HL7_DAM_HARMON_HQAREL_INFORM_R1_2014MAY

HL7 Domain Analysis Model: Harmonization of Health Quality Artifact Reasoning and Expression Logic, Release 1

May 2014

HL7 Informative Document

Sponsored by:
Clinical Decision Support Work Group
Structured Documents Work Group
Clinical Quality Information Work Group

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.

Co-Chair:	Guilherme Del Fiol, MD, PhD University of Utah Health Care guilherme.delfiol@utah.edumailto:email@email_email.com
Co-Chair:	Robert Jenders, MD, MS jenders@ucla.edumailto:email@email_email.com
Co-Chair:	Kensaku Kawamoto, MD, PhD University of Utah Health Care kensaku.kawamoto@utah.edumailto:email@email_email.com
Co-Chair:	Howard Strasberg Walters Kluwer Health howard.strasberg@wolterskluwer.commailto:email@email_email.com
Co-Chair/Co-Editor:	Patricia Craig The Joint Commission pcraig@jointcommission.org
Co-Chair:	Floyd Eisenberg iParsimony LLC FEisenberg@iParsimony.com
Co-Chair:	Crystal Kallem RHIA, CPHQ Lantana Consulting Group Crystal.Kallem@Lantanagroup.com
Co-Chair:	Chris Millet cmillet@qualityforum.org
Co-Chair:	Walter Suarez, MD, MPH Kaiser Permanente walter.q.suarez@kp.org
Co-Chair:	Calvin Beebe Mayo Clinic cbeebe@mayo.edu
Co-Chair:	Diana Behling Iatric Systems Diana.Behling@iatric.com
Co-Chair:	Robert Dolin, MD Lantana Consulting Group Bob.Dolin@LantanaGroup.com
Co-Chair:	Austin Kreisler SAIC – Science Applications International Corp AUSTIN.J.Kreisler@leidos.com
Co-Chair:	Patrick Loyd iCode Solutions patrick.e.loyd@gmail.com
Co-Chair:	Brett Marquard River Rock Associates brett@riverrockassociates.com
Co-Editor:	Bryn Rhodes Veracity Solutions, Inc. bryn@veracitysolutions.commailto:email@email_email.com

Co-Editor:	Marc J. Hadley The MITRE Corporation mhadley@mitre.orgmailto:email@email_email.com
Co-Editor:	Gavin S. Black The MITRE Corporation gblack@mitre.org
Co-Editor:	Keith W. Boone GE Healthcare keith.boone@ge.commailto:email@email_email.com
Co-Editor:	Aziz Boxwala, PhD Meliorix aziz.boxwala@meliorix.com
Co-Editor:	Chengjian Che Lantana Consulting Group chengjian.che@lantanagroup.com
Co-Editor:	Nagesh Bashyam nagesh.bashyam@drajer.com
Co-Editor:	Cynthia L. Barton cbarton@ofmq.com
Co-Editor:	Kanwarpreet Sethi Lantana Consulting Group kp.sethi@lantanagroup.com
Co-Editor:	Andrew McIntyre andrew@Medical-Objects.com.au

Acknowledgments

This guide was produced as part of a combined effort with members from multiple HL7 Workgroups related to health quality. This group gratefully acknowledges input from numerous HL7 community members, as well as members of the broader health care community.

Revision History

Rev	Date	By Whom	Changes
0	12/2/13	B. Rhodes	Initial document creation
1	12/8/13	B. Rhodes	Expanded interval semantics discussions to account for open/closed intervals and boundary determination. Incorporated pre-ballot comments and performed final review prior to ballot submission.
2	3/10/14	B. Rhodes	Incorporated changes suggested by ballot comment reconciliation.

Contents

Figures

Tables

1 INTRODUCTION

The ability to unambiguously represent and share reasoning is a critical component of many different aspects of Health Quality, including measurement, management, and improvement. For example, quality measures require the description and communication of various population criteria, as well as the computations that must be performed to evaluate the measure overall. Similarly, decision support artifacts must capture criteria describing whether a patient should be the recipient of a particular intervention. These and many other motivating examples make clear that the ability to express and unambiguously share expression logic should be a central component of an overall health quality approach.

Although standards exist for this purpose, the domains of quality measurement and clinical decision support use different standards to do so. Harmonization of these different approaches would enable broader sharing of computable clinical knowledge, as well as reduce the burden on authors and implementers responsible for producing and consuming that knowledge. To enable that harmonization, this document describes the underlying concepts and behavior necessary to enable the expression and accurate communication of health quality reasoning across both quality domains.

Within this document, the term *artifact* refers to the documents that are used to capture and communicate health quality knowledge. Examples of artifacts from the clinical quality domain, include Electronic Clinical Quality Measures (eCQMs) and Health Quality Measure Format (HQMF) documents. From the clinical decision support domain, examples include Event-Condition-Action Rules, Order Sets, and Documentation Templates.

1.1 Purpose

This document seeks to define the common concepts and semantics involved in modeling reasoning within the various aspects of the health quality domain, with the goal of providing a common conceptual foundation that other specifications can use whenever the need to express and communicate expression logic arises.

In particular, this document is intended to inform the ongoing effort to harmonize expression logic representation between the quality measurement and clinical decision support domains.

1.2 Audience

The audience for this document includes knowledge workers in the health quality domains of measurement, management, and reporting: artifact authors and implementers, standards analysts and developers, tooling developers, as well as systems integrators.

Because the document is effectively a blueprint for the conceptual and semantic aspects of a logic language, some familiarity with common programming language concepts is helpful, but not required.

1.3 Background

Given that the need to represent expression logic within the health quality domain is ubiquitous, it is not surprising that there are different approaches to doing so. Even within the

body of HL7 standards, there are at least four different approaches being taken. The reasons for these differences ultimately have to do with the requirements driving each particular application. However, it is clear that there is a great deal of semantic overlap between each of the different approaches, and further, that providing a common foundation that describes that overlap clearly and completely can serve as a basis for the development of more concrete specifications that can enable harmonization.

Briefly, the four different standards within HL7 are Arden Syntax, GELLO, Health Quality Measures Format (HQMF), and CDS Knowledge Artifact IG (HeD). In addition, the workgroups involved in these various approaches acknowledge that there is an immense body of related work in the field of languages in general. However, as with any specific knowledge domain, the issue of finding the most appropriate language to be used is complex, and due to various factors including at least clinical requirements, business needs, technical constraints, logistical issues and consumer target, no single approach has as yet been found to be appropriate for all the various use cases within the domain of health quality.

It may well be the case that there is not a single standard that would be capable of meeting all the diverse requirements that are being met by each of the four standards within HL7. However, it is certainly the case that a core set of concepts can be identified and clearly defined in such a way that each of the four approaches can be considered either equivalent to, or a superset of that minimal set of functionality.

Specifically, by clearly identifying the broadest requirements that are common to each of the above approaches, we can define at the conceptual level a consistent set of features and semantics that can provide a common semantic footing for the representation, sharing, translation, and potentially execution, of expression logic within the health quality domain.

1.4 Approach

To build this document, the conceptual requirements for each of the following standards were considered:

- Arden Syntax
- GELLO
- Health Quality Measures Format (HQMF)
- CDS Knowledge Artifact IG (HeD)

The intent was to capture completely the semantics and requirements that are common to all four standards. The following sections briefly review each of these standards.

1.4.1 Arden Syntax

Arden Syntax is an HL7 standard for capturing, sharing, and executing clinical decision support logic. The syntax was developed in the late 1980s and early 1990s, and has been an HL7 standard since version 2.0 in 1999. The current version is 2.9.

Because Arden Syntax is capable of modeling not only logic but imperative processing (i.e. actions to perform rather than computations), only the functional aspects of the standard were considered. This subset is referred to within the Arden community as Essential Arden.

The requirements in this document are a superset of the conceptual requirements of Essential Arden.

1.4.2 GELLO

GELLO is an expression language based on the Object Management Group's (OMG) Object Constraint Language (OCL). The primary focus of OCL is to provide the ability to express constraints related to object models expressed in the OMG's Unified Modeling Language (UML). GELLO extends that goal with the intent to provide a standard query and expression language for clinical decision support. The language was developed in the early 2000s and was adopted as an international standard by HL7 in 2005. The current version is Release 2.

The requirements in this document are a superset of the conceptual requirements of GELLO.

1.4.3 Health Quality Measure Format

Health Quality Measure Format (HQMF) is an HL7 standard for capturing and sharing the definition of clinical quality measures. It is an XML document format based on the HL7 Reference Information Model (RIM), just like CDA is, but instead of describing what happens in a patient encounter, the HQMF standard describes how to compute a quality measure.

The requirements in this document are a superset of the expression components of the conceptual requirements of HQMF.

1.4.4 CDS Knowledge Artifact IG

CDS Knowledge Artifact IG, or Health eDecisions (HeD) Schema is a format for sharing the definition of clinical decision support artifacts. HeD was developed as an S&I Framework initiative beginning in late 2012. The format has passed HL7 ballot and is currently awaiting publication.

The requirements in this document are a superset of the expression components of the conceptual requirements of HeD.

2 REQUIREMENTS

In this section we identify the conceptual requirements that are common to all expression language representation needs within the health quality domain. Because these requirements have to do with the expression of logic in general, and must be capable of dealing with a broad range of potential data representation approaches, these requirements are quite general.

The requirements are broken down into two broad categories, *functional*, and *non-functional*. By functional requirements, we mean specifically the behaviors and semantics that must be represented within the language, and by non-functional requirements, we mean any constraints or requirements that have to do with the environment or context, external to the language itself.

2.1 Non-Functional Requirements

We begin with the non-functional requirements, defining the overall characteristics of the language itself, as well as any business or technical constraints required by the health quality domain.

2.1.1 Clinical Context

Perhaps the most important non-functional requirement is that the language be capable of dealing with the types of information that are encountered in clinical contexts. Specifically, the language must be capable of reasoning over the structures that are typically used to represent patient and health quality information. At the broadest level, this includes all types of data stores, from highly structured and accessible stores such as relational and object databases, to hierarchical structures such as object representations and XML documents, and even semi-structured and unstructured data such as excel spreadsheets, comma-separated value files, and text documents.

However, for the purposes of reasoning over health quality information, most, if not all, applications require that data be represented in some structured way, usually dictated by the structures within the applications, or some intermediate format used to transport that information. Moreover, this aspect of health quality reasoning is somewhat simplified by the fact that there are existing standard representations that can be used to capture and transport patient information.

Specifically, the existing HeD and HQMF standards use the Virtual Medical Record (vMR) and the HL7 Reference Information Model V3 (RIM), respectively, for this purpose. It is noncontroversial then to assert that so long as a language is capable of dealing with the kinds of structures typically encountered in these standards, as well as the object-oriented representations typically encountered in EHRs that such a language would be sufficient for representing any health quality logic.

In particular, the language must be capable of dealing with:

- Atomic values, such as integers, reals, strings, dates, and times.
- Physical quantities such as volumes and measurements.
- Intervals of values such as times, integers, reals, and physical quantities.

- Composite structures such as a Patient or SubstanceAdministration.
- Collections of values such as a list of SubstanceAdministration values.

In addition to the structure of the data, quality artifacts must be capable of dealing with medical terminologies as they are represented within these structures. In particular, reasoning often involves high-level terminology comparisons such as mapping between terminologies, or determining whether a given concept is subsumed by another.

2.1.2 Data Requirements

Related to the clinical context requirement is the notion that the data required to successfully evaluate an artifact should be easily computable. This problem of determining what data needs to be involved in the evaluation of any given artifact if that artifact contains 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 a 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 domain, this problem is further complicated by the problem of terminology mapping. In addition to the meaning of a particular clinical statement as defined by the data model (i.e. Procedure, Encounter, AdverseEvent, etc), a more precise meaning is typically represented with a vocabulary consisting of codes which determine the specific kind of statement being represented. For example, a diagnostic clinical statement may be classified using the ICD-9 vocabulary, further identifying the specific diagnosis represented.

In order for reasoning within health quality artifacts to operate correctly, the meaning of each clinical statement, as identified by the vocabularies involved, must be preserved. However, the 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 this requirement 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 any given artifact.

First, all clinical data within a knowledge artifact must be represented with a common data model. The data model selected is beyond the scope of this document, but so long as the model is capable of representing the information that must be reasoned about, the model selected is independent of this requirement.

Second, all references to clinical data within a health 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 reference the data, namely by effective time and clinical concept. 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 simple and reliable implementation of the interface between the evaluation engine and the clinical data source, because the criteria used to request information are simple and well-defined.

Third, by using standard terminologies within the actual data involved, the language 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 clinical data source. At a high level, this is the terminology problem, and is also beyond the scope of this document, other than to state that there is an expectation that standard terminologies will be used to ensure semantic consistency between the clinical data source and the reasoning expressed in the artifacts.

These three motivating factors inform the design of the Clinical Request expression discussed in detail in the Functional Requirements section later in this document.

2.1.3 Location Independence

Another aspect of paramount importance in dealing with the expression of health quality logic is the idea of *location independence*. This idea implies that the source of the data, i.e. where it is physically located, should be completely opaque to the language performing the reasoning.

This concept is an aspect of what is called *physical data independence* in the field of database technology, a well-supported and extremely useful concept, as it frees the language from having to deal with any of the aspects of physical storage, including where and how the data is physically stored. Data in today's health systems is physically stored in any number of possible ways, from enterprise relational databases and cloud-based object repositories all the way to proprietary indexing structures and even text files. By requiring that a language explicitly exclude reference to any physical storage details, implementers are free to provide adapters from the physical systems to the logical representation required by the language.

2.1.4 Model Independence

Another type of independence that is extremely useful is the notion of *model independence*. This idea implies that the specific structures being modeled within any particular data model should be completely independent of the expression logic. This is the same type of independence that enables a general-purpose language such as SQL or C# to deal with data from any kind of domain, without having to change the underlying language. The structures that the language is capable of dealing with are flexible enough to represent various knowledge domains, and so the same language can be used to perform operations on data from any domain.

The first reason for this separation is that clinical data models evolve to react to changing clinical, business, regulatory, and other requirements. By contrast, the operations used to express logic within health quality artifacts tend to be more stable. In other words, the language used to reason about clinical data models evolves at a different rate than the clinical data models themselves. Keeping a clear separation between the representation of the logic and the clinical data models on which that logic operates, minimizes the impact of those changes.

In addition, because there are so many different clinical models, keeping the data model separate from the representation of the logic results in a more flexible specification, as the same general purpose language can be used to deal with data from multiple models if necessary.

And finally, the separation results in a simpler implementation, as the details of dealing with specific clinical data models can be isolated from the implementation of the operations of the expression language, allowing the language to focus on reasoning, rather than domain-specific artifacts.

2.1.5 Missing Information

The ability to deal with missing information easily and gracefully is a critical aspect of any health quality reasoning system. For example, if an artifact requires a patient's age, the logic must be able to deal with the possibility that the patient information does not include a birthdate or age. Rather than throw an error, or require the content developer to handle the case explicitly, the language should be defined to allow the expression to be evaluated, but resulting in a *null*, rather than an actual value. This is the same model used in database systems to allow logic to be expressed easily in the presence of missing information.

Although the reason that a particular piece of information is missing may be relevant, in general, that aspect must be represented within the data model in some way. By accessing the relevant constructs within the data model, the logic for dealing with missing information can take the reason into account if necessary.

2.1.6 Simplicity

A primary goal for the representation of expression logic within health quality artifacts is simplicity. Expression logic must be easy to author, understand, communicate, and maintain. Although this is a difficult concept to quantify, some well-understood ideas from the field of computer language design can help identify more concrete requirements that enable simplicity.

Specifically, the concept of *orthogonality* has a direct bearing on the simplicity of a language. In language design, orthogonality refers to the relative independence of language concepts, and directly impacts both the simplicity and power of the resulting language. In other words, the number of concepts in the language should be as few as possible, and the operations available should be as broadly applicable as possible.

In addition to orthogonality, several of the other non-functional requirements, such as location independence and model independence, contribute to the overall simplicity of the language.

2.1.7 Readability

Although readability is an important aspect of the representation of expression logic within health quality artifacts, there is generally a tension between readability and implementability in that adding syntactic layers to improve readability tends to increase the difficulty of processing the expression logic. An ideal solution therefore should strike a reasonable balance between these two requirements.

2.1.8 Declarativeness

The declarative aspect of the conceptual requirements refers to the need for the language to be as high-level as possible, focusing on "what" reasoning is to be performed, rather than "how" that reasoning is carried out. In general, the more declarative a system is, the more the infrastructure of the system can provide the implementation details, leading to greater

simplicity, power, and reliability. For example, SQL is a fairly high-level language that focuses on describing the desired result set in set-oriented terms, rather than the low-level details such as which indexes to use, what the most efficient access path would be, etc.

In concrete terms, this requirement is essentially satisfied by the fact that the desired language is a functional expression language, but it is important to identify declarativeness as a desirable overall property of any system of reasoning in a health quality context.

2.1.9 Immutability

In the context of a computer language, *immutability* means that the language does not allow side-effects in any operation. In other words, no operation is allowed to change the state of any object within the system. Adopting this characteristic for the language has several key advantages for the health quality expression logic domain.

First, because health quality artifacts for the most part involve the specification of criteria and/or computations involving clinical information, there is little, if any, need to allow non-functional operations within the expression language. Any non-functional requirements, such as recording a recommendation made by an artifact, can be handled outside the context of the expression language itself.

Second, strictly enforcing this requirement allows the implementation environment to make better decisions regarding optimization and translation. Without the ability to guarantee that no operation is allowed to change the state of any object, an implementation would not be able to guarantee that any rewrites it performs would have no unintended side-effects, severely limiting the potential for optimization.

And finally, an immutable execution environment helps provide for more efficient and scalable potential implementations. For example, a real-time clinical decision support service that can guarantee all evaluations are immutable can provide a much simpler data interface to the engine because it does not have to be concerned with issues of consistency and concurrency. This enables an implementation engine to provide more sophisticated and encompassing caching at run-time.

2.1.10 Implementability

The *implementability* requirement refers to the relative difficulty of implementing processing functionality for expressions in the language. The language should address as much as possible engineering difficulties likely to be encountered during implementation, including:

- Semantic Verification
- Translation
- Compilation
- Execution
- Integration w/ Visual Designers

Part of this requirement involves consideration of the availability of commercial or open-source implementations of the technology, and the ease with which those technologies can be integrated to provide a solution for evaluating quality artifacts.

Perhaps the most important aspect of representing reasoning is ensuring that the representation can support the concrete implementation of the applications that will use that reasoning. In the health quality domain specifically, those representations consist of defining the reasoning involved in the expression of a health quality artifact such as a clinical decision support rule, or a quality measure. In both these cases, the primary intent of the artifact is to communicate knowledge. To achieve this goal, not only does the logic system in which the reasoning is expressed need to be well-defined, but it must be as easy as possible for an implementer to incorporate this knowledge into any given system.

In the broadest sense, this "incorporation" involves the same types of processes that are involved in computer language processing in general, and casting the problem in those terms helps to inform the conceptual model by clearly understanding the issues an implementer will have to deal with as part of integrating health quality artifacts. The following diagram, then, depicts the steps performed by a traditional compiler:

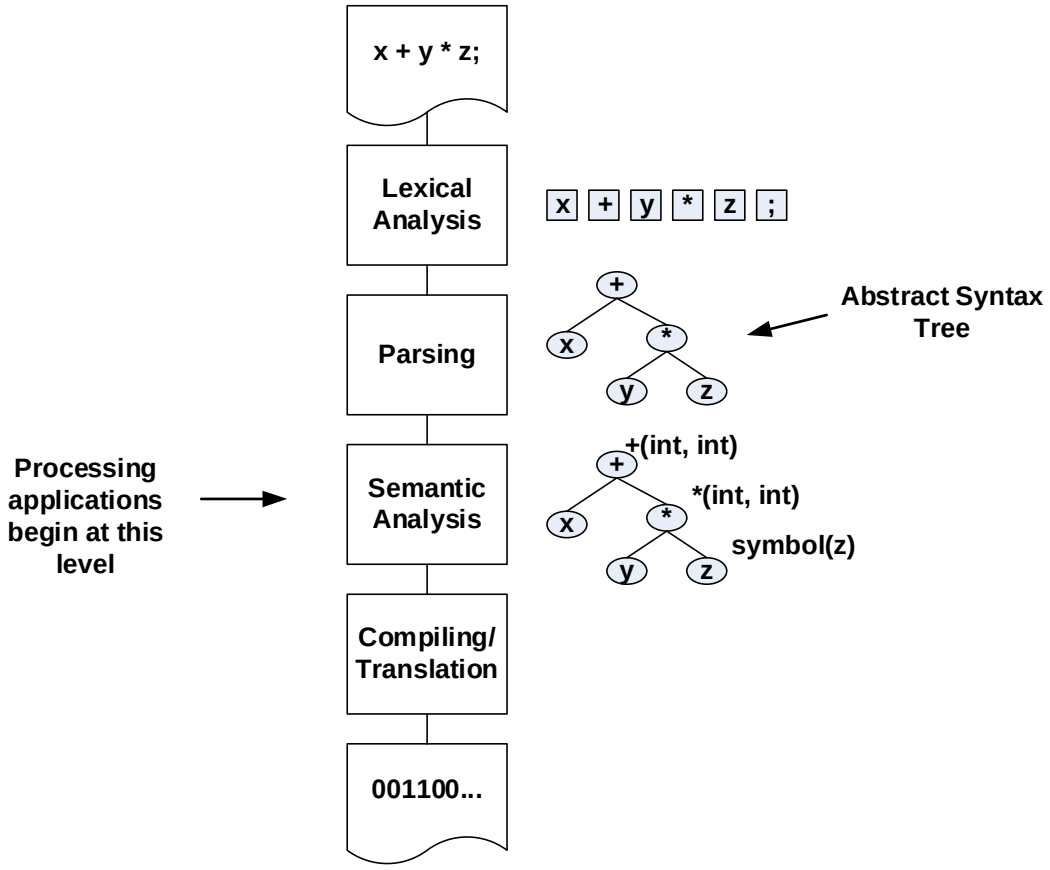

Figure 1 - Traditional Compiler Processes

The first step is *lexical analysis*, which involves breaking down the raw text of the expression into the "words" or *tokens* it contains. The next step is *parsing* which involves ensuring that the expression is *syntactically valid*, meaning that it obeys the rules of the expression language.

This is accomplished by organizing the tokens output by the lexical analysis phase into an *abstract syntax tree*, which is a conceptual representation of the original expression. The next step is *semantic analysis*, which involves ensuring that the operations and symbols used within the expression can all be resolved unambiguously to known operations. This step determines the types of values involved in the expression, and ultimately determines the result type of the expression. And finally, the traditional compiler converts the expression into the machine representation for the target environment.

For health quality artifacts then, implementability refers to the relative difficulty encountered when building language processing applications associated with consuming health quality artifacts. As noted in the above diagram, a typical language processing application such as evaluation or translation would start at the abstract syntax tree level. As a result, if the sharing of artifacts occurs at this level, language processing applications can skip the lexical analysis and parsing steps that would normally be required if sharing took place at the traditional language level.

To simplify integration and language processing requirements, the conceptual model, and any more concrete implementations of that model, should therefore focus on representing reasoning as close as possible to an abstract syntax tree, with the intent of striking the best balance between readability and simplicity of implementation and machine processing.

2.1.11 Expressivity

A basic requirement of any expression logic is that it should be complete with respect to the types of reasoning that must be represented. For health quality artifacts, the types of reasoning essentially involve evaluation of criteria over clinical data. These criteria generally involve standard operations encountered in data processing applications such as comparison, computation, filtering, relationship traversal, aggregation, and logical operations. In addition, clinical data often involves more specialized constructs such as intervals, ratios, temporal comparisons, and unit quantities.

Given these types of reasoning, it is useful to define completeness in terms of the following specific, well-understood categories. The elements of each category are defined in detail in the functional requirements section.

- **Language Completeness** – The language must provide sufficient language elements such as types, values, expressions, etc.
- **Value Completeness** – The language must support comprehensive value representation and manipulation. (e.g. atomic values, structured values, intervals, and lists)
    - For each category of value, the language must support complete manipulation including selection, conversion, and access where appropriate.
- **Logical Completeness** – The language must support all possible unary and binary logical operations. Note this specifically applies to the 3-valued-logic necessitated by dealing with missing information.
- **Nullological Completeness** – The language must provide complete support for dealing with the potential for missing information for each category of value.
- **Computational Completeness** – The language must provide complete support for arithmetic computations for supported numeric data types.

- **Relational Completeness** – The language must provide complete support for set-based operations, as defined by the notion of *relational completeness* from the field of relational database management.
- **Interval Completeness** – The language must provide complete support for interval-based operations, as defined by Allen's operators (merges, overlaps, meets, etc.).

In addition to completeness, a key aspect of expressivity is *composability*, referring to the fact that the result of any operation can be used as the argument to a subsequent operation, allowing expressions of arbitrary complexity to be built.

2.1.12 Extensibility

Although the language should make every effort to provide complete support, there are inevitably aspects of reasoning that will not be supported, and the language must therefore provide some mechanism to support extensibility so that new types and operators can be introduced as required.

For types, note that the extensibility requirements only include the ability to introduce new atomic types. It does not mean that the language must support the ability to introduce new type categories, a much more difficult requirement to support.

For operators, the extensibility requirements mean that the language must support the ability to introduce new operators that can then be used in any type-appropriate context within the language.

2.1.13 Type Safety

A key requirement for the expression of reasoning is the ability to ensure, as much as possible, that the reasoning is *correct* in some meaningful sense. Although the true measure of the correctness of an artifact is a function of how well it performs in the real world, the language used to express the reasoning can help or hinder the overall goal of correctness.

From the field of computer languages, the correctness of a program is typically established in terms of *syntactic* and *semantic* validity. Although neither of these measures ensures the program actually does what it was intended to do, it is generally accepted that that level of correctness cannot be automated because it involves, in the general case, theorem proving and other as yet unresolved questions at the edge of computer science and language research.

However, it is also generally accepted that establishing the syntactic and semantic correctness of a program significantly improves the chances that the program is correct. In particular, by ensuring *type safety*, a program can be guaranteed to be free of type errors such as attempting to divide a number by a string.

In addition, because a primary aspect of health quality artifacts is sharing, it should be possible to verify as much as possible the correctness of an artifact without having to actually execute the logic it contains. This can be accomplished by ensuring type safety using *static typing*, meaning that the types of the values and expressions within the logic can be determined by analysis.

In order to ensure type safety can be enforced statically, the language must be statically typed, meaning in particular that:

- All values within the language are of some type.
- All operators are defined to return some type.
- All operands of all operators are defined to be of some type.
- All arguments to all operator invocations are required to be the same type as the operand.

2.1.14 Usability

The usability requirement refers to the degree to which the language allows for the concise expression of common constructs. For example, in the area of missing information, if a language supports a conditional expression and a null test, then that is sufficient to ensure that null results can be handled gracefully within an expression:

```
if Patient.Age is null then
  CalculateAge(Patient.Birthdate, AsOf)
else
  Patient.Age
```

However, because this type of expression is extremely common, a more succinct shorthand should be provided:

```
IfNull(Patient.Age, CalculateAge(Patient.Birthdate, AsOf))
```

Where appropriate, the language should provide these types of shorthands to improve usability and understandability of the resulting artifacts.

2.2 Functional Requirements

For the functional requirements, we will identify the specific behaviors, operations, and associated semantics that must be supported by any language used within the health quality domain.

2.2.1 Language Elements

The most basic component of any language is the set of elements that can be used. Conceptually, the language must support the following elements:

Table 1 - Required Language Elements

Element	Description
Types	The language must support the ability to represent types of values, such as atomic types, structured types, interval types, and list types. This is required in order to support appropriate semantics for operator invocation and expression evaluation.
Values	The language must support the ability to represent values, such as atomic values, structured values, intervals, and lists.
Operators	The language must support the ability to invoke operations such as equality, comparisons, arithmetic computation, set and list operations, etc.
Expressions	The language must support the ability to construct expressions

	consisting of arbitrary combinations of other language elements such as literals and operator invocations. In particular, the language must support the notion of closure so that the result of one expression can be used as the input to another.
Named Expressions	The language must support the ability to "name" an expression such that it can be referenced by name from other expressions.
Parameters	The language must support some notion of global parameterization such that named values provided from an external source can be referenced within expressions in the language.

2.2.1.1 Types

Formally, a *type* is 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 in a specific range, typically the signed integers that can be represented using two's complement binary notation with a 32-bit word.

In addition to the concept of types, it is useful to define a *type category* as a way of representing different kinds of values. The type categories that must be supported will be discussed in detail the section on Data Representation.

2.2.1.2 Values

Formally, a *value* is 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. The values that must be supported within the language will be discussed in detail in the section on Data Representation.

The language must support *literals*, or representations of values within the language. In particular, the language must support literals for each supported type category:

- **Atomic literals** – The language must provide a mechanism to represent literals for each supported atomic type
- **Structured literals** – The language must provide a *structured literal* to allow for the representation of structured values of any type.
- **Interval literals** – The language must provide an *interval literal* to allow for the representation of intervals of any type.
- **List literals** – The language must provide a *list literal* to allow for the representation of lists of any type.

2.2.1.3 Operators

An *operator* provides the mechanism for manipulation of existing values in order to produce new values. Operators have the following characteristics:

- A unique *name* by which the operator may be invoked.
- A set, possibly empty, of *arguments*, each of which has an associated *name* and *type*.
- A *result type*, which defines the type of the value that will be returned from the invocation.

The operators that must be available within the language will be discussed in detail in the Operations section.

Note specifically that operators that do not return a value are explicitly disallowed because of the requirement that the language is *functional*. This does not mean that a function is not allowed to evaluate to *null*, only that all operators must be defined to return a result.

2.2.1.4 Expressions

Formally, an *expression* consists of any syntactically valid combination of language elements, as defined by the grammar of the language. For the purposes of the conceptual definition, the specific syntax involved is out of scope, other than to note that the expression construct must provide for the construction of expressions of arbitrary complexity.

In particular, the syntax must support:

- The representation of a single *value* is a valid expression, called a *literal*.
- Invocation of an *operator* is a valid expression, provided the appropriate number and types of arguments are provided.
- Wherever a *value* of a particular type is expected, it must be possible to provide an operator invocation that returns a value of that type.

A critical consequence of these requirements is that because the result type of each literal and operator invocation is known, the result type of any expression of arbitrary complexity can be determined statically by verifying the types of the literals and arguments involved in the expression.

2.2.1.5 Named Expressions

Formally, a *named expression* is defined as an arbitrary expression that is given a unique name. This name can then be used to reference the expression whenever it is needed in other contexts.

Conceptually, this requires two constructs:

- **Expression definition** – The ability to define the expression and provide a name.
- **Expression reference** – The ability to reference the expression within other expressions.

Note specifically that this is not as powerful a notion as a general purpose *function*, a named expression that is allowed to specify a set of arguments. This is not to say that an implementation must not support functions; only that within the domain of health quality, such support is an additional complication, especially in terms of implementation, that is unnecessary due to the constrained scope of the artifacts. In particular, by restricting the functionality to the use of parameterless named expressions, circular expression references are always invalid, preventing the possibility of infinite recursion at runtime.

2.2.1.6 Parameters

Formally, a *parameter* is defined as a named value that can be referenced within any expression within the artifact. Conceptually, parameters provide a mechanism for the execution environment to pass information to the artifact per evaluation. For example, a quality measure may define the measure period as a parameter, allowing the same measure to

be evaluated for different measure periods. Or, a clinical decision support rule may define a threshold such as Hemoglobin A1C level as a parameter, allowing an EHR to provide configuration data to the evaluation.

Conceptually, this requires two constructs:

- **Parameter definition** – The ability to define the *name* and *type* of the parameter, as well as provide an optional *default value* in terms of an expression. The type of the default value must match the defined type of the parameter.
- **Parameter reference** – The ability to reference the parameter by name from any expression.

The semantics of a parameter are that its value is fixed by the execution environment and cannot change during evaluation.

2.2.2 Data Representation

The language must support the ability to represent and reason about at least the following categories of types:

Table 2 - Required Type Categories

Category	Description
Atomic Types	The language must support the ability to represent atomic values such as integers, reals, strings, etc. See the Atomic Types and Values functional requirement for more information.
Structured Types	The language must support the ability to represent structured types, meaning types that represent values that are collections of named properties of any type. See the Structured Types and Values functional requirement for more information.
Interval Types	The language must support the ability to represent intervals, where an interval type is defined as having a point type, and a beginning and ending point. See the Interval Types and Values functional requirement for more information.
List Types	The language must support the ability to represent list values, where a list type specifies the types of elements that are allowed within a list.

Note that the conceptual requirements do not include the need to define new types within any of these categories within the language. Type definitions will in general be provided by the execution environment together with the data models involved in the definition of any particular artifact.

Note also that for each type category, the notion of *type equivalence* is defined, meaning the criteria that is used to determine whether or not two types are the same.

2.2.2.1 Atomic Types

The language must support the ability to represent atomic values, each of which is of some atomic type. This type category consists of values that have no language-understood

components, meaning that from the perspective of the language, the value is an atomic unit of data.

In particular, the language must support at least the following atomic types:

Table 3 - Atomic Types

Type	Description
Boolean	The Boolean type consists of the logical values *true* and *false*.
Integer	The Integer type consists of the whole integers, with range determined by platform, except that it must support at least 32-bit signed integer range.
Real	The Real type consists of real numbers, with range determined by the platform, except that it must support at least 28 digits of precision and/or scale. Note that this document does not dictate whether the platform encoding be floating point or BCD, although there is preference for BCD due to the more reliable encoding of values.
String	The String type consists of strings of characters, with maximum length determined by the platform, except that it must support at least strings of length 2^16-1 (64K, or 65,535 bytes).
Timestamp	The Timestamp type consists of date and time values, including timezone offset representation, with range determined by platform, except that it must support at least the representation of dates and times from January 1st, 1900 through December 31st, 2999, with granularity to at least seconds.

Atomic type equivalence is determined by name only. In other words two atomic types are equivalent if and only if they have the same name.

Note that a value such as a *physical quantity* is a *structured type* because it has two components, namely the *value* and *units*.

2.2.2.2 Structured Types

A *structured type* defines a non-empty set of named *properties*, each of which has a name and type. A *structured value* then is a value of some structured type, consisting of a named set of properties, each of which has a value of a type corresponding to the property type.

Structured types come in two flavors:

- **Named structured types** – A structured type with a unique name, e.g. Patient or Encounter
- **Anonymous structured types** – A structured type whose "name" is determined by the set of properties for the type.

This distinction is useful because it allows for structured value comparisons to be performed without requiring name-based type equivalence. In particular, two structured types are considered equivalent if:

- They have the same type name.

- Or, they have the same set of properties, by name and type.

In addition, the notion of anonymous structured types is required in order to support operations on lists of structured types such as *foreach* that allow for the construction of new structured values as part of the result.

2.2.2.3 Interval Types

The language must support the ability to deal with intervals of values of various types. In particular, intervals of time, and ranges of integers and reals. To ensure that interval support is a first class feature of the language, the requirements are described in terms of the operations that must be supported for a particular type in order for it to be used as the basis for an interval type. Such a type is referred to here as the *point type* of the interval type.

In particular, the operations described in this section are necessary in order to completely support the interval operators defined in the Operations section. Furthermore, to ensure meaningful support of the interval operators, intervals are explicitly defined to be discrete. For conceptually continuous types such as timestamp and real, this requires that a minimum granularity be specified to ensure well-defined behavior. This minimum granularity is defined by the Successor and Predecessor operators for each type.

An *interval type* thendefines the possible set of interval values for a particular *point type*. The point type for an interval can be any type that supports:

- **Comparison** – The system must be able to determine whether a value of the point type is less than, equal to, or greater than, another value of the point type.
- **Successor and Predecessor functions** – Given a particular value of the point type, the system must be able to determine what the "next" or "previous" value in the point type is. These operators are used to perform various interval operations such as Meets and Overlaps.
- **Minimum and Maximum value functions** – Given a particular point type, the system must be able to determine what the minimum and maximum values are for the point type. These operators define the boundaries for the range of the point type.

Formally then, an interval type consists of:

- A point type
- A Begin property that defines the starting point for the interval
- An End property that defines the ending point value for the interval
- A BeginIsInclusive property that indicates whether the starting boundary for the interval is inclusive or exclusive of the starting point.
- An EndIsInclusive property that indicates whether the ending boundary for the interval is inclusive or exclusive of the ending point.

The beginning and ending boundaries of the interval may be specified as inclusive or exclusive. If the boundary is inclusive, the interval includes the boundary point. For example, the following pseudo-syntax interval expression contains the integers 3 and 4, but not 5:

```
[3, 5)
```

In addition, the beginning and ending points of an interval may be null. Whether the interval boundary is inclusive or exclusive determines the interpretation of a null point. If the point is

null and the interval is exclusive, the boundary is considered unknown and operations involving that point will return null. For example, given the interval:

```
Interval1 := [3, null);
```

Testing whether the interval contains the value 5, results in null:

```
Result := Contains(Interval1, 5);
```

However, if the point is null and the interval is inclusive, the boundary is interpreted as the beginning or ending of the range of the point type, and operations involving the boundary will be performed with that interpretation. For example, given the interval:

```
Interval1 := [3, null];
```

Testing whether the interval contains the value 5, results in true because the interval boundary is the maximum integer:

```
Result := Contains(Interval1, 5);
```

Interval types are equivalent if they have the same point type.

2.2.2.4 List Types

A *list type* consists of an ordered collection of values, called *elements*, that are all of some type, the *element type* of the list type. The element type may be any type, for example:

- A list of integers
- A list of intervals
- A list of Patient values
- A list of lists of integers

However, all the values within any given list value must be of the same element type.

List types are equivalent if they have the same element type.

List values 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, this specification 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.

2.2.2.5 Type Inheritance

Because clinical models are often defined in terms of class hierarchies, health quality artifacts can often involve logic that must be capable of reasoning not only about the data involved, but also about whether or not a particular value is a specialization or generalization of some type in the class hierarchy.

This requirement results in the need for the language to support a minimal notion of type inheritance. In particular:

- **A derived type** – The language must support the notion of a *derived* type, meaning a type that is based on another type, and inherits its structure and values.
- **A minimal base type** – The language must support the notion of a base type, from which all types are derived.
- **Type equivalence** – The language must take the notion of derived types into consideration when determining type equivalence.
- **Value substitutability** – The language must support the notion of *value substitutability*, meaning that wherever a value of a particular type is expected, it must be possible to provide a value of a more derived type.

Note that although support for some form of dealing with type inheritance is desirable, it is not necessary, so long as the language provides some mechanism for dealing with structured values in an inheritance hierarchy.

2.2.3 Operations

To support the non-functional completeness requirements, the language must support a fairly broad range of operations in several categories. The following sections define these operations.

Note that to define the operands, result types, and where useful, the semantics, a pseudo-syntax is used. This syntax is purely for definition purposes and is not considered part of the requirements specified. Only the semantics of the operations are prescribed.

In addition, because the document is focused on the concepts involved, there are aspects that would normally be represented with specialized syntax within a language that are represented in the pseudo-syntax as operators. For example, arithmetic addition is typcially represented in a concrete syntax with infix notation and a specialized symbol (+ in this case). However, from a purely conceptual perspective, the relevant concept is *arithmetic addition*, and for definiteness and simplicity, it is represented as an operator invocation within this document.

2.2.3.1 Values

The language must support the expression of values of each type category.

2.2.3.1.1 Atomic Type Literals

The language must provide some way to express literals for each atomic type.

2.2.3.1.2 Structured Value Literals

The language must provide some way to express a structured value consisting of:

- The type name of the structured value
- A list of property specifiers, each of which contains
    - The name of a property
    - The value for the property in terms of an arbitrary expression.

The result of this expression is a structured value of the specified type, with each property set to the result of evaluating the associated expression. If the structured type has properties that do not have corresponding property specifiers, those properties will be null in the resulting structured value.

2.2.3.1.3 Structured Value Redefine

The language must provide some way to easily redefine specific values of a structured value. In particular, some mechanism must be provided to allow an expression consisting of:

- A source expression to specify the structured value to be redefined
- A list of property redefine specifiers, each of which contains
  - The name of the property to be redefined
  - The value for the property in terms of an arbitrary expression.

The result of this expression is a value of the same type as the source expression, with the same values as the source expression, except where a property is redefined. In that case the property value will be the result of evaluating the expression in the specifier.

In other words, this expression allows a new structured value to be built based on an existing one.

The property redefine expressions must be able to access the current values of the properties of the source structured value.

The following pseudo-syntax example illustrates the use of a redefine operation:

```
Patient1 = Object<Patient>(Name = "John", DOB = "19840101");
Patient2 = Redefine(Patient1, Name = Name + " Doe");
```

Evaluating the Patient2 expression in this example results in a Patient object with name "John Doe", and DOB "19840101".

2.2.3.1.4 Property Access

```
Property(Source : Any, Path : String)
```

The language must support the ability to access properties of structured values through the use of a path specifier.

The type of the result is the type of the property specified by the path.

The path specifier must support the following:

- Ability to reference properties by name (e.g. ID)
- Ability to use dereferencing to access properties of values, recursively (e.g. EffectiveTime.Low, Substance.Code.CodeSet)
- Ability to use absolute indexing to access elements of list values (e.g. Medications[1])
- Ability to use property referencing and indexing in combination (e.g. Medications[1].EffectiveTime.Low)

2.2.3.1.5 List

```
List(Operand1 : Any, Operand2 : Any, … OperandN : Any) : List<Any>
```

The List operator returns a list of type Any containing the value of each argument as an element, in the order in which they were provided to the operation.

The static type of the first argument determines the type of the resulting list, and the actual type of each subsequent argument must be of that same type.

If any argument is null, the resulting list will have null for that element.

2.2.3.1.6 Interval

```
Interval(Begin : Any, BeginIsInclusive : Boolean,
  End : Any, EndIsInclusive : Boolean) : Interval<Any>
```

The Interval operator returns an interval value with the specified beginning and ending points.

An Interval must be defined using a point type that supports comparison, as well as Successor and Predecessor operations.

The boundaries of the interval may each be defined as inclusive (closed) or exclusive (open).

The static type of the Begin argument determines the point type of the interval, and the End argument must be of the same type.

If the Begin argument is null, the resulting interval will have a null Begin point.

If the End argument is null, the resulting interval will have a null End point.

2.2.3.2 Logical Operators

The language must support a complete set of 3-valued logical operators to ensure that any logical operation can be performed.

2.2.3.2.1 And

```
And(Operand1 : Boolean, Operand2 : Boolean, … OperandN : Boolean) : Boolean
```

The And operator returns the logical conjunction of its arguments. Note that this operator is defined as n-ary, allowing any number of arguments. The result of And with no arguments is defined to be false. The result of an And with a single argument is defined to be the result of the argument. The result of And with two arguments is defined using 3-valued logic semantics. This means that if either argument is false, the result is false; if both arguments are true, the result is true; otherwise, the result is null. The result of more than two arguments is defined as successive invocations of And.

The following table defines the truth table for this operator. The table contains a row for each possibility for the first operand, and a column for each possibility for the second operand. Each cell then contains the result of the operation with the inputs for that row and column.

Table 4 - "And" Truth Table

	TRUE	**FALSE**	**NULL**
TRUE	TRUE	FALSE	NULL
FALSE	FALSE	FALSE	FALSE
NULL	NULL	FALSE	NULL

2.2.3.2.2 Or

```
Or(Operand1 : Boolean, Operand2 : Boolean, … OperandN : Boolean) : Boolean
```

The Or operator returns the logical disjunction of its arguments. Note that this operator is defined as n-ary, allowing any number of arguments. The result of Or with no argument is defined to be true. The result of Or with a single argument is defined to be the result of the argument. The result of Or with two arguments is defined using 3-valued logic semantics. This means that if either argument is true, the result is true; if both arguments are false, the result

is false; otherwise, the result is null. The result of more than two arguments is defined as successive invocations of Or.

The following table defines the truth table for this operator. The table contains a row for each possibility for the first operand, and a column for each possibility for the second operand. Each cell then contains the result of the operation with the inputs for that row and column.

Table 5 - "Or" Truth Table

	TRUE	**FALSE**	**NULL**
TRUE	TRUE	TRUE	TRUE
FALSE	TRUE	FALSE	NULL
NULL	TRUE	NULL	NULL

2.2.3.2.3 Not

`Not(Operand : Boolean) : Boolean`

The Not operator returns the logical negation of its argument. If the argument is true, the result is false; if the argument is false, the result is true; otherwise, the result is null.

The following table defines the truth table for this operator. The table contains a row for each possibility for the operand, with each cell containing the result of the operation with the input for that row.

Table 6 - "Not" Truth Table

	NOT
TRUE	FALSE
FALSE	TRUE
NULL	NULL

2.2.3.2.4 Xor

`Xor(Operand1 : Boolean, Operand2 : Boolean) : Boolean`

The Xor operator returns the exclusive disjunction of its arguments. If one argument is true and the other argument is false, the result is true; 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. The table contains a row for each possibility for the first operand, and a column for each possibility for the second operand. Each cell then contains the result of the operation with the inputs for that row and column.

Table 7 - "Xor" Truth Table

	TRUE	**FALSE**	**NULL**
TRUE	FALSE	TRUE	NULL
FALSE	TRUE	FALSE	NULL
NULL	NULL	NULL	NULL

2.2.3.3 Conditional Operators

Conditional operators provide the ability to select between multiple expressions using conditional logic to determine the result.

2.2.3.3.1 Conditional

Conditional(Condition : Boolean, Then : Any, Else : Any) : Any

The conditional operator is a ternary operator that allows for selection between two operands using a conditional expression. If the Condition operand evaluates to true, the result of the conditional is the result of evaluating the Then operand; otherwise, the result is the result of evaluating the Else operand.

The static type of the Then argument determines the result type of the conditional, and the Else argument must be of that same type.

2.2.3.3.2 Case

Case(Comparand : Any, CaseItem1 : Any, CaseItem2 : Any,
… CaseItemN : Any, Else : Any) : Any

CaseItem(When : Any, Then : Any) : Any

The Case operator allows for multiple conditional expressions to be chained together in a single expression, rather than having to nest multiple Conditional operators.

In addition, the Comparand operand provides a variant on the case that allows a single value to be compared in each case item.

If a Comparand is not provided, the type of each When argument within each CaseItem is expected to be Boolean.

If a Comparand is provided, the type of each When argument within each CaseItem is expected to be of the same type as the comparand.

An Else argument must always be provided.

The static type of the Then argument within the first CaseItem determines the type of the result, and the Then argument of each subsequent CaseItem and the Else argument must be of that type.

2.2.3.4 Nullological Operators

Nullological operators deal with the ability of the language to detect and deal with nulls. Because the language supports nulls as a marker for missing information, evaluation of any expression may result in a null. The following operators ensure that the language is capable of dealing with these nulls.

2.2.3.4.1 Null

Null() : Any

The Null operator returns a null, or missing information marker. To avoid the need to cast this result, the operator may be defined to return a typed null.

2.2.3.4.2 IsNull

IsNull(Operand1 : Any) : Boolean

The IsNull operator determines whether or not its argument evaluates to null. If the argument evaluates to null, the operator returns true; otherwise, the operator returns false.

2.2.3.4.3 IfNull

IfNull(Operand1 : Any, Operand2 : Any) : Any

The IfNull operator replaces a null with the result of a given expression. If the first argument evaluates to null, the result of evaluating the second argument is returned; otherwise, the result of the first argument is returned.

The static type of the first argument determines the type of the result, and that the second argument must be of that type.

Note that IfNull is shorthand for an equivalent Conditional expression.

2.2.3.4.4 Coalesce

Coalesce(Operand1 : Any, Operand2 : Any, … OperandN : Any) : Any

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 that all subsequent arguments must be of that type.

Note that Coalesce is the generalization of the IfNull operator to any number of operands.

2.2.3.5 Conversion and Casting Operators

2.2.3.5.1 Is

Is<Any>(Operand : Any) : Boolean

The Is operator allows the type of a result to be tested. The language must support the ability to test against any type. If the run-time type of the argument is *of* the type being tested, meaning it is the type, or any descendent of the type, the result of the operator is true; otherwise, the result is false.

For example, assuming a type hierarchy that defines a base ClinicalStatement with a descendent SubstanceAdministration, the following test would result in true:

```
Is<ClinicalStatement>(SubstanceAdministrationValue);
```

2.2.3.5.2 As

As<Any>(Operand : Any) : Type

AsStrict<Any>(Operand : Any) : Type

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 actual run-time type of the argument is not *of* the target type, or any descendent type, the result of the operator is null.

If the *strict* variant is used and the actual run-time type of the argument is not *of* the target type, or any descendent, an exception is raised.

2.2.3.5.3 Convert

ConvertTo<Any>(Operand : Any) : Any

The Convert operator converts a value to a given target 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 raised if there is no valid conversion from the actual value to the target type.

The following table lists the minimum conversions that must be supported:

Table 8 - Type Conversion Matrix

From\To	Boolean	Integer	Real	String	Timestamp
Boolean	N/A	-	-	Required	-
Integer	-	N/A	Required	Required	-
Real	-	-	N/A	Required	-
String	Required	Required	Required	N/A	Required
Timestamp	-	-	-	Required	N/A

2.2.3.6 Comparison Operators

The language must support a complete set of comparison operators to allow values to be compared. In particular, the language must support the equality operator for all types, and must support relative comparison operators for all types that have a total ordering.

2.2.3.6.1 Equal

Equal(Operand1 : Any, Operand2 : Any) : Boolean

The Equal operator returns true if the arguments are equal; otherwise, false.

Equality semantics are defined to be value-based. This means for atomic types that equality returns true if and only if the result of each argument evaluates to the same value.

For object types, this means that equality returns true if and only if the objects are of the same type, and the values for all properties are the same.

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, the elements are equal using value semantics.

For interval types, this means that equality returns true if and only if the intervals are over the same point type, and they have the same value for the beginning and ending boundaries of the interval as determined by the Begin and End operators. Note in particular that this means that two intervals are equal if they define the same range, not necessarily that they have the same value for all properties. For example, the completely closed interval [1..5] is equal to the closed-open interval [1..6).

If either argument is null, the result is null.

2.2.3.6.2 Not Equal

NotEqual(Operand1 : Any, Operand2 : Any) : Boolean

The NotEqual operator returns true if its arguments are not the same value.

This operator is equivalent to an invocation of Not and Equal.

2.2.3.6.3 Less Than

Less(Operand1 : Any, Operand2 : Any) : Boolean

The Less operator compares two values and returns true if the first argument is less than the second argument; false otherwise.

The Less operator is defined for Integers, Reals, Strings, and Timestamps.

If either argument is null, the result is null.

2.2.3.6.4 Greater Than

Greater(Operand1 : Any, Operand2 : Any) : Boolean

The Greater operator compares two values and returns true if the first argument is greater than the second argument; false otherwise.

The Greater operator is defined for Integers, Reals, Strings, and Timestamps.

If either argument is null, the result is null.

2.2.3.6.5 Less Than Or Equal

LessOrEqual(Operand1 : Any, Operand2 : Any) : Boolean

The LessOrEqual operator compares two values and returns true if the first argument is less than or equal to the second argument; false otherwise.

The LessOrEqual operator is defined for Integers, Reals, Strings, and Timestamps.

If either argument is null, the result is null.

2.2.3.6.6 Greater Than Or Equal

GreaterOrEqual(Operand1 : Any, Operand2 : Any) : Boolean

The GreaterOrEqual operator compares two values and returns true if the first argument is greater than or equal to the second argument; false otherwise.

The GreaterOrEqual operator is defined for Integers, Reals, Strings, and Timestamps.

If either argument is null, the result is null.

2.2.3.7 Arithmetic Operators

The language must support a complete set of computational operators to ensure that reasoning involving mathematical computation can be expressed. This includes basic arithmetic, dealing with reals versus integers, as well as discrete integer operations and exponential operations.

2.2.3.7.1 Add

Add(Operand1 : Any, Operand2 : Any) : Any

The Add operator performs numeric addition of its arguments.

The Add operator is defined for Integers and Reals.

If either argument is null, the result is null.

2.2.3.7.2 Subtract

Subtract(Operand1 : Any, Operand2 : Any) : Any

The Subtract subtracts the second argument from the first.

The Subtract operator is defined for Integers and Reals.

If either argument is null, the result is null.

2.2.3.7.3 Multiply

Multiply(Operand1 : Any, Operand2 : Any) : Any

The Multiply operator performs numeric multiplication of its arguments.

The Multiply operator is defined for Integers and Reals.

If either argument is null, the result is null.

2.2.3.7.4 Divide

Divide(Operand1 : Any, Operand2 : Any) : Real

The Divide operator divides the first argument by the second.

The Divide operator is defined for Integers, and Reals. Note that the result type is Real, even if the arguments are Integers.

If either argument is null, the result is null.

2.2.3.7.5 Integer Division

TruncatedDivide(Operand1 : Any, Operand2 : Any) : Any

The TruncatedDivide operator performs discrete integer division of the first argument by the second.

The TruncatedDivide operator is defined for Integers and Reals.

If either argument is null, the result is null.

2.2.3.7.6 Modulo

Modulo(Operand1 : Any, Operand2 : Any) : Any

The Modulo operator computes the remainder of the division of the first argument by the second.

The Modulo operator is defined for Integers and Reals.

If either argument is null, the result is null.

2.2.3.7.7 Ceiling

Ceiling(Operand : Real) : Integer

The Ceiling operator returns the first integer greater than or equal to the argument.

If the argument is null, the result is null.

2.2.3.7.8 Floor

Floor(Operand : Real) : Integer

The Floor operator returns the first integer less than or equal to the argument.

If the argument is null, the result is null.

2.2.3.7.9 Truncate

Truncate(Operand : Real) : Integer

The Truncate operator returns the integer component of its argument.

If the argument is null, the result is null.

2.2.3.7.10 Absolute Value

Abs(Operand : Any) : Any

The Abs operator returns the absolute value of its argument.

The Abs operator is defined for Integers and Reals.

If the argument is null, the result is null.

2.2.3.7.11 Negate

Negate(Operand : Any) : Any

The Negate operator returns the negative of its argument.

The Negate operator is defined for Integers and Reals.

If the argument is null, the result is null.

2.2.3.7.12 Round

Round(Operand : Real, Precision : Integer) : Real

The Round operator returns the nearest value to its argument within the given precision. The semantics of Round are defined as a traditional round, meaning that with a Precision of 0, a decimal value of 0.5 or higher will round to 1.

If the argument is null, the result is null.

If specified, precision determines the decimal place at which the rounding will occur.

If precision is not specified or null, 0 is assumed.

2.2.3.7.13 Natural Log

Ln(Operand : Real) : Real

The Ln operator computes the natural logarithm of its argument.

If the argument is null, the result is null.

2.2.3.7.14 Log

Log(Operand : Real, Base : Real) : Real

The Log operator computes the logarithm of its first argument using the second argument as the base.

If either argument is null, the result is null.

2.2.3.7.15 Power

Power(Operand : Real, Exponent : Real) : Real

The Power operator raises the first argument to the power given by the second argument.

If either argument is null, the result is null.

2.2.3.8 String Operators

The language must support a complete set of operators for dealing with strings, including concatenation, indexing, length, substring, and position.

2.2.3.8.1 Concatenate

Concat(Operand1 : String, Operand2 : String, … OperandN : String) : String

The Concat operator performs string concatenation of its arguments.

If any argument is null, the result is null.

2.2.3.8.2 Combine

Combine(Source : List<String>, Separator : String) : String

The Combine operator combines a list of strings, optionally separating each string with the given separator.

2.2.3.8.3 Split

Split(StringToSplit : String, Separator : String) : List<String>

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.

Otherwise, the result is a list of strings containing one element for each string in the StringToSplit argument separated by the given separator.

2.2.3.8.4 Length

Length(Operand : String) : Integer

The Length operator returns the number of characters in the given string. Trailing whitespace characters are included in the length.

If the argument is null, the result is null.

2.2.3.8.5 Upper

Upper(Operand : String) : String

The Upper operator returns the upper case of its argument.

If the argument is null, the result is null.

2.2.3.8.6 Lower

Lower(Operand : String) : String

The Lower operator returns the lower case of its argument.

If the argument is null, the result is null.

2.2.3.8.7 Indexer

Indexer(Operand : String, Index : Integer) : String

The Indexer operator returns the Indexth character of a string.

Indexes in strings are defined to be 1-based.

If either argument is null, the result is null.

2.2.3.8.8 Pos

Pos(Pattern : String, StringToSearch : String) : Integer

The Pos operator returns the 1-based index of the string Pattern in the string StringToSearch.

If the pattern is not found, the result is 0.

If either argument is null, the result is null.

2.2.3.8.9 Substring

Substring(StringToSub : String, StartIndex : Integer, Length : Integer) : String

The Substring operator returns the string within StringToSub beginning at the 1-based index StartIndex, and consisting of Length characters.

If Length is not specified, the substring returned starts at StartIndex and continues to the end of StringToSub.

If StringToSub or StartIndex is null, the result is null.

2.2.3.8.10 LeftTrim

LeftTrim(StringToTrim : String) : String

The LeftTrim operator returns the contents of StringToTrim with any leading whitespace characters removed. Whitespace characters are defined as characters with the Unicode character property White_Space. See the Unicode standard reference for more detail.

If StringToTrim is null, the result is null.

2.2.3.8.11 RightTrim

RightTrim(StringToTrim : String) : String

The RightTrim operator returns the contents of StringToTrim with any trailing whitespace characters removed. Whitespace characters are defined as characters with the Unicode character property White_Space. See the Unicode standard reference for more detail.

If StringToTrim is null, the result is null.

2.2.3.9 Date and Time Operators

2.2.3.9.1 Add Years

AddYears(Date : Timestamp, Years : Integer) : Timestamp

The AddYears operator returns the given Date with Years years added.

Years may be a positive or negative value, allowing for addition or subtraction of years from a date.

This operation is equivalent to AddMonths using twelve times the number of Years.

If either argument is null, the result is null.

2.2.3.9.2 Add Months

AddMonths(Date : Timestamp, Months : Integer) : Timestamp

The AddMonths operator returns the given Date with Months months added.

Months may be a positive or negative value, allowing for addition or subtraction of months from a date.

If the day component of the timestamp is greater than the number of days in the resulting month, the last day of the resulting month is returned.

If either argument is null, the result is null.

2.2.3.9.3 Add Days

AddDays(Date : Timestamp, Days : Integer) : Timestamp

The AddDays operator returns the given Date with Days days added.

Days may be a positive or negative value, allowing for addition or subtraction of days from a date.

If either argument is null, the result is null.

2.2.3.9.4 Add Hours

AddHours(Date : Timestamp, Hours : Integer) : Timestamp

The AddHours operator returns the given Date with Hours hours added.

Hours may be a positive or negative value, allowing for addition or subtraction of hours from a date.

If either argument is null, the result is null.

2.2.3.9.5 Add Minutes

AddMinutes(Date : Timestamp, Minutes : Integer) : Timestamp

The AddMinutes operator returns the given Date with Minutes minutes added.

Minutes may be a positive or negative value, allowing for addition or subtraction of minutes from a date.

If either argument is null, the result is null.

2.2.3.9.6 Add Seconds

AddSeconds(Date : Timestamp, Seconds : Integer) : Timestamp

The AddSeconds operator returns the given Date with Seconds seconds added.

Seconds may be a positive or negative value, allowing for addition or subtraction of seconds from a date.

If either argument is null, the result is null.

2.2.3.9.7 Add Milliseconds

AddMilliseconds(Date : Timestamp, Milliseconds : Integer) : Timestamp

The AddMilliseconds operator returns the given Date with Millisecond milliseconds added.

Milliseconds may be a positive or negative value, allowing for addition or subtraction of milliseconds from a date.

If either argument is null, the result is null.

2.2.3.9.8 Years Between

YearsBetween(StartDate : Timestamp, EndDate : Timestamp) : Integer

The YearsBetween operator returns the number of whole years occurring between StartDate and EndDate.

If StartDate is after EndDate, the result is negative.

If either argument is null, the result is null.

2.2.3.9.9 Months Between

MonthsBetween(StartDate : Timestamp, EndDate : Timestamp) : Integer

The MonthsBetween operator returns the number of whole months occurring between StartDate and EndDate.

If StartDate is after EndDate, the result is negative.

If either argument is null, the result is null.

2.2.3.9.10 Days Between

DaysBetween(StartDate : Timestamp, EndDate : Timestamp) : Integer

The DaysBetween operator returns the number of whole days occurring between StartDate and EndDate.

If StartDate is after EndDate, the result is negative.

If either argument is null, the result is null.

2.2.3.9.11 Hours Between

YearsBetween(StartDate : Timestamp, EndDate : Timestamp) : Integer

The HoursBetween operator returns the number of whole hours occurring between StartDate and EndDate.

If StartDate is after EndDate, the result is negative.

If either argument is null, the result is null.

2.2.3.9.12 Minutes Between

MinutesBetween(StartDate : Timestamp, EndDate : Timestamp) : Integer

The MinutesBetween operator returns the number of whole minutes occurring between StartDate and EndDate.

If StartDate is after EndDate, the result is negative.

If either argument is null, the result is null.

2.2.3.9.13 Seconds Between

SecondsBetween(StartDate : Timestamp, EndDate : Timestamp) : Integer

The SecondsBetween operator returns the number of whole seconds occurring between StartDate and EndDate.

If StartDate is after EndDate, the result is negative.

If either argument is null, the result is null.

2.2.3.9.14 Milliseconds Between

MillisecondsBetween(StartDate : Timestamp, EndDate : Timestamp) : Integer

The MillisecondsBetween operator returns the number of whole milliseconds occurring between StartDate and EndDate.

If StartDate is after EndDate, the result is negative.

If either argument is null, the result is null.

2.2.3.9.15 Extract Year

ExtractYear(Date : Timestamp) : Integer

The ExtractYear operator returns the year component of the given timestamp.

If either argument is null, the result is null.

2.2.3.9.16 Extract Month

ExtractMonth(Date : Timestamp) : Integer

The ExtractMonth operator returns the month component of the given timestamp.

If either argument is null, the result is null.

2.2.3.9.17 Extract Day

ExtractDay(Date : Timestamp) : Integer

The ExtractDay operator returns the day component of the given timestamp.

If either argument is null, the result is null.

2.2.3.9.18 Extract Hour

ExtractHour(Date : Timestamp) : Integer

The ExtractHour operator returns the hour component of the given timestamp.

If either argument is null, the result is null.

2.2.3.9.19 Extract Minute

ExtractMinute(Date : Timestamp) : Integer

The ExtractMinute operator returns the minute component of the given timestamp.

If either argument is null, the result is null.

2.2.3.9.20 Extract Second

ExtractSecond(Date : Timestamp) : Integer

The ExtractSecond operator returns the second component of the given timestamp.

If either argument is null, the result is null.

2.2.3.9.21 Extract Millisecond

```
ExtractMillisecond(Date : Timestamp) : Integer
```

The ExtractMillisecond operator returns the millisecond component of the given timestamp.

If either argument is null, the result is null.

2.2.3.9.22 Now

```
Now()
```

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 during evaluation of an artifact, ensuring that the result of an expression containing Now will always return the same result (determinism).
2) The operation will return 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.

2.2.3.9.23 Today

```
Today()
```

The Today operator returns the date (with no time component) of the start timestamp associated with the evaluation request. As with the Now operator, this methodology ensures determinism and allows for consistent time zone handling within the evaluation.

2.2.3.9.24 Date

```
Date(Year : Integer,
  Month : Integer,
  Day : Integer,
  Hour : Integer,
  Minute : Integer,
  Second : Integer,
  Millisecond : Real,
  TimezoneOffset : Integer) : Timestamp
```

The Date operator returns a date value with the given Year, Month, Day, Hour, Minute, Second, and Millisecond.

If any of Year, Month, or Day is null, the result is null. However, Hour, Minute, Second, and Millisecond may all be null, provided that no value appears in a granularity that is strictly smaller than a granularity that has already been provided.

For example, Hour may be non-null, and if Minute, Second, and Millisecond are all null, they are assumed to be 0. However, if Hour is null, Minute, Second, and Millisecond must all be null as well.

When an argument is null, it is assumed to be the minimum valid value for the domain of the argument, as listed in the following table:

Component	Default
Year	1900 (Minimum year)
Month	1
Day	1
Hour	0
Minute	0
Second	0
Millisecond	0
TimezoneOffset	0

2.2.3.9.25 DateOf

`DateOf(Operand : Timestamp) : Timestamp`

The DateOf operator returns a timestamp value that only specifies the Year, Month, and Day components of the argument.

If the argument is null, the result is null.

2.2.3.9.26 TimeOf

`TimeOf(Operand : Timestamp) : Timestamp`

The TimeOf operator returns a timestamp value that only specifies the Hour, Minute, Second, and Millisecond components of the argument (with the Year, Month, and Day values defaulted to the minimum specifiable date of January 1st, 1900).

If the argument is null, the result is null.

2.2.3.10 Interval Operators

The language must support a complete set of interval operators to ensure that logic involving ranges of values, including temporal operations such as during and overlaps, can be expressed.

In particular, the language must support construction of new intervals, access to the properties of an interval (beginning and ending points, and whether the interval is inclusive or exclusive at both boundaries), as well as computation and comparison involving intervals.

2.2.3.10.1 Point Type Support Operators

The following operators are not interval operators per se, but are required to support general purpose interval maniuplation.

2.2.3.10.1.1 Minimum Value

`Minimum<Any>() : Any`

The Minimum operator returns the minimum representable value for a given type.

The Minimum operator is defined for the Integer, Real and Timestamp types.

2.2.3.10.1.2 Maximum Value

Maximum<Any>() : Any

The Maximum operator returns the maximum representable value for a given type.

The Maximum operator is defined for the Integer, Real and Timestamp types.

2.2.3.10.1.3 Successor

Successor<Any>(Operand : Any) : Any

The Successor operator returns the immediate successor of its argument.

For Integers, the Successor is defined to return the next integer.

For Reals, the Successor is defined to return its argument plus the minimum real increment of 1E-08.

For Timestamps, the Successor is defined to return its argument plus the minimum timestamp increment of 1 millisecond.

If the argument is null, the result is null.

Attempting to invoke the Successor function on a value that is already the maximum value for the given type results in an error.

2.2.3.10.1.4 Predecessor

Predecessor<Any>(Operand : Any) : Any

The Predecessor operator returns the immediate predecessor of its argument.

For Integers, the Predecessor is defined to return the previous integer.

For Reals, the Predecessor is defined to return its argument minus the minimum real increment of 1E-08.

For Timestamps, the Predecessor is defined to return its argument minus the minimum timestamp increment of 1 millisecond.

If the argument is null, the result is null.

Attempting to invoke the Predecessor function on a value that is already the minimum value for the given type results in an error.

2.2.3.10.2 Begin

Begin(Operand : Interval<Any>) : Any

The Begin operator returns the starting point of an interval. Note that the Begin operator is different than accessing the defined beginning point of the interval value, as the defined beginning point is not part of the interval if the beginning of the interval is exclusive.

If the beginning of the interval is exclusive, the Begin operator will return the successor of the defined beginning point of the interval.

If the beginning of the interval is inclusive, and the defined beginning point of the interval is null, the Begin operator will return the minimum value for the point type of the interval.

The following pseudo-code specifies the semantics of the Begin operator:

```
Begin(i) =
  if i.beginIsInclusive then
    IfNull(i.begin, Minimum<T>())
  else
    Successor(i.begin)
```

Unless otherwise noted in the description for a specific operator, interval comparisons and computations are performed using the Begin operator to determine the beginning point for the interval.

The type of the result is the point type of the interval type of the argument.

If the argument is null, the result is null.

2.2.3.10.3 End

```
End(Operand : Interval<Any>) : Any
```

The End operator returns the ending point of an interval. Note that the End operator is different than accessing the defined ending point of the interval, as the defined ending point is not part of the interval if the ending of the interval is exclusive.

If the ending of the interval is exclusive, the End operator will return the predecessor of the defined ending point of the interval.

If the ending of the interval is inclusive, and the defined ending point of the interval is null, the End operator will return the maximum value for the point type of the interval.

The following pseudo-code specifies the semantics of the End operator:

```
End(i) =
  if i.endIsInclusive then
    IfNull(i.end, Maximum<T>())
  else
    Predecessor(i.begin)
```

Unless otherwise noted in the description for a specific operator, interval comparisons and computations are performed using the End operator to determine the ending point for the interval.

The type of the result is the point type of the interval type of the argument.

If the argument is null, the result is null.

2.2.3.10.4 Length

```
Length(Operand : Interval<Any>) : Any
```

The Length operator for intervals determines the length of the interval, and is defined as the difference between the beginning and ending points of the interval for a completely closed interval.

The following pseudo-code specifies the semantics of the Length operator:

```
Length(i) = End(i) – Begin(i)
```


If the beginning of the interval is exclusive, the Successor operator is used to obtain the starting point of the range of the interval. Similarly, if the ending of the interval is exclusive, the Predecessor operator is used to obtain the ending point of the range of the interval.

If the argument is null, the result is null.

2.2.3.10.5 Contains

Contains(Source : Interval<Any>, Point : Any) : Boolean

The Contains operator returns true if a given point exists within a given interval. In other words, if the given point is greater than or equal to the beginning point of the interval, and less than or equal to the ending point of the interval.

For exclusive interval boundaries, exclusive, rather than inclusive, comparison operators are used.

For inclusive interval boundaries, if the interval boundary point is null, the result of the boundary comparison is considered true.

The type of the point must be the same as the point type of the interval argument.

If either argument is null, the result is null.

Note that the Contains operator does not use the Begin or End operators to determine the boundary points of the interval. This is to avoid the use of the Successor and Predecessor operators, instead relying on exclusive, rather than inclusive comparison operators to provide exclusive interval semantics.

The following pseudo-code specifies the semantics of the Contains operator:

```
Contains(i, p) =
  (if i.beginIsInclusive then p >= IsNull(i.begin, p) else p > i.begin)
  and (if i.endIsInclusive then p <= IsNull(i.end, p) else p < i.end)
```

Note that the Contains operator is the inverse of the In operator.

2.2.3.10.6 In (During)

In(Point : Any, Range : Interval<Any>) : Boolean

The In operator returns true if a given point is within a given range. In other words, if the given point is greater than or equal to the beginning point of the interval, and less than or equal to the ending point of the interval.

For exclusive interval boundaries, exclusive, rather than inclusive, comparison operators are used.

For inclusive interval boundaries, if the interval boundary point is null, the result of the boundary comparison is considered true.

The type of the point must be the same the point type of the interval argument.

If either argument is null, the result is null.

Note that the In operator does not use the Begin or End operator to determine the boundary points of the interval. This is to avoid the use of the Successor and Predecessor operators, instead relying on exclusive, rather than inclusive comparison operators to provide exclusive interval semantics.

The following pseudo-code specifies the semantics of the Contains operator:

```
In(i, p) =
  (if i.beginIsInclusive then p >= IsNull(i.begin, p) else p > i.begin)
  and (if i.endIsInclusive then p <= IsNull(i.end, p) else p < i.end)
```

Note that the In operator is the inverse of the Contains operator.

2.2.3.10.7 Includes

Includes(Source : Interval<Any>, Range : Interval<Any>) : Boolean

The Includes operator returns true if a given interval completely includes another. In other words, if the beginning point of the first interval is less than or equal to the beginning 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:

```
Includes(i1, i2) = Begin(i1) <= Begin(i2) and End(i1) >= End(i2)
```

The type of the second argument must be the same as the type of the first argument.

If either argument is null, the result is null.

The Includes operator is the inverse of the IncludedIn operator.

2.2.3.10.8 Included In

IncludedIn(Source : Interval<Any>, Range : Interval<Any>) : Boolean

The IncludedIn operator returns true if a given source interval is completely included in a given range.

The type of the second argument must be the same as the type of the first argument. In other words, if the beginning point of the first interval is greater than or equal to the beginning 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:

```
IncludedIn(i1, i2) = Begin(i1) >= Begin(i2) and End(i1) <= End(i2)
```

If either argument is null, the result is null.

The IncludedIn operator is the inverse of the Includes operator.

2.2.3.10.9 Proper Includes

ProperIncludes(Source : Interval<Any>, Range : Interval<Any>) : Boolean

The ProperIncludes operator returns true if the Source interval completely includes the Range interval, and the Source interval is strictly larger:

```
ProperIncludes(i1, i2) = Includes(i1, i2) and i1 <> i2
```

The type of the second argument must be the same as the type of the first argument.

If either argument is null, the result is null.

The ProperIncludes operator is the inverse of the ProperIncludedIn operator.

2.2.3.10.10 Proper Included In

ProperIncludedIn(Source : Interval<Any>, Range : Interval<Any>) : Boolean

The ProperIncludedIn operator returns true if the Source interval is completely included in the Range interval, and the Source interval is strictly smaller than the Range interval:

```
ProperIncludedIn(i1, i2) = IncludedIn(i1, i2) and i1 <> i2
```

The type of the second argument must be the same as the type of the first argument.

If either argument is null, the result is null.

The ProperIncludedIn operator is the inverse of the ProperIncludes operator.

2.2.3.10.11 Before

```
Before(Operand1 : Interval<Any>, Operand2 : Interval<Any>) : Boolean
```

The Before operator 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:

```
Before(i1, i2) = End(i1) < Begin(i2)
```

The type of the second argument must be the same as the type of the first argument.

If either argument is null, the result is null.

The Before operator is the inverse of the After operator.

2.2.3.10.12 After

```
After(Operand1 : Interval<Any>, Operand2 : Interval<Any>) : Boolean
```

The After operator 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:

```
After(i1, i2) = Begin(i1) > End(i2)
```

The type of the second argument must be the same as the type of the first argument.

If either argument is null, the result is null.

The After operator is the inverse of the Before operator.

2.2.3.10.13 Meets

```
Meets(Operand1 : Interval<Any>, Operand2 : Interval<Any>) : Boolean
```

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 the beginning point of the first interval is equal to the successor of the ending point of the second interval:

```
Meets(i1, i2) = End(i1) = Predecessor(Begin(i2))
  or Begin(i1) = Successor(End(i2))
```

The type of the second argument must be the same as the type of the first argument.

If either argument is null, the result is null.

2.2.3.10.14 Meets Before

MeetsBefore(Operand1 : Interval<Any>, Operand2 : Interval<Any>) : Boolean

The MeetsBefore operator returns true if the first interval ends immediately before the second interval starts. In other words if the ending point of the first interval is equal to the predecessor of the starting point of the second:

MeetsBefore(i1, i2) = End(i1) = Predecessor(Begin(i2))

The type of the second argument must be the same as the type of the first argument.

If either argument is null, the result is null.

The MeetsBefore operator is the inverse of the MeetsAfter operator.

2.2.3.10.15 Meets After

MeetsAfter(Operand1 : Interval<Any>, Operand2 : Interval<Any>) : Boolean

The MeetsAfter operator returns true if the first interval starts immediately after the second interval ends. In other words if the beginning point of the first interval is equal to the successor of the ending point of the second interval:

MeetsAfter(i1, i2) = Begin(i1) = Successor(End(i2))

The type of the second argument must be the same as the type of the first argument.

If either argument is null, the result is null.

The MeetsAfter operator is the inverse of the MeetsBefore operator.

2.2.3.10.16 Overlaps

Overlaps(Operand1 : Interval<Any>, Operand2 : Interval<Any>) : Boolean

The Overlaps operator returns true if the first interval overlaps the second, or if the second interval overlaps the first. In other words, if the starting point of the first interval is less than or equal to the ending point of the second interval, and the starting point of the second interval is less than or equal to the ending point of the first interval:

Overlaps(i1, i2) = Begin(i1) <= End(i2) and Begin(i2) <= End(i1)

The type of the second argument must be the same as the type of the first argument.

If either argument is null, the result is null.

2.2.3.10.17 Overlaps Before

OverlapsBefore(Operand1 : Interval<Any>, Operand2: Interval<Any>) : Boolean

The OverlapsBefore operator returns true if the first interval starts before and overlaps the second. In other words, if the first interval contains the starting point of the second:

OverlapsBefore(i1, i2) = Contains(i1, Begin(i2))

The type of the second argument must be the same as the type of the first argument.

If either argument is null, the result is null.

The OverlapsBefore operator is the inverse of the OverlapsAfter operator.

2.2.3.10.18 Overlaps After

OverlapsAfter(Operand1 : Interval<Any>, Operand2 : Interval<Any>) : Boolean

The OverlapsAfter operator returns true if the first interval overlaps and ends after the second. In other words, if the first interval contains the ending point of the second interval:

```
OverlapsAfter(i1, i2) = Contains(i1, End(i2))
```

The type of the second argument must be the same as the type of the first argument.

If either argument is null, the result is null.

The OverlapsAfter operator is the inverse of the OverlapsBefore operator.

2.2.3.10.19 Begins

Begins(Operand1 : Interval<Any>, Operand2: Interval<Any>) : Boolean

The Begins operator returns true if the first interval begins the second. In other words, if the starting point of the first interval is equal to the starting point of the second; and the ending point of the first interval is less than or equal to the ending point of the second interval:

```
Begins(i1, i2) = Begin(i1) = Begin(i2) and End(i1) <= End(i2)
```

The point type of the intervals must be the same.

If either argument is null, the result is null.

The Begins operator is the inverse of the BegunBy operator.

2.2.3.10.20 Begun By

BegunBy(Operand1 : Interval<Any>, Operand2: Interval<Any>) : Boolean

The BegunBy operator returns true if the second interval begins the first. In other words, if the starting point of the second interval is equal to the starting point of the first; and the ending point of the second interval is less than or equal to the ending point of the first interval:

```
BegunBy(i1, i2) = Begin(i2) = Begin(i1) and End(i2) <= End(i1)
```

The point type of the intervals must be the same.

If either argument is null, the result is null.

The BegunBy operator is the inverse of the Begins operator.

2.2.3.10.21 Ends

Ends(Operand1 : Interval<Any>, Operand2 : Interval<Any>) : Boolean

The Ends operator returns true if the first interval ends the second. In other words, 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:

```
Ends(i1, i2) = Begin(i1) >= Begin(i2) and End(i1) = End(i2)
```

The point type of the intervals must be the same.

If either argument is null, the result is null.

The Ends operator is the inverse of the EndedBy operator.

2.2.3.10.22 EndedBy

EndedBy(Operand1 : Interval<Any>, Operand2 : Interval<Any>) : Boolean

The EndedBy operator returns true if the second interval ends the first. In other words, if the starting point of the second interval is greater than or equal to the starting point of the first; and the ending point of the second interval is equal to the ending point of the first:

```
EndedBy(i1, i2) = Begin(i2) >= Begin(i1) and End(i2) = End(i1)
```

The point type of the intervals must be the same.

If either argument is null, the result is null.

The EndedBy operator is the inverse of the Ends operator.

2.2.3.10.23 Union

Union(Operand1 : Interval<Any>, Operand2 : Interval<Any>, …, OperandN : Interval<Any>) : Interval<Any>

The Union operator for intervals returns the interval that starts at the earliest starting point in the arguments, and ends at the latest starting point in the arguments. Note that if the arguments do not overlap, this operator returns null.

The type of the first argument determines the type of the result, and all subsequent arguments must be of the same type.

If any argument is null, the result is null.

2.2.3.10.24 Intersect

Intersect(Operand1 : Interval<Any>, Operand2 : Interval<Any>, …, OperandN : Interval<Any>) : Interval<Any>

The Intersect operator for intervals returns the interval that defines the overlapping portion of the arguments. Note that if the arguments do not overlap, this operator returns null.

If any argument is null, the result is null.

2.2.3.10.25 Difference

Difference(Operand1 : Interval<Any>, Operand2 : Interval<Any>) : Interval<Any>

The Difference operator for intervals returns the portion of the first interval that does not overlap with the second. Note that if the intervals do not overlap, or if the second interval is properly contained by the first, this operator returns null.

The type of the first argument determines the type of the result, and the type of the second argument must be the same.

If either argument is null, the result is null.

2.2.3.11 List Operators

The language must support a complete set of list operators to allow for the expression of set-based and list-based computation within artifacts.

In particular, the language must support list construction, element access, iteration, and sorting for complete list support, as well as set-based operations such as filtering, union, intersection, difference, and distinction.

Note that many of the list operations involve a *scope*. For example, when filtering, the semantics of the operation involve iterating through the list and evaluating a condition for each element within the list. Access to the current element is provided by naming the scope, and allowing access to the element through the Current and Property operations.

In addition, naming the scope enables nesting of operations that involve scope. For example a nested sub-query can be expressed using a Filter within the Condition of an outer Filter. Naming of the scopes allows the inner condition to access the element within the outer scope.

Note that explicitly attempting to name a scope the same name as an existing scope will result in an error.

For a complete discussion of how scoping within the language is expected to behave, refer to the Execution Model discussion. For specific examples of the use of scoping, refer to the Filter and ForEach operator descriptions.

2.2.3.11.1 Expand

Expand(Source : List<List<Any>>) : List<Any>

The expand operator expands a list of lists into a single list.

If the argument is null, the result is null.

2.2.3.11.2 Indexer

Indexer(Operand : List<Any>, Index : Integer) : Any

The list Indexer operator returns the Indexth element of its argument. The type of the result is the element type of the list.

If either argument is null, the result is null.

2.2.3.11.3 IndexOf

IndexOf(Operand : List<Any>, Element : Any) : Integer

The IndexOf operator searches a list for a specific element, using value-based equality semantics, and returns the 1-based index of the first instance of Element in the list, if found. Otherwise, the result is 0.

If the list contains multiple instances of Element, the index of the first element is returned.

If either argument is null, the result is null.

2.2.3.11.4 Length

Length(Operand : List<Any>) : Integer

The Length operator returns the size of a list.

If the argument is null, the result is null.

2.2.3.11.5 IsEmpty

IsEmpty(Operand : List<Any>) : Boolean

The IsEmpty operator returns true if the list contains no elements; false otherwise.

If the argument is null, the result is null.

2.2.3.11.6 IsNotEmpty

IsNotEmpty(Operand : List<Any>) : Boolean

The IsNotEmpty operator returns true if the list contains any elements; false otherwise.

If the argument is null, the result is null.

2.2.3.11.7 First

First(Operand : List<Any>, OrderBy : List<OrderByItem>) : Boolean

The First operator returns the first element of the list, optionally specifying an ordering.

If an OrderBy argument is provided, the semantics are equivalent to invoking a Sort on the source argument, and returning the first element of the resulting list. For more information on the structure of the OrderBy operand, refer to the Sort operator discussion.

The result type of this operator is the element type of the source list.

If the source argument is null, the result is null.

If the source argument is an empty list, the result is null.

2.2.3.11.8 Last

Last(Operand : List<Any>, OrderBy : List<OrderByItem>) : Boolean

The Last operator returns the last element of the list, optionally specifying an ordering.

If an OrderBy argument is provided, the semantics are equivalent to invoking a Sort on the source argument, and returning the last element of the resulting list. For more information on the structure of the OrderBy operand, refer to the Sort operator discussion.

The result type of this operator is the element type of the source list.

If the source argument is null, the result is null.

If the source argument is an empty list, the result is null.

2.2.3.11.9 Union

Union(Operand1 : List<Any>, Operand2 : List<Any>, …, OperandN : List<Any>) : List<Any>

The Union operator returns a list with all elements from all arguments. Note that this is equivalent to an SQL "UNION ALL" rather than a "UNION DISTINCT". The Distinct operator can be used to provide union distinct behavior if necessary.

The type of the first argument determines the result type, and each subsequent argument must be of the same type.

If any argument is null, the result is null.

2.2.3.11.10 Intersect

Intersect(Operand1 : List<Any>, Operand2 : List<Any>, …, OperandN : List<Any>) : List<Any>

The Intersect operator returns a list with only those elements that appear in every argument.

The type of the first argument determines the result type, and each subsequent argument must be of the same type.

If any argument is null, the result is null.

2.2.3.11.11 Difference

Difference(Operand1 : List<Any>, Operand2 : List<Any>) : List<Any>

The Difference operator returns a list with those elements that appear in the first list, but do not appear in the second list.

The type of the first argument determines the result type, and the second argument must be of the same type.

If either argument is null, the result is null.

2.2.3.11.12 Filter

Filter(Source : List<Any>, Scope : String, Condition : Boolean) : List<Any>

The Filter operator filters a list by a given condition. For each element in the list, the result will contain that element if and only if the condition expression evaluates to true. The Filter operator creates a *scope* which allows access to the current element of the list. The scope is visible within the Condition expression.

The type of the Source argument determines the result type.

The Scope operand is optional and can be used to provide a name for the scope of the filter. This name can then be used by the Current and Property operators to determine which scope to access.

NOTE: The pseudo-syntax provided here does not capture the notion that the Condition expression must be evaluated multiple times, once for each element of the list being filtered.

If the Source is null, the result is null.

If the Scope is null, the default scope "Current" is assumed.

If the Condition evaluates to null for a specific element, that element will not be included in the result.

The following example illustrates a simple filter, given an expression named LabResults:

```
Filter(LabResults, null, GreaterOrEqual(Property("Current.value"), 8.0%));
```

The result of this expression will be the set of LabResults whose value is greater than or equal to 8.0%.

As another example, given an expression named LabResults and an expression named Encounters:

```
Filter(LabResults, "LR",
  IsNotEmpty(Filter(Encounters, "E",
    Includes(Property("E.effectiveTime"), Property("LR.effectiveTime")))));
```

The result of this expression is the set of LabResults that occurred during any encounter in the set of Encounters.

2.2.3.11.13 Contains

Contains(Operand : List<Any>, Element : Any) : Boolean

The Contains operator returns true if a given element exists in a given list.

The type of the element must be the same as the element type of the list argument.

If either argument is null, the result is null.

The Contains operator is the inverse of the In operator.

2.2.3.11.14 In

In(Element : Any, Operand : List<Any>) : Boolean

The In operator returns true if a given element exists in a given list.

The element type of the list must be the same as the type of the element argument.

If either argument is null, the result is null.

The In operator is the inverse of the Contains operator.

2.2.3.11.15 Includes

Includes(Operand1 : List<Any>, Operand2 : List<Any>) : Boolean

The Includes operator returns true if the first list includes all the elements of the second list.

The type of the second argument must be the same as the type of the first argument.

If either argument is null, the result is null.

The Includes operator is the inverse of the IncludedIn operator.

2.2.3.11.16 IncludedIn

IncludedIn(Operand1 : List<Any>, Operand2 : List<Any>) : Boolean

The IncludedIn operator returns true if every element in the first list is included in the second list.

The type of the second argument must be the same as the type of the first argument.

If either argument is null, the result is null.

The IncludedIn operator is the inverse of the Includes operator.

2.2.3.11.17 ProperIncludes

ProperIncludes(Operand1 : List<Any>, Operand2 : List<Any>) : Boolean

The ProperIncludes operator returns true if the first list includes every element of the second list, and the first list is strictly larger.

The type of the second argument must be the same as the type of the first argument.

If either argument is null, the result is null.

The ProperIncludes operator is the inverse of the ProperIncludedIn operator.

2.2.3.11.18 ProperIncludedIn

ProperIncludedIn(Operand1 : List<Any>, Operand2 : List<Any>) : Boolean

The ProperIncludedIn operator returns true if every element of the first list is included in the second list, and the first list is strictly smaller than the second list.

The type of the second argument must be the same as the type of the first argument.

If either argument is null, the result is null.

The PropertyIncludedIn operator is the inverse of the ProperIncludes operator.

2.2.3.11.19 Sort

Sort(Source : List<Any>, OrderBy : List<OrderByItem>) : List<Any>

OrderByItem(Path : String, Ascending : Boolean)

The Sort operator returns a list with the same elements as the argument, sorted in the order specified by the given OrderBy.

The ordering is specified in terms of a list of OrderByItems, each of which specifies a Path that determines the value to be used to determine the ordering, and a flag, Ascending, to indicate whether the order is ascending or descending. The following pseudo-syntax example returns the LabResults sorted by value descending:

```
Sort(LabResults, OrderByItem("value", false));
```

The OrderByItems are optional, if no OrderByItems are specified, the ordering is based on the element type of the list, and is assumed to be ascending. For example, the following pseudo-syntax returns the input list of integers sorted ascending:

```
Sort(List(8, 6, 3, 2, 10), null);
```

The type of the Source argument determines the result type.

If the Source argument is null, the result is null.

2.2.3.11.20 ForEach

ForEach(Source : List<Any>, Scope : String, Element : Any) : List<Any>

The ForEach operator returns a list with the same number of elements, where each element in the result list is obtained by evaluating the Element expression for each element in the source list. The ForEach operator creates a *scope* which allows access to the current element of the list. The scope is visible within the Element expression.

The type of the result is a list whose element type is the type of the Element result.

The Scope operand is optional and can be used to provide a name for the scope of the foreach operation. This name can then be used by the Current and Property operators to determine which scope to access.

NOTE: The pseudo-syntax provided here does not capture the notion that the Element expression must be evaluated multiple times, once for each element of the list being iterated.

If the Source argument is null, the result is null.

If the Scope argument is null, the default scope "Current" is assumed.

If the Element argument evaluates to null for a given element, the resulting list will contain a null for that element.

The following pseudo-syntax example illustrates the use of ForEach to extract effective times given a set of Encounters:

```
ForEach(Encounters, null, Current.effectiveTime);
```

The result is a list containing the effectiveTime value for each encounter in the input.

As another example, the following pseudo-syntax illustrates using nested ForEach operations to compute the cartesian product of LabResults and Encounters:

```
Expand(ForEach(LabResults, "LR",
  ForEach(Encounters, "E",
    Object(labResult = Current("LR"), encounter = Current("E")))));
```

The result is a list of objects representing all possible combinations of LabResults and Encounters. Each object has two properties, one containing the lab result, and one containing the encounter. Note that the expand is necessary because the result of the first ForEach operation is actually a list of lists, the expand flattens that to a single list.

2.2.3.11.21 Distinct

```
Distinct(Source : List<Any>) : List<Any>
```

The Distinct operator returns a list with only the unique elements of the source list, as determined by value-based equality.

The type of the result is determined by the type of the argument.

If the Source argument is null, the result is null.

2.2.3.11.22 Current

```
Current(Scope : String) : Any
```

The Current operator returns the current element in the given scope. Certain operators such as Filter and ForEach introduce a *scope* which allows expressions to reference the current element being processed. Operators that introduce a scope allow the scope to be named so that nested operations can distinguish between multiple scopes.

The Scope operand is optional. If it is null, the default scope of Current is assumed.

The result type of this operator is the type of the element in the given scope.

For an example of the use of the Current operator, refer to the ForEach operator.

2.2.3.12 Aggregate Operators

The language must support a complete set of aggregate operators to ensure that reasoning involving computation on sets of values can be expressed.

Aggregate operators have two variants:

- **Simple list aggregates** – Where the aggregate is performed directly on the elements within the list. For example, taking the sum of a list of integers is a simple aggregate.
- **Path-based aggregates** – Where the aggregate is performed on a property specified by a path that is applied to each element.

In general, nulls encountered during aggregation are ignored.

2.2.3.12.1 Count

Count(Source : List<Any>, Path : String) : Integer

The Count operator returns the number of elements in the given list.

If the Source argument is null, the result is null.

If specified, the Path argument identifies the path to the value on each element in the Source list.

Null elements within the list are ignored.

If the list is empty, the result is 0.

2.2.3.12.2 Sum

Sum(Source : List<Any>, Path : String) : Any

The Sum operator returns the numeric sum of the elements in the list.

The type of the result is the type of the element being aggregated. For simple list aggregates, this is the element type of the list. For path-based aggregates, this is the type of the property referenced by the path.

If the Source argument is null, the result is null.

If specified, the Path argument identifies the path to the value on each element in the Source list.

Null elements within the list are ignored.

If the list is empty, the result is null.

The following pseudo-syntax example illustrates a simple sum:

```
Sum(List(1, 2, 3, 4, 5), null);
```

And an example summing the values from a given set of LabResults:

```
Sum(LabResults, "value");
```

2.2.3.12.3 Min

Min(Source : List<Any>, Path : String) : Any

The Min operator returns the minimum value of any element in the list.

The type of the result is the type of the element being aggregated. For simple list aggregates, this is the element type of the list. For path-based aggregates, this is the type of the property referenced by the path.

The aggregate element type must support comparison operations.

If the Source argument is null, the result is null.

If specified, the Path argument identifies the path to the value on each element in the Source list.

Null elements within the list are ignored.

If the list is empty, the result is null.

2.2.3.12.4 Max

Max(Source : List<Any>, Path : String) : Any

The Max operator returns the maximum value of any element in the list.

The type of the result is the type of the element being aggregated. For simple list aggregates, this is the element type of the list. For path-based aggregates, this is the type of the property referenced by the path.

The aggregate element type must support comparison operations.

If the Source argument is null, the result is null.

If specified, the Path argument identifies the path to the value on each element in the Source list.

Null elements within the list are ignored.

If the list is empty, the result is null.

2.2.3.12.5 Average

Avg(Source : List<Any>, Path : String) : Any

The Avg operator returns the numeric average of all elements in the list.

The type of the result is the type of invoking division on the element type being aggregated. For simple list aggregates, this is the element type of the list. For path-based aggregates, this is the type of the property referenced by the path.

The aggregate element type must support the addition and division operations.

If the Source argument is null, the result is null.

Null elements within the list are ignored.

If the list is empty, the result is null.

2.2.3.12.6 Median

Median(Source : List<Any>, Path : String) : Any

The Median operator returns the statistical median of all elements in the list.

The type of the result is the type of invoking division on the element being aggregated. For simple list aggregates, this is the element type of the list. For path-based aggregates, this is the type of the property referenced by the path.

The aggregate element type must be numeric, or support comparison, subtraction, and division operations.

If the Source argument is null, the result is null.

If specified, the Path argument identifies the path to the value on each element in the Source list.

Null elements within the list are ignored.

If the list is empty, the result is null.

2.2.3.12.7 Mode

Mode(Source : List<Any>, Path : String) : Any

The Mode operator returns the statistical mode (most common value) of all elements in the list.

The type of the result is the type of the element being aggregated. For simple list aggregates, this is the element type of the list. For path-based aggregates, this is the type of the property referenced by the path.

The aggregate element type must support comparison.

If the Source argument is null, the result is null.

If specified, the Path argument identifies the path to the value on each element in the Source list.

Null elements within the list are ignored.

If the list is empty, the result is null.

2.2.3.12.8 Standard Deviation

StdDev(Source : List<Any>, Path : String) : Real

The StdDev operator computes the sample standard deviation of the elements in the list.

Sample standard deviation is the square root of the sample variance.

The aggregate element type must be numeric. For simple list aggregates, this is the element type of the list. For path-based aggregates, this is the type of the property referenced by the path.

If the Source argument is null, the result is null.

If specified, the Path argument identifies the path to the value on each element in the Source list.

Null elements within the list are ignored.

If the list is empty, the result is null.

2.2.3.12.9 Population Standard Deviation

PopStdDev(Source : List<Any>, Path : String) : Real

The PopStdDev operator computes the population standard deviation of the elements in the list.

The population standard deviation is the square root of the population variance.

The aggregate element type must be numeric. For simple list aggregates, this is the element type of the list. For path-based aggregates, this is the type of the property referenced by the path.

If the Source argument is null, the result is null.

If specified, the Path argument identifies the path to the value on each element in the Source list.

Null elements within the list are ignored.

If the list is empty, the result is null.

2.2.3.12.10 Variance

`Variance(Source : List<Any>, Path : String) : Real`

The Variance operator computes the sample variance of the elements in the list.

The sample variance is the 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.

The aggregate element type must be numeric. For simple list aggregates, this is the element type of the list. For path-based aggregates, this is the type of the property referenced by the path.

If the Source argument is null, the result is null.

If specified, the Path argument identifies the path to the value on each element in the Source list.

Null elements within the list are ignored.

If the list is empty, the result is null.

2.2.3.12.11 Population Variance

`PopVariance(Source : List<Any>, Path : String) : Real`

The PopVariance operator computes the population variance of the elements in the list.

The population variance is the average distance of the data elements from the population mean.

The aggregate element type must be numeric. For simple list aggregates, this is the element type of the list. For path-based aggregates, this is the type of the property referenced by the path.

If the source argument is null, the result is null.

Null elements within the list are ignored.

If the list is empty, the result is null.

2.2.3.12.12 All True

`AllTrue(Source : List<Any>, Path : String) : Boolean`

The AllTrue operator returns true if all of the elements in the list are true.

The aggregate element type must be boolean. For simple list aggregates, this is the element type of the list. For path-based aggregates, this is the type of the property referenced by the path.

If the Source argument is null, the result is null.

If specified, the Path argument identifies the path to the value on each element in the Source list.

Null elements within the list are ignored.

If the list contains no true values, the result is true. This applies to the empty list, as well as to a list containing only nulls, or a list containing structured values where the path evaluation on each value results in a null.

2.2.3.12.13 Any True

`AnyTrue(Source : List<Any>, Path : String) : Boolean`

The AnyTrue operator returns true if any of the elements in the list are true.

The aggregate element type must be boolean. For simple list aggregates, this is the element type of the list. For path-based aggregates, this is the type of the property referenced by the path.

If the Source argument is null, the result is null.

If specified, the Path argument identifies the path to the value on each element in the Source list.

Null elements within the list are ignored.

If the list contains no true values, the result is false. This applies to the empty list, as well as to a list containing only nulls, or a list containing structured values where the path evaluation on each value results in a null.

2.2.3.13 Data Request Operators

The language must provide some mechanism for accessing external data that is provided as an input to the evaluation. To ensure model independence, these operators should not be directly expressed in terms of any particular data model.

In this context, *external data* refers to the information required to evaluate the artifact. For example, patient information, problem lists, medication history, etc. The language makes no assumptions about *how* this information is provided to the evaluation, only that it is available.

2.2.3.13.1 Request

The language must support the ability to retrieve data from an external source. The Request operator provides this functionality, but is limited to specifying the following:

- **Type** – The type of the data involved, specified by name
- **Cardinality** – Whether the data request will return a single value or a list of values of the specified type (Single or Multiple cardinality)
- **Id Property** – The name of the "id" or uniquely identifying property for the type

If the Cardinality is Single, the type of the result is the type specified by the Data Type Name of the request.

If the Cardinality is Multiple, the type of the result is a list whose elements are of the type specified by the Data Type Name of the request.

At run-time, a cardinality mismatch results in an error. For example, if the defined cardinality is Multiple, the run-time result must be a list type. If the run-time result is a single value, an error occurs. Similarly, if the defined cardinality is Single, the run-time result must not be a list type or an error occurs.

The result of a request is expected to return the same data for subsequent invocations within the same evaluation request. This means in particular that any data updates made during the evaluation request are not visible to the artifact. In effect, the data is a snapshot of the data as of the start of the evaluation. This ensures strict deterministic and functional behavior of the

artifact, and allows the implementation engine freedom to cache intermediate results in order to improve performance.

2.2.3.14 Clinical Operators

In addition to a basic retrieval operation as described above, the language must provide the ability to request clinically relevant data as described in the Clinical Context and Data Requirements non-functional requirements.

2.2.3.14.1 Clinical Types

In general, the data types required for dealing with clinical data will be dictated and represented by the data model chosen in any particular artifact. However, in order to describe some clinical operations, some data types must be commonly understood. In particular, the concept of a clinical Code is an important aspect of supporting the clinical operators described in this section. From the perspective of the expression language in general, a Code is simply a structured value type, and for the purposes of these operators, it is assumed to have the structure of a CD type as defined by the HL7 V3 DataTypes R2.

2.2.3.14.2 Templates

Many clinical data models are built as a core structural representation that is then augmented with *templates* that further constrain the model to describe how specific concepts are represented within the model. For example, the base model may describe a general purpose Problem structure, and then provide several different templates that all use that base structure in different ways to represent different types of problems.

Conceptually then, the data type consists not only of the structural type but the particular template being used. Note that the template is not visible within the language, it is only an input to the clinical request to help identify the data type and applicable constraints. In other words, the template identifier is only an input to the clinical request, and does not have any other visible effects within the language.

2.2.3.14.3 Clinical Request

The ClinicalRequest operator provides a data retrieval operation that includes clinically-relevant criteria:

- **Clinical Data Type** – The type of the clinical data to be retrieved, specified by name as well as an optional template identifier.
- **Codes** – The set of codes defining the clinical data. Only clinical data matching the codes in the set will be returned. If no codes are specified, clinical data with any code will be returned.
- **Code Property** – The name of the property on the clinical data type that will be used to compare the codes.
- **Date Range** – The date range for clinical data. Only data within the specified date range will be returned. If no date range is specified, clinical data of any date will be returned.
- **Date Property** – The name of the property on the clinical data type that will be used to compare the dates.
- **Status** – The status of the clinical data. Only data with the specified status (or in a set of statuses) will be returned. If no status is specified, clinical data of any status will be returned.

- **Status Property** – The name of the property on the clinical data type that will be used to compare the statuses.

These criteria are designed to allow the implementation environment to determine and communicate the data requirements for an artifact, or group of artifacts, to a consumer to allow the clinical data source to quickly and easily gather all and only the relevant clinical information for delivery to the evaluation environment. This supports both the near-real-time clinical decision support scenario where the evaluation environment is potentially separate from the medical records system environment, as well as the quality measurement scenario where the interface between the implementation engine and the artifact should be as simple as possible.

2.2.3.14.4 Terminology Operators

In addition to requesting clinical data, artifacts will often require the ability to reason about terminologies. The operators in this section define the expected semantics of these high-level operators. In practice, these operations could be implemented by calling an appropriate terminology service.

2.2.3.14.4.1 Value Set

ValueSet(Id : String, Version : String, Authority : String) : List<Code>

The ValueSet operator returns a list of codes whose elements are defined by the given value set authority for the given value set id and version.

If no version is specified, the current version of the value set is returned.

If no authority is specified, the implementation environment should provide a default authority.

If Id is null, the result is null.

2.2.3.14.4.2 In Value Set

InValueSet(Code : Code, Id : String, Version : String, Authority : String)
InValueSet(Codes : List<Code>, Id : String, Version : String, Authority : String) : Boolean

The InValueSet operator returns true if the given Code is, or all of a list of Codes are, in the given value set.

If no version is specified, the current version of the value set is used.

If no authority is specified, the implementation environment should provide a default authority.

If Code, Codes, or Id is null, the result is null.

2.2.3.14.4.3 Subsumes

Subsumes(Ancestor : Code, Descendent : Code) : Boolean

The Subsumes operator returns true if the arguments are of the same code system, and the ancestor code subsumes the descendent code in the hierarchy of the code system. If the codes are the same code, the operator returns true.

If either argument is null, the result is null.

Note that this requirement does not imply that the evaluation engine be capable of dealing with hierarchical value sets, rather the expectation is that the evaluation environment would integrate with a terminology service to provide this functionality.

For the purposes of the conceptual requirements, Subsumption functionality is considered desirable, but not necessary. An implementation could choose not to provide these operators.

2.2.3.14.4.4 SetSubsumes

`SetSubsumes(Ancestors : List<Code>, Descendents : List<Code>) : List<Code>`

This operator returns the list of codes in the Descendents that are subsumed by some code in the list of Ancestors.

If either argument is null, the result is null.

2.2.3.14.4.5 IsEquivalent

`IsEquivalent(Left : Code, Right : Code) : Boolean`

The IsEquivalent operator returns true if the left argument is equivalent to the right argument. For example, codes from different code systems that mapped to the same concept would compare true with this operator.

If either argument is null, the result is null.

2.2.4 Semantic Validation

Semantic validation within the language is the process of verifying that the meaning of an expression is valid. This involves determining the type of each expression, and verifying that the arguments to each invocation have the correct type. This document provides a conceptual description of this process in order to ensure that the intended semantics of the type system described by the functional requirements is clear.

This section assumes the reader has some familiarity with language processing techniques and terminology.

The process proceeds as follows:

The abstract syntax tree of the expression being validated is traversed and the type of each node is determined. 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:

Table 9 - Type Inference by Language Element

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 type of the Parameter being referenced.
ExpressionRef	The type of the node is the type of the expression being reference.

Request	The type of the node is either the type of the data being requested (Single cardinality), or a list of the type of the data being requested (Multiple cardinality).
ValueSet	The type of the node is a list of codes.
Operator	Generally, the type of the node is determined by resolving the type of each argument, and then using this signature to determine the resulting type of the operator.

During validation, the implementation must maintain a stack of symbols that track the type of the object currently in scope. This allows the type of context-sensitive operators such as Current and Property to be determined. Refer to the Execution Model section for a description of the expected semantics of the evaluation-time stack.

2.2.5 Execution Model

The execution model described here is provided to ensure that the intended semantics of the language are communicated clearly and completely. It is not intended to prescribe a particular implementation, only to describe the behavior that any implementation is expected to exhibit.

All logic in the language is represented as *expressions*. The language is pure functional, meaning no operations are allowed to have side effects visible in the language of any kind. An expression may consist of any number of other expressions and operations, so long as they are combined according to the semantic rules for each operation as described in the Semantic Validation section above.

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. For a quality measure scenario where a data warehouse or other relatively static set of data is the source, the implementation might simply access the source. However, for a near-real-time scenario, 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 evaluation 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 the requested clinical information. In order to be represented in this data set, a given structured value must have at least the following:

Table 10 - Required Properties of "Trackable" Information

Property	Description
Identifier	A property, or set of properties, that uniquely identify the item within the scope of its type.
Codes	An optional code, or list of codes, that identify the associated clinical codes for the item.
Date	An optional date time defining the clinically relevant date and/or time of the item.
Status	An optional status defining the clinically relevant status for the item.

During evaluation, 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 in order 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:

Table 11 - Evaluation Semantics by Language Element

Category	Evaluation
Literal	The result of the node is the value of the literal represented.
Operation	The result of the node is the result of the operation described by the node given the results of the arguments to the invocation.
Clinical Request	The result of the node is the result of retrieving the data represented by the request (i.e. a list of structured values of the type defined in the request, and matching any criteria specified as part of the request.
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.

During evaluation, the implementation must maintain a stack that is used to represent the value that is currently in context. Certain operations within the language are defined with a scope, and these operations use the stack to represent this scope. The following table details these operations:

Table 12 - Stack Effect by Operator

Operation	Stack Effect
ObjectRedefine	The Source argument is pushed on to the stack prior to evaluating each Property expression. The stack is popped before the result is returned.
Filter	For each item in the Source argument, 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 argument, the item is pushed on to the stack, the Element expression is evaluated, and the item is popped off of the stack.

The Scope argument to these operators provides an optional name for the item being pushed on to the stack. This name can then be used within the Current and Property operators 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.

The final result of any expression is the result of the topmost invocation element.

3 CONCLUSIONS

This document provides a conceptual foundation for the representation of reasoning within health quality artifacts. Any candidate language can be compared against this set of requirements to determine how well the language will fit within the health quality domain. In addition, the requirements laid out here provide a way to clearly identify commonalities between different implementations, providing a path for conceptual mapping, and potentially translation between various concrete realizations of these concepts.

4 REFERENCES

- *HL7 Implementation Guide: Clinical Decision Support Knowledge Artifact Implementation Guide, Release 1 (pending publication)*
- *HL7 Version 3 DSTU: Representation of the Health Quality Measures Format (eMeasure), DSTU Release 2 (pending publication)*
- *HL7 Version 3 Standard: GELLO; A Common Expression Language, Release 2*
- *Object Constraint Language, OMG Available Specification Version 2.0*
- *Health eDecisions CDS Artifact Sharing, Use Case 1*
- *Health Level Seven Arden Syntax for Medical Logic Systems, Version 2.9*
- *Foundations of Databases. Abiteboul, Hull, Vianu, 1995*
- *Temporal Data and The Relational Model. Date, Darwen, Lorentzos, 2003*
- *Databases, Types, and the Relational Model, 3rd edition. Date, Darwen, 2007*
- *Compilers: Principles, Techniques, and Tools. Aho, Sethi, Ullman, 1998*
- *Unicode Standard Annex #44: Unicode Character Database (http://www.unicode.org/reports/tr44/)*