	-	Explicit	-	-	-	-
Integer	-	N/A	Implicit	-	Explicit	-	-	-	-
Decimal	-	-	N/A	-	Explicit	-	-	-	-
Quantity	-	-	-	N/A	Explicit	-	-	-	-
String	Explicit	Explicit	Explicit	Explicit	N/A	Explicit	Explicit	-	-
Datetime	-	-	-	-	Explicit	N/A	-	-	-
Time	-	-	-	-	Explicit	-	N/A	-	-
Code	-	-	-	-	-	-	-	N/A	Implicit
Concept	-	-	-	-	-	-	-	-	N/A

TABLE 3–H

Although implicit conversions can be performed using the explicit convert, the language will also automatically apply implicit conversions when appropriate to produce a correctly typed expression. For example, consider the following multiplication:

```
define MixedMultiply: 1 * 1.0
```

The type of the literal `1` is `Integer`, and the type of the literal `1.0` is `Decimal`. To infer the type of the expression correctly, the language will implicitly convert the type of the `1` to `Decimal` by inserting a `ToDecimal` invocation. The multiplication is then performed on two `Decimals`, and the result type is `Decimal`.

3.4.5 Casting

Casting is the operation of treating a value of some base type as a more specific type at run-time. The `as` operator provides this functionality. For example, given a model that defines an `ImagingProcedure` as a specialization of a `Procedure`, in the following example:

```
define AllProcedures: [Procedure]
define ImagingProcedures:
  AllProcedures P
    where P is ImagingProcedure
    return P as ImagingProcedure
```

the `ImagingProcedures` expression returns all procedures that are instances of `ImagingProcedure` as instances of `ImagingProcedure`. This means that attributes that are specific to `ImagingProcedure` can be accessed.

If the run-time type of the value is not of the type specified in the `as` operator, the result is `null`.

In addition, CQL supports a *strict* cast, which has the same semantics as casting, except that if the run-time type of the value is not of the type specified, a run-time error is thrown. The keyword `cast` is used to indicate a strict cast:

```
define StrictCast: cast First(Procedures) as ImagingProcedure
```

3.4.5.1 Implicit Casting

CQL also supports the notion of *implicit casting* to prevent the need to cast a `null` literal to a specific type. For example, consider the following expression:

```
define ImplicitCast: 5 * null
```

The type of the first argument to the multiplication is `Integer`, and the type of the second argument is `Any`, an untyped `null` literal. But multipication of `Integer` and `Any` is not defined and `Any` is a supertype of `Integer`, not a subtype. This means that with strict typing, this expression would not compile without the addition of an explicit cast:

```
define ImplicitCast: 5 * (null as Integer)
```

To avoid the need for this explicit cast, CQL implicitly casts the `Any` to `Integer`.

In addition, CQL defines implicit casting of a named structured type to its equivalent tuple type. For example, given the type `Person` with elements `Name` of type `String` and `DOB` of type `DateTime`, the following comparison is valid:

```
define TupleComparison: Person { Name: 'Joe', DOB: @1970-01-01 } = Tuple { Name: 'Joe',
DOB: @1970-01-01 }
```

In this case, the structured value will be implicitly cast to the equivalent tuple type, and the comparison will evaluate to true.

Note that the opposite implicit cast, from a tuple to a named structured type, does not occur because a named structured type has additional information (namely the type hierarchy) that cannot be inferred from the definition of a tuple type. In such cases, an explicit cast can be used:

```
define TupleExpression: Tuple { Name: 'Joe', DOB: @1970-01-01 }
define TupleCast: TupleExpression as Person
```

The cast from a tuple to a structured type requires that the set of elements in the tuple type be a subset (possibly proper) of the elements in the structured type.

3.5 Conditional Expressions

To simplify the expression of complex logic, CQL provides two flavors of conditional expressions, the `if` expression, and the `case` expression.

The if expression allows a single condition to select between two expressions:

```
if Count(X) > 0 then X[1] else 0
```

This expression checks the count of `X` and returns the first element if it is greater than `0`; otherwise, the expression returns `0`.

The `case` expression allows multiple conditions to be tested, and comes in two flavors: standard case, and selected case.

A standard case allows any number of conditions, each with a corresponding expression that will be the result of the `case` if the associated condition evaluates to `true`. If none of the conditions evaluate to `true`, the `else` expression is the result:

```
case
  when X > Y then X
  when Y > X then Y
  else 0
end
```

A selected case specifies a comparand, and each case item specifies a possible value for the comparand. If the comparand is equal to a case item, the corresponding expression is the result of the selected case. If the comparand does not equal any of the case items, the else expression is the result:

```
case X
  when 1 then 12
  when 2 then 14
  else 15
end
```

3.6   Nullological Operators

To provide complete support for missing information, CQL supports several operators for testing for and dealing with null results.

To provide a null result, use the `null` keyword:

```
null
```

To test whether an expression is `null`, use the *null test*:

```
X is null
X is not null
```

To replace a null with the result of an expression, use a simple `if` expression:

```
if X is null then Y else X
```

To return the first non-null expression among two or more expressions, use the `Coalesce` operator:

```
Coalesce(X, Y, Z)
```

which is equivalent to:

```
case
  when X is not null then X
  when Y is not null then Y
  else Z
end
```

In addition, CQL supports the boolean-test operators `is [not] true` and `is [not] false`. These operators, like the null-test operator, only return `true` and `false`, they will not propagate a `null` result.

```
X is true
X is not false
```

The first example will return `true` if X evaluates to `true`, `false` if X evaluates to `false` or `null`. The second example will return `true` if X evaluates to `true` or `null`, `false` if X evaluates to `false`. Note

in particular that these operators are *not* equivalent to comparison of Boolean results using equality or inequality.

3.7   String Operators

Although less common in typical clinical logic, some use cases require string manipulation. As such, CQL supports a core set of string operators.

Like lists, strings are 1-based in CQL. To index into a string, use the *indexer* operator:

```
X[1]
```

To determine the length of string, use the `Length` operator:

```
Length(X)
```

To determine the position of a given pattern within a string, use the `PositionOf` operator:

```
PositionOf('cde', 'abcdefg')
```

The `PositionOf()` operator returns the index of the starting character of the first argument in the second argument, if the first argument can be located in the second argument. Otherwise, `PositionOf()` returns `0` to indicate the pattern was not found in the string.

To return a substring from a given string, use the `Substring` operator:

```
Substring('abcdefg', 1, 3)
```

This example returns the string `'abc'`. The second argument is the starting index of the substring to be returned, and the third argument is the length of the substring to be returned. If the length is greater than number of characters present in the string from the starting index on, the result includes only the remaining characters. If the starting index is less than 1, or greater than the length of the string, the result is `null`. The third argument is optional; if it is not provided, the substring is taken from the starting index to the end of the string.

To concatenate strings, use the `+` operator:

```
'abc' + 'defg'
```

To combine a list of strings, use the `Combine` operator:

```
Combine({ 'ab', 'cd', 'ef' })
```

The result of this expression is:

```
'abcdef'
```

To combine a list with a separator, provide the separator argument to the `Combine` operator:

```
Combine({ 'completed', 'refused', 'pending' }, ';')
```

The result of this expression is:

```
'completed;refused;pending'
```

To split a string into a list of strings based on a specific separator, use the `Split` operator:

```
Split('completed;refused;pending', ';')
```

The result of this expression is:

```
{ 'completed', 'refused', 'pending' }
```

Use the `Upper` and `Lower` operators to return strings with upper or lowercase letters for all characters in the argument.

3.8   Multi-Source Queries

In addition to the single-source queries discussed in the Author's Guide, CQL provides multi-source queries to allow for the simple expression of complex relationships between sets of data. Consider the following excerpt from the numerator of a measure for appropriate warfarin and parenteral anticoagulation overlap therapy:

- **Numerator =**
    - ○ Patients who received warfarin and parenteral anticoagulation:
        - ▪ Five or more days, with an INR greater than or equal to 2 prior to discontinuation of parenteral therapy
        - ▪ OR: Five or more days, with an INR less than 2 and discharged on overlap therapy
        - ▪ OR: Less than five days and discharged on overlap therapy

We begin by breaking this down into the source components, Encounters, Warfarin Therapy, and Parenteral Therapy:

```
define "Encounters": [Encounter: "Inpatient"] E
  where E.period during "Measurement Period"
define "Warfarin Therapy": [MedicationAdministration: "Warfarin"]
define "Parenteral Therapy": [MedicationAdministration: "Parenteral Anticoagulation"]
```

First, we establish that the encounter had both warfarin and parenteral anticoagulation therapies. This is easy enough to accomplish using `with` clauses:

```
define "Encounters with Warfarin and Parenteral Therapies":
  "Encounters" E
    with "Warfarin Therapy" W such that W.effectiveTime starts during E.period
    with "Parenteral Therapy" P such that P.effectiveTime starts during E.period
```

However, the next step involves calculating the duration of overlap between the warfarin and parenteral therapies, and a with clause only filters by a relationship, it does not introduce any data from the related source. To allow queries like this to be easily expressed, CQL allows a `from` clause to be used to start a query:

```
define "Encounters with Warfarin and Parenteral Therapies":
  from "Encounters" E,
    "Warfarin Therapy" W,
    "Parenteral Therapy" P
  where W.effectiveTime starts during E.period
    and P.effectiveTime starts during E.period
```

We now have both the encounter and the warfarin and parenteral therapies in context and can perform calculations involving all three:

```
define "Encounters with overlapping Warfarin and Parenteral Therapies":
  from "Encounters" E,
    "Warfarin Therapy" W,
```

```
    "Parenteral Therapy" P
  where W.effectiveTime starts during E.period
    and P.effectiveTime starts during E.period
    and duration in days of (W.effectiveTime intersect P.effectiveTime) >= 5
    and Last([Observation: "INR Value"] I
      where I.applies during P.effectiveTime sort by I.applies).value >= 2
```

This gives us the first condition, namely that a patient was on overlapping warfarin and parenteral therapies for at least 5 days, and the ending INR result associated with the parenteral therapy is greater than or equal to 2.

Next, we need to build criteria for the other cases, but these cases involve the same calculations, just compared against different values, or in different ways. Rather than having to restate the calculations multiple times, CQL allows a define clause to be used to introduce an intermediate comutational result within a query:

```
define "Encounters with overlapping Warfarin and Parenteral Therapies":
  from "Encounters" E,
    "Warfarin Therapy" W,
    "Parenteral Therapy" P
  define
    overlapDuration: duration in days of (W.effectiveTime intersect P.effectiveTime),
    endingINR:
      Last([Observation: "INR Value"] I
        where I.applies during P.effectiveTime sort by I.applies
      ).value
  where W.effectiveTime starts during E.period
    and P.effectiveTime starts during E.period
    and (
      (overlapDuration >= 5 and endingINR >= 2)
        or (overlapDuration >= 5 and endingINR < 2
              and P.effectiveTime overlaps after E.period)
        or (overlapDuration < 5
              and P.effectiveTime overlaps after E.period)
    )
  return E
```

Because the return clause in a query is optional, the type of the result of multi-source queries with no return clause is defined as a list of tuples with an element for each source named the alias for the source within the query and of the type of the elements of the source. For example:

```
from [Encounter] E, [MedicationStatement] M
```

The result type of this query is:

```
List<Tuple { E Encounter, M MedicationStatement }>
```

The result will be a list of tuples containing the cartesian product of all Encounters and Medication Statements.

In addition, the default for return clauses is distinct, as opposed to all, so if no return clause is specified, duplicates will be eliminated from the result.

3.9   Non-Retrieve Queries

In addition to the query examples already discussed, it is possible to use any arbitrary expression as the source for a query. For example:

```
({ 1, 2, 3, 4, 5 }) L return L * 2
```

This query results in { 2, 4, 6, 8, 10 }. Note that the parentheses are required for arbitrary expressions. A query source is either a retrieve, a qualified identifier, or a parenthesized expression.

The above example also illustrates that queries need not be based on lists of tuples. In fact, they need not be based on lists at all. The following example illustrates the use of a query to redefine a single tuple:

```
define FirstInpatientEncounter:
  First([Encounter] E where E.class = 'inpatient' sort by E.period.start desc)

define RedefinedEncounter:
  FirstInpatientEncounter E
    return Tuple {
      type: E.type,
      admissionDate: E.period.start
      dischargeDate: E.period.end
    }
```

In addition, even if a given query is based on a list of tuples, the results are not required to be tuples. For example, if only the length of stay is required, the following example could be used to return a list of integers representing the length of stay in days for each encounter:

```
[Encounter: "Inpatient"] E
  return duration in days of E.period
```

3.10   Defining Functions

CQL provides for the definition of functions. A function in CQL is a named expression that is allowed to take any number of arguments, each of which has a name and a declared type. For example:

```
define function CumulativeDuration(Intervals List<Interval<DateTime>>):
  Sum((collapse Intervals) X return duration in days of X)
```

This statement defines a function named CumulativeDuration that takes a single argument named Intervals of type List<Interval<DateTime>>. The function returns the sum of duration in days of the collapsed intervals given. This function can then be used just as any other system-defined function:

```
define Encounters: [Encounter: "Inpatient Visit"]
define CD: CumulativeDuration(Encounters E return E.period)
```

These statements establish an expression named CD that computes the cumulative duration of inpatient encounters for a patient.

Within the library in which it is defined, a function can be invoked directly by name. When a function is defined in a referenced library, the local library alias must be used to invoke the

function. For example, assuming a library with the above function definition and referenced with the local alias `Core`:

```
define Encounters: [Encounter: "Inpatient Visit"]
define CD: Core.CumulativeDuration(Encounters E return E.period)
```

In this example, the `CumulativeDuration` function must be invoked using the local library alias `Core`.

4 LOGICAL SPECIFICATION

This chapter describes the Expression Logical Model (ELM) and how it is used to represent clinical knowledge within a quality artifact.

The ELM defines a mechanism for representing artifact logic independent of syntax and special-purpose constructs introduced at the syntactic level. ELM is equivalent to CQL syntax in terms of expressive power: every possible expression in CQL has an equivalent canonical-form expression in ELM. Higher-level constructs such as timing phrases and implicit conversions are represented in terms of the more primitive operators in ELM. This takes the burden of interpretation of higher-level constructs off of implementers, allowing them to focus on the implementation of a more primitive set of functionality.

Expressions within ELM are represented as Abstract Syntax Trees. ELM defines the base *Expression* class, and all language elements and operators are then defined as descendants of the base *Expression*. For example, the *Add* class descends from *BinaryExpression*, which introduces two operands, each of type *Expression*. The *Literal* class descends from *Expression* and allows primitive-typed values such as strings and integers to be represented directly. Using these classes, the expression 2 + 2 can be represented as instances of the appropriate classes:
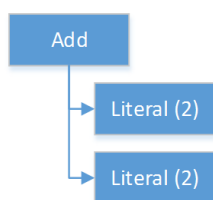
FIGURE 4–A

By combining instances of the appropriate classes of ELM, the logic for any expression can be represented. Note that the type of the expression can be inferred from the representation, Integer in this example.

The ELM consists of the following components:

- Expression – This component defines the core structures for representing expressions, as well as the operations available within those expressions.

- Clinical Expression – This component extends the Expression component to introduce expressions specific to the clinical quality domain.

- Library – This component defines the structure of a library, the container, and the basic unit of sharing.

Each of these components is defined fully within the ELM UML model. This model is defined formally as an XMI, and the model definition is also presented as an Enterprise Architect Project file (.eap) for viewing.

The documentation provided here serves only as a high-level structural reference for the ELM. The actual content of the specification is defined by the XMI file, and that provides the "source-of-truth" for the ELM specification.

Note that the semantics for the operations described here are defined both in the UML model as comments on the node for each operator, as well as the equivalent CQL operation as defined in Appendix B – CQL Reference.

4.1 Expressions

The ELM Expression component defines a mechanism for representing the structure of logic. The following table lists the core elements defined by the ELM:

Expression	Description
Expression	Abstract base class for all expressions in ELM
UnaryExpression	Abstract base class for unary expressions in ELM
BinaryExpression	Abstract base class for binary expressions in ELM
TernaryExpression	Abstract base class for ternary expressions in ELM
NaryExpression	Abstract base class for n-ary expressions in ELM

TABLE 4–A

Every expression in ELM is represented as a descendant of the abstract base element *Expression*. In addition, several abstract descendants are introduced to support the representation of unary, binary, ternary, and n-ary operators. Note that an expression need not descend from one of these descendants, it may descend from *Expression* directly.

4.2 Simple Values

Support for simple values is provided by the *Literal* class. This class defines properties to represent the type of the value, as well as the value itself.

The following table lists the simple value classes available in ELM:

Expression	Description
Literal	Represents simple value literals

TABLE 4–B

The *Literal* class is used to represent values for simple primitive types: Boolean, String, Integer, Decimal, DateTime, and Time.

The result type of a *Literal* is the type of the primitive being represented.

For more information on the primitive types, see the Types section in the CQL Reference.

4.3 Comparison Operators

ELM defines a standard set of comparison operators for use with simple values. Each comparison operator takes two arguments of the same type, and returns a boolean indicating the result of the comparison. Note that for comparison operators, if either or both operands evaluate to null, the result of the comparison is *null*, not false.

The following table lists the comparison operators available in ELM:

Expression	Description
Equal	Returns true if the operands are equal
Matches	Returns true if the operands match
NotEqual	Returns true if the operands are not equal
Less	Returns true if the first operand is less than the second operand
LessOrEqual	Returns true if the first operand is less than or equal to the second operand
Greater	Returns true if the first operand is greater than the second operand
GreaterOrEqual	Returns true if the first operand is greater than or equal to the second operand

<div align="center">TABLE 4–C</div>

The following example illustrates a simple *Equal* comparison:
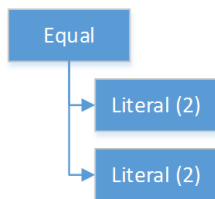
<div align="center">FIGURE 4–B</div>

For more information on the semantics of the various comparison operators, see the Comparison Operators section of the CQL Reference.

4.4   Logical Operators

ELM defines logical operators that can be used to combine the results of logical expressions. *And* and *Or* can be used to combine any number of results, and *Not* can be used to invert the result of any expression.

Note that these operators are defined with 3-valued logic semantics, allowing the operators to deal consistently with missing information.

The following table lists the logical operators available in ELM:

Expression	Description
And	Returns the logical conjunction of its operands
Or	Returns the logical disjunction of its operands
Not	Returns the logical negation of its operand
Xor	Returns the exclusive or of its operands

<div align="center">TABLE 4–D</div>

The following example illustrates a simple *And* expression:

May 2015
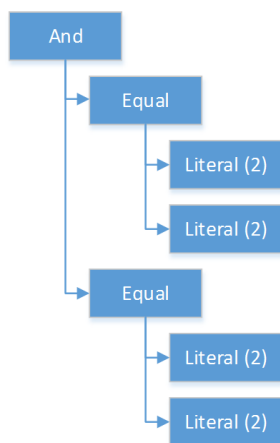
FIGURE 4–C

For more information on the semantics of these operators, refer to the Logical Operators section in the CQL Reference.

4.5   Nullological Operators

ELM defines several nullological operators that are useful for dealing with potentially missing information. These are *Null, IsNull*, *IsTrue*, *IsFalse*, and *Coalesce*.

The following table lists the logical operators available in ELM:

Expression	Description
Null	Returns a typed null
IsNull	Returns true if the argument is *null*, false otherwise
IsTrue	Returns true if the argument is *true*, false otherwise
IsFalse	Returns true if the argument is *false*, false otherwise
Coalesce	Returns the first non-null argument, null if there are no non-null arguments

TABLE 4–E

For more information on the semantics of these operators, refer to the Nullological Operators section in the CQL Reference.

4.6   Conditional Operators

ELM defines several conditional expressions that can be used to return different values based on a condition, or set of conditions. These are the *If* (conditional) expression, and the *Case* expression.

The conditional expression allows a simple condition to be used to decide between one expression or another.

The case expression has two varieties, one that is equivalent to repeated conditionals, and one that allows a specific comparand to be identified and compared with each item to determine a result.

The following table lists the conditional operators available in ELM:

Expression	Description
If	Allows for conditional evaluation between two expressions.
Case	Allows for multiple conditional expressions, or a comparand with multiple cases.

The following examples illustrates a simple *If* expression (i.e. if / then / else):
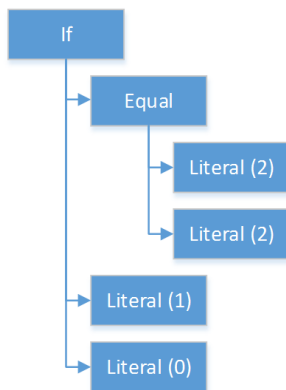
FIGURE 4–D

The following example illustrates a more complex multi-conditional *Case* expression:
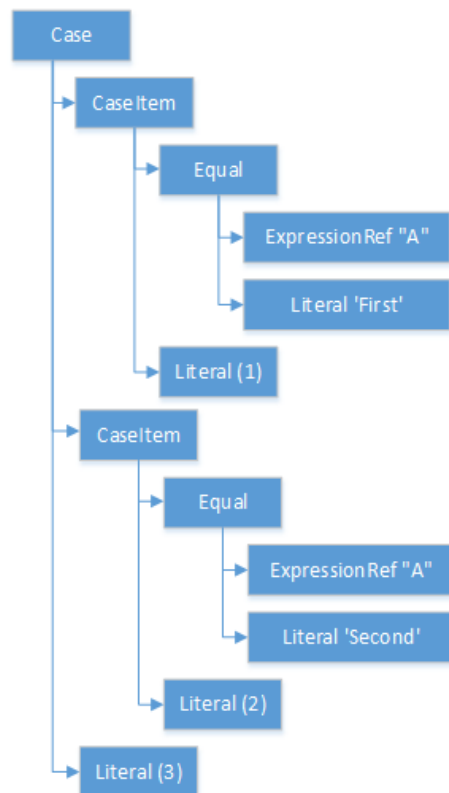
FIGURE 4–E

And finally, an equivalent comparand-based *Case* expression:
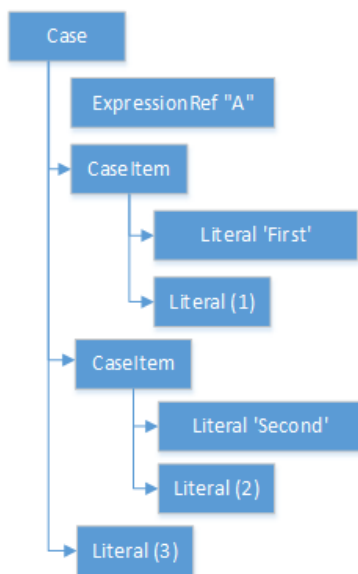
FIGURE 4–F

4.7   Arithmetic Operators

ELM provides a complete set of arithmetic operators to allow for manipulation of integer and real values within artifacts. In general, these operators have the expected semantics for arithmetic operators.

Note that if an operand evaluates to null, the result of the operation is defined to be null. This provides consistent semantics when dealing with missing information.

The following table lists the arithmetic operators available in ELM:

Expression	Description
Add	Performs numeric addition of its arguments
Subtract	Performs numeric subtraction of its arguments
Multiply	Performs numeric multiplication of its arguments
Divide	Performs numeric division of its arguments
TruncatedDivide	Performs integer division of its arguments
Modulo	Computes the remainder of the division of its arguments
Ceiling	Returns the first integer greater than or equal to its argument
Floor	Returns the first integer less than or equal to its argument
Truncate	Returns the integer component of its argument
Abs	Returns the absolute value of its argument
Negate	Returns the negative value of its argument
Round	Returns the nearest numeric value to its argument, optionally specified to a number of decimal places for rounding
Ln	Computes the natural logarithm of its argument
Log	Computes the logarithm of its first argument, using the second argument as

	the base
Power	Raises the first argument to the power given by the second argument
Successor	Returns the successor of its argument
Predecessor	Returns the predecessor of its argument
MinValue	Returns the minimum representable value for a type
MaxValue	Returns the maximum representable value for a type

<div align="center">TABLE 4–G</div>

The following example illustrates a simple *Add* expression:
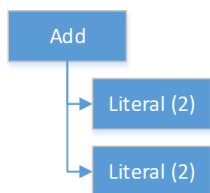
<div align="center">FIGURE 4–G</div>

For more information on the semantics of these operators, refer to the Arithmetic Operators section in the CQL Reference.

4.8 String Operators

ELM defines a set of string operators to allow for manipulation of string values within artifact definitions.

Indexes within strings are defined to be 1-based.

Note that except as noted within the documentation for each operator, if any argument evaluates to null, the result of the operation is also defined to be null.

The following table lists the string operators available in ELM:

Expression	Description
Concatenate	Returns the concatenation of its arguments
Combine	Combines a list of strings, optionally separating them with the given separator
Split	Splits a string into a list of strings along a given separator
Length	Returns the length of its argument
Upper	Returns the upper case representation of its argument
Lower	Returns the lower case representation of its argument
Indexer	Returns the nth character of its argument
PositionOf	Returns the starting position of a given pattern within a string
Substring	Returns a substring of its argument

<div align="center">TABLE 4–H</div>

For more information on the semantics of these operators, refer to the String Operators section in the CQL Reference.

HL7 Standard: Clinical Quality Language, Release 1 - US Realm
May 2015

4.9   Date and Time Operators

ELM defines several operators for representing the manipulation of date and time values. These operators are defined using a common precision type that allows the various precisions (e.g. day, month, week, hour, minute, second) of time to be manipulated.

Except as noted within the documentation for each operator, if any argument evaluates to null, the result of the operation is also defined to be null.

The following table lists the date and time operators available in ELM:

| Expression	Description
DateTimeComponentFrom	Returns a specified component of its argument
Today	Returns the date (with no time components specified) of the start timestamp associated with the evaluation request
Now	Returns the date and time of the start timestamp associated with the evaluation request
TimeOfDay	Returns the time-of-day of the start timestamp associated with the evaluation request
DateTime	Constructs a date/time value from its arguments
Time	Constructs a time value from its arguments
DateFrom	Returns the date (with no time component) of the argument
TimeFrom	Returns the time of the argument
TimezoneFrom	Returns the timezone offset (in hours) of the argument
SameAs	Performs precision-based equality comparison of two date/time values
SameOrBefore	Performs precision-based less-or-equal comparison of two date/time values
SameOrAfter	Performs precision-based greater-or-equal comparison of two date/time values
Before	Performs precision-based less-than comparison of two date/time values
After	Performs precision-based greater-than comparison of two date/time values
DurationBetween	Computes the number of whole periods between two dates
DifferenceBetween	Computes the number of whole period boundaries crossed between two dates

TABLE 4–I

For more information on the semantics of these operators, refer to the Date/Time Operators section in the CQL Reference.

4.10   Interval Operators

ELM defines a complete set of operators for use in defining and manipulating interval values.

Constructing an interval is performed with the *Interval* expression, which allows the beginning and ending of the interval to be specified, as well as whether the interval beginning and ending is exclusive (open), or inclusive (closed).

ELM defines support for basic operations on intervals including determining length, accessing interval properties, and determining interval boundaries.

ELM also supports complete operations involving comparisons of intervals, including equality, membership testing, and inclusion testing.

In addition, the language supports operators for combining and manipulating intervals.

The following table provides a complete listing of the interval operators available in ELM:

Expression	Description
Interval	Constructs a new interval value
Equal	Returns true if the arguments are the same interval
NotEqual	Returns true if the arguments are not the same interval
Matches	Returns true if the intervals match
Contains	Returns true if the interval contains the given point
In	Returns true if the given point is in the interval
Includes	Returns true if the first interval completely includes the second (i.e., starts on or before and ends on or after)
IncludedIn	Returns true if the first interval is completely included in the second (i.e., starts on or after and ends on or before)
ProperIncludes	Returns true if the first interval completely includes the second and the first interval is strictly larger (i.e., includes and not equal)
ProperIncludedIn	Returns true if the first interval is completely included in the second and the second interval is strictly larger (i.e., included in and not equal)
Before	Returns true if the first interval ends before the second one starts
After	Returns true if the first interval starts after the second one ends
Meets	Returns true if the first interval ends immediately before the second interval starts, or if the first interval starts immediately after the second interval ends
MeetsBefore	Returns true if the first interval ends immediately before the second interval starts
MeetsAfter	Returns true if the first interval starts immediately after the second interval ends
Overlaps	Returns true if the first interval overlaps the second
OverlapsBefore	Returns true if the first interval starts before and overlaps the second
OverlapsAfter	Returns true if the first interval ends after and overlaps the second
Union	Returns the interval that results from combining the arguments
Intersect	Returns the interval that results from the intersection of the arguments
Except	Returns the interval that results from subtracting the second interval from the first
Length	Returns the length of the interval
Start	Returns the starting point of the interval
End	Returns the ending point of the interval
Starts	Returns true if the first interval starts the second
Ends	Returns true if the first interval ends the second

| Expression	Description
Collapse	Returns the unique set of intervals that completely cover the range covered by the given intervals
Width	Returns the width of the interval

<div align="center">TABLE 4–J</div>

For more information on the semantics of these operators, refer to the Interval Operators section in the CQL Reference.

4.11 Structured Values

Structured values in ELM are values with sets of named elements (tuples), each of which may have a value of any type. Structured values are most commonly used to represent clinical information such as encounters, problems, and procedures.

The *Tuple* class represents construction of a new structured value, with the values for each element supplied by *TupleElement* instances.

To access elements of a structured value, use the *Property* expression. A property expression has a *path* attribute, an optional *source* element, and a *value* element. The source element returns the structured value to be accessed. In some contexts, such as within a *Filter* expression, the source is implicit. If used outside such a context, a source must be provided.

The path attribute specifies a property path relative to the source structured value. The property expression returns the value of the property specified by the property path. Property paths are allowed to include qualifiers as well as indexers to indicate that subelements should be traversed.

The following table lists the structured value operators available in ELM:

Expression	Description
Tuple	Constructs a new tuple value
Instance	Constructs a new instance of a structured value
Property	Returns the value of an element of a structured value
Equal	Returns true if its arguments are equal
NotEqual	Retruns true if its arguments are not equal
Matches	Returns true if its arguments match

<div align="center">TABLE 4–K</div>

The following example illustrates the construction of a tuple using the *Tuple* class:
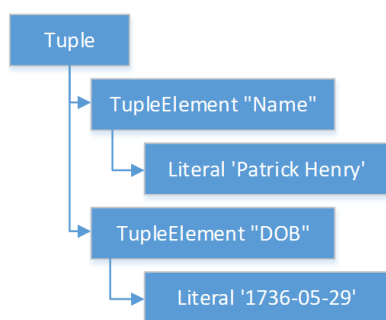
FIGURE 4–H

The following example illustrates the construction of a structured value using the *Instance* class:
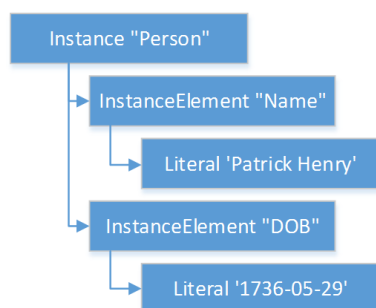
FIGURE 4–I

4.12 List Values

ELM allows for the expression and manipulation of lists of values of any type. The most basic list operation is the *List* class, which represents a simple list selector.

Basic list operations include testing for membership, indexing, and content. ELM also supports comparison of lists, including equality and inclusion determination (subset/superset). Supported operations on single lists include filtering, sorting, and computation. For multiple lists, ELM supports combining through union and intersection, as well as computing the difference.

The use of the scope attribute allows for more complex expressions such as correlated subqueries.

ELM also supports a flattening operator, *Expand* to construct a single list from a list of lists.

The following table provides a complete listing of the list operators available in ELM:

Expression	Description
List	Constructs a list from its arguments
Exists	Returns true if its argument contains any elements
Equal	Returns true if its arguments have the same number of elements, and for each element considered in order, the elements are equal
NotEqual	Returns true if its arguments are not equal
Matches	Returns true if its arguments match
Union	Returns a list containing all the elements of its arguments

Expression	Description
Except	Returns a list containing only the elements in the first list that are not in the second list
Intersect	Returns a list containing only the elements that are in all of its arguments
Times	Combines the elements from two lists, returning a list with an element for each possible combination of elements from the source list.
Filter	Returns a list containing only the elements for which the given condition evaluates to true
SingletonFrom	Extracts the single element from a list with at most one element.
IndexOf	Returns the 1-based index of an element within the list, or 0 if the element is not present
Indexer	Returns the element at the given 1-based index in the list
In	Returns true if the given element is in a given list
Contains	Returns true if the given list contains a given element
Includes	Returns true if every element in the second list is in the first list
IncludedIn	Returns true if every element in the first list is in the second list
ProperIncludes	Returns true if every element in the second list is in the first list, and the first list is strictly larger than the second
ProperIncludedIn	Returns true if the second list contains every element in the first list, and the second list is strictly larger than the first
Sort	Returns a list with the same elements, sorted by the given sort criteria
ForEach	Returns a list whose elements are determined by evaluating a given expression for each element in its argument
Expand	Flattens a list of lists into a single list with all the elements from every list in the input. Duplicates are not eliminated by this operation
Distinct	Returns a list that contains the unique elements within its argument
Current	Returns the contents of the current scope
First	Returns the first element in the given list
Last	Returns the last element in the given list

<div align="center">TABLE 4–L</div>

For more information on the semantics of these operators, refer to the List Operators section in the CQL Reference.

4.13 Aggregate Operators

For computing aggregate quantities, ELM defines several aggregate operators. These operators perform computations on lists of values, either on the elements of the list directly, or on a specific property of each element in the list.

Unless noted in the documentation for each operator, aggregate operators deal with missing information by excluding elements which have no value before performing the aggregation. In addition, an aggregate operation performed over an empty list is defined to return null, except as noted in the documentation for each operator (e.g. Count).

The following table lists the aggregate operators available in ELM:

Expression	Description
Count	Returns the number of non-null elements in the source
Sum	Computes the sum of non-null elements in the source
Min	Returns the minimum element in the source
Max	Returns the max element in the source
Avg	Returns the average of the elements in the source
Median	Returns the median of the elements in the source
Mode	Returns the mode of the elements in the source
Variance	Returns the statistical variance of the elements in the source
PopulationVariance	Returns the population variance of the elements in the source
StdDev	Returns the standard deviation of the elements in the source
PopulationStdDev	Returns the population standard deviation of the elements in the source
AllTrue	Returns true if all the non-null elements in source are true
AnyTrue	Returns true if any non-null element in source is true

<div align="center">TABLE 4–M</div>

For more information on the semantics of these operators, refer to the Aggregate Functions section in the CQL Reference.

4.14 Type Specifiers and Operators

ELM provides the following elements for type specifiers, testing, casting, and conversion:

Element	Description
Is	Returns true if the type of the argument is the given type
As	Returns the argument as the type if it is of the given type, null otherwise
Convert	Returns the argument converted to the given type, if possible. If no conversion is possible, a run-time error is thrown
NamedTypeSpecifier	Specifies a named type
IntervalTypeSpecifier	Specifies an interval type
ListTypeSpecifier	Specifies a list type
TupleTypeSpecifier	Specifies a tuple type

<div align="center">TABLE 4–N
FOR MORE INFORMATION ON THE SEMANTICS OF THESE OPERATORS, REFER TO THE TABLE 9–D</div>

Type Operators section in the CQL Reference.

4.15 Queries

ELM provides a mechanism for expressing the structure of a query using the following classes:

Class	Description
Query	Defines a query in ELM, containing clauses as defined by the other elements in this section.
AliasedQuerySource	The AliasedQuerySource element defines a single source for inclusion in the query context. The type of the source is determined by the expression element, and the source can be accessed within the query context by the given alias.
DefineClause	The DefineClause element allows any number of expression definitions to be introduced within a query context. Defined expressions can be referenced by name within the query context.
With	The With clause restricts the elements of a given source to only those elements that have elements in the related source that satisfy the suchThat condition. This operation is known as a semi-join in database languages.
Without	The Without clause restricts the elements of a given source to only those elements that do not have elements in the related source that satisfy the suchThat condition. This operation is known as a semi-difference in database languages.
SortClause	The SortClause element defines the sort order for the query, and is made up of any number of elements that are descendants of the SortByItem class (ByDirection, ByColumn, or ByExpression).
ByDirection	Indicates that the sort should be performed ascending or descending. This sortByItem can only appear by itself in a sort clause, and is used when the query is based on a list of non-tuple-valued elements.
ByColumn	Indicates that the sort should be performed based on the values of a specified column.
ByExpression	Indicates that the sort should be performed based on the result of an expression.
ReturnClause	The ReturnClause element defines the shape of the result set of the query.
AliasRef	Within a Query, references a defined alias
QueryDefineRef	Within a Query, references a defined value

<p style="text-align:center">**TABLE 4–O**</p>

For more information on query semantics, refer to the Queries section of the Author's Guide, as well as the Multi-Source Queries and Non-Retrieve Queries sections of the Developer's Guide.

4.16 Reusing Logic

ELM provides a mechanism for reusing expressions by declaring a named expression. This construct is similar to a function call with no parameters in a traditional imperative language, with the exception that since ELM is a pure-functional system, the result of the evaluation could be cached by an implementation to avoid performing the same computation multiple times.

In addition, ELM provides a more traditional function call with named parameters that can then be accessed by the expression in the function body, and passed as part of the call from the invoking context.

The *ExpressionDef* class is used to define a named expression that can then be referenced by other expressions. The *FunctionDef* class is used to define a function and its parameters.

Note that circular expression references are not allowed, but that named expressions can be defined in any order, so long as the actual references do not result in a cycle.

The following table lists the expression definition components available in ELM:

Expression	Description
ExpressionDef	Defines a named expression that can be referenced by other expressions
ExpressionRef	Returns the result of evaluating a named expression
FunctionDef	Defines a function that can be referenced by other expressions, or within the body of other functions.
FunctionRef	Returns the result of evaluating a function with the given arguments

TABLE 4–P

The *ExpressionDef* class introduces the notion of *context* which can be either Patient or Population. This context defines how the contained expression is evaluated, either with respect to a single patient, defined by the evaluation environment, or with respect to a population. For more information about patient context, please refer to the External Data section.

4.17 External Data

All access to external data within ELM is represented by *Retrieve* expressions.

The *Retrieve* class defines the data type of the request, which determines the type of elements to be returned. The result will always be a list of values of the type specified in the request.

The type of the elements to be returned is specified with the *dataType* attribute of the *Retrieve*, and must refer to the name of a type within a known data model specified in the *dataModels* element of the library definition.

In addition, the *Retrieve* introduces the ability to specify optional criteria for the request. The available criteria are intentionally restricted to the set of codes involved, and the date range involved. If these criteria are omitted, the request is interpreted to mean all data of that type.

Note that because every expression is being evaluated within a context (either Patient or Population) as defined by the containing *ExpressionDef*, the data returned by a retrieve depends on the context. For the Patient context, the data is returned for a single patient only, as defined by the evaluation environment. Whereas for the Population context, the data is returned for all patients.

The following table lists the expressions relevant to defining external data in ELM:

Expression	Description
Retrieve	Defines clinical data that will be used within the artifact

<div align="center">TABLE 4–Q</div>

4.18 Clinical Operators

For working with clinical data, ELM defines operators for terminology sets, quantities, and calculating age.

The following table lists the classes representing clinical information in ELM:

Class	Description
CodeSystemDef	Defines a code system identifier that can be referenced by name
CodeSystemRef	References a code system by its previously defined name
InCodeSystem	Tests a string, code, or concept for membership in a codesystem
ValueSetDef	Defines a valueset identifier that can be referenced by name
ValueSetRef	References a valueset by its previously defined name
InValueSet	Tests a string, code, or concept for membership in a valueset
Code	Selects an existing code from a defined codesystem
Concept	Selects an existing concept containing a list of codes
Quantity	Returns a clinical quantity with a specified unit
CalculateAge	Calculates the age in the specified precision of a person born on the given date as of today.
CalculateAgeAt	Calculates the age in the specified precision of a person born on the first date as of the second date.

<div align="center">TABLE 4–R</div>

4.19 Parameters

In addition to external data, ELM provides a mechanism for defining parameters to an artifact. A library can define any number of parameters, each of which has a name, and a defined type, as well as an optional default value.

Parameter values, if any, are expected to be provided as part of the evaluation request, and can be accessed with a *ParameterRef* expression in any expression throughout the library.

The following table lists the expressions relevant to parameters in ELM:

Expression	Description
ParameterDef	Defines a parameter to the artifact
ParameterRef	Returns the value of a parameter

<div align="center">TABLE 4–S</div>

4.20 Data Model

ELM does not reference any specific data model, and so can be used to represent logic expressed against any data model. These data models are specified using the *UsingDef* class. This class

provides attributes for specifying the name and version of the data model. An ELM library can reference any number of models.

The name of the model is an implementation-specific identifier that provides the environment with a mechanism for finding the model description. The details of how that model description is provided are part of the physical representation.

The following table lists the elements relevant to data models in ELM:

Element	Description
UsingDef	Defines a data model that can be used by expressions within the library

<p align="center">**TABLE 4–T**</p>

4.21 Libraries

ELM defines the notion of a library as the basic container for logic constructs. Libraries consist of sets of declarations including data model references, library references, valueset definitions, parameters, functions, and named expressions. The *Library* class defines this unit and defines properties for each of these types of declarations.

Once defined, libraries can then be referenced by other libraries with the *IncludeDef* class, which defines properties for the name and version of the library being referenced, as well as a local name that is used to access components of the library.

The following table lists the elements relevant to libraries in ELM:

Element	Description
IncludeDef	Defines a library reference; public components of the included library can be referenced by components of the referencing library.
VersionedIdentifier	Defines the versioned identifier construct used to label the various declarations throughout ELM

<p align="center">**TABLE 4–U**</p>

5 LANGUAGE SEMANTICS

This section contains more detailed information relating to the intended semantics of the Clinical Quality Language. These topics are specifically relevant for readers interested in building translation, semantic validation, or evaluation applications for CQL.

Note that the semantics are described here with reference to the representation defined by the ELM, but because CQL syntax is equivalent to ELM, the semantics apply to CQL as well.

5.1 Clinical Data Retrieval in Quality Artifacts

This section discusses the problem of clinical data retrieval in the clinical quality space in general, and how the problem is addressed in the CQL specification.

5.1.1 Defining Clinical Data

The problem of determining what data is required in the evaluation of an artifact containing arbitrary queries against the data model is equivalent to the problem of query containment from database theory. This problem is known to be undecidable for arbitrary queries of the relational algebra, but is also shown to be both decidable and equivalent to the problem of query evaluation for the restricted class of conjunctive queries (Foundations of Databases, Abiteboul, Hull, Vianu).

In the health quality domains of measurement and improvement, this problem is further complicated by the problem of terminology mapping. The meaning of a particular clinical statement within a patient's data is represented with a vocabulary consisting of codes which determine the type of statement being represented. For example, a diagnosis clinical statement may be classified using the ICD-9 vocabulary, identifying the specific diagnosis represented.

In order for health quality artifacts to operate correctly, the meaning of each clinical statement, as identified by the vocabularies involved, must be preserved. However, this meaning is often represented in different vocabularies in different systems. A mapping between the vocabularies is therefore required in order to facilitate expression and evaluation of the artifact.

In addition, patient data is represented in differing schemas across various patient data sources, and must therefore be mapped structurally into the patient data model used by an artifact.

These problems collectively constitute what is referred to as the "curly braces problem" in the Arden space. This problem arises because of the difficulty in defining the structural and semantic aspects of the data involved.

The solution to this problem proposed by the CQL specification is to create a well-defined and relatively simple interface between the clinical data provided by patient data sources, and the usage of that data within the artifact.

First, all clinical data within a CQL artifact is represented using a common, standard data model. This allows content to be authored without regard to the specific data models used by various patient data sources.

Second, all references to clinical data within a quality artifact are represented using a specific type of expression that only allows a well-defined set of clinically relevant criteria to be used to

HL7 Standard: Clinical Quality Language, Release 1 - US Realm
May 2015

reference the data. The purpose of this restriction is two-fold: First, it allows the data required for evaluation to be determined solely by inspection of the artifact. And second, it allows for easy and reliable implementation of the interface between the evaluation engine and the patient data source, because the criteria used to request information from the patient data source are simple and well-defined.

Third, by using standard terminologies within this data interface, the CQL specification can guarantee that any given clinical statement referenced in an artifact has the same meaning as the data that is provided to the artifact from the patient data source. At a high level, this is the terminology problem; ensuring that the vocabularies used within the artifact are accurately mapped to the vocabularies used by the patient data source.

These three motivating factors inform the design of the *retrieve* expression (*Retrieve* element in ELM) used within the CQL specification.

5.1.2 Conformance Levels

Even though CQL and ELM are intended to be used with a standard data model, there are many possibilities for variance in the way that data is provided, even within a particular model. This problem leads to the potential for artifacts to reference properties within the model that may or not be provided within a given specific instance of patient data. To address this potential problem, the retrieve elements within ELM specify not only the type of the data (meaning the specific model type being retrieved), but optionally a template, or profile identifier that further constrains the data that is expected in a given retrieve. If a template identifier is provided, then the retrieve expression is expected to return only data that matches the constraints in the given template.

For example, consider the following retrieve element:

```
<operand xsi:type="Retrieve"
   dataType="quick:Condition"
   templateId="qicore-condition"
   codeProperty="code">
 <codes xsi:type="ValueSetRef" name="Other Female Reproductive Conditions"/>
</operand>
```

In this example, the data type is specified as "quick:Condition", indicating that the result of the retrieve is a list of Condition instances. In addition, each instance must conform to the profile defined by the identifier "qicore-condition".

To help communicate validity of an artifact for a specific use, this specification defines two conformance levels related to this use of templates:

5.1.2.1 Strict Conformance

A quality artifact can be said to be strictly conforming if all references to clinical statement model properties (elements and attributes of model types) within the artifact are explicitly constrained by the templates used in the retrieves.

5.1.2.2 Loose Conformance

A quality artifact can be said to be loosely conforming if the artifact references properties that are not explicitly constrained by the templates used in the retrieves. This is not to say that the artifact

is necessarily invalid, just that the instances of clinical data provided to the retrieve may or may not contain the elements referenced by properties within the artifact.

5.1.3 Artifact Data Requirements

Because of the way data access is modeled within CQL, the data requirements of a particular artifact can be clearly and accurately defined by inspecting only the *Retrieve* expressions defined within the artifact. The following table broadly describes the data defined by each retrieve:

Item	Description
Clinical Data Type	The type of clinical data to be retrieved. This includes both the data type and the template identifier.
Codes	The set of codes defining the clinical data. Only clinical data with matching codes (based on the code path of the retrieve) in the set will be retrieved. If no codes are specified, clinical data with any code will be retrieved.
Date Range	The date range for clinical data. Only data within the specified date range (based on the date range path of the retrieve) will be retrieved. If no date range is specified, clinical data of any date will be retrieved.

<div align="center">TABLE 5–A</div>

These criteria are designed to allow the implementation environment to communicate the data requirements for an artifact, or group of artifacts, to a consumer to allow the consumer to gather all and only the relevant clinical information for transport to the evaluation environment. This supports the near-real-time clinical decision support scenario where the evaluation environment is potentially separate from the medical records system environment.

To support further reducing the overall size of data required to be transported, the following steps can be taken to combine retrieve descriptors that deal with the same type of clinical data.

First, create a retrieve context for each unique type of retrieve using the retrieve data type (and template identifier) for each retrieve. Note that if the determination here involves dynamic information, the retrieve is not considered "initial" and could result in additional data being requested by the engine in order to complete the evaluation. An implementation environment may opt to restrict artifacts to only those that contain statically determined data requirements.

Next, for each retrieve, add the codes to the matching retrieve context (by data type), recording the associated date range, if any, for each code. Note that the empty set of codes should be represented as the single code "ALL" for the purposes of this method. As date ranges are recorded, they must be merged so that for each code in each retrieve context, no two date range intervals overlap or meet.

Once the date ranges for each code within each unique retrieve context are determined, the unique set of date ranges for all codes is calculated, accumulating the set of associated codes. Each unique date range for the context then results in a final descriptor. As part of this process, the "ALL" placeholder code is replaced with the empty set of codes.

This process produces a set of clinical data descriptors with the following structure:

Property	Description
Clinical Data Type	The type of clinical data required (including template identifier)
Codes	The set of applicable codes, possibly empty (meaning all codes)
Date Range	The applicable date range, possibly empty (meaning all dates)

<div align="center">TABLE 5−B</div>

Collectively, these descriptors then represent the minimum initial data requirements for the artifact, with any overlapping requests for the same type of data collapsed into a single request descriptor.

Note that for the purposes of this method, the notion of the Clinical Data Type must be inclusive of the attributes used for filtering the codes and date ranges. For example, a retrieve of *Condition* data filtered by *code* must be considered separately from a retrieve of *Condition* data filtered by *severity*.

In addition to being used to describe the initial data requirements, this same process can be used to collapse additional data retrieves that are encountered as part of further evaluation of the artifact.

5.2 Expression Language Semantics

In order to completely specify the semantics of the expression logic defined by CQL, the intended execution model for expressions must be clearly defined. The following sections discuss the conceptual components of the expression language, and how these components are defined to operate.

5.2.1 Data Model

The data model for CQL provides the overall structure and definition for the types of operations and capabilities that can be represented within the language. Note that the schema itself is layered into a core expression schema, and a more specific, clinical expression schema. The expression schema deals with defining the core operations that are available without respect to any specific model. The clinical expression schema then extends those operations to include references to clinical data.

Note that although the expression language deals with various categories of types, these are only conceptually defined within the expression language schema. There is no expectation within the core expression language that any particular data model be used, only that whatever concrete data model is actually used can be concretely mapped to the type categories defined within CQL. Because these type categories are extremely broad, this allows the CQL expression language component to be used with a large class of concrete data models without modifying the underlying specification.

5.2.1.1 Values

A *value* within CQL represents some piece of data. All values are of some *type*, which designates what operations can be performed on the value. There are four categories of types within CQL:

1. Simple types – Types representing simple values such as strings, integers, dates, and decimals

2. Structured types – Types representing composite values consisting of sets of named properties, each of which has a declared type, that may or may not have a current value of that type.

3. Collection types – Types representing lists of values of some declared type

4. Interval types – Types representing an interval of some declared type, called the *point* type

5.2.1.2 Simple Types

Simple types allow for the representation of simple, atomic types, such as integers and strings. For example, the value **5** is a value of type *Integer*, meaning that it can be used in operations that require integer-valued input such as addition or comparison.

Note that because CQL defines a set of basic supported types, an implementation must map these types to the equivalent types in the selected data model. Ideally, this mapping would occur as part of the data access layer to isolate the mapping and minimize complexity.

5.2.1.3 Structured Types

Structured types allow for the representation of composite values. Typically, these types correspond to the model types defined in the clinical data model used for the artifact. Structured types are defined as containing a set of named properties, each of which are of some type, and may have a value of that type.

As with simple types, the core expression layer does not define any structured types, it only provides facilities for constructing values of structured types and for operating on structured values.

5.2.1.4 Collection Types

Collection types allow for the representation of lists and sets of values of any type. All the values within a collection are expected to be of the same type.

Collections may be empty, and are defined to be 1-based for indexing purposes.

5.2.1.5 Interval Types

Interval types allow for the representation of ranges over some point type. For example, an interval of integers allows the expression of the interval 1 to 5. Intervals can be open or closed at the beginning and/or end of the interval, and the beginning or end of the interval can be unspecified.

The core expression layer does not define any interval types, it only provides facilities for constructing values of interval types, and for operating on intervals.

5.2.2 Language Elements

The expression language represented by the ELM is defined as an Abstract Syntax Tree. Whereas a traditional language would have syntax and require lexical analysis and parsing, using the ELM exclusively allows expressions to be represented directly as trees. This removes potential ambiguities such as operator order precedence, and makes analysis and processing of the expressions in the language much easier.

Concretely, this is accomplished by defining the language elements as classes in a UML model. Each language element is represented by a type in the UML model. For example, the *Literal* class represents the appearance of a literal expression, and has attributes for specifying the type of the literal, as well as its actual value.

Arguments to operations are represented naturally using the hierarchical structure of the model. For example, the *Add* operator is represented as a *BinaryExpression* descendant, indicating that the operation takes two arguments, each of which is itself an expression.

This general structure allows expressions of arbitrary complexity to be built up using the language elements defined in the schema. Essentially, the language consists of only two kinds of elements: 1) Expressions, and 2) Expression Definitions (including Functions).

Each expression returns a value of some type, and an expression or function definition allows a given expression to be defined with an identifier so that it can be referenced in other expressions.

These expressions and expression definitions are then used throughout the CQL specification wherever logic needs to be defined within an artifact.

5.2.3 Semantic Validation

Semantic Validation of an expression within CQL is the process of verifying that the meaning of the expression is valid. This involves determining the type of each expression, and verifying that the arguments to each operation have the correct type.

This process proceeds as follows:

The graph of the expression being validated is traversed to determine the result type of each node. If the node has children (operands) the type of each child is determined in order to determine the type of the node. The following table defines the categories of nodes and the process for determining the type of each category:

Node Category	Type Determination
Literal	The type of the node is the type of the literal being represented.
Property	The type of the node is the declared type of the property being referenced.
ParameterRef	The type of the node is the parameterType of the parameter being referenced.
ExpressionRef	The type of the node is the type of the expression being referenced.
Retrieve	The type of the node is a list of the type of the data being requested.
FunctionRef/ Operator	Generally, the type of the node is determined by resolving the type of each operand, and then using that signature to determine the resulting type of the operator.
ValueSetRef	The type of the node is a list of codes.
Query	If the query has a return clause, the result is a list of the type of the return expression. Otherwise, the result type is determined by the source of the query.
AliasRef	The type of the node is the element type of the type of the query source referenced by the alias.
QueryDefineRef	The type of the node is the type of the referenced expression

defined within the query context.

TABLE 5–C

During validation, the implementation must maintain a stack of symbols that track the types of the objects currently in scope. This allows the type of context-sensitive operators such as Current and Property to be determined. Refer to the Execution Model (5.2.4) section for a description of the evaluation-time stack.

Details for the specifics of type determination for each operator are provided with the documentation for those operators.

5.2.4 Execution Model

All logic in CQL is represented as *expressions*. The language is pure functional, meaning no operations are allowed to have side effects of any kind. An expression may consist of any number of other expressions and operations, so long as they are all combined according to the semantic rules for each operation as described in the Semantic Validation (5.2.3) section.

Because the language is pure functional, every expression and operator is defined to return the same value on every evaluation within the same artifact evaluation. In particular this means:

1. All clinical data returned by request expressions within the artifact must return the same set on every evaluation. An implementation would likely use a snapshot of the required clinical data in order to achieve this behavior.

2. Invocations of non-deterministic operations such as Now() and Today() are defined to return the timestamp associated with the evaluation request, rather than the clock of the engine performing the evaluation.

Once an expression has been semantically validated, its return type is known. This means that the expression is guaranteed to return either a value of that type, or a *null*, indicating the evaluation did not result in a value.

In general, operations are defined to result in null if any of their arguments are null. For example, the result of evaluating 2 + null is null. In this way, missing information results in an unknown result. There are exceptions to this rule, notably the logical operators, and the null-handling operators. The behavior for these operators (and others that do not follow this rule) are described in detail in the documentation for each operator.

Evaluation takes place within an execution model that provides access to the data and parameters provided to the evaluation. Data is provided to the evaluation as a set of lists of structured values representing a patient's clinical information. In order to be represented in this data set, a given structured value must be a *cacheable* item. A cacheable item must have the following:

Property	Description
Identifier	A property or set of properties that uniquely identify the item
Codes	A code or list of codes that identify the associated clinical codes for the item
Date	A date time defining the clinically relevant date and/or time of the item

TABLE 5–D

Evaluation consists of two phases, a *pre-processing* phase, and an *evaluation* phase. The pre-processing phase is used to determine the initial data requirements for a rule. During this phase any retrieve expressions in the rule are analyzed to determine what data must be provided to the evaluation in order to successfully complete a rule evaluation. This set of data descriptors is produced using the method described in the Artifact Data Requirements (5.1.3) section. This means in particular that only retrieves whose Codes and DateRange expressions are compile-time evaluable should be considered to determine initial data requirements. This means that these expressions may not reference any clinical information, though they are allowed to reference parameter values.

During the evaluation phase, the result of the expression is determined. Conceptually, evaluation proceeds as follows:

The graph of the expression being evaluated is traversed and the result of each node is calculated. If the node has children (operands), the result of each child is evaluated before the result of the node can be determined. The following table describes the general categories of nodes and the process of evaluation for each:

Node Category	Evaluation
Literal	The result of the node is the value of the literal represented.
FunctionRef/Operation	The result of the node is the result of the operation described by the node given the results of the operand nodes of the expression.
Retrieve	The result of the node is the result of retrieving the data represented by the retrieve—i.e., a list of structured values of the type defined in the retrieve representing the patient information being retrieved.
ExpressionRef	The result of the node is the result of evaluating the referenced expression.
ParameterRef	The result of the node is the value of the referenced parameter.
ValueSetRef	The result of the node is the expansion set of the referenced value set definition. Note that in the case of the InValueSet operator specifically, the expansion set need not be materialized; the membership test can be passed to a terminology service using only the valueset definition information.

TABLE 5–E

During evaluation, the implementation must maintain a stack that is used to represent the value that is currently in context. Certain operations within the expression language are defined with a scope, and these operations use the stack to represent this scope. The following table details these operations:

Operation	Stack Effect
Query	Query evaluation is discussed in detail below.
Filter	For each item in the *source* operand, the item is pushed on to the stack, the *condition* expression is evaluated, and the item is popped off of the stack.
ForEach	For each item in the *source* operand, the item is pushed on to the stack, the *element* expression is evaluated, and the item is popped off of the stack.

TABLE 5–F

The *scope* attribute of these operators provides an optional name for the item being pushed on to the stack. This name can be used within the **Current** and **Property** expressions to determine which element on the stack is being accessed. If no scope is provided, the top of the stack is assumed.

Details for the evaluation behavior of each specific operator are provided as part of the documentation for each operator.

5.3 Query Evaluation

In general, query evaluation can be performed in many different ways, especially when queries involve large numbers of sources. Rather than address the many ways queries could be evaluated, the intent of this section is to describe the expected semantics for query evaluation, regardless of how the underlying implementation actually executes any given query.

The outline of the process is:

- Evaluate the sources
- For each item in the source
    - evaluate any defines within the query
    - evaluate each with or without clause in the query
    - evaluate the where clause, if present
    - evaluate the return clause
- Sort the results if a sort clause is present

The following sections discuss each of these steps in more detail.

5.3.1 Evaluate Sources

The first step in evaluation of a given query is to establish the query sources. Conceptually, this step involves generating the cartesian product of all the sources involved. In a single-source query, this is simply the source. But for a multi-source query, the evaluation needs to be performed for every possible combination of the sources involved.

How this actually occurs is up to the specific implementation, but note that the evaluation must still be able to reference components originating from each individual source using the alias for the source defined in the query. A simple solution to allowing this is to define the query source internally as a list of tuples, each with an element for each source whose value is the tuple from that source. This list is then simply populated with the cartesian product of all sources, and alias access within the rest of the query can be implemented as tuple-element access.

5.3.2 Iteration

Once the source for the query has been established, the iterative clauses must be evaluated for each element of the source, in order, as described in the following sections.

5.3.2.1 Define Clause

The define clause, if present, allows a CQL author to introduce expression definitions scoped to the query context. For each definition specified in the define clause, the result of the expression is evaluated and made available within the query context such that subsequent clauses can access the value. Note that an implementation may opt for lazy evaluation, saving the cost of evaluating an expression that is never actually referenced.

5.3.2.2 With Clause

Each with clause present in the query acts as a filter to remove items from the result if they do not satisfy the conditions of the with clause. Evaluation proceeds by introducing the related source into the query context and evaluating the "such that" condition of the with clause for each element of the introduced source. If no element of the introduced source satisfies the such that condition, the current row of the query source is filtered out of the result.

Note that because this is a positive existence condition, the test can stop after the first positive result. Only in the case of a negative result would all the elements of the introduced source need to be processed.

5.3.2.3 Without Clause

Each without clause present in the query acts as a filter to remove items from the result if they satisfy the conditions of the without clause. This is the opposite of the with clause. Evaluation proceeds the same way as a with clause, except that an element from the query source will only pass the filter if there a no rows from the introduced source that satisfy the conditions of the without clause.

5.3.2.4 Where Clause

The where clause, if present simply determines whether each element should be included in the result. If the condition evaluates to true, the element is included. Otherwise, the element is excluded from the result.

5.3.2.5 Return Clause

The return clause, if present, defines the final shape of each element produced by the query, as well as whether or not to eliminate duplicates from the result. If distinct is specified as part of the return clause, any duplicates must not appear in the result set. The expression defined in the return clause is evaluated and the result is added to the output. If neither all or distinct is specified, distinct is the default behavior.

5.3.3 Sort

After the iterative clauses are executed for each element of the query source, the sort clause, if present, specifies a sort order for the final output. This step simply involves sorting the output of the iterative steps by the conditions defined in the sort clause. This may involve sorting by a particular element of the result tuples, or it may simply involve sorting the resulting list by the defined comparison for the data type (for example, if the result of the query is simply a list of integers).

5.3.4 Implementing Query Evaluation

It is worth noting that the implementation of query evaluation can be simplified by decomposing the query into a set of more primitive operations. For example, the following operations are sufficient to evaluate any query of CQL:

- ForEach
- Times
- Filter
- Distinct
- Sort

The following sketch details an implementation plan for any query using these primitives:

1. For each query source beyond the first, use a Times operation to produce a result with a tuple for each combination, named the same as the alias used to introduce the source in the query.

2. If the define clause is present, use a ForEach operation to introduce a tuple element for each defined expression.

3. For each with clause, use a Filter and express the with in terms of an Exists in the condition of the Filter.

4. For each without clause, use a Filter and express the without in terms of a Not Exists in the condition of the Filter.

5. If the return clause is specified, use a ForEach to produce the result of the return. If the return clause specifies Distinct, also attach a Distinct operation to the result.

6. If the sort clause is specified, use a Sort operation to produce the final sorted output.

Using this sketch, the evaluation of a query can be performed by pipelining the query into a series of more primitive operations that can be implemented more easily. This approach also lends itself to translation and/or optimization if necessary.

5.4 Timing Calculations

This section discusses the precise semantics for the representation of date/time values within CQL, as well as the calculation of date/time arithmetic. The discussion in this section assumes fully-specified date/time values. The next section will discuss the implications of partially-specified date/time values.

5.4.1 Definitions

This section provides precise definitions for the terms involved in dealing with date/time values. These definitions are based on the ISO 8601:2004 standard for the representation of date/time values.

Term	Definition	Notes
DateTime interval	Part of the time axis bounded by two DateTime values.	A DateTime interval comprises all DateTime values between the two

		boundary DateTimes and, unless otherwise stated, the boundary DateTime values themselves.
Duration	Quantity attributed to a DateTime interval, the value of which is equal to the difference between the time points of the final instant and the initial instants of the time interval.	In case of discontinuities in the time scale, such as a leap second or the change from winter time to summer time and back, the computation of the duration requires the subtraction or addition of the change of duration of the discontinuity.
Nominal duration	Duration expressed in years, months, or days.	The duration of a calendar year, a calendar month, or a calendar day depends on its position in the calendar. Therefore, the exact duration of a nominal duration can only be evaluated if the duration of the calendar years, calendar months, or calendar days used is known.
Second	Base unit of measurement of time in the SI as defined by the International Committee of Weights and Measures.	
Millisecond	Unit of time equal to 0.001 seconds.	
Minute	Unit of time equal to 60 seconds.	
Hour	Unit of time equal to 60 minutes.	
Day	Unit of time equal to 24 hours.	
Calendar day	Time interval starting at midnight and ending at the next midnight, the latter being also the starting instant of the next calendar day.	A calendar day is often also referred to as a day. The duration of a calendar day is 24 hours, except if modified by: - The insertion or deletion of leap seconds, by decision of the International Earth Rotation Service (IERS), or - The insertion or deletion of other time intervals, as may be prescribed by local authorities to alter the time scale of local time.
Day	Duration of a calendar day.	The term "day" applies also to the duration of any time interval which starts at a certain time of day at a certain calendar day and ends at the same time of day at the next calendar day.
Calendar month	Time interval resulting from the division of a calendar year into 12 time intervals, each with a specific name and containing a specific number of calendar days.	A calendar month is often referred to as a month.
Month	Duration of 28, 29, 30, or 31 calendar days, depending on the start and/or the end of the	The term "month" applies also to the duration of any time interval which

	corresponding time interval within the specific calendar month.	starts at a certain time of day at a certain calendar day of the calendar month and ends at the same time of day at the same calendar day of the next calendar month, if it exists. In other cases, the ending calendar day has to be agreed on.
Calendar year	Cyclic time interval in a calendar which is required for one revolution of the Earth around the Sun and approximated to an integral number of calendar days.	A calendar year is also referred to as a year. Unless otherwise specified, the term designates a calendar year in the Gregorian calendar.
Year	Duration of 365 or 366 calendar days depending on the start and/or the end of the corresponding time interval within the specific calendar year.	The term "year" applies also to the duration of any time interval which starts at a certain time of day at a certain calendar date of the calendar year and ends at the same time of day at the same calendar date of the next calendar year, if it exists. In other cases, the ending calendar day has to be agreed on.
Common year	Calendar year in the Gregorian calendar that has 365 calendar days.	
Leap year	Calendar year in the Gregorian calendar that has 366 calendar days.	

<p style="text-align:center">TABLE 5–G</p>

ISO 8601 postulates that duration can be expressed by a combination of components with accurate duration (hour, minute, and second) and components with nominal duration (year, month, week, and day). The standard allows for the omission of lower-level components for "reduced accuracy" applications. Following this guidance, CQL represents date/time values using the following components:

Component	Type	Range	Notes
Year	Integer	[0001, 9999]	A CQL environment must be able to represent the minimum year of 0001, and a maximum year of 9999. Environments may represent dates in years before or after these years, the range specified here is the minimum required.
Month	Integer	[1, 12]	Months are specified by their ordinal position (i.e. January = 1, February = 2, etc.)
Day	Integer	[1, 31]	If the day specified is not present in the month (i.e. February 30th), the day value is reduced by the number of days in the given month, and the month is incremented by 1.
Hour	Integer	[0, 23]	
Minute	Integer	[0, 59]	
Second	Integer	[0, 59]	
Millisecond	Integer	[0, 999]	999 milliseconds is the maximum required

HL7 Standard: Clinical Quality Language, Release 1 - US Realm
May 2015

			precision. Note that many operations require the ability to compute the "next" or "prior" instant, and these semantics depend on the step-size of 1 millisecond, so systems that support more than millisecond precision will need to quantize to the millisecond to achieve these semantics.
Timezone Offset	Real	[-12.00, 14.00]	The timezone offset is represented as a real with two digits of precision to account for timezones with partial hour differences. Note that the timezone offset is a decimal representation of the time offset, so an offset of +2:30 would be represented as +2.50.

<div align="center">

TABLE 5–H

</div>

5.4.2 Date/Time Arithmetic

CQL allows time durations, represented as Quantities, to be added to or subtracted from date/time values. The result of these operations take the calendar into account when determining the correct answer. In general, when the addition of a quantity exceeds the limit for that precision, it results in a corresponding increase in the next higher precision. The following table describes these operations for each precision:

Precision	Type	Range	Semantics
Year	Integer	[0001, 9999]	The year, positive or negative, is added to the year component of the date/time value. If the resulting year is out of range, an error is thrown. If the month and day of the date/time value is not a valid date in the resulting year, the last day of the calendar month is used. For example, DateTime(2012, 2, 29) + 1 year = DateTime(2013, 2, 28). The resulting date/time value will have the same time components.
Month	Integer	[1, 12]	The month, positive or negative is divided by 12, and the integer portion of the result is added to the year component. The remaining portion of months is added to the month component. If the resulting date is not a valid date in the resulting year, the last day of the resulting calendar month is used. The resulting date/time value will have the same time components.
Day	Integer	[1, 31]	The days, positive or negative, are added to the day component, respecting calendar month and calendar year lengths. The resulting date/time value will have the same time components.
Hour	Integer	[0, 23]	The hours, positive or negative, are added to the hour component, with each 24 hour block counting as a calendar day, and respecting calendar month and calendar year lengths.
Minute	Integer	[0, 59]	The minutes, positive or negative, are added to the minute component, with each 60 minute block counting as an hour, and respecting

			calendar month and calendar year lengths.
Second	Integer	[0, 59]	The seconds, positive or negative, are added to the second component, with each 60 second block counting as a minute, and respecting calendar month and calendar year lengths.
Millisecond	Integer	[0, 999]	The milliseconds, positive or negative, are added to the millisecond component, with each 1000 millisecond block counting as a second, and respecting calendar month and calendar year lengths.

<center>TABLE 5-1</center>

5.5 Precision-Based Timing

One of the most complex aspects of quality expression logic is dealing with timing relationships in the presence of partially-specified date/time values. This section discusses the precise semantics used by CQL to help mitigate this complexity and allow measure and decision support authors to express temporal logic intuitively and accurately, even in the presence of uncertain date/time data.

The core issue being addressed is the proper handling of temporal comparisons in the presence of varying degrees of certainty about the time at which events occur. For example, if a measure is looking for the occurrence of a particular procedure within two years of the measurement start date, but an EHR records that a qualifying procedure occurred in a given year, not the month or day of the occurrence. In this scenario, the EHR must be allowed to provide as much information as it accurately has, but must not be required to provide information that is not known. This requirement means that the record will contain a date/time value, but specified only to the year precision. If the semantics for timing comparison do not take this possibility into account, the resulting comparisons may yield incorrect results.

In general, the approach taken by CQL formally defines the notion of *uncertainty* to specify the semantics for date/time comparisons, and all the operations that rely on them. Note that the concept of uncertainty is not exposed directly in CQL or in ELM, but is defined as an implementation detail. This approach is deliberate and is taken to achieve the intuitively correct semantics without exposing the complexity involved to CQL authors and developers.

The discussion here begins by formally defining uncertainty and the semantics of operations involving uncertainty. The calculation of duration between imprecise dates is then discussed in terms of uncertainty, and then the CQL timing phrases are all defined in terms of either date/time comparison, or duration calculation. The discussion concludes with some notes on implementation of these semantics within an engine or translated environment.

5.5.1 Uncertainty

Formally, an *uncertainty* is a closed interval over a given point type, with specific semantics defined for comparison operators. For simplicity, we use the point type Integer in the discussion that follows.

Intuitively, an uncertainty between X and Y means *some value between X and Y*. For example:

```
uncertainty[1, 10]
```

This uncertainty means *some value between 1 and 10*. Note that this representation of uncertainty assumes a continuous probability distribution along the range. In other words, the assumption is that there is no information about how likely the value is to be any particular value within the range.

Note that the special case of an uncertainty of width zero:

```
uncertainty[1, 1]
```

Must be treated as equivalent to the point value, 1 in this case.

5.5.1.1 Comparison Operators

Comparison semantics for uncertainty are defined to result in the intuitively expected behavior. For example, when comparing two uncertainties for equality:

```
uncertainty[1, 10] = uncertainty[1, 10]
```

The above expression results in *null*, because the meaning of the statement is actually:

Is *some value between 1 and 10* equal to *some value between 1 and 10*?

And the intuitively correct answer to that question is, *I don't know*. However, for cases where there is no overlap between the uncertainties, the result is *false*:

```
uncertainty[1, 10] = uncertainty[21, 30]
```

Again, the intended semantics of this statement are:

Is *some value between 1 and 10* equal to *some value between 21 and 30*?

And the correct answer is, *No*, because there is no possible value in either uncertainty range that could evaluate to *true*.

In the special case of equality comparisons of two uncertainties of width zero, the result is true:

```
uncertainty[2, 2] = uncertainty[2, 2]
```

This expression can be read:

Is *some value between 2 and 2* equal to *some value between 2 and 2*?

And the correct answer is, *Yes*.

More precisely, given an uncertainty A with range A_{low} to A_{high}, and uncertainty B with range B_{low} to B_{high}, the comparison:

```
A = B
```

Is equivalent to:

```
if A_low <= B_high and A_high >= B_low
  then if A_low = A_high and B_low = B_high
    then true
    else null
  else false
```

For relative comparisons, again, the semantics are defined to give the intuitively correct answer given the intended meaning of uncertainty. For example:

```
uncertainty[30, 40] < uncertainty[50, 60]
```

This expression can be read:

Is some value between 30 and 40 less than some value between 50 and 60?

And the correct answer is, *Yes*. If the ranges overlap:

```
uncertainty[30, 40] < uncertainty[35, 45]
```

Then the result is *null*, with one exception having to do with boundaries. Consider the following:

```
uncertainty[30, 40] < uncertainty[20, 30]
```

This expression can be read:

Is some value between 30 and 40 less than some value between 20 and 30?

And the correct answer is, *No*, because even though the ranges overlap (by width one at the lower boundary of the left-hand value), the result would still be false because 30 is not less than 30.

More precisely, given an uncertainty A with range A_{low} to A_{high}, and uncertainty B with range B_{low} to B_{high}, the comparison:

```
A < B
```

Is equivalent to:

```
case
  when Ahigh < Blow then true
  when Alow >= Bhigh then false
  else null
end
```

And finally, for relative comparisons involving equality, consider the following:

```
uncertainty[30, 40] <= uncertainty[40, 50]
```

This expression can be read:

Is some value between 30 and 40 less than or equal to some value between 40 and 50?

And the correct answer is, *Yes*, because every possible value between 30 and 40 inclusive is either less than or equal to every possible value between 40 and 50 inclusive.

More precisely, given an uncertainty A with range A_{low} to A_{high}, and uncertainty B with range B_{low} to B_{high}, the comparison:

```
A <= B
```

Is equivalent to:

```
case
  when Ahigh <= Blow then true
  when Alow > Bhigh then false
  else null
end
```

Note carefully that these semantics introduce some asymmetries into the comparison operators. In particular, *A = B or A < B* is *not* equivalent to *A <= B* because of the uncertainty.

5.5.1.2 Arithmetic Operators

In addition to comparison operators, the basic arithmetic operators are defined for uncertainty, again based on the intuitively expected semantics. For example:

```
uncertainty[17, 44] + uncertainty[5, 10] // returns uncertainty[22, 54]
```

The above expression can be read:

some value between 17 and 44 + some value between 5 and 10

The result of this calculation simply adds the respective boundaries to determine what the range of possible values of this calculation would be, in this case *some value between 22 and 54*.

Similarly for multiplication:

```
uncertainty[17, 44] * uncertainty[2, 4] // returns uncertainty[34, 176]
```

The result of this calculation multiplies the boundaries of the uncertainties to determine the range of possible values for the result, in this case *some value between 34 and 176*.

5.5.1.3 Implicit Conversion

An important step to achieving the intended semantics for precision-based timing comparisons in CQL is to allow for implicit conversion between uncertainties and point-values. This means that anywhere an uncertainty is involved in an operation with a point-value, the point-value will be implicitly converted to an uncertainty of width zero and the uncertainty semantics defined above are then used to perform the calculation. For example:

```
uncertainty[17, 44] > 2
```

The point-value of *2* in this example is implicitly converted to an uncertainty of width zero:

```
uncertainty[17, 44] > uncertainty[2, 2]
```

This implicit conversion means that in general, the notion of uncertainty will not be visible in the resulting syntax of CQL. For example:

```
days between DateTime(2014, 1, 15) and DateTime(2014, 2) > 2
```

Even though determining the correct answer to this question involves the use of uncertainty, it is implicit in the way the operations are defined, and does not surface to the CQL authors.

5.5.2 Determining Difference and Duration

To determine the duration between two date/time values, CQL supports a *between* operator for each date/time component. For example:

```
days between A and B
```

This expression returns the number of whole days between A and B. If A is before B, the result will be a positive integer. If A is after B, the result will be a negative integer. And if A is the same day as B, the result will be zero.

However, to support the case where one or the other comparand in the duration operation does not specify components to the level of precision being determined, the between operator does not return a strict integer, it returns an *uncertainty*, which is defined as a range of values, similar to an interval. For example:

```
days between DateTime(2014, 1, 15) and DateTime(2014, 2)
```

The number of days between these two dates cannot be determined reliably, but a definite range of possible values can be determined. The lower bound of that range is found by determining the duration between the maximum possible value of the first comparand and the minimum possible value of the second comparand; and the upper bound is determined using the minimum possible value of the first comparand and the maximum possible value of the second:

```
days between DateTime(2014, 1, 15) and DateTime(2014, 2, 1) // 17 days
days between DateTime(2014, 1, 15) and DateTime(2014, 2, 28) // 44 days
```

Intuitively, what this means is that the number of days between January 15th, 2014 and some date in February, 2014, is no less than 17 days, but no more than 44. By incorporating this information into an uncertainty, CQL can support the intuitively expected semantics when performing timing comparisons. For example:

```
days between DateTime(2014, 1, 15) and DateTime(2014, 2) > 2
```

This comparison returns true, because the lower bound of the uncertainty, 17, is greater than 2, so no matter what the actual date of the second comparand, it would always be at least 17 days. By contrast:

```
days between DateTime(2014, 1, 15) and DateTime(2014, 2) > 50
```

This comparison returns false, because the upper bound of the uncertainty, 44, is less than 50, so no matter what the actual date of the second comparand, it would always be at most 44 days. And finally:

```
days between DateTime(2014, 1, 15) and DateTime(2014, 2) > 20
```

This comparison returns unknown (null), because the value being compared, 20, falls within the uncertainty, so no determination can be reliably made.

CQL also supports a difference in operator which, rather than calculating the number of calendar periods between two dates, calculates the number of boundaries crossed between the two dates. As with the duration operator, difference is defined to take imprecision in date/time values into account by returning an uncertainty.

5.5.3 Timing Phrases

Using the foundational elements described in the previous sections, the semantics for the various CQL timing phrases can now be described in detail. The general approach for each timing phrase is to transform it to an equivalent representation in terms of either a direct comparison, or a comparison involving a duration calculation.

5.5.3.1 Same As

The *same as* timing phrase is simply defined to be equivalent to a *same as* comparison of the date/time values involved:

```
A starts same day as start B
```

This expression is equivalent to:

```
start of A same day as start of B
```

Similarly for the *or after* and *or before* comparisons:

```
A starts same day or after start B
A starts same day or before start B
```

These expressions are equivalent to:

```
start of A same day or after start of B
start of A same day or before start of B
```

5.5.3.2 Before/After

The basic *before* and *after* timing phrases are defined to be equivalent to a *before* or *after* comparison of the date/time values involved:

```
A starts before start B
A starts after start B
```

These expressions are equivalent to:

```
start of A before start of B
start of A after start of B
```

If the phrase involves a duration, it is defined in terms of a duration calculation and a comparison of the result:

```
A starts 3 days before start B
A starts 3 days after start B
```

These expressions are equivalent to:

```
days between start of A and start of B = 3
days between start of A and start of B = -3
```

For timing phrases involving relative comparison, the suffixes *or more* and *or less* can be used:

```
A starts 3 days or more before start B
A starts 3 days or less after start B
```

These expressions are equivalent to:

```
days between start of A and start of B >= 3
days between start of A and start of B in [-3, 0)
```

5.5.3.3 Within

The *within* timing phrase is defined in terms of a duration calculation and an interval test of the result:

```
A starts within 3 days of start B
```

This expression is equivalent to:

```
days between start of A and start of B in [-3, 3]
```

5.5.3.4 Interval Operators

In general, interval comparisons are already defined in terms of the fundamental comparison operators (=, >, <, >=, <=, and the precision-based counterparts) so the semantics of the interval comparisons follow directly from these extended semantics.

5.5.4 Implementing Precision-Based Timing with Uncertainty

Implementation of these semantics can be simplified by recognizing that all the date/time comparisons can be expressed in terms of a difference calculation and a comparison of the resulting (potentially uncertain) values against 0. Combined with the timing phrase translations, this means that the implementation for precision-based timing can be isolated to:

- Support for run-time operations on integer-based uncertainties, including:
  - =, <, >, <=, >=, +, -, unary +/-, *, /
  - implicit conversion between integer point values and uncertainties
- Precision-based duration and difference between date/times

All the other operations and semantics can be achieved using only these primitives. For example, given *A* and *B*, both date/time values, the comparison:

```
A > B
```

Can be evaluated as:

```
difference in milliseconds between A and B > 0
```

Similarly:

```
A same day as B
```

Can be evaluated as:

```
difference in days between A and B = 0
```

Because the difference operation will return an uncertainty when imprecise date/time values are involved, the correct semantics will be applied to the comparison to the point value, 0 in this case. By structuring the evaluation engine such that all operations involving date/times are performed in terms of these primitives, correct semantics can be achieved with a comparatively straightforward implementation.

Note also that a compile-time implicit conversion to uncertainty may also simplify the implementation, avoiding the need for integer-valued primitives to determine at run-time whether they are operating on an uncertainty.

6 TRANSLATION SEMANTICS

As discussed in the introductory section, this specification covers three levels of definition, the Conceptual or Author level, the Logical level, and the Physical level. The Conceptual level is concerned with the representation of logic in a format suitable for authoring and consumption by clinical experts; the Physical level is concerned with the representation of logic in a format suitable for processing and transferring by machines; and the Logical level is concerned with providing a mapping between the Conceptual and Physical levels in a way that preserves the semantics of the logic represented while also enabling integration and execution functionality.

To achieve these goals, the Logical level establishes a semantically complete bi-directional mapping between the Conceptual and Physical levels. This chapter describes this mapping in more detail, and sketches a process for translation from the Conceptual to the Logical, and from the Logical to the Conceptual. The Physical level is an isomorphic concrete realization of the Logical level; translation between the Logical and Physical levels is therefore a matter of serialization and realization of the data model, and is covered in detail in the **Error! Not a valid bookmark self-reference.** chapter.

6.1 CQL-to-ELM

Every statement of CQL has a semantically equivalent representation in ELM. As such, it is possible to programmatically translate any statement of CQL into its equivalent ELM representation. The following sections define the mappings between the language elements of CQL and their equivalent ELM representations, as well as providing a sketch for how these mappings could be used to translate from CQL to ELM.

6.1.1 Declarations

In both CQL and ELM, the basic container for all declarations is the *Library*. In CQL, a library corresponds to a single source document, usually represented as a text file. In ELM, a library is represented as a single instance of the *Library* class which contains all the declarations for the library.

The identifier and version of the library are set as part of the library metadata.

The following table specifies the ELM equivalent for each CQL declaration:

CQL Declaration	ELM Equivalent
library	Library
using	UsingDef
include	IncludeDef
codesystem	CodeSystemDef
valueset	ValueSetDef
parameter	ParameterDef
define	ExpressionDef
function	FunctionDef

TABLE 6–A

6.1.2 Types

To represent types, CQL uses the *type-specifier* construct. In ELM, an equivalent TypeSpecifier abstract class is defined, with appropriate subclasses to represent the various types of specifiers, as detailed in the following table:

CQL Specifier	ELM Equivalent
named-type-specifier	NamedTypeSpecifier
interval-type-specifier	IntervalTypeSpecifier
list-type-specifier	ListTypeSpecifier
tupe-type-specifier	TupleTypeSpecifier

TABLE 6–B

Note that for named type specifiers, the name of the type is a qualified identifier, with the qualifier representing the name of the data model that defines the type. For example, the system-defined integer type in CQL is named System.Integer, with System as the name of the data model, and Integer as the name of the type.

6.1.3 Literals and Selectors

The following table defines the mapping between the various CQL literals and their equivalent representation in ELM:

CQL Literal	ELM Equivalent
null	Null
boolean-literal	Boolean
integer-literal	Literal (valueType="Integer")
decimal-literal	Literal (valueType="Decimal")
quantity-literal	Quantity
string-literal	Literal (valueType="String")
date-time-literal	DateTime
time-literal	Time
interval-selector	Interval
list-selector	List
tuple-selector	Tuple
instance-selector	Instance

TABLE 6–C

6.1.4 Functions

Most of the functions and operations available in CQL have a direct counterpart in ELM. For ease of reference, the operations and functions are grouped the same way they are in the CQL Reference.

HL7 Standard: Clinical Quality Language, Release 1 - US Realm
May 2015

6.1.4.1 Logical Operators

CQL Operator	ELM Equivalent
and	And
not	Not
or	Or
xor	Xor

6.1.4.2 Type Operators

CQL Operator	ELM Equivalent
as	As
convert	Convert
is	Is

Note that for supported conversions, a more efficient implementation would be to emit a specific operator to perform the conversion, rather than a generic Convert as specified here. For example, consider the following CQL conversion expression:

```
convert B to String
```

Rather than emitting a *Convert*, an implementation could emit a *ToString* which took an integer parameter. This would prevent the run-time type check required for implementation of a general purpose *Convert* operator.

Note also that when translating to ELM, an implementation could emit all implicit conversions directly, avoiding the need for an ELM translator or execution engine to deal with implicit conversion.

6.1.4.3 Nullological Operators

CQL Operator	ELM Equivalent
Coalesce	Coalesce
is null	IsNull
is false	IsFalse
is true	IsTrue

6.1.4.4 Comparison Operators

CQL Operator	ELM Equivalent
between	Between
=	Equal
>	Greater
>=	GreaterOrEqual

May 2015

CQL Operator	ELM Equivalent
<	Less
<=	LessOrEqual
matches	Matches
<>	NotEqual

6.1.4.5 Arithmetic Operators

CQL Operator	ELM Equivalent
Abs	Abs
+	Add
Ceiling	Ceiling
/	Divide
Floor	Floor
Log	Log
Ln	Ln
maximum	MaxValue
minimum	MinValue
mod	Modulo
*	Multiply
- (unary minus)	Negate
predecessor	Predecessor
^	Power
Round	Round
-	Subtract
successor	Successor
Truncate	Truncate
div	TruncatedDivide

6.1.4.6 String Operators

CQL Operator	ELM Equivalent
Combine	Combine
+	Concatenate
[]	Indexer
Length	Length
Lower	Lower
PositionOf	PositionOf
Split	Split
Substring	Substring

CQL Operator	ELM Equivalent
Upper	Upper

6.1.4.7 Date/Time Operators

CQL Operator	ELM Equivalent
+	Add
after	After
before	Before
DateTime	DateTime
component from	DateTimeComponentFrom
difference..between	DifferenceBetween
duration..between	DurationBetween
Now	Now
same as	SameAs
same or after	SameOrAfter
same or before	SameOrBefore
-	Subtract
Time	Time
TimeOfDay	TimeOfDay
Today	Today

6.1.4.8 Interval Operators

CQL Operator	ELM Equivalent
after	After
before	Before
collapse	Collapse
contains	Contains
end of	End
ends	Ends
=	Equal
except	Except
in	In
includes	Includes
during	IncludedIn
included in	IncludedIn
intersect	Intersect
matches	Matches
meets	Meets

HL7 Standard: Clinical Quality Language, Release 1 - US Realm
May 2015

CQL Operator	ELM Equivalent
meets after	MeetsAfter
meets before	MeetsBefore
<>	NotEqual
overlaps	Overlaps
overlaps after	OverlapsAfter
overlaps before	OverlapsBefore
properly includes	ProperlyIncludes
properly included in	ProperlyIncludedIn
properly during	ProperlyIncludedIn
start of	Start
starts	Starts
union	Union
width of	Width

<div align="center">TABLE 6–K</div>

6.1.4.9 List Operators

CQL Operator	ELM Equivalent
contains	Contains
distinct	Distinct
=	Equal
except	Except
exists	Exists
expand	Expand
First	First
in	In
includes	Includes
included in	IncludedIn
[]	Indexer
IndexOf	IndexOf
intersect	Intersect
Last	Last
Length	Length
matches	Matches
<>	NotEqual
properly includes	ProperlyIncludes
properly included in	ProperlyIncludedIn
singleton from	SingletonFrom
union	Union

TABLE 6–L

6.1.4.10 Aggregate Operators

CQL Operator	ELM Equivalent
AllTrue	AllTrue
AnyTrue	AnyTrue
Avg	Avg
Count	Count
Max	Max
Min	Min
Median	Median
Mode	Mode
PopulationStdDev	PopulationStdDev
PopulationVariance	PopulationVariance
StdDev	StdDev
Sum	Sum
Variance	Variance

TABLE 6–M

6.1.4.11 Clinical Operators

CQL Operator	ELM Equivalent
AgeIn-*precision*	CalculateAge (with patient birthdate reference supplied)
AgeIn-*precision*-At	CalculateAgeAt (with patient birthdate reference supplied)
CalculateAgeIn-*precision*	CalculateAge
CalculateAgeIn-*precision*-At	CalculateAgeAt
=	Equal
matches	Matches
in **(Codesystem)**	InCodeSystem
in **(Valueset)**	InValueSet

TABLE 6–N

6.1.5 Phrases

In general, the various phrases of CQL do not have a direct representation in ELM, but rather result in operator and function invocations which then do have representations. For more information, see the Timing Phrases section.

6.1.6 Queries

The CQL query construct has a direct representation in ELM, as shown by the following table:

CQL Construct	ELM Equivalent
query	Query

HL7 Standard: Clinical Quality Language, Release 1 - US Realm
May 2015

CQL Construct	ELM Equivalent
aliased-query-source	AliasedQuerySource
define-clause	DefineClause
with-clause	With
without-clause	Without
where-clause	Query (where element)
return-clause	ReturnClause
sort-clause	SortClause

TABLE 6−O

Althought these elements can be used to directly represent the *query* construct of CQL, it is also possible to represent queries using a series of equivalent operations that simplify implementation. ELM defines simplified operations specifically for this purpose. See the Implementing Query Evaluation section for more information on how to transform any given CQL query into an equivalent representation using these operators.

6.2 ELM-to-CQL

In addition to being able to translate CQL to ELM, any given expression of ELM can be represented in CQL. Support for this direction of translation would be useful for applications that produce ELM from another source, and need to display a human-readable representation of the logic.

This bi-directionality means that a given expression of CQL could be translated to ELM, and then back again. However, because ELM is typically a more primitive representation, this process is not necessarily a "round-trip". For example, consider the following CQL:

```
A starts within 3 days of start B
```

This will actually result in the following ELM output:

```
<expression xsi:type="In">
  <operand xsi:type="DurationBetween" precision="Day">
    <operand xsi:type="Start">
      <operand xsi:type="ExpressionRef" name="A"/>
    </operand>
    <operand xsi:type="Start">
      <operand xsi:type="ExpressionRef" name="B"/>
    </operand>
  </operand>
  <operand xsi:type="Interval">
    <low xsi:type="Literal" valueType="xs:int" value="-3"/>
    <high xsi:type="Literal" valueType="xs:int" value="3"/>
  </operand>
</expression>
```

The above expression, rendered directly back to CQL would be:

```
days between start of A and start of B in [-3, 3]
```

These expressions are semantically equivalent, but not syntactically the same, as the first is targeted at understandability, while the second is targeted at implementation. To preserve "round-trip" capability, an implementation could emit annotations with the ELM using the extension mechanism of the base *Element* class to provide the original source CQL.

In general, the mapping from ELM to CQL is simply the opposite of the mapping described in the previouis section. However, there are several special-purpose operators that are only defined in ELM which are used to simplify query implementation. For completeness, the mappings from those operators to CQL are described here to ensure that any given ELM document could be translated to CQL.

The examples in the following section will make use of the following expression definitions:

```xml
<def name="List1">
 <expression xsi:type="List">
  <element xsi:type="Tuple">
   <element name="X">
    <value xsi:type="Literal" valueType="xs:int" value="1"/>
   </element>
  </element>
  <element xsi:type="Tuple">
   <element name="X">
    <value xsi:type="Literal" valueType="xs:int" value="2"/>
   </element>
  </element>
  <element xsi:type="Tuple">
   <element name="X">
    <value xsi:type="Literal" valueType="xs:int" value="3"/>
   </element>
  </element>
 </expression>
</def>
<def name="List2">
 <expression xsi:type="List">
  <element xsi:type="Tuple">
   <element name="Y">
    <value xsi:type="Literal" valueType="xs:int" value="1"/>
   </element>
  </element>
  <element xsi:type="Tuple">
   <element name="Y">
    <value xsi:type="Literal" valueType="xs:int" value="2"/>
   </element>
  </element>
  <element xsi:type="Tuple">
   <element name="Y">
    <value xsi:type="Literal" valueType="xs:int" value="3"/>
   </element>
  </element>
 </expression>
</def>
```

6.2.1 ForEach

The *ForEach* operator in ELM takes an argument of type list and returns a list with an element for each source element that is the result of evaluating the *element* expression. For example:

```
<expression xsi:type="ForEach">
  <source xsi:type="ExpressionRef" name="List1"/>
  <element xsi:type="Property" path="X"/>
</expression>
```

This expression returns the list of integers from the List1 expression. Although there is no direct counterpart in CQL, this expression can be represented using the *query* construct. The source for the *ForEach* is used as the primary query source, and the *element* expression is represented using the *return-clause*:

```
List1 A return A.X
```

6.2.2 Times

The *Times* operator in ELM computes the Cartesian-product of two lists. Again, although there is no direct counterpart in CQL, the *query* construct can be used to produce an equivalent result:

```
<expression xsi:type="Times">
  <source xsi:type="ExpressionRef" name="List1"/>
  <source xsi:type="ExpressionRef" name="List2"/>
</expression>
```

Assuming List1 and List2 are defined as specified above, the equivalent CQL is a multi-source query with a source for each operand in the *Times*, and a return clause that builds the resulting tuples:

```
from List1 A, List2 B
  return { X: A.X, Y: B.Y }
```

6.2.3 Filter

The *Filter* operator in ELM filters the contents of a list, returning only those elements that satisfy the expression defined in the *condition* element. For example:

```
<expression xsi:type="Filter">
  <source xsi:type="ExpressionRef" name="List1"/>
  <condition xsi:type="Equal">
    <operand xsi:type="Property" path="X">
    <operand xsi:type="Literal" valueType="xs:int" value="1"/>
  </condition>
</expression>
```

Again, although no direct counterpart in CQL exists, the *where* clause of the *query* construct provides the equivalent functionality:

```
List1 A where A.X = 1
```

6.2.4 Sort

The *Sort* operator in ELM sorts the contents of a list. For example:

```
<expression xsi:type="Sort">
  <source xsi:type="ExpressionRef" name="List1"/>
  <by xsi:type="ByColumn" path="X" direction="desc"/>
</expression>
```

Again, the CQL query construct provides the equivalent functionality:

```
List1 A sort by A.X desc
```

7 PHYSICAL REPRESENTATION

The physical representation for CQL is specifically concerned with communicating the logic involved in any given artifact. As discussed in the previous sections, the unit of distribution for CQL is the library, which corresponds to a single file of CQL at the author level, or a single XML document at the physical level.

7.1 Schemata

The physical representation is simply a set of XML schemata which define XML types for each class defined in the ELM UML model. A CQL physical library is then an XML document with a single *Library* element as the root.

The physical representation for ELM is defined by the following schemata:

Schema	Description
expression.xsd	Defines expression logic components without reference to clinically relevant constructs
clinicalexpression.xsd	Introduces expression components that contain clinically-relevant constructs
library.xsd	Defines the overall library container for ELM

<div align="center">TABLE 7–A</div>

As with the logical portion of the specification, this documentation is intended to provide an overview only, the schemata are the actual specification and should be considered the source of truth.

7.2 Library References

The implementation environment must provide a mechanism for library references to be resolved based on their names and versions.

7.3 Data Model References

In addition, the implementation environment must provide a mechanism for data model references to be resolved. At a minimum, the data model definition must define the structure of all the types available within the data model, generally by providing an XSD or similar class structure definition. If the implementation environment is only concerned with translation or execution of ELM documents, then the type structures for each data model are sufficient. However, to fully enable the authoring features of CQL syntax, the data model reference must also define the following:

Component	Description
URL	The XML namespace associated with the model. This namespace is used by the CQL-to-ELM translator to establish the URL used to reference types from the model schema within an ELM document.
Schema Location	The physical location of the model xsd relative to the ELM document. This information can be provided, but is not required.
Target Qualifier	If specified, determines the namespace qualifier that should be used when

Component	Description
	referencing types of the data model within the ELM document.
Patient Type	The name of the type that is used to represent patient information within the model.
Patient Birth Date	The name of the birth date property on the patient type. This information is used by the CQL-to-ELM translator to render references to patient-age-related functions (AgeInYears, AgeInYearsAt, etc.) into the non-patient-aware age-related functions in ELM (CalculateAgeInYears, CalculateAgeInYearsAt, etc.). This information is not required, but if it is not present, references to patient-age-related functions will be passed directly through to ELM as FunctionRefs.

<p align="center">TABLE 7–B</p>

For each type available in the data model, the following information should be provided:

Component	Description
Name	The name of the type within the data model. This corresponds to the name of the class within the class model, or the name of the type in the case of an xsd. In FHIR, for example, this corresponds to the name of the underlying resource.
Identifier	A unique identifier for the class that may be independent of the name. In FHIR, for example, this corresponds to the profile identifier.
Label	This information specifies the name of the type as it is referenced from CQL. Note that this need not be a language-valid identifier, as CQL allows quoted-identifiers to be used. However, the label must be unique. In the simplest case, the label corresponds directly with the class name. Whether or not a label is provided, a class can still be referenced from CQL by its name.
Primary Code Filter	If the type has the notion of a primary code filter (e.g., Encounter), the name of the attribute that is to be used if no code filter attribute is named within a retrieve
Retrievable	A boolean flag indicating whether the class can be referenced as a topic in a retrieve. If this flag is not set, values of this class cannot be retrieved directly, but may still be accessible as elements of other class values.

<p align="center">TABLE 7–C</p>

The information defined here is formally described in the modelinfo.xsd document included in the specification. The QUICK module in the CQL-to-ELM translator contains an instance of this schema, quick-modelinfo.xml, which defines this metadata for the QUICK model.

Note that the actual model info definition and associated artifacts are part of the reference implementation for CQL and not a normative aspect of the CQL specification. CQL only specifies the expected behavior at the conceptual level. How that behavior is achieved with respect to any particular data model is an implementation aspect and not prescribed by this specification.

8 APPENDIX A – CQL SYNTAX FORMAL SPECIFICATION

The formal specification for the CQL syntax is defined using the ANTLR4 grammar framework. This framework is a general purpose cross-platform technology for describing computer languages. For more information on this framework, refer to the ANTLR website http://www.antlr.org/.

Note that the material in this section is necessarily technical and assumes familiarity with language definition in general, and ANTLR4 in particular. Note also that the g4 presented here is somewhat simplified. For the complete g4 definition, refer to the CQL.g4 file included with the specification package.

8.1 Declarations

The CQL grammar is defined in a single ANTLR4 grammar file, CQL.g4. The root production rule is *logic*, which specifies the overall structure for a library file:

```
logic
    :
    libraryDefinition?
    usingDefinition*
    includeDefinition*
    codesystemDefinition*
    valuesetDefinition*
    parameterDefinition*
    statement+
    ;
```

Other than *statement*, these production rules define the declarations available for a library.

```
libraryDefinition
    : 'library' identifier ('version' versionSpecifier)?
    ;

usingDefinition
    : 'using' identifier ('version' versionSpecifier)?
    ;

includeDefinition
    : 'include' identifier ('version' versionSpecifier)? 'called' localIdentifier
    ;

localIdentifier
    : identifier
    ;

accessModifier
    : 'public'
    | 'private'
    ;

parameterDefinition
    : accessModifier? 'parameter' identifier typeSpecifier? ('default' expression)?
```

```
        ;

codesystemDefinition
    : accessModifier? 'codesystem' identifier ':' codesystemId
        ('version' versionSpecifier)?
    ;

valuesetDefinition
    : accessModifier? 'valueset' identifier ':' valuesetId
        ('version' versionSpecifier)? codesystems?
    ;

codesystems
    : 'codesystems' '(' codesystemIdentifier (',' codesystemIdentifier)* ')'
    ;

codesystemIdentifier
    : (libraryIdentifier '.')? identifier
    ;

libraryIdentifier
    : identifier
    ;

codesystemId
    : STRING
    ;

valuesetId
    : STRING
    ;

versionSpecifier
    : STRING
    ;
```

8.2   Type Specifiers

The *typeSpecifier* production rule defines all type specifiers available in the language.

```
typeSpecifier
    : namedTypeSpecifier
    | listTypeSpecifier
    | intervalTypeSpecifier
    | tupleTypeSpecifier
    ;

namedTypeSpecifier
    : (modelIdentifier '.')? identifier
    ;

modelIdentifier
    : identifier
    ;

listTypeSpecifier
    : 'List' '<' typeSpecifier '>'
```

```
    ;

intervalTypeSpecifier
    : 'Interval' '<' typeSpecifier '>'
    ;

tupleTypeSpecifier
    : 'Tuple' '{' tupleElementDefinition (',' tupleElementDefinition)* '}'
    ;

tupleElementDefinition
    : identifier typeSpecifier
    ;
```

8.3   Statements

The main body of the library then consists of any number of statements, defined by the *statement* production rule:

```
statement
    : expressionDefinition
    | contextDefinition
    | functionDefinition
    ;

expressionDefinition
    : 'define' accessModifier? identifier ':' expression
    ;

contextDefinition
    : 'context' identifier
    ;

functionDefinition
    : 'define' accessModifier? 'function' identifier
      '(' (operandDefinition (',' operandDefinition)*)? ')' ':' functionBody
    ;

operandDefinition
    : identifier typeSpecifier
    ;

functionBody
    : expression
    ;
```

8.4   Queries

The *query* production rule defines the syntax for queries within CQL:

```
querySource
    : retrieve
    | qualifiedIdentifier
    | '(' expression ')'
    ;
```

```
aliasedQuerySource
    : querySource alias
    ;

alias
    : identifier
    ;

queryInclusionClause
    : withClause
    | withoutClause
    ;

withClause
    : 'with' aliasedQuerySource 'such that' expression
    ;

withoutClause
    : 'without' aliasedQuerySource 'such that' expression
    ;

retrieve
    : '[' namedTypeSpecifier (':' (valuesetPathIdentifier 'in')? valueset)? ']'
    ;

valuesetPathIdentifier
    : identifier
    ;

valueset
    : qualifiedIdentifier
    ;

qualifier
    : identifier
    ;

query
    : sourceClause
        defineClause?
        queryInclusionClause*
        whereClause?
        returnClause?
        sortClause?
    ;

sourceClause
    : singleSourceClause
    | multipleSourceClause
    ;

singleSourceClause
    : aliasedQuerySource
    ;

multipleSourceClause
```

```
    : 'foreach' aliasedQuerySource (',' aliasedQuerySource)*
    ;

defineClause
    : 'define' defineClauseItem (',' defineClauseItem)*
    ;

defineClauseItem
    : identifier ':' expression
    ;

whereClause
    : 'where' expression
    ;

returnClause
    : 'return' ('all' | 'distinct')? expression
    ;

sortClause
    : 'sort' ( sortDirection | ('by' sortByItem (',' sortByItem)*) )
    ;

sortDirection
    : 'asc' | 'ascending'
    | 'desc' | 'descending'
    ;

sortByItem
    : expressionTerm sortDirection?
    ;

qualifiedIdentifier
    : (qualifier '.')* identifier
    ;
```

8.5 Expressions

The *expression* production rule defines the syntax for all expressions within CQL:

```
expression
    : expressionTerm
    | retrieve
    | query
    | expression 'is' 'not'? ('null' | 'true' | 'false')
    | expression ('is' | 'as') typeSpecifier
    | 'cast' expression 'as' typeSpecifier
    | 'not' expression
    | 'exists' expression
    | expression 'properly'? 'between' expressionTerm 'and' expressionTerm
    | pluralDateTimePrecision 'between' expressionTerm 'and' expressionTerm
    | 'difference' 'in'
        pluralDateTimePrecision 'between' expressionTerm 'and' expressionTerm
    | expression ('<=' | '<' | '>' | '>=') expression
    | expression intervalOperatorPhrase expression
    | expression ('=' | '<>' | 'matches') expression
    | expression ('in' | 'contains') dateTimePrecisionSpecifier? expression
```

```
    | expression 'and' expression
    | expression ('or' | 'xor') expression
    | expression ('union' | 'intersect' | 'except') expression
    ;

dateTimePrecision
    : 'year' | 'month' | 'day' | 'hour' | 'minute' | 'second' | 'millisecond'
    ;

dateTimeComponent
    : dateTimePrecision
    | 'date'
    | 'time'
    | 'timezone'
    ;

pluralDateTimePrecision
    : 'years' | 'months' | 'days' | 'hours' | 'minutes' | 'seconds' | 'milliseconds'
    ;

expressionTerm
    : term
    | expressionTerm '.' identifier
    | expressionTerm '[' expression ']'
    | (qualifier '.')? identifier '(' (expression (',' expression)*)? ')'
    | 'convert' expression 'to' typeSpecifier
    | ('+' | '-') expressionTerm
    | ('start' | 'end') 'of' expressionTerm
    | dateTimeComponent 'from' expressionTerm
    | 'duration' 'in' pluralDateTimePrecision 'of' expressionTerm
    | 'width' 'of' expressionTerm
    | 'successor' 'of' expressionTerm
    | 'predecessor' 'of' expressionTerm
    | 'singleton' 'from' expressionTerm
    | ('minimum' | 'maximum') namedTypeSpecifier
    | expressionTerm '^' expressionTerm
    | expressionTerm ('*' | '/' | 'div' | 'mod') expressionTerm
    | expressionTerm ('+' | '-') expressionTerm
    | 'if' expression 'then' expression 'else' expression
    | 'case' expression? caseExpressionItem+ 'else' expression 'end'
    | ('distinct' | 'collapse' | 'expand') expression
    ;

caseExpressionItem
    : 'when' expression 'then' expression
    ;

dateTimePrecisionSpecifier
    : dateTimePrecision 'of'
    ;

relativeQualifier
    : 'or before'
    | 'or after'
    ;

offsetRelativeQualifier
```

```
    : 'or more'
    | 'or less'
    ;

quantityOffset
    : quantityLiteral offsetRelativeQualifier?
    ;

intervalOperatorPhrase
    : ('starts' | 'ends' | 'occurs')? 'same' dateTimePrecision?
        (relativeQualifier | 'as') ('start' | 'end')?
    | 'properly'? 'includes' dateTimePrecisionSpecifier? ('start' | 'end')?
    | ('starts' | 'ends' | 'occurs')? 'properly'? ('during' | 'included in')
        dateTimePrecisionSpecifier?
    | ('starts' | 'ends' | 'occurs')? quantityOffset? ('before' | 'after')
        ('start' | 'end')?
    | ('starts' | 'ends' | 'occurs')? 'properly'? 'within' quantityLiteral 'of'
        ('start' | 'end')?
    | 'meets' ('before' | 'after')? dateTimePrecisionSpecifier?
    | 'overlaps' ('before' | 'after')? dateTimePrecisionSpecifier?
    | 'starts' dateTimePrecisionSpecifier?
    | 'ends' dateTimePrecisionSpecifier?
    ;
```

8.6   Terms

The *term* production rule defines the syntax for core expression terms within CQL:

```
term
    : identifier
    | literal
    | intervalSelector
    | tupleSelector
    | instanceSelector
    | listSelector
    | codeSelector
    | conceptSelector
    | '(' expression ')'
    ;

intervalSelector
    'Interval' ('['|'(') expression ',' expression (']'|')')
    ;

tupleSelector
    : 'Tuple'? '{' (':' | (tupleElementSelector (',' tupleElementSelector)*)) '}'
    ;

tupleElementSelector
    : identifier ':' expression
    ;

instanceSelector
    : namedTypeSpecifier '{' (':' | (instanceElementSelector
        (',' instanceElementSelector)*)) '}'
    ;
```

```
instanceElementSelector
    : identifier ':' expression
    ;

listSelector
    : ('List' ('<' typeSpecifier '>')?)? '{' expression? (',' expression)* '}'
    ;

displayClause
    : 'display' stringLiteral
    ;

codeSelector
    : 'Code' stringLiteral 'from' codesystemIdentifier displayClause?
    ;

conceptSelector
    : 'Concept' '{' codeSelector (',' codeSelector)* '}' displayClause?
    ;

literal
    : nullLiteral
    | booleanLiteral
    | stringLiteral
    | dateTimeLiteral
    | timeLiteral
    | quantityLiteral
    ;

nullLiteral
    : 'null'
    ;

booleanLiteral
    : 'true'
    | 'false'
    ;

stringLiteral
    : STRING
    ;

dateTimeLiteral
    : DATETIME
    ;

timeLiteral
    : TIME
    ;

quantityLiteral
    : QUANTITY unit?
    ;

unit
    : dateTimePrecision
```

HL7 Standard: Clinical Quality Language, Release 1 - US Realm
May 2015

```
    | pluralDateTimePrecision
    | STRING // UCUM syntax for units of measure
    ;
identifier
    : IDENTIFIER | QUOTEDIDENTIFIER
    | 'Code'
    | 'Concept'
    | 'date'
    | 'display'
    | 'time'
    | 'timezone'
    | 'version'
    ;
```

8.7   Lexer Rules

The lexer rules define the terminal production rules in the language:

```
IDENTIFIER
    : ([A-Za-z] | '_')([A-Za-z0-9] | '_')*
    ;

QUANTITY
    : [0-9]+('.'[0-9]+)?
    ;

QUOTEDIDENTIFIER
    : '"' ( ~[\\"] | '""' )* '"'
    ;

STRING
    : ('\'') ( ~[\\'] | '\'\'' )* ('\'')
    ;

WS
    : (' ' | '\r' | '\t') -> channel(HIDDEN)
    ;

NEWLINE
    : ('\n') -> channel(HIDDEN)
    ;

COMMENT
    : '/*' .*? '*/' -> skip
    ;

LINE_COMMENT
    :   '//' ~[\r\n]* -> skip
    ;
```

HL7 Standard: Clinical Quality Language, Release 1 - US Realm
May 2015

9 APPENDIX B – CQL REFERENCE

This appendix provides a reference for all the system-defined types, operators, and functions that can be used within CQL. It is intended to provide complete semantics for each available type and operator as a companion to the Author's and Developer's Guides. The reference is organized by operator category.

For each type, the definition and semantics are provided. Note that because CQL does not define a type declaration syntax, the definitions are expressed in a pseudo-syntax.

For each operator or function, the signature, semantics, and usually an example are provided. Note that for built-in operators, the signature is expressed in a pseudo-syntax intended to clearly define the operator and its parameters. Although the symbolic operators may in general be prefix, infix, or postfix operators, the signatures for each operator are defined using function definition syntax for consistency and ease of representation. For example, the signature for the `and` operator is given as:

```
and(left Boolean, right Boolean) Boolean
```

Even though `and` is an infix operator and would be invoked as in the following expression:

```
InDemographic and NeedsScreening
```

9.1 Types

9.1.1 Any

Definition:

```
simple type Any
```

Description:

The `Any` type is the maximal supertype in the CQL type system, meaning that all types derive from `Any`, including list, interval, and structured types. In addition, the type of a `null` result is `Any`.

9.1.2 Boolean

Definition:

```
simple type Boolean
```

Description:

The `Boolean` type represents the logical boolean values `true` and `false`. The result of logical operations within CQL use the `Boolean` type, and constructs within the language that expect a conditional result, such as a where clause or conditional expression, expect results of the `Boolean` type.

9.1.3 Code

Definition:

```
structured type Code
{
  code String,
  display String,
  system String,
  version String
}
```

Description:

The `Code` type represents single terminology codes within CQL.

9.1.4 Concept

Definition:

```
structured type Concept
{
  codes List<String>,
  display String
}
```

Description:

The `Concept` type represents a single terminological concept within CQL.

9.1.5 DateTime

Definition:

```
simple type DateTime
```

Description:

The `DateTime` type represents date and time values with potential uncertainty within CQL.

CQL supports date and time values in the range `@0001-01-01T00:00:00.0` to `@9999-12-31T23:59:59.999` with a 1 millisecond step size.

9.1.6 Decimal

Definition:

```
simple type Decimal
```

Description:

The `Decimal` type represents real values within CQL.

CQL supports decimal values in the range `-10^28-10^-8` to `10^28-10^-8` with a step size of `10^-8`.

9.1.7 Integer

Definition:

```
simple type Integer
```

Description:

The `Integer` type represents whole number values within CQL.

CQL supports integer values in the range `-2^31` to `2^31-1` with a step size of `1`.

9.1.8 Quantity

Definition:

```
structured type Quantity
{
  value Decimal
  unit String
}
```

Description:

The `Quantity` type represents quantities with a specified unit within CQL.

9.1.9 String

Definition:

```
simple type String
```

Description:

The `String` type represents string values within CQL.

CQL supports string values up to `2^31-1` characters in length.

9.1.10   Time

Definition:

```
simple type Time
```

Description:

The `Time` type represents time-of-day values within CQL.

CQL supports time values in the range `@T00:00:00.0` to `@T23:59:59.999` with a step size of 1 millisecond.

9.2   Logical Operators

9.2.1 And

Signature:

```
and (left Boolean, right Boolean) Boolean
```

Description:

The `and` operator returns true if both its arguments are true. If either argument is false, the result is false. Otherwise, the result is null.

The following table defines the truth table for this operator:

	TRUE	FALSE	NULL
TRUE	TRUE	FALSE	NULL
FALSE	FALSE	FALSE	FALSE
NULL	NULL	FALSE	NULL

Example:

The following examples illustrate the behavior of the `and` operator:

```
define IsTrue = true and true
define IsFalse = true and false
define IsAlsoFalse = false and null
define IsNull = true and null
```

9.2.2 Not

Signature:

```
not (argument Boolean) Boolean
```

Description:

The `not` operator returns true if the argument is false and false if the argument is true. Otherwise, the result is null.

The following table defines the truth table for this operator:

	NOT
TRUE	FALSE
FALSE	TRUE
NULL	NULL

9.2.3 Or

Signature:

```
or (left Boolean, right Boolean) Boolean
```

Description:

The `or` operator returns true if either of its arguments are true. If both arguments are false, the result is false. Otherwise, the result is null.

The following table defines the truth table for this operator:

	TRUE	FALSE	NULL
TRUE	TRUE	TRUE	TRUE
FALSE	TRUE	FALSE	NULL
NULL	TRUE	NULL	NULL

TABLE 9–C

Example:

The following examples illustrate the behavior of the `or` operator:

```
define IsTrue = true or false
define IsAlsoTrue = true or null
define IsFalse = false or false
define IsNull = false or null
```

9.2.4 Xor

Signature:

```
xor (left Boolean, right Boolean) Boolean
```

Description:

The `xor` (exclusive or) operator returns true if one argument is true and the other is false. If both arguments are true or both arguments are false, the result is false. Otherwise, the result is null.

The following table defines the truth table for this operator:

	TRUE	FALSE	NULL
TRUE	FALSE	TRUE	NULL
FALSE	TRUE	FALSE	NULL
NULL	NULL	NULL	NULL

TABLE 9–D

9.3 Type Operators

9.3.1 As

Signature:

```
as<T>(argument Any) T
cast as<T>(argument Any) T
```

Description:

The `as` operator allows the result of an expression to be cast as a given target type. This allows expressions to be written that are statically typed against the expected run-time type of the argument.

If the argument is not of the specified type at run-time the result is `null`.

The `cast` prefix indicates that if the argument is not of the specified type at run-time then an exception is thrown.

Example:

The following examples illustrate the use of the `as` operator.

```
define AllProcedures: [Procedure]
define ImagingProcedures:
  AllProcedures P
    where P is ImagingProcedure
    return P as ImagingProcedure
define RuntimeError:
  ImagingProcedures P
    return cast P as Observation
```

9.3.2 Convert

Signature:

```
convert to<T>(argument Any) T
```

Description:

The convert operator converts a value to a specific type. The result of the operator is the value of the argument converted to the target type, if possible. Note that use of this operator may result in a run-time exception being thrown if there is no valid conversion from the actual value to the target type.

The following table lists the conversions supported in CQL:

From\To	Boolean	Integer	Decimal	Quantity	String	Datetime	Time	Code	Concept
Boolean	N/A	-	-	-	Explicit	-	-	-	-
Integer	-	N/A	Implicit	-	Explicit	-	-	-	-
Decimal	-	-	N/A	-	Explicit	-	-	-	-
Quantity	-	-	-	N/A	Explicit	-	-	-	-
String	Explicit	Explicit	Explicit	Explicit	N/A	Explicit	Explicit	-	-
Datetime	-	-	-	-	Explicit	N/A	-	-	-
Time	-	-	-	-	Explicit	-	N/A	-	-
Code	-	-	-	-	-	-	-	N/A	Implicit
Concept	-	-	-	-	-	-	-	-	N/A

TABLE 9–E

For conversions between date/time and string values, ISO-8601 standard format is used:

yyyy-MM-ddThh:mm:ss.fff(Z | +/- hh:mm)

For example, the following are valid string representations for date/time values:

```
'2014-01-01T14:30:00.0Z'        // January 1st, 2014, 2:30PM UTC
'2014-01-01T14:30:00.0-07:00'   // January 1st, 2014, 2:30PM Mountain Standard (GMT-7:00)
'T14:30:00.0Z'                  // 2:30PM UTC
'T14:30:00.0-07:00'             // 2:30PM Mountain Standard (GMT-7:00)
```

For specific semantics for each conversion, refer to the explicit conversion operator documentation.

9.3.3 Is

Signature:

May 2015

```
is<T>(argument Any) Boolean
```

Description:

The `is` operator allows the type of a result to be tested. If the run-time type of the argument is of the type being tested, the result of the operator is `true`; otherwise, the result is `false`.

9.3.4 ToBoolean

Signature:

```
ToBoolean(argument String) Boolean
```

Description:

The `ToBoolean` operator converts the value of its argument to a `Boolean` value. The operator accepts the following string representations:

String Representation	Boolean Value
true t yes y 1	true
false f no n 0	false

<div align="center">TABLE 9–F</div>

Note that the operator will ignore case when interpreting the string as a `Boolean` value.

If the input cannot be interpreted as a valid `Boolean` value, a run-time error is thrown.

If the argument is `null`, the result is `null`.

9.3.5 ToConcept

Signature:

```
ToConcept(argument Code) Concept
```

Description:

The `ToConcept` operator converts a value of type `Code` to a `Concept` value with the given `Code` as its primary and only `Code`. If the `Code` has a `display` value, the resulting `Concept` will have the same `display` value.

If the argument is `null`, the result is `null`.

9.3.6 ToDateTime

Signature:

```
ToDateTime(argument String) DateTime
```

Description:

The `ToDateTime` operator converts the value of its argument to a `DateTime` value. The operator expects the string to be formatted using the ISO-8601 date/time representation:

YYYY-MM-DDThh:mm:ss.fff(+|-)hh:mm

In addition, the string must be interpretable as a valid date/time value.

HL7 Standard: Clinical Quality Language, Release 1 - US Realm
May 2015

For example, the following are valid string representations for date/time values:

```
'2014-01-01'                // January 1st, 2014
'2014-01-01T14:30:00.0Z'    // January 1st, 2014, 2:30PM UTC
'2014-01-01T14:30:00.0-07:00' // January 1st, 2014, 2:30PM Mountain Standard (GMT-7:00)
```

If the input string is not formatted correctly, or does not represent a valid date/time value, a run-time error is thrown.

As with date/time literals, date/time values may be specified to any precision. If no timezone is supplied, the timezone of the evaluation request timestamp is assumed.

If the argument is `null`, the result is `null`.

9.3.7 ToDecimal

Signature:

```
ToDecimal(argument String) Decimal
```

Description:

The `ToDecimal` operator converts the value of its argument to a `Decimal` value. The operator accepts strings using the following format:

(+|-)?#0(.0#)?

Meaning an optional polarity indicator, followed by any number of digits (including none), followed by at least one digit, followed optionally by a decimal point, at least one digit, and any number of additional digits (including none).

Note that the decimal value returned by this operator must be limited in precision and scale to the maximum precision and scale representable for `Decimal` values within CQL.

If the input string is not formatted correctly, or cannot be interpreted as a valid `Decimal` value, a run-time error is thrown.

If the argument is `null`, the result is `null`.

9.3.8 ToInteger

Signature:

```
ToInteger(argument String) Integer
```

Description:

The `ToInteger` operator converts the value of its argument to an `Integer` value. The operator accepts strings using the following format:

(+|-)?#0

Meaning an optional polarity indicator, followed by any number of digits (including none), followed by at least one digit.

Note that the integer value returned by this operator must be a valid value in the range representable for `Integer` values in CQL.

If the input string is not formatted correctly, or cannot be interpreted as a valid `Integer` value, a run-time error is thrown.

If the argument is `null`, the result is `null`.

9.3.9 ToQuantity

Signature:

```
ToQuantity(argument String) Quantity
```

Description:

The `ToQuantity` operator converts the value of its argument to a `Quantity` value. The operator accepts strings using the following format:

(+|-)?#0(.0#)?('<unit>')?

Meaning an optional polarity indicator, followed by any number of digits (including none) followed by at least one digit, optionally followed by a decimal point, at least one digit, and any number of additional digits, all optionally followed by a unit designator as a string literal specifying a valid UCUM unit of measure. Spaces are allowed between the quantity value and the unit designator.

Note that the decimal value of the quantity returned by this operator must be a valid value in the range representable for `Decimal` values in CQL.

If the input string is not formatted correctly, or cannot be interpreted as a valid `Quantity` value, a run-time error is thrown.

If the argument is `null`, the result is `null`.

9.3.10   ToString

Signature:

```
ToString(argument Boolean) String
ToString(argument Integer) String
ToString(argument Decimal) String
ToString(argument Quantity) String
ToString(argument DateTime) String
ToString(argument Time) String
```

Description:

The `ToString` operator converts the value of its argument to a `String` value. The operator uses the following string representations for each type:

Type	String Representation
Boolean	true|false
Integer	(-)?#0
Decimal	(-)?#0.0#
Quantity	(-)?#0.0# '<unit>'
DateTime	YYYY-MM-DDThh:mm:ss.fff(+|-)hh:mm

Time	Thh:mm:ss.fff(+|-)hh:mm

<div align="center">**TABLE 9–G**</div>

If the argument is `null`, the result is `null`.

9.3.11 ToTime

Signature:

```
ToTime(argument String) Time
```

Description:

The `ToTime` operator converts the value of its argument to a `Time` value. The operator expects the string to be formatted using ISO-8601 time representation:

Thh:mm:ss.fff(+|-)hh:mm

In addition, the string must be interpretable as a valid time-of-day value.

For example, the following are valid string representations for time-of-day values:

```
'T14:30:00.0Z'                // 2:30PM UTC
'T14:30:00.0-07:00'           // 2:30PM Mountain Standard (GMT-7:00)
```

If the input string is not formatted correctly, or does not represent a valid time-of-day value, a run-time error is thrown.

As with time-of-day literals, time-of-day values may be specified to any precision. If no timezone is supplied, the timezone of the evaluation request timestamp is assumed.

If the argument is `null`, the result is `null`.

9.4 Nullological Operators

9.4.1 Coalesce

Signature:

```
Coalesce<T>(argument1 T, argument2 T) T
Coalesce<T>(argument1 T, argument2 T, argument3 T) T
Coalesce<T>(argument1 T, argument2 T, argument3 T, argument4 T) T
Coalesce<T>(argument1 T, argument2 T, argument3 T, argument4 T, argument5 T) T
Coalesce<T>(arguments List<T>) T
```

Description:

The `Coalesce` operator returns the first non-null result in a list of arguments. If all arguments evaluate to `null`, the result is `null`.

The static type of the first argument determines the type of the result, and all subsequent arguments must be of that same type.

9.4.2 IsNull

Signature:

```
is null(argument Any) Boolean
```

Description:

The `is null` operator determines whether or not its argument evaluates to `null`. If the argument evaluates to `null`, the result is `true`; otherwise, the result is `false`.

9.4.3 IsFalse

Signature:

```
is false(argument Boolean) Boolean
```

Description:

The `is false` operator determines whether or not its argument evaluates to `false`. If the argument evaluates to `false`, the result is `true`; otherwise, the result is `false`.

9.4.4 IsTrue

Signature:

```
is true(argument Boolean) Boolean
```

Description:

The `is true` operator determines whether or not its argument evaluates to `true`. If the argument evaluates to `true`, the result is `true`; otherwise, the result is `false`.

9.5 Comparison Operators

9.5.1 Between

Signature:

```
between(argument Integer, low Integer, high Integer) Boolean
between(argument Decimal, low Decimal, high Decimal) Boolean
between(argument Quantity, low Quantity, high Quantity) Boolean
between(argument DateTime, low DateTime, high DateTime) Boolean
between(argument Time, low Time, high Time) Boolean
between(argument String, low String, high String) Boolean
```

Description:

The between operator determines whether the first argument is within a given range, inclusive. If the first argument is greater than or equal to the low argument, and less than or equal to the high argument, the result is `true`, otherwise, the result is `false`.

For comparisons involving quantities, the dimensions of each quantity must match, but not necessarily the unit. For example, units of `'cm'` and `'m'` are comparable, but units of `'cm2'` and `'cm'` are not.

For comparisons involving date/time or time values with imprecision, note that the result of the comparison may be `null`, depending on whether the values involved are specified to the level of precision used for the comparison.

HL7 Standard: Clinical Quality Language, Release 1 - US Realm
May 2015

If any argument is `null`, the result is `null`.

9.5.2 Equal

Signature:

```
=<T>(left T, right T) Boolean
```

Description:

The *equal* (`=`) operator returns `true` if the arguments are equal; `false` if the arguments are known unequal, and `null` otherwise. Equality semantics are defined to be value-based.

For simple types, this means that equality returns `true` if and only if the result of each argument evaluates to the same value.

For quantities, this means that the dimensions of each quantity must match, but not necessarily the unit. For example, units of `'cm'` and `'m'` are comparable, but units of `'cm2'` and `'cm'` are not.

For tuple types, this means that equality returns `true` if and only if the tuples are of the same type, and the values for all elements by name are equal.

For list types, this means that equality returns `true` if and only if the lists contain elements of the same type, have the same number of elements, and for each element in the lists, in order, the elements are equal using the same semantics.

For interval types, equality returns `true` if and only if the intervals are over the same point type, and they have the same value for the starting and ending points of the interval as determined by the `Start` and `End` operators.

For comparisons involving date/time or time values with imprecision, note that the result of the comparison may be `null`, depending on whether the values involved are specified to the level of precision used for the comparison.

If either argument is `null`, the result is `null`.

9.5.3 Greater

Signature:

```
>(left Integer, right Integer) Boolean
>(left Decimal, right Decimal) Boolean
>(left Quantity, right Quantity) Boolean
>(left DateTime, right DateTime) Boolean
>(left Time, right Time) Boolean
>(left String, right String) Boolean
```

Description:

The *greater* (`>`) operator returns `true` if the first argument is greater than the second argument.

For comparisons involving quantities, the dimensions of each quantity must match, but not necessarily the unit. For example, units of `'cm'` and `'m'` are comparable, but units of `'cm2'` and `'cm'` are not.

For comparisons involving date/time or time values with imprecision, note that the result of the comparison may be `null`, depending on whether the values involved are specified to the level of precision used for the comparison.

If either argument is `null`, the result is `null`.

9.5.4 Greater Or Equal

Signature:

```
>=(left Integer, right Integer) Boolean
>=(left Decimal, right Decimal) Boolean
>=(left Quantity, right Quantity) Boolean
>=(left DateTime, right DateTime) Boolean
>=(left Time, right Time) Boolean
>=(left String, right String) Boolean
```

Description:

The *greater or equal* (`>=`) operator returns `true` if the first argument is greater than or equal to the second argument.

For comparisons involving quantities, the dimensions of each quantity must match, but not necessarily the unit. For example, units of `'cm'` and `'m'` are comparable, but units of `'cm2'` and `'cm'` are not.

For comparisons involving date/time or time values with imprecision, note that the result of the comparison may be `null`, depending on whether the values involved are specified to the level of precision used for the comparison.

If either argument is `null`, the result is `null`.

9.5.5 Less

Signature:

```
<(left Integer, right Integer) Boolean
<(left Decimal, right Decimal) Boolean
<(left Quantity, right Quantity) Boolean
<(left DateTime, right DateTime) Boolean
<(left Time, right Time) Boolean
<(left String, right String) Boolean
```

Description:

The *less* (`<`) operator returns `true` if the first argument is less than the second argument.

For comparisons involving quantities, the dimensions of each quantity must match, but not necessarily the unit. For example, units of `'cm'` and `'m'` are comparable, but units of `'cm2'` and `'cm'` are not.

For comparisons involving date/time or time values with imprecision, note that the result of the comparison may be `null`, depending on whether the values involved are specified to the level of precision used for the comparison.

If either argument is `null`, the result is `null`.

9.5.6 Less Or Equal

Signature:

```
<=(left Integer, right Integer) Boolean
<=(left Decimal, right Decimal) Boolean
<=(left Quantity, right Quantity) Boolean
<=(left DateTime, right DateTime) Boolean
<=(left Time, right Time) Boolean
<=(left String, right String) Boolean
```

Description:

The *less or equal* (`<=`) operator returns `true` if the first argument is less than or equal to the second argument.

For comparisons involving quantities, the dimensions of each quantity must match, but not necessarily the unit. For example, units of `'cm'` and `'m'` are comparable, but units of `'cm2'` and `'cm'` are not.

For comparisons involving date/time or time values with imprecision, note that the result of the comparison may be `null`, depending on whether the values involved are specified to the level of precision used for the comparison.

If either argument is `null`, the result is `null`.

9.5.7 Matches

Signature:

```
matches<T>(left T, right T) Boolean
```

Description:

The `matches` operator returns `true` if the arguments are same value, or if they are both `null`; and `false` otherwise.

For tuple types, this means that two tuple values match if and only if the tuples are of the same type, and the values for all elements by name match.

For list types, this means that two list values match if and only if the lists contain elements of the same type, have the same number of elements, and for each element in the lists, in order, the elements match.

For interval types, this means that two intervals match if and only if the intervals are over the same point type, and the starting and ending points of the intervals as determined by the `Start` and `End` operators match.

For `Code` values, matching is defined based on the code, system, and version elements only. The display element is ignored for the purposes of determining `Code` matching.

For `Concept` values, matching is defined as a non-empty intersection of the codes in each `Concept`.

Note that this operator will always return `true` or `false`, even if either or both of its arguments are `null`, or contain `null` components.

This operator, and the corresponding notion of *matching*, are used throughout CQL to define the behavior of membership and containment operators such as `in`, `contains`, `includes`, `IndexOf()`, etc. This provides consistent and intuitive behavior in the presence of missing information in list and membership contexts.

9.5.8 Not Equal

Signature:

```
<><T>(left T, right T) Boolean
```

Description:

The *not equal* (`<>`) operator returns true if its arguments are not the same value.

The *not equal* operator is a shorthand for invocation of logical negation (`not`) of the *equal* operator.

9.6   Arithmetic Operators

9.6.1 Abs

Signature:

```
Abs(argument Integer) Integer
Abs(argument Decimal) Decimal
Abs(argument Quantity) Quantity
```

Description:

The `Abs` operator returns the absolute value of its argument.

When taking the absolute value of a quantity, the unit is unchanged.

If the argument is `null`, the result is `null`.

9.6.2 Add

Signature:

```
+(left Integer, right Integer) Integer
+(left Decimal, right Decimal) Decimal
+(left Quantity, right Quantity) Quantity
```

Description:

The *add* (`+`) operator performs numeric addition of its arguments.

When invoked with mixed `Integer` and `Decimal` arguments, the `Integer` argument will be implicitly converted to `Decimal`.

When adding quantities, the dimensions of each quantity must match, but not necessarily the unit. For example, units of `'cm'` and `'m'` can be added, but units of `'cm2'` and `'cm'` cannot. The unit of the result will be the most granular unit of either input.

If either argument is `null`, the result is `null`.

9.6.3 Ceiling

Signature:

```
Ceiling(argument Decimal) Integer
```

Description:

The `Ceiling` operator returns the first integer greater than or equal to the argument.

When invoked with an `Integer` argument, the argument will be implicitly converted to `Decimal`.

If the argument is `null`, the result is `null`.

9.6.4 Divide

Signature:

```
/(left Decimal, right Decimal) Decimal
/(left Quantity, right Decimal) Quantity
/(left Quantity, right Quantity) Quantity
```

Description:

The *divide* (*/*) operator performs numeric division of its arguments. Note that this operator is `Decimal` division; for `Integer` division, use the *truncated divide* (`div`) operator.

When invoked with `Integer` arguments, the arguments will be implicitly converted to `Decimal`.

For division operations involving quantities, the input quantities must be in compatible dimensions (not necessarily the same units), and the resulting quantity will have the appropriate unit. For example:

```
12 'cm2' / 3 'cm'
```

In this example, the result will have a unit of `'cm'`.

If either argument is `null`, the result is `null`.

9.6.5 Floor

Signature:

```
Floor(argument Decimal) Integer
```

Description:

The `Floor` operator returns the first integer less than or equal to the argument.

When invoked with an `Integer` argument, the argument will be implicitly converted to `Decimal`.

If the argument is `null`, the result is `null`.

9.6.6 Log

Signature:

```
Log(argument Decimal, base Decimal) Decimal
```

Description:

The `Log` operator computes the logarithm of its first argument, using the second argument as the base.

When invoked with `Integer` arguments, the arguments will be implicitly converted to `Decimal`.

If either argument is `null`, the result is `null`.

9.6.7 Ln

Signature:

```
Ln(argument Decimal) Decimal
```

Description:

The `Ln` operator computes the natural logarithm of its argument.

When invoked with an `Integer` argument, the argument will be implicitly converted to `Decimal`.

If the argument is `null`, the result is `null`.

9.6.8 Maximum

Signature:

```
maximum<T>() T
```

Description:

The `maximum` operator returns the maximum representable value for the given type.

The `maximum` operator is defined for the `Integer`, `Decimal`, `DateTime`, and `Time` types.

For `Integer`, `maximum` returns the maximum signed 32-bit integer, 2^{31} - 1.

For `Decimal`, `maximum` returns the maximum representable decimal value, $(10^{37} - 1) / 10^{8}$ (99999999999999999999999999.99999999).

For `DateTime`, `maximum` returns the maximum representable date/time value, `DateTime(9999, 12, 31, 23, 59, 59, 999)`.

For `Time`, `maximum` returns the maximum representable time value, `Time(23, 59, 59, 999)`.

For any other type, attempting to invoke `maximum` results in an error.

9.6.9 Minimum

Signature:

```
minimum<T>() T
```

Description:

The `minimum` operator returns the minimum representable value for the given type.

The `minimum` operator is defined for the `Integer`, `Decimal`, `DateTime`, and `Time` types.

For `Integer`, `MinValue` returns the minimum signed 32-bit integer, -2^{31}.

For `Decimal`, `MinValue` returns the minimum representable decimal value, $(-10^{37} - 1) / 10^8$ (-99999999999999999999999999.99999999).

For `DateTime`, `minimum` returns the minimum representable date/time value, `DateTime(1, 1, 1, 0, 0, 0, 0)`.

For `Time`, `minimum` returns the minimum representable time value, `Time(0, 0, 0, 0)`.

For any other type, attempting to invoke `minimum` results in an error.

9.6.10   Modulo

Signature:

```
mod(left Integer, right Integer) Integer
mod(left Decimal, right Decimal) Decimal
```

Description:

The `mod` operator computes the remainder of the division of its arguments.

When invoked with mixed `Integer` and `Decimal` arguments, the `Integer` argument will be implicitly converted to `Decimal`.

If either argument is `null`, the result is `null`.

9.6.11   Multiply

Signature:

```
*(left Integer, right Integer) Integer
*(left Decimal, right Decimal) Decimal
*(left Decimal, right Quantity) Decimal
*(left Quantity, right Decimal) Decimal
*(left Quantity, right Quantity) Quantity
```

Description:

The *multiply* (*) operator performs numeric multiplication of its arguments.

When invoked with mixed `Integer` and `Decimal` arguments, the `Integer` argument will be implicitly converted to `Decimal`.

For multiplication operations involving quantities, the input quantities must be of the same dimension (not necessarily the same units), and the resulting quantity will have the appropriate unit. For example:

```
12 'cm' * 3 'cm'
```

In this example, the result will have a unit of `'cm2'`.

If either argument is `null`, the result is `null`.

9.6.12   Negate

Signature:

```
-(argument Integer) Integer
-(argument Decimal) Decimal
-(argument Quantity) Quantity
```

Description:

The *negate* (-) operator returns the negative of its argument.

When negating quantities, the unit is unchanged.

If the argument is `null`, the result is `null`.

9.6.13   Predecessor

Signature:

```
predecessor of<T>(argument T) T
```

Description:

The `predecessor` operator returns the predecessor of the argument. For example, the predecessor of 2 is 1. If the argument is already the minimum value for the type, a run-time error is thrown.

The `predecessor` operator is defined for the `Integer`, `Decimal`, `DateTime`, and `Time` types.

For `Integer`, `predecessor` is equivalent to subtracting 1.

For `Decimal`, `predecessor` is equivalent to subtracting the minimum precision value for the `Decimal` type, or 10^{-08}.

For `DateTime` and `Time` values, `predecessor` is equivalent to subtracting a time-unit quantity for the lowest specified precision of the value. For example, if the `DateTime` is fully specified, `predecessor` is equivalent to subtracting 1 millisecond; if the `DateTime` is specified to the second, `predecessor` is equivalent to subtracting one second, etc.

If the argument is `null`, the result is `null`.

9.6.14   Power

Signature:

```
^(argument Integer, exponent Integer) Integer
^(argument Decimal, exponent Decimal) Decimal
```

Description:

The *power* (^) operator raises the first argument to the power given by the second argument.

When invoked with mixed `Integer` and `Decimal` arguments, the `Integer` argument will be implicitly converted to `Decimal`.

If either argument is `null`, the result is `null`.

9.6.15   Round

Signature:

```
Round(argument Decimal) Decimal
Round(argument Decimal, precision Integer) Decimal
```

Description:

The `Round` operator returns the nearest whole number to its argument. The semantics of round are defined as a traditional round, meaning that a decimal value of 0.5 or higher will round to 1.

When invoked with an `Integer` argument, the argument will be implicitly converted to `Decimal`.

If the argument is `null`, the result is `null`.

Precision determines the decimal place at which the rounding will occur. If precision is not specified or `null`, 0 is assumed.

9.6.16　　Subtract

Signature:

```
-(left Integer, right Integer) Integer
-(left Decimal, right Decimal) Decimal
-(left Quantity, right Quantity) Quantity
```

Description:

The *subtract* (-) operator performs numeric subtraction of its arguments.

When invoked with mixed `Integer` and `Decimal` arguments, the `Integer` argument will be implicitly converted to `Decimal`.

When subtracting quantities, the dimensions of each quantity must match, but not necessarily the unit. For example, units of `'cm'` and `'m'` can be subtracted, but units of `'cm2'` and `'cm'` cannot. The unit of the result will be the most granular unit of either input.

If either argument is `null`, the result is `null`.

9.6.17　　Successor

Signature:

```
successor of<T>(argument T) T
```

Description:

The `successor` operator returns the successor of the argument. For example, the successor of 1 is 2. If the argument is already the maximum value for the type, a run-time error is thrown.

The `successor` operator is defined for the `Integer`, `Decimal`, `DateTime`, and `Time` types.

For `Integer`, `successor` is equivalent to adding 1.

For `Decimal`, `successor` is equivalent to adding the minimum precision value for the `Decimal` type, or 10^{-08}.

For `DateTime` and `Time` values, `successor` is equivalent to adding a time-unit quantity for the lowest specified precision of the value. For example, if the `DateTime` is fully specified, `successor` is equivalent to adding 1 millisecond; if the `DateTime` is specified to the second, `successor` is equivalent to adding one second, etc.

　　　　　　　　　　HL7 Standard: Clinical Quality Language, Release 1 - US Realm
　　　　　　　　　　May 2015

If the argument is `null`, the result is `null`.

9.6.18 Truncate

Signature:

```
Truncate(argument Decimal) Integer
```

Description:

The `Truncate` operator returns the integer component of its argument.

When invoked with an `Integer` argument, the argument will be implicitly converted to `Decimal`.

If the argument is `null`, the result is `null`.

9.6.19 Truncated Divide

Signature:

```
div(left Integer, right Integer) Integer
div(left Decimal, right Decimal) Decimal
```

Description:

The `div` operator performs truncated division of its arguments.

When invoked with mixed `Integer` and `Decimal` arguments, the `Integer` argument will be implicitly converted to `Decimal`.

If either argument is `null`, the result is `null`.

9.7 String Operators

9.7.1 Combine

Signature:

```
Combine(source List<String>) String
Combine(source List<String>, separator String) String
```

Description:

The `Combine` operator combines a list of strings, optionally separating each string with the given separator.

If either argument is `null`, or any element in the source list of strings is `null`, the result is `null`.

9.7.2 Concatenate

Signature:

```
+(left String, right String) String
```

Description:

The *concatenate* (+) operator performs string concatenation of its arguments.

May 2015

If either argument is `null`, the result is `null`.

9.7.3 Indexer

Signature:

```
[](argument String, index Integer) String
```

Description:

The *indexer* (`[]`) operator returns the character at the indexth position in a string.

Indexes in strings are defined to be 1-based.

If either argument is `null`, the result is `null`.

If the index is greater than the length of the string being indexed, the result is `null`.

9.7.4 Length

Signature:

```
Length(argument String) Integer
```

Description:

The `Length` operator returns the number of characters in a string.

If the argument is `null`, the result is `null`.

9.7.5 Lower

Signature:

```
Lower(argument String) String
```

Description:

The `Lower` operator returns the lower case of its argument.

If the argument is `null`, the result is `null`.

9.7.6 PositionOf

Signature:

```
PositionOf(pattern String, argument String) Integer
```

Description:

The `PositionOf` operator returns the 1-based index of the given pattern in the given string.

If the pattern is not found, the result is 0.

If either argument is `null`, the result is `null`.

9.7.7 Split

Signature:

```
Split(stringToSplit String, separator String) List<String>
```

Description:

The Split operator splits a string into a list of strings using a separator.

If the stringToSplit argument is null, the result is null.

If the stringToSplit argument does not contain any appearances of the separator, the result is a list of strings containing one element that is the value of the stringToSplit argument.

9.7.8 Substring

Signature:

```
Substring(stringToSub String, startIndex Integer) String
Substring(stringToSub String, startIndex Integer, length Integer) String
```

Description:

The Substring operator returns the string within stringToSub, starting at the 1-based index startIndex, and consisting of length characters.

If length is ommitted, the substring returned starts at startIndex and continues to the end of stringToSub.

If stringToSub or startIndex is null, or startIndex is out of range, the result is null.

9.7.9 Upper

Signature:

```
Upper(argument String) String
```

Description:

The Upper operator returns the upper case of its argument.

If the argument is null, the result is null.

9.8 Date/Time Operators

9.8.1 Add

Signature:

```
+(left DateTime, right Quantity) DateTime
+(left Time, right Quantity) Time
```

Description:

The *add* (+) operator returns the value of the given date/time, incremented by the time-valued quantity, respecting variable length periods for calendar years and months.

For DateTime values, the quantity unit must be one of: years, months, days, hours, minutes, seconds, or milliseconds.

For `Time` values, the quantity unit must be one of: `hours`, `minutes`, `seconds`, or `milliseconds`.

The operation is performed by attempting to derive the highest granularity precision first, working down successive granularities to the granularity of the time-valued quantity. For example, the following addition:

```
DateTime(2014) + 24 months
```

This example results in the value `DateTime(2016)` even though the date/time value is not specified to the level of precision of the time-valued quantity.

If either argument is `null`, the result is `null`.

9.8.2 After

Signature:

```
after precision of(left DateTime, right DateTime) Boolean
after precision of(left Time, right Time) Boolean
```

Description:

The `after`-*precision-of* operator compares two date/time values to the specified precision to determine whether the first argument is the after the second argument. Precision must be one of: `year`, `month`, `day`, `hour`, `minute`, `second`, or `millisecond`.

For comparisons involving date/time or time values with imprecision, note that the result of the comparison may be `null`, depending on whether the values involved are specified to the level of precision used for the comparison.

If either or both arguments are `null`, the result is `null`.

9.8.3 Before

Signature:

```
before precision of(left DateTime, right DateTime) Boolean
before precision of(left Time, right Time) Boolean
```

Description:

The `before`-*precision-of* operator compares two date/time values to the specified precision to determine whether the first argument is the before the second argument. Precision must be one of: `year`, `month`, `day`, `hour`, `minute`, `second`, or `millisecond`.

For comparisons involving date/time or time values with imprecision, note that the result of the comparison may be `null`, depending on whether the values involved are specified to the level of precision used for the comparison.

If either or both arguments are `null`, the result is `null`.

9.8.4 DateTime

Signature:

```
DateTime(year Integer) DateTime
DateTime(year Integer, month Integer) DateTime
DateTime(year Integer, month Integer, day Integer) DateTime
DateTime(year Integer, month Integer, day Integer,
  hour Integer) DateTime
DateTime(year Integer, month Integer, day Integer,
  hour Integer, minute Integer) DateTime
DateTime(year Integer, month Integer, day Integer,
  hour Integer, minute Integer, second Integer) DateTime
DateTime(year Integer, month Integer, day Integer,
  hour Integer, minute Integer, second Integer, millisecond Integer) DateTime
DateTime(year Integer, month Integer, day Integer,
  hour Integer, minute Integer, second Integer, millisecond Integer,
  timezoneOffset Decimal) DateTime
```

Description:

The `DateTime` operator constructs a date/time value from the given components.

At least one component other than `timezoneOffset` must be specified, and no component may be specified at a precision below an unspecified precision. For example, `hour` may be `null`, but if it is, `minute`, `second`, and `millisecond` must all be `null` as well.

If `timezoneOffset` is not specified, it is defaulted to the timezone offset of the evaluation request.

9.8.5 Date/Time Component From

Signature:

```
precision from(argument DateTime) Integer
precision from(argument Time) Integer
timezone from(argument DateTime) Decimal
timezone from(argument Time) Decimal
date from(argument DateTime) DateTime
time from(argument DateTime) Time
```

Description:

The *component-from* operator returns the specified component of the argument.

For `DateTime` values, *precision* must be one of: `year`, `month`, `day`, `hour`, `minute`, `second`, or `millisecond`.

For `Time` values, *precision* must be one of: `hour`, `minute`, `second`, or `millisecond`.

If the argument is `null`, or is not specified to the level of precision being extracted, the result is `null`.

9.8.6 Difference

Signature:

```
difference in precision between(low DateTime, high DateTime) Integer
difference in precision between(low Time, high Time) Integer
```

Description:

The *difference-between* operator returns the number of boundaries crossed for the specified precision between the first and second arguments. If the first argument is after the second argument, the result is negative. The result of this operation is always an integer; any fractional boundaries are dropped.

For `DateTime` values, *precision* must be one of: `years`, `months`, `days`, `hours`, `minutes`, `seconds`, or `milliseconds`.

For `Time` values, *precision* must be one of: `hours`, `minutes`, `seconds`, or `milliseconds`.

If either argument is `null`, the result is `null`.

9.8.7 Duration

Signature:

```
duration between(low DateTime, high DateTime) Integer
duration between(low Time, high Time) Integer
```

Description:

The *duration-between* operator returns the number of whole calendar periods for the specified precision between the first and second arguments. If the first argument is after the second argument, the result is negative. The result of this operation is always an integer; any fractional periods are dropped.

For `DateTime` values, *duration* must be one of: `years`, `months`, `days`, `hours`, `minutes`, `seconds`, or `milliseconds`.

For `Time` values, *duration* must be one of: `hours`, `minutes`, `seconds`, or `milliseconds`.

If either argument is `null`, the result is `null`.

9.8.8 Now

Signature:

```
Now() DateTime
```

Description:

The `Now` operator returns the date and time of the start timestamp associated with the evaluation request. `Now` is defined in this way for two reasons:

1. The operation will always return the same value within any given evaluation, ensuring that the result of an expression containing Now will always return the same result.

2. The operation will return the timestamp associated with the evaluation request, allowing the evaluation to be performed with the same timezone information as the data delivered with the evaluation request.

9.8.9 Same As

Signature:

```
same precision as(left DateTime, right DateTime) Boolean
same precision as(left Time, right Time) Boolean
```

Description:

The same-*precision*-as operator compares two date/time values to the specified precision for equality. Individual component values are compared starting from the year component down to the specified precision. If all values are specified and have the same value for each component, then the result is true. If a compared component is specified in both dates, but the values are not the same, then the result is false. Otherwise the result is null, as there is not enough information to make a determination.

For DateTime values, *precision* must be one of: year, month, day, hour, minute, second, or millisecond.

For Time values, *precision* must be one of: hour, minute, second, or millisecond.

For comparisons involving date/time or time values with imprecision, note that the result of the comparison may be null, depending on whether the values involved are specified to the level of precision used for the comparison.

If either or both arguments are null, the result is null.

9.8.10   Same Or After

Signature:

```
same precision or after(left DateTime, right DateTime) Boolean
same precision or after(left Time, right Time) Boolean
```

Description:

The same-*precision*-or after operator compares two date/time values to the specified precision to determine whether the first argument is the same or after the second argument.

For DateTime values, *precision* must be one of: year, month, day, hour, minute, second, or millisecond.

For Time values, *precision* must be one of: hour, minute, second, or millisecond.

For comparisons involving date/time or time values with imprecision, note that the result of the comparison may be null, depending on whether the values involved are specified to the level of precision used for the comparison.

If either or both arguments are null, the result is null.

9.8.11   Same Or Before

Signature:

```
same precision or before(left DateTime, right DateTime) Boolean
same precision or before(left DateTime, right DateTime) Boolean
```

Description:

The `same-`*precision-*`or before` operator compares two date/time values to the specified precision to determine whether the first argument is the same or before the second argument.

For `DateTime` values, *precision* must be one of: `year`, `month`, `day`, `hour`, `minute`, `second`, or `millisecond`.

For `Time` values, *precision* must be one of: `hour`, `minute`, `second`, or `millisecond`.

For comparisons involving date/time or time values with imprecision, note that the result of the comparison may be `null`, depending on whether the values involved are specified to the level of precision used for the comparison.

If either or both arguments are `null`, the result is `null`.

9.8.12 Subtract

Signature:

```
-(left DateTime, right Quantity) DateTime
-(left Time, right Quantity) Time
```

Description:

The *subtract* (`-`) operator returns the value of the given date/time, decremented by the time-valued quantity, respecting variable length periods for calendar years and months.

For `DateTime` values, the quantity unit must be one of: `years`, `months`, `days`, `hours`, `minutes`, `seconds`, or `milliseconds`.

For `Time` values, the quantity unit must be one of: `hours`, `minutes`, `seconds`, or `milliseconds`.

The operation is performed by attempting to derive the highest granularity precision first, working down successive granularities to the granularity of the time-valued quantity. For example, the following subtraction:

```
DateTime(2014) - 24 months
```

This example results in the value `DateTime(2012)` even though the date/time value is not specified to the level of precision of the time-valued quantity.

If either argument is `null`, the result is `null`.

9.8.13 Time

Signature:

```
Time(hour Integer) Time
Time(hour Integer, minute Integer) Time
Time(hour Integer, minute Integer, second Integer) Time
Time(hour Integer, minute Integer, second Integer, millisecond Integer) Time
Time(hour Integer, minute Integer, second Integer, millisecond Integer,
  timezoneOffset Decimal) Time
```

Description:

The `Time` operator constructs a time value from the given components.

At least one component other than `timezoneOffset` must be specified, and no component may be specified at a precision below an unspecified precision. For example, `minute` may be `null`, but if it is, `second`, and `millisecond` must all be `null` as well.

If `timezoneOffset` is not specified, it is defaulted to the timezone offset of the evaluation request.

9.8.14 TimeOfDay

Signature:

```
TimeOfDay() Time
```

Description:

The `TimeOfDay` operator returns the time of day of the start timestamp associated with the evaluation request. See the `Now` operator for more information on the rationale for defining the `TimeOfDay` operator in this way.

9.8.15 Today

Signature:

```
Today() DateTime
```

Description:

The `Today` operator returns the date (with no time component) of the start timestamp associated with the evaluation request. See the `Now` operator for more information on the rationale for defining the `Today` operator in this way.

9.9 Interval Operators

9.9.1 After

Signature:

```
after(left Interval<T>, right Interval<T>) Boolean
after(left T, right Interval<T>) Boolean
after(left Interval<T>, right T) Boolean
```

Description:

The `after` operator for intervals returns `true` if the first interval starts after the second one ends. In other words, if the starting point of the first interval is greater than the ending point of the second interval.

For the point-interval overload, the operator returns `true` if the given point is greater than the end of the interval.

For the interval-point overload, the operator returns `true` if the given interval starts after the given point.

This operator uses the semantics described in the `Start` and `End` operators to determine interval boundaries.

If either argument is `null`, the result is `null`.

9.9.2 Before

Signature:

```
before(left Interval<T>, right Interval<T>) Boolean
before(left T, right Interval<T>) Boolean
before(left interval<T>, right T) Boolean
```

Description:

The `before` operator for intervals returns `true` if the first interval ends before the second one starts. In other words, if the ending point of the first interval is less than the starting point of the second interval.

For the point-interval overload, the operator returns `true` if the given point is less than the start of the interval.

For the interval-point overload, the operator returns `true` if the given interval ends before the given point.

This operator uses the semantics described in the `Start` and `End` operators to determine interval boundaries.

If either argument is `null`, the result is `null`.

9.9.3 Collapse

Signature:

```
collapse(argument List<Interval<T>>) List<Interval<T>>
```

Description:

The `collapse` operator returns the unique set of intervals that completely covers the ranges present in the given list of intervals.

If the list of intervals is empty, the result is empty. If the list of intervals contains a single interval, the result is a list with that interval. If the list of intervals contains nulls, they will be excluded from the resulting list.

If the argument is `null`, the result is `null`.

9.9.4 Contains

Signature:

```
contains(argument Interval<T>, point T) Boolean
```

Description:

The `contains` operator for intervals returns `true` if the given point is greater than or equal to the starting point of the interval, and less than or equal to the ending point of the interval. For open interval boundaries, exclusive comparison operators are used. For closed interval boundaries, if the interval boundary is `null`, the result of the boundary comparison is considered `true`.

If either argument is `null`, the result is `null`.

9.9.5 End

Signature:

```
end of(argument Interval<T>) T
```

Description:

The `End` operator returns the ending point of an interval.

If the high boundary of the interval is open, this operator returns the `predecessor` of the high value of the interval. Note that if the high value of the interval is `null`, the result is `null`.

If the high boundary of the interval is closed and the high value of the interval is not null, this operator returns the high value of the interval. Otherwise, the result is the maximum value of the point type of the interval.

If the argument is `null`, the result is `null`.

9.9.6 Ends

Signature:

```
ends(left Interval<T>, right Interval<T>) Boolean
```

Description:

The `ends` operator returns `true` if the first interval ends the second. More precisely, if the starting point of the first interval is greater than or equal to the starting point of the second, and the ending point of the first interval is equal to the ending point of the second.

This operator uses the semantics described in the `start` and `end` operators to determine interval boundaries.

If either argument is `null`, the result is `null`.

9.9.7 Equal

Signature:

```
=(left Interval<T>, right Interval<T>) Boolean
```

Description:

The *equal* (`=`) operator for intervals returns `true` if and only if the intervals are over the same point type, and they have the same value for the starting and ending points of the intervals as determined by the `Start` and `End` operators.

If either argument is `null`, the result is `null`.

9.9.8 Except

Signature:

```
except(left Interval<T>, right Interval<T>) Interval<T>
```

Description:

May 2015

The `except` operator for intervals returns the set difference of two intervals. More precisely, this operator returns the portion of the first interval that does not overlap with the second. Note that to avoid returning an improper interval, if the second argument is properly contained within the first and does not start or end it, this operator returns `null`.

If either argument is `null`, the result is `null`.

9.9.9 In

Signature:

```
in(point T, argument Interval<T>) Boolean
```

Description:

The `in` operator for intervals returns `true` if the given point is greater than or equal to the starting point of the interval, and less than or equal to the ending point of the interval. For open interval boundaries, exclusive comparison operators are used. For closed interval boundaries, if the interval boundary is `null`, the result of the boundary comparison is considered true.

If either argument is `null`, the result is `null`.

9.9.10   Includes

Signature:

```
includes(left Interval<T>, right Interval<T>) Boolean
```

Description:

The `includes` operator for intervals returns `true` if the first interval completely includes the second. More precisely, if the starting point of the first interval is less than or equal to the starting point of the second interval, and the ending point of the first interval is greater than or equal to the ending point of the second interval.

This operator uses the semantics described in the `Start` and `End` operators to determine interval boundaries.

If either argument is `null`, the result is `null`.

9.9.11   Included In

Signature:

```
included in(left Interval<T>, right Interval<T>) Boolean
```

Description:

The `included in` operator for intervals returns `true` if the first interval is completely included in the second. More precisely, if the starting point of the first interval is greater than or equal to the starting point of the second interval, and the ending point of the first interval is less than or equal to the ending point of the second interval.

This operator uses the semantics described in the `Start` and `End` operators to determine interval boundaries.

If either argument is `null`, the result is `null`.

9.9.12   Intersect

Signature:

```
intersect(left Interval<T>, right Interval<T>) Boolean
```

Description:

The `intersect` operator for intervals returns the intersection of two intervals. More precisely, the operator returns the interval that defines the overlapping portion of both arguments. If the arguments do not overlap, this operator returns `null`.

If either argument is `null`, the result is `null`.

9.9.13   Matches

Signature:

```
matches(left Interval<T>, right Interval<T>) Boolean
```

Description:

The `matches` operator for intervals returns `true` if and only if the intervals are over the same point type, and the starting and ending points of the intervals as determined by the `Start` and `End` operators match.

9.9.14   Meets

Signature:

```
meets(left Interval<T>, right Interval<T>) Boolean
meets before(left Interval<T>, right Interval<T>) Boolean
meets after(left Interval<T>, right Interval<T>) Boolean
```

Description:

The `meets` operator returns `true` if the first interval ends immediately before the second interval starts, or if the first interval starts immediately after the second interval ends. In other words, if the ending point of the first interval is equal to the predecessor of the starting point of the second, or if the starting point of the first interval is equal to the successor of the ending point of the second.

The `meets before` operator returns `true` if the first interval ends immediately before the second interval starts, while the `meets after` operator returns `true` if the first interval starts immediately after the second interval ends.

This operator uses the semantics described in the `Start` and `End` operators to determine interval boundaries.

If either argument is `null`, the result is `null`.

9.9.15   Not Equal

Signature:

```
<>(left Interval<T>, right Interval<T>) : Boolean
```

Description:

The *not equal* (`<>`) operator for intervals returns `true` if its arguments are not the same value.

The *not equal* operator is a shorthand for invocation of logical negation (`not`) of the *equal* operator.

9.9.16 Overlaps

Signature:

```
overlaps(left Interval<T>, right Interval<T>) Boolean
overlaps before(left Interval<T>, right Interval<T>) Boolean
overlaps after(left Interval<T>, right Interval<T>) Boolean
```

Description:

The `overlaps` operator returns `true` if the first interval overlaps the second. More precisely, if the ending point of the first interval is greater than or equal to the starting point of the second interval, and the starting point of the first interval is less than or equal to the ending point of the second interval.

The operator `overlaps before` returns `true` if the first interval overlaps the second and starts before it, while the `overlaps after` operator returns `true` if the first interval overlaps the second and ends after it.

This operator uses the semantics described in the `Start` and `End` operators to determine interval boundaries.

If either argument is `null`, the result is `null`.

9.9.17 Properly Includes

Signature:

```
properly includes(left Interval<T>, right Interval<T>) Boolean
```

Description:

The `properly includes` operator for intervals returns `true` if the first interval completely includes the second and the first interval is strictly larger than the second. More precisely, if the starting point of the first interval is less than or equal to the starting point of the second interval, and the ending point of the first interval is greater than or equal to the ending point of the second interval, and they are not the same interval.

This operator uses the semantics described in the `Start` and `End` operators to determine interval boundaries.

If either argument is `null`, the result is `null`.

9.9.18 Properly Included In

Signature:

May 2015

```
properly included in(left Interval<T>, right Interval<T>) Boolean
```

Description:

The `properly included in` operator for intervals returns `true` if the first interval is completely included in the second and the first interval is strictly smaller than the second. More precisely, if the starting point of the first interval is greater than or equal to the starting point of the second interval, and the ending point of the first interval is less than or equal to the ending point of the second interval, and they are not the same interval.

This operator uses the semantics described in the `Start` and `End` operators to determine interval boundaries.

If either argument is `null`, the result is `null`.

9.9.19   Start

Signature:

```
start of(argument Interval<T>) T
```

Description:

The `Start` operator returns the starting point of an interval.

If the low boundary of the interval is open, this operator returns the `successor` of the low value of the interval. Note that if the low value of the interval is `null`, the result is `null`.

If the low boundary of the interval is closed and the low value of the interval is not null, this operator returns the low value of the interval. Otherwise, the result is the minimum value of the point type of the interval.

If the argument is `null`, the result is `null`.

9.9.20   Starts

Signature:

```
starts(left Interval<T>, right Interval<T>) Boolean
```

Description:

The `starts` operator returns `true` if the first interval starts the second. More precisely, if the starting point of the first is equal to the starting point of the second interval and the ending point of the first interval is less than or equal to the ending point of the second interval.

This operator uses the semantics described in the `Start` and `End` operators to determine interval boundaries.

If either argument is `null`, the result is `null`.

9.9.21   Union

Signature:

```
union(left Interval<T>, right Interval<T>) Interval<T>
```

Description:

The `union` operator for intervals returns the union of the intervals. More precisely, the operator returns the interval that starts at the earliest starting point in either argument, and ends at the latest starting point in either argument. If the arguments do not overlap or meet, this operator returns `null`.

If either argument is `null`, the result is `null`.

9.9.22 Width

Signature:

```
width of(argument Interval<T>) T
```

Description:

The `width` operator returns the width of an interval. The result of this operator is equivalent to invoking: (`start of` argument – `end of` argument) + *point-size.*

Note that because CQL defines *duration* and *difference* operations for date/time and time valued intervals, *width* is not defined for intervals of these types.

If the argument is `null`, the result is `null`.

9.10 List Operators

9.10.1 Contains

Signature:

```
contains(argument List<T>, element T) Boolean
```

Description:

The `contains` operator for lists returns `true` if the given element is in the list.

This operator uses the notion of *matching* to determine whether or not the element being searched for matches any element in the list. In particular this means that if the list contains a `null`, and the element being searched for is `null`, the result will be `true`.

If the list argument is `null`, the result is `null`.

9.10.2 Distinct

Signature:

```
distinct(argument List<T>) List<T>
```

Description:

The `distinct` operator returns the given list with duplicates eliminated.

This operator uses the notion of *matching* to determine whether two elements in the list are the same for the purposes of duplicate elimination. In particular this means that if the list contains multiple `null` elements, the result will only contain one `null` element.

If the argument is `null`, the result is `null`.

9.10.3   Equal

Signature:

```
=(left List<T>, right List<T>) Boolean
```

Description:

The *equal* (`=`) operator for lists returns `true` if and only if the lists have the same element type, and have the same elements by value, in the same order.

If either argument is `null`, or contains null elements, the result is `null`.

9.10.4   Except

Signature:

```
except(left List<T>, right List<T>) List<T>
```

Description:

The `except` operator returns the set difference of two lists. More precisely, the operator returns a list with the elements that appear in the first operand that do not appear in the second operand.

This operator uses the notion of *matching* to determine whether two elements are the same for the purposes of computing the difference.

If either argument is `null`, the result is `null`.

9.10.5   Exists

Signature:

```
exists(argument List<T>) Boolean
```

Description:

The `exists` operator returns `true` if the list contains any elements, including `null` elements.

If the argument is `null`, the result is `null`.

9.10.6   Expand

Signature:

```
expand(argument List<List<T>>) List<T>
```

Description:

The `expand` operator flattens a list of lists into a single list.

If the argument is `null`, the result is `null`.

9.10.7   First

Signature:

```
First(argument List<T>) T
```

Description:

The `First` operator returns the first element in a list. The operator is equivalent to invoking the indexer on the 1st index.

If the argument is `null`, the result is `null`.

9.10.8   In

Signature:

```
in(element T, argument List<T>) Boolean
```

Description:

The `in` operator for lists returns `true` if the given element is in the given list.

This operator uses the notion of *matching* to determine whether or not the element being searched for matches any element in the list. In particular this means that if the list contains a `null`, and the element being searched for is `null`, the result will be `true`.

If either argument is `null`, the result is `null`.

9.10.9   Includes

Signature:

```
includes(left List<T>, right List<T>) Boolean
```

Description:

The `includes` operator for lists returns `true` if the first list contains every element of the second list.

This operator uses the notion of *matching* to determine whether or not two elements are the same.

If either argument is `null`, the result is `null`.

Note that the order of elements does not matter for the purposes of determining inclusion.

9.10.10   Included In

Signature:

```
included in(left List<T>, right list<T>) Boolean
```

Description:

The `included in` operator for lists returns `true` if every element of the first list is in the second list.

This operator uses the notion of *matching* to determine whether or not two elements are the same.

If either argument is `null`, the result is `null`.

Note that the order of elements does not matter for the purposes of determining inclusion.

9.10.11 Indexer

Signature:

```
[](argument List<T>, index Integer) T
```

Description:

The *indexer* (`[]`) operator returns the element at the `index`th position in a list.

Indexes in lists are defined to be 1-based.

If the `index` is greater than the number of elements in the list, the result is `null`.

If either argument is `null`, the result is `null`.

9.10.12 IndexOf

Signature:

```
IndexOf(argument List<T>, element T) Integer
```

Description:

The `IndexOf` operator returns the 1-based index of the given element in the given source list.

The operator uses the notion of *matching* to determine the index. The search is linear, and returns the index of the first element that matches the element being searched for.

If the list is empty, or no element is found, the result is 0.

If the list argument is `null`, the result is `null`.

9.10.13 Intersect

Signature:

```
intersect(left List<T>, right List<T>) List<T>
```

Description:

The `intersect` operator for lists returns the intersection of two lists. More precisely, the operator returns a list containing only the elements that appear in both lists.

This operator uses the notion of *matching* to determine whether or not two elements are the same.

If either argument is `null`, the result is `null`.

9.10.14 Last

Signature:

```
Last(argument List<T>) T
```

Description:

The `Last` operator returns the last element in a list. In a list of length N, the operator is equivalent to invoking the indexer on the Nth element in the list.

If the argument is `null`, the result is `null`.

9.10.15 Length

Signature:

```
Length(argument List<T>) Integer
```

Description:

The `Length` operator returns the number of elements in a list.

If the argument is `null`, the result is `null`.

9.10.16 Matches

Signature:

```
matches(left List<T>, right List<T>) Boolean
```

Description:

The `matches` operator for lists returns `true` if and only if the lists contain elements of the same type, have the same number of elements, and for each element in the lists, in order, the elements match.

9.10.17 Not Equal

Signature:

```
<>(left List<T>, right List<T>) Boolean
```

Description:

The *not equal* (`<>`) operator for lists returns `true` if its arguments are not the same value.

The *not equal* operator is a shorthand for invocation of logical negation (`not`) of the *equal* operator.

9.10.18 Properly Includes

Signature:

```
properly includes(left List<T>, right List<T>) Boolean
```

Description:

The `properly includes` operator for lists returns `true` if the first list contains every element of the second list, and the first list is strictly larger than the second list.

This operator uses the notion of *matching* to determine whether or not two elements are the same.

If either argument is `null`, the result is `null`.

Note that the order of elements does not matter for the purposes of determining inclusion.

9.10.19 Properly Included In

Signature:

```
properly included in(left List<T>, right list<T>) Boolean
```

May 2015

Description:

The `properly included in` operator for lists returns `true` if every element of the first list is in the second list and the first list is strictly smaller than the second list.

This operator uses the notion of *matching* to determine whether or not two elements are the same.

If either argument is `null`, the result is `null`.

Note that the order of elements does not matter for the purposes of determining inclusion.

9.10.20   Singleton From

Signature:

```
singleton from(argument List<T>) T
```

Description:

The `singleton from` operator extracts a single element from the source list. If the source list is empty, the result is `null`. If the source list contains one element, that element is returned. If the list contains more than one element, a run-time error is thrown.

If the source list is `null`, the result is `null`.

9.10.21   Union

Signature:

```
union(left List<T>, right List<T>) List<T>
```

Description:

The `union` operator for lists returns a list with all elements from both arguments. Note that duplicates are not eliminated during this process, if an element appears once in both sources, that element will be present twice in the resulting list.

If either argument is `null`, the result is `null`.

9.11 Aggregate Functions

9.11.1   AllTrue

Signature:

```
AllTrue(argument List<Boolean>) Boolean
```

Description:

The `AllTrue` operator returns `true` if all the non-null elements in the source are `true`.

If the source contains no non-null elements, `true` is returned.

If the source is `null`, the result is `null`.

9.11.2   AnyTrue

Signature:

```
AnyTrue(argument List<Boolean>) Boolean
```

Description:

The AnyTrue operator returns true if any non-null element in the source is true.

If the source contains no non-null elements, false is returned.

If the source is null, the result is null.

9.11.3   Avg

Signature:

```
Avg(argument List<Decimal>) Decimal
Avg(argument List<Quantity>) Quantity
```

Description:

The Avg operator returns the average of the non-null elements in the source.

If the source contains no non-null elements, null is returned.

If the source is null, the result is null.

9.11.4   Count

Signature:

```
Count(argument List<T>) Integer
```

Description:

The Count operator returns the number of non-null elements in the source. If the list contains no non-null elements, the result is 0. If the list is null, the result is null.

9.11.5   Max

Signature:

```
Max(argument List<Integer>) Integer
Max(argument List<Decimal>) Decimal
Max(argument List<Quantity>) Quantity
Max(argument List<DateTime>) DateTime
Max(argument List<Time>) Time
Max(argument List<String>) String
```

Description:

The Max operator returns the maximum element in the source.

If the source contains no non-null elements, null is returned.

If the source is null, the result is null.

9.11.6 Min

Signature:

```
Min(argument List<Integer>) Integer
Min(argument List<Decimal>) Decimal
Min(argument List<Quantity>) Quantity
Min(argument List<DateTime>) DateTime
Min(argument List<Time>) Time
Min(argument List<String>) String
```

Description:

The `Min` operator returns the minimum element in the source.

If the source contains no non-null elements, `null` is returned.

If the source is `null`, the result is `null`.

9.11.7 Median

Signature:

```
Median(argument List<Decimal>) Decimal
Median(argument List<Quantity>) Quantity
```

Description:

The `Median` operator returns the median of the elements in source.

If the source contains no non-null elements, `null` is returned.

If the source is `null`, the result is `null`.

9.11.8 Mode

Signature:

```
Mode(argument List<T>) T
```

Description:

The `Mode` operator returns the statistical mode of the elements in source.

If the source contains no non-null elements, `null` is returned.

If the source is `null`, the result is `null`.

9.11.9 Population StdDev

Signature:

```
PopulationStdDev(argument List<Decimal>) Decimal
PopulationStdDev(argument List<Quantity>) Quantity
```

Description:

The `PopulationStdDev` operator returns the statistical standard deviation of the elements in source.

If the source contains no non-null elements, `null` is returned.

If the source is null, the result is null.

9.11.10   Population Variance

Signature:

```
PopulationVariance(argument List<Decimal>) Decimal
PopulationVariance(argument List<Quantity>) Quantity
```

Description:

The PopulationVariance operator returns the statistical population variance of the elements in source.

If the source contains no non-null elements, null is returned.

If the source is null, the result is null.

9.11.11   StdDev

Signature:

```
StdDev(argument List<Decimal>) Decimal
StdDev(argument List<Quantity>) Quantity
```

Description:

The StdDev operator returns the statistical standard deviation of the elements in source.

If the source contains no non-null elements, null is returned.

If the list is null, the result is null.

9.11.12   Sum

Signature:

```
Sum(argument List<Integer>) Integer
Sum(argument List<Decimal>) Decimal
Sum(argument List<Quantity>) Quantity
```

Description:

The Sum operator returns the sum of non-null elements in the source.

If the source contains no non-null elements, null is returned.

If the list is null, the result is null.

9.11.13   Variance

Signature:

```
Variance(argument List<Decimal>) Decimal
Variance(argument List<Quantity>) Quantity
```

Description:

The Variance operator returns the statistical variance of the elements in source.

If the source contains no non-null elements, `null` is returned.

If the source is `null`, the result is `null`.

9.12 Clinical Operators

9.12.1   Age

Signature:

```
AgeInYears() Integer
AgeInMonths() Integer
AgeInDays() Integer
AgeInHours() Integer
AgeInMinutes() Integer
AgeInSeconds() Integer
```

Description:

The `Age` operators calculate the age of the patient as of now in the precision named in the operator.

If the patient's birthdate is `null`, the result is `null`.

The `Age` operators are defined in terms of a `DateTime` duration calculation. This means that if the age of the patient is not specified to the level of precision corresponding to the operator being invoked, the result will be an *uncertainty* over the range of possible values, potentially causing some comparisons to return `null`.

9.12.2   AgeAt

Signature:

```
AgeInYearsAt(asOf DateTime) Integer
AgeInMonthsAt(asOf DateTime) Integer
AgeInDaysAt(asOf DateTime) Integer
AgeInHoursAt(asOf DateTime) Integer
AgeInMinutesAt(asOf DateTime) Integer
AgeInSecondsAt(asOf DateTime) Integer
```

Description:

The `AgeAt` operators calculate the age of the patient as of the given date in the precision named in the operator.

If the patient's birthdate is `null`, or the `asOf` argument is `null`, the result is `null`.

The `AgeAt` operators are defined in terms of a `DateTime` duration calculation. This means that if the age of the patient or the given `asOf` value are not specified to the level of precision corresponding to the operator being invoked, the will be an *uncertainty* over the range of possible values, potentially causing some comparisons to return `null`.

9.12.3   CalculateAge

Signature:

```
CalculateAgeInYears(birthDate DateTime) Integer
CalculateAgeInMonths(birthDate DateTime) Integer
CalculateAgeInDays(birthDate DateTime) Integer
CalculateAgeInHours(birthDate DateTime) Integer
CalculateAgeInMinutes(birthDate DateTime) Integer
CalculateAgeInSeconds(birthDate DateTime) Integer
```

Description:

The `CalculateAge` operators calculate the age of a person born on the given birthdate as of now in the precision named in the operator.

If the birthdate is `null`, the result is `null`.

The `CalculateAge` operators are defined in terms of a `DateTime` duration calculation. This means that if the given `birthDate` is not specified to the level of precision corresponding to the operator being invoked, the result will be an *uncertainty* over the range of possible values, potentially causing some comparisons to return `null`.

9.12.4   CalculateAgeAt

Signature:

```
CalculateAgeInYearsAt(birthDate DateTime, asOf DateTime) Integer
CalculateAgeInMonthsAt(birthDate DateTime, asOf DateTime) Integer
CalculateAgeInDaysAt(birthDate DateTime, asOf DateTime) Integer
CalculateAgeInHoursAt(birthDate DateTime, asOf DateTime) Integer
CalculateAgeInMinutesAt(birthDate DateTime, asOf DateTime) Integer
CalculateAgeInSecondsAt(birthDate DateTime, asOf DateTime) Integer
```

Description:

The `CalculateAgeAt` operators calculate the age of a person born on the given birthdate as of the given date in the precision named in the operator.

If the `birthDate` is `null` or the `asOf` argument is `null`, the result is `null`.

The `CalculateAgeAt` operators are defined in terms of a `DateTime` duration calculation. This means that if the given `birthDate` or `asOf` are not specified to the level of precision corresponding to the operator being invoked, the result will be an *uncertainty* over the range of possible values, potentially causing some comparisons to return `null`.

9.12.5   Equal

Signature:

```
=(left Code, right Code) Boolean
=(left Concept, right Concept) Boolean
```

Description:

The *equal* (=) operator for Codes and Concepts uses tuple equality semantics. This means that the operator will return `true` if and only if the values for each element by name are equal.

If either argument is `null`, or contains any `null` components, the result is `null`.

9.12.6 Matches

Signature:

```
matches(left Code, right Code) Boolean
```

Description:

The `matches` operator for `Code` values returns `true` if the `code`, `system`, and `version` elements match. The `display` element is ignored for the purposes of determining `Code` matching.

For `Concept` values, matching is defined as a non-empty intersection of the codes in each `Concept`. The `display` element is ignored for the purposes of determining `Concept` matching.

Note that this operator will always return `true` or `false`, even if either or both of its arguments are `null`, or contain `null` components.

Note carefully that this notion of *matching* is *not* the same as the notion of equivalence used in terminology: "these codes represent the same concept." CQL specifically avoids defining terminological equivalence. The notion of matching defined here is used to provide consistent and intuitive semantics when dealing with missing information in membership contexts.

9.12.7 In (Codesystem)

Signature:

```
in(code String, codesystem CodeSystemRef) Boolean
in(code Code, codesystem CodeSystemRef) Boolean
in(concept Concept, codesystem CodeSystemRef) Boolean
```

Description:

The `in` (Codesystem) operators determine whether or not a given code is in a particular codesystem. Note that these operators can only be invoked by referencing a defined `codesystem`.

For the `String` overload, if the given code system contains a code with a matching code element, the result is `true`.

For the `Code` overload, if the given code system contains a matching code, the result is `true`.

For the `Concept` overload, if the given code system contains a code matching any code in the given concept, the result is `true`.

If the `code` argument is `null`, the result is `null`.

9.12.8 In (Valueset)

Signature:

```
in(code String, valueset ValueSetRef) Boolean
in(code Code, valueset ValueSetRef) Boolean
in(concept Concept, valueset ValueSetRef) Boolean
```

Description:

The `in` (Valueset) operators determine whether or not a given code is in a particular valueset. Note that these operators can only be invoked by referencing a defined `valueset`.

For the `String` overload, if the given valueset contains a code with a matching code element, the result is `true`.

For the `Code` overload, if the given valueset contains a matching code, the result is `true`.

For the `Concept` overload, if the given valueset contains a code matching any code in the given concept, the result is `true`.

If the `code` argument is `null`, the result is `null`.

10 APPENDIX C – REFERENCE IMPLEMENTATIONS

As part of the Clinical Quality Framework effort, reference implementations of a CQL-ELM translator as well as a native ELM execution engine are in progress. This appendix provides a brief overview of where to find more information on these reference implementations.

10.1 CQL-ELM Translator Reference Implementation

The CQL-ELM Translator is a reference implementation for the translation of text-based CQL library documents into an XML or JSON representation using the ELM. The implementation is intended to be used in CQF pilots and eventually integrated into production authoring environments for both Clinical Decision Support and Clinical Quality Measurement. The implementation can also be used as the first step in a process to enable distribution, translation, execution, and integration of CQL-based quality artifacts.

The CQL-ELM Translator is licensed under the open source Apache Version 2.0 license, and available as part of the clinical_quality_language project on GitHub: https://github.com/cqframework/clinical_quality_language.

For an overview of the project, along with current status, refer to the following document:

https://github.com/cqframework/clinical_quality_language/blob/master/Src/java/cql-to-elm/OVERVIEW.md

10.2 CQL Execution Framework Reference Implementation

A reference implementation for executing CQL is currently under development. This reference implementation is intended to be used in CQF pilots and eventually integrated into production eCQM testing and certification tools.

The CQL execution framework is licensed under the open source Apache Version 2.0 license, and available as part of the clinical_quality_language project on GitHub: https://github.com/cqframework/clinical_quality_language.

For an overview of the project along with current status, refer to the following document:

https://github.com/cqframework/clinical_quality_language/blob/master/Src/coffeescript/cql-execution/OVERVIEW.md

11 APPENDIX D – REFERENCES

1. *Clinical Quality Framework Use Cases.*
   http://wiki.siframework.org/Clinical+Quality+Framework+Use+Cases

2. *HL7 Version 3 Standard: Clinical Decision Support Knowledge Artifact Specification, Release 1.2.* http://www.hl7.org/implement/standards/product_brief.cfm?product_id=337

3. *HL7 Version 3 Standard: Representation of the Health Quality Measure Format (eMeasure) DSTU, Release 2.*
   http://www.hl7.org/implement/standards/product_brief.cfm?product_id=97

4. *HL7 Version 3 Implementation Guide: Quality Data Model (QDM)-based Health Quality Measure Format (HQMF), Release 1 - US Realm.*
   http://www.hl7.org/implement/standards/product_brief.cfm?product_id=346

5. *HL7 Domain Analysis Model: Harmonization of Health Quality Artifact Reasoning and Expression Logic.*
   http://www.hl7.org/implement/standards/product_brief.cfm?product_id=359

6. *HL7 Version 3 Standard: Decision Support Service, Release 2.*
   http://www.hl7.org/implement/standards/product_brief.cfm?product_id=12

7. *HL7 Version 3 Implementation Guide: Decision Support Service, Release 1.*
   http://www.hl7.org/implement/standards/product_brief.cfm?product_id=334

8. *Health Level Seven Arden Syntax for Medical Logic Systems, Version 2.10.*
   http://www.hl7.org/implement/standards/product_brief.cfm?product_id=372

9. *HL7 Version 3 Standard: GELLO; A Common Expression Language, Release 2.*
   http://www.hl7.org/implement/standards/product_brief.cfm?product_id=5

10. *HL7 Fast Healthcare Interoperability Resources Specification (FHIR), Release 1.*
    http://www.hl7.org/implement/standards/product_brief.cfm?product_id=343

11. *ISO 8601:2004: Data elements and interchange formats -- Information Interchange -- Representation of dates and times.*
    http://www.iso.org/iso/catalogue_detail?csnumber=40874

12. *Quality Data Model, Version 4.1.1.*
    http://www.healthit.gov/sites/default/files/qdm_4_1_1.pdf

13. *Object Constraint Language, OMG Available Specification Version 2.4.*
    http://www.omg.org/spec/OCL/2.4/

14. *Foundations of Databases. Abiteboul, Hull, Vianu, 1995*

15. *Temporal Data and The Relational Model. Date, Darwen, Lorentzos, 2003*

16. *Databases, Types, and the Relational Model, 3rd edition. Date, Darwen, 2007*

17. *Compilers: Principles, Techniques, and Tools. Aho, Sethi, Ullman, 1998*

18. *Unicode Standard Annex #44: Unicode Character Database*
    http://www.unicode.org/reports/tr44/

19. *Common Terminology Services 2, Version 1.0.* http://www.omg.org/spec/CTS2/1.0/

12 APPENDIX E – ACRONYMS

Acronym	Definition/Description
AHRQ	Agency for Healthcare Research and Quality
ANTLR4	ANother Tool for Language Recognition (version 4)
CDA	Clinical Document Architecture
CDS	Clinical Decision Support
CDSC L3	Clinical Decision Support Consortium Level 3
CDS KAS	Clinical Decision Support Knowledge Artifact Specification
CMS	Centers for Medicare & Medicaid Services
CPT	Current Procedural Terminology
CQL	Clinical Quality Language
CQM	Clinical Quality Measure
CREF	Allscripts Common Rule Engine Format (CREF) specification
CTS2	Common Terminology Services 2
DAM	Domain Analysis Model
DSTU	Draft Standard for Trial Use
eCQI	Electronic Clinical Quality Improvement
eCQM	Electronic Clinical Quality Measure
EHR	Electronic Health Record
ELM	Expression Logical Model
EMR	Electronic Medical Record
eRecs	AHRQ Electronic Recommendations
FHIR	Fast Healthcare Interoperability Resources
GEM	Guidelines Element Model
HIE	Health Information Exchange
HIT	Health Information Technology
HITECH Act	Health Information Technology for Economic and Clinical Health Act
HIPAA	Health Insurance Portability and Accountability Act
HeD	Health eDecisions
HITSP	Health Information Technology Standards Panel
HL7	Health Level 7
HQMF	Health Quality Measure Format
ICD-9-CM	International Classification of Diseases, Ninth Revision
ICD-10	International Classification of Diseases, Tenth Revision
IHTSDO	International Health Terminology Standards Development Organization
ISO	International Organization for Standardization
LOINC	Logical Observation Identifiers Names and Codes
MAT	Measure Authoring Tool

NCQA	National Committee for Quality Assurance
OID	Object Identifier
ONC	Office of the National Coordinator for Health Information Technology
PHR	Personal Health Record
QDM	Quality Data Model
QRDA	Quality Reporting Document Architecture
QUICK	Quality Improvement and Clinical Knowledge
RIM	Reference Information Model
SI	International System of Units
SNOMED-CT	Systematized Nomenclature of Medicine – Clinical Terms
SQL	Structured Query Language
UCUM	Unified Code for Units of Measure
UML	Unified Modeling Language
URI	Uniform Resource Identifier
URL	Uniform Resource Locator
USHIK	United States Health Information Knowledgebase
UTC	Coordinated Universal Time
VSAC	Value Set Authority Center
XMI	XML Metadata Interchange
XML	eXtended Markup Language

TABLE 12–A

May 2015

13 APPENDIX F – GLOSSARY

canonical representation – As used within the CQL specification, this term means a representation of information in terms of primitives. For example, CQL contains high-level constructs such as timing phrases that are intended to allow natural language expression of timing relationships. The canonical representation of these constructs involves equivalent expression in terms of more primitive constructs targeted at implementation and integration applications.

clinical statement – Within the CQL specification, the term clinical statement is used to refer to the representation of clinical information in terms of a specific data model. For example, an instance of a FHIR Condition resource is a clinical statement asserting a condition for a subject in some status. Clinical statements are the content that CQL reasons about.

ELM – Expression Logical Model is a UML specification for representing artifact logic independent of syntax and special-purpose constructs introduced at the syntactic level. It is intended to enable distribution and sharing of computable quality logic.

FHIR® – Fast Healthcare Interoperability Resources (hl7.org/fhir) – is a next generation standards framework created by HL7. FHIR combines the best features of HL7's Version 2, Version 3 and CDA® product lines while leveraging the latest web standards and applying a tight focus on implementability.

FHIR Profile – A FHIR Profile is a statement of use of one or more FHIR Resources. It may include constraints on Resources and Data Types, Terminology Binding Statements and Extension Definitions. See the FHIR documentation for more information.

nullological – A category of operations for dealing with missing information. The term is actually due to Hugh Darwen, who introduced it in a paper to describe the behavior of operations in edge cases (e.g. empty sets or strings of length zero) and given his long history of opposing the use of "nulls" in relational systems, would probably not be pleased with the way the term has been co-opted in this context.

quality vendor – A company or organization that provides commercially available health quality services, such as distribution of quality-related knowledge artifacts, integration of quality measurement and improvement functionality, or provision of health quality evaluation services.

query – Within CQL, the term query refers to a specific language construct that forms the basis for expressing logic involving lists of clinical statements within an artifact. A query may use any or all of various clauses to describe the types of operations to be performed, and returns a list of values that can be used directly, or serve as the input to other queries.

retrieve – Within CQL, the term retrieve refers to a specific language construct used to access clinical statements within an artifact.

shaping – Within CQL, the term shaping refers to the operation performed by the return clause of a query, which allows the shape of the resulting values of the query to be described.

three-valued logic (3VL) – A logic system prevalent within SQL-based Database Management Systems (DBMSs) that is defined using three values, TRUE, FALSE, and UNKNOWN, as opposed to traditional boolean-valued logic systems that involve only two values, TRUE and

FALSE. The system is used as a mechanism for reasoning in the presence of missing information. Within CQL, the three values are represented by the language keywords true, false, and null.

tuple – Within CQL, a tuple is the basic construct for representing structured values, with each tuple value consisting of a set of tuple elements, each of which has a name and a value. Tuples in CQL are analogous to records, or structs, in traditional programming languages, and rows in database programming languages. They are used within CQL to represent class instances from object-oriented data models, as well as XML or JSON instances.

type – Within CQL, the term type refers to a conceptual component of the language that defines a set of values that are all of that same type. For example, Integer is a type and is defined as the set of all integer values within a specific range, specifically the signed integers that can be represented using two's complement binary notation with a 32-bit word.

uncertainty – Conceptually, the notion that a value is present, but not precisely known. Formally within CQL implementation contexts, uncertainty is represented using closed intervals to describe the range of possible values.

value – Within CQL, the term value refers to a piece of data of some type. For example, the value 5 is of type Integer. Values are immutable, meaning they do not change over time.

valueset – Within CQL, a valueset allows logic to reference externally defined value sets.