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 Specified Material in whole or part, into products and services, or to enjoy additional authorizations granted to HL7 ORGANIZATIONAL MEMBERS as noted below, must become ORGANIZATIONAL MEMBERS of HL7.

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

C. NON-MEMBERS, who register and agree to the terms of HL7’s IP policy for Specified Material, are authorized, without additional charge, to read and use the Specified Material for evaluating whether to implement, or in implementing, the Specified Material, and to use Specified Material to develop and sell products and services that implement, but do not directly incorporate, the Specified Material in whole or in part. NON-MEMBERS wishing to incorporate additional items of Specified Material in whole or part, into products and services, or to enjoy the additional authorizations granted to HL7 ORGANIZATIONAL MEMBERS, as noted above, must become ORGANIZATIONAL MEMBERS of HL7.

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

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

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

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

<table>
<thead>
<tr>
<th>Terminology</th>
<th>Owner/Contact</th>
</tr>
</thead>
<tbody>
<tr>
<td>SNOMED CT</td>
<td>International Healthcare Terminology Standards Development Organization (IHTSDO) [<a href="http://www.ihtsdo.org/snomed-ct/get-snomed-ct">http://www.ihtsdo.org/snomed-ct/get-snomed-ct</a> or <a href="mailto:info@ihtsdo.org">info@ihtsdo.org</a>](<a href="http://www.ihtsdo.org/snomed-ct/get-snomed-ct">http://www.ihtsdo.org/snomed-ct/get-snomed-ct</a> or <a href="mailto:info@ihtsdo.org">info@ihtsdo.org</a>)</td>
</tr>
<tr>
<td>Logical Observation Identifiers Names &amp; Codes (LOINC)</td>
<td>Regenstrief Institute</td>
</tr>
<tr>
<td>International Classification of Diseases (ICD) codes</td>
<td>World Health Organization (WHO)</td>
</tr>
<tr>
<td>NUCC Health Care Provider Taxonomy code set</td>
<td>American Medical Association. Please see 222.nucc.org. AMA licensing contact: 312-464-5022 (AMA IP services)</td>
</tr>
</tbody>
</table>
GELLO: A Common Expression Language, Release 2

Table of Contents

1 Introduction
  1.1 What is GELLO?
2 Requirements for an Expression Language in the Clinical Context
  2.1 Non-Functional Requirements
  2.2 Use Cases
  2.3 A Clinical Decision Support Data Model
  2.4 Expressions and Clinical Guidelines
3 GELLO: Goals and Properties
4 OCL
  4.1 Why OCL?
5 GELLO
  5.1 GELLO Types
    5.1.1 Basic Types
    5.1.2 Model Types
    5.1.3 Collection Types
    5.1.4 Tuple Type
    5.1.5 Enumeration Types
  5.2 Names
5.3 Properties
5.3.1 Attributes
5.3.2 Operations
5.4 Context and References to Contextual Instances
5.4.1 Context, Self and Implicit References to Contextual Instances
5.4.2 Package Context and Pathname
5.4.3 Navigation through Associations
5.5 Variable Declaration
5.5.1 Declaring References to Instances of Classes
5.5.2 Scope of Declarations
5.6 Reflection
5.7 Casting
5.8 Built-in Operators
5.8.1 Arithmetic Operators "+", ",", "**"
5.8.2 Arithmetic Operator "/"
5.8.3 Arithmetic Operators "div" and "mod"
5.8.4 Arithmetic Operator unary minus "-"
5.8.5 Comparison Operators ",", ",", ",", ",", "="
5.8.6 Mathematical Operator "abs"
5.8.7 Mathematical Operator "acos"
5.8.8 Mathematical Operator "asin"
5.8.9 Mathematical Operator "atan"
5.8.10 Mathematical Operator "ceiling"
5.8.11 Mathematical Operator "cos"
5.8.12 Mathematical Operator "exp"
5.8.13 Mathematical Operator "floor"
5.8.14 Mathematical Operator "log"
5.8.15 Mathematical Operator "max"
5.8.16 Mathematical Operator "min"
5.8.17 Mathematical Operator "power"
5.8.18 Mathematical Operator "rand"
5.8.19 Mathematical Operator "sin"
5.8.20 Mathematical Operator "sqrt"
5.8.21 Mathematical Operator "tan"
5.8.22 String Operators "size", "concat", "toUpper", "toLowerCase", "substring", ",", and "<>"
5.8.23 String Operators "toChar", "lpad", "rpad", "ltrim", "rtrim" and "replace"
5.10 Collection Operators
5.10.1 The 'Arrow' Notation
5.10.2 Single Instances as Collections
5.10.3 Operator Select
5.10.4 Operator Reject
5.10.5 Operator Collect
5.10.6 Operator ForAll
5.10.7 Operator Iterate
5.10.8 Operator Exists
5.10.9 Operator Flatten
5.10.10 Operator Size
5.10.11 Operator Count
5.10.12 Operators "Max" and "Min"
5.10.13 Operator Includes
5.10.14 Operator IncludesAll
5.10.15 Operator IsEmpty
5.10.16 Operator notEmpty
5.10.17 Operator Sum
5.10.18 Operator FirstN
5.10.19 Operator LastN
5.10.20 Operator ElemAt
5.10.21 Operator Reverse
5.10.22 Operator SortBy
5.10.23 Operator Intersection
5.10.24 Operator Union
5.10.25 Operator Including
5.10.26 Operator Excluding
5.10.27 Operator Join
5.10.28 Operator Average
5.10.29 Operator Stdev
5.10.30 Operator Variance
5.10.31 Operator Median
1 Introduction

GELLO is intended to be a standard expression language for decision support. Its specification has been developed in coordination with the HL7 Clinical Decision Support TC (CDSTC). The effort, begun in 2001, has been carried out with input from other TCs and SIGs as well, in order to take account of common needs for constraint specification and query formulation, and the following groups have been consulted in developing the specification: Control/Query, Modeling and Methodology, and Templates.

The syntax of the GELLO language can be used with any object-oriented data model. In the context of clinical decision support, such an OO data model can be any R-MIM view of the HL7 RIM. An example -- out of many possible-- of an R-MIM view of the HL7 RIM is the "virtual medical record" (or vMR), as it is referred to in the CDSTC. The vMR functions as a limited view of the multiple classes in the HL7 RIM, showing only those classes relevant to a clinical decision support application.

Based on the premise that GELLO can fully provide expression support for any properly defined view of the HL7 RIM, it has been possible for us to develop GELLO regardless of any particular specification of an OO data model. It is thus only necessary, when producing a set of decision support applications using GELLO to specify the particular object-oriented model used.

As discussed further below, GELLO is based on the Object Constraint Language (OCL). OCL is well-developed as a constraint language and has a number of features that make it desirable for use as an expression language. GELLO incorporates most of OCL functionality with the exception of some unneeded capabilities which have been removed: a) pre and post conditions for constraints – not used in GELLO and b) invariants – also for constraints. These features are not required because GELLO is designed for writing queries, and not for constraining models. The UML ITS implements the semantics of the HL7 Abstract Data Types in such a way that HL7 data types are mapped into the core UML and OCL kernel data types where such mappings are appropriate. Hence, GELLO, plus the functionality provided by UML ITS for handling HL7 Abstract Data Types, is intended to provide full expression support inasmuch as it is intended to use only standard features wherever possible.

In regard to existing standards for decision support, the evolution of Arden Syntax data references to compatibility with GELLO is already under way. GELLO is not intended to be a substitute for Arden Syntax, although it could be used to write MLMs. In addition, however, GELLO can be used to support a variety of specifications of expressions in decision support settings other than MLMs.

1.1 What is GELLO?

GELLO is a class-based, object-oriented (OO) language that is built on existing standards. GELLO expression language is based on the Object Constraint Language (OCL), developed by the Object Management Group. Relevant components of OCL have been selected and integrated into the GELLO to provide a suitable framework for manipulation of clinical data for decision support in health care.

The GELLO language can be used to:

- Build up expressions to extract and manipulate data from medical records.
- Construct decision criteria by building up expressions to reason about particular data features/values. These criteria can be used in decision-support knowledge bases such as those designed to provide alerts and reminders, guidelines, or other decision rules.
- Create expressions, formulae, etc. for other applications.

GELLO has been designed in the context of a guideline execution model proposed in the HL7 CDSTC (GeM). The guideline execution model consists of a series of steps: actions, decisions, patient-state, branches and synchronization. GELLO perfectly fits in this execution model, providing a standard interface to medical record systems and other data or knowledge sources, specifying decision criteria, and abstracting or deriving summary values. The object-oriented approach for the language has the flexibility and extensibility that is needed for implementation in a broad range of applications.

The expression language is strongly typed and object-oriented. In order to facilitate the process of encoding and evaluation of expressions and more importantly, to maximize the ability to share such expressions, GELLO includes basic built-in data types (§5.1), while providing the necessary syntactic mechanisms to manipulate an OO data model compatible with the HL7 RIM, and access all the data model associated classes and methods. This is especially important in enabling decision rules and guidelines to successfully support different data models, inasmuch as classes and relationships specified could vary from one data model to another.

This document contains the full software specification for GELLO expression language. It is organized as follows:

Section 2 describes the requirements for an expression language for clinical use. Section 3 describes the main goals
and properties of GELLO to meet such requirements. Section 4 briefly describes the Object Constraint Language (OCL) features.

Section 5 describes OCL features included in GELLO expression language, including basic data types in §5.1.1, model types in §5.1.2 (classes in the data model), collection types in §5.1.3, properties of model types (attributes and methods) in §5.3, and variables in §5.5. Variables are named GELLO expressions with a predefined, limited scope that can be used anywhere a GELLO expression can be used (see §5.5.2). In order to preserve GELLO as a side-effect-free language, the mechanisms for creating variables as instances of classes is delegated to the underlying data model. This section also describes GELLO built-in operators (see §5.9), and their syntax and semantics, and discusses the tuple type as an aggregation type (see §5.1.4) and tuple operators (see §5.11) supported by GELLO.

Section 6 describes the syntax of the GELLO grammar. Section 7 gives examples of GELLO expressions for retrieving information. It also describes GELLO expressions used to build decision criteria, perform abstraction or derive summary values. Finally, appendices A, B and C contain reference diagrams of the HL7 v3 Data Types, HL7 RIM, UML OCL core kernel declarations. Appendix D depicts a simple data model used to illustrate examples along the document.

2 Requirements for an Expression Language in the Clinical Context

2.1 Non-Functional Requirements

This section summarizes the non-functional requirements which the GELLO expression language meet.

1. The syntax of the expression language is object-oriented (OO).
2. The expression language is the means for accessing and manipulating data retrieved from an object-oriented data model. In the clinical context, such data model may be a RMIM – a refined view of the HL7 RIM.
3. Expressions can be used to retrieve information from data sources, build decision criteria, and reason about specific data/values. Expressions are text strings that must conform to the definition of an expression in the GELLO language specification (section 6).
4. Temporal data can be handled by expressions containing time-related relationships. Temporal operators are part of the HL7 RIM and can be referred to using the specified notation as described in section 5.12 of this document.
5. The object-oriented approach allows modularity, encapsulation and extensibility. Thus use of GELLO:
   1. Provides platform-independent support for mapping to any standard OO data model used. Therefore eliminating the need for implementation-specific encoding methods for information retrieval as part of knowledge content (guidelines, alerts, etc.).
   2. Simplifies the evaluation of clinical data objects.
   3. Facilitates sharing of decision logic and other computer expressions.

2.2 Use Cases

The following use cases define a goal-oriented set of interactions between GELLO and external actors. Actors are users and/or other systems that interact with GELLO. A use case is initiated by a user with a particular goal in mind, and complete successfully when such goal is satisfied. Use cases are defined in terms of actors (who) interacting with the system (what) for a particular purpose (goal) without dealing with system internals.

User profiles:

- **Medical experts** not familiar with computer languages, with some basic experience with computers (e.g. word-processing, software packages)
- **Researchers** not familiar with computer languages, with string scientific knowledge and some basic experience with computers (e.g. word-processing, software packages)
- **Health Economists** with expertise in health economics and communication with health care providers, though not familiar with computer languages, with some basic experience with computers (e.g. word-processing, software packages)
- **Epidemiologist** with expertise in public health issues. S/he may not familiar with computer languages, with some basic experience with computers (e.g. word-processing, software packages)
- **Knowledge engineers** with programming experience and some basic medical knowledge.
- **Programmers** with ample programming experience and no medical expertise.

Scenarios:

- i. To write implementation-independent expressions that can be embedded in multiple clinical applications (guidelines, RIM derivation expression, template constraints, etc.) to retrieve patient information from an OO data model.
- ii. To write implementation-independent expressions that can be embedded in multiple clinical applications (guidelines, RIM derivation expression, template constraints, etc.) for representing knowledge and decision logic in a medical context.
- iii. To enable existing clinical applications (guidelines, RIM derivation expression, template constraints, etc.) to be
made executable and populate them with data from electronic patient records (derives from (i) and (ii))

- iv. To allow researchers, health economists, epidemiologists and clinical and health care providers to implement expressions that can be embedded in research, clinical, public health protocols, templates, or any application that requires expressions that requires retrieval and evaluation of data from an OO data model
- v. To ease the evaluation of clinical data objects (derives from (i) and (ii)).
- vi. To facilitate sharing of decision logic and other computer-based expressions.

### Table 1: Use Case 1

<table>
<thead>
<tr>
<th>Use Case 1</th>
<th>Expressions into clinical applications to retrieve patient-related data</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Description</strong></td>
<td>Map clinical knowledge into multiple clinical applications by embedding implementation-independent GELLO expressions to retrieve patient information from an OO data model</td>
</tr>
<tr>
<td><strong>Actors</strong></td>
<td>Clinicians, researchers, epidemiologists, knowledge engineers, programmers</td>
</tr>
</tbody>
</table>
| **Assumptions** | • Expressions are strings that comply with the GELLO specification for expressions  
• The general structure of the clinical application is correct, or there is a mechanism that validates it  
• Patient and other clinical data sources are available and in an OO format compatible with the HL7 RIM |
| **Steps** | • GELLO expressions are embedded in the appropriate locations in the clinical application  
• GELLO expressions satisfy the GELLO syntax  
• GELLO expressions are validated by a parser |
| **Variations** | • Parser detects errors and highlights them  
• User corrects errors |

### Table 2: Use Case 2

<table>
<thead>
<tr>
<th>Use Case 2</th>
<th>Expressions in clinical applications for decision logic</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Description</strong></td>
<td>Represent knowledge and decision logic in multiple clinical applications by embedding implementation-independent GELLO expressions that can be evaluated.</td>
</tr>
<tr>
<td><strong>Actors</strong></td>
<td>Clinicians, researchers, epidemiologists, knowledge engineers, programmers</td>
</tr>
</tbody>
</table>
| **Assumptions** | • Expressions are strings that comply with the GELLO specification for expressions  
• The general structure of the clinical application is correct, or there is a mechanism that validates it  
• Patient data sources are available and in an OO format compatible with the HL7 RIM |
| **Steps** | • GELLO expressions are embedded in the appropriate locations in the clinical application  
• GELLO expressions satisfy the GELLO syntax  
• GELLO expressions are validated by a parser |
| **Variations** | • Parser detects errors and highlights them  
• User corrects errors |

### Table 3: Use Case 3

<table>
<thead>
<tr>
<th>Use Case 3</th>
<th>Automatic population and execution of applications</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Description</strong></td>
<td>Derived from Use cases 1 and 2, clinical applications that contain GELLO expressions can be populated and executed independently of the environment.</td>
</tr>
<tr>
<td><strong>Actors</strong></td>
<td>Clinicians, researchers, epidemiologists, health economists</td>
</tr>
</tbody>
</table>
| **Assumptions** | • Expressions have been previously validated and they conform to the GELLO syntax  
• The general structure of the application is correct, or there is a mechanism that validates it  
• Data sources are available and in an OO format compatible with the HL7 RIM |
2.3 A Clinical Decision Support Data Model

GELLO can provide full expression support for any properly defined object-oriented data model in general. In a clinical context, GELLO can interact with any HL7 RIM-based OO data model serving as intermediary to heterogeneous medical record systems.

Although this approach represents a paradigm shift in data representation, moving from time-stamped atomic data types to an object-oriented data model, it is an ongoing effort towards a standard for exchange, management and integration of clinical data much needed by the community.

2.4 Expressions and Clinical Guidelines

The need for a language to formulate expressions to extract and manipulate clinical data is clear. Ideally such a language should be (Click here to go to appropriate reference):

- vendor-independent
- platform-independent
- object-oriented and compatible with the HL7 RIM
- easy to read/write
- side-effect free. The result of an operation leaves the state of the system unchanged. The UML attribute isQuery will always return true for any GELLO expression.
The following section describes how GELLO complies with the above requirements and provides the mechanisms for handling clinical data stored in any OO compatible data model.

3  GELLO: Goals and Properties

We propose GELLO as a platform-independent standard expression language for sharing and manipulating knowledge in a medical context. Specifically:

- GELLO is targeted to clinical applications that need an expression language for decision support.
- GELLO is vendor-independent by relying on a language specification that is not vendor-specific.
- GELLO is platform-independent.
- GELLO provides the mechanisms to access data through an OO data model, with strongly-typed expressions, via general purpose expression language.
- GELLO is a declarative language; expressions have no side effects.
- GELLO is extensible by adding new user-defined classes to the underlying OO data model.
- GELLO can refer to any data manipulation methods explicitly defined in the OO data model. The purpose of GELLO is to provide a robust syntax for expressions so data can be easily handled.
- GELLO expression evaluation is compatible with the v.3 RIM; hence each decision rule or guideline need not provide a separate mechanism for translation of data elements to/from host environments.
- The object-oriented approach allows modularity, encapsulation and extensibility.

Thus use of GELLO:

1. provides platform-independent support for mapping to an OO data model, compatible with the v.3 HL7 RIM therefore eliminating the need for implementation-specific encoding methods for information retrieval as part of knowledge content (guidelines, alerts, etc.).
2. simplifies the creation and updating of clinical data objects, and their evaluation.
3. facilitates sharing of decision logic and other computer expressions.

GELLO and its relation to GLIF, RIM and other DSs and KBs. GELLO expression language can be embedded into various applications to provide the mechanisms for access and manipulation of OO data. The RMIM can be any view derived from the HL7 reference information model (RIM).

4  OCL

OCL is the expression language used for specifying invariant constraints and pre- and –post-conditions in object models in the Unified Modeling Language (UML). OCL is a strongly-typed pure expression language without any side effects.
Although the evaluation of an OCL expression returns a value, the state of the underlying data model cannot change because of the evaluation of such an expression. OCL is the result of a consensus effort towards a standard in object-oriented modeling and design. Since OCL is not a programming language, it does not rely on a specific platform. All implementation issues are out of the scope of the language. The OCL expression language satisfies the requirements we have outlined above for GELLO, namely:

- vendor-independent
- platform-independent
- object-oriented
- easy to read/write
- side-effect free
- flexible
- extensible.

In addition, OCL is:

- concise
- declarative.

The latest version of OCL documentation can be found at the Object Management Group website.

4.1 Why OCL?

Besides OCL, we considered XQL, OWL and Java as options for defining a query and expression language. XQL is a query language designed specifically for XML documents. XML documents are unordered, labeled trees, with nodes representing the document entity, elements, attributes, processing instructions and comments. The implied data model behind XML neither matches that of a relational data model nor that of an object-oriented data model. XQL is a query language for XML in the same sense as SQL is a query language for relational tables. Since the HL7 RIM information model and the vMR data model are both object-oriented, it is clear that XQL is not an appropriate approach for an object-oriented query and expression language.

The Web Ontology Language (OWL) is an ongoing effort by W3C to specify a language for publishing and sharing ontologies on the Web. OWL is intended to provide a language that can be used to describe entities and inherent relations between them in Web documents and applications. OWL can be used in applications that need to understand the content. In other words, OWL is more focused on machine interpretable definitions of concepts, rather than on the evaluation of expressions that include arithmetic and user-defined operators. Given that OWL has focused on Web applications and the semantics of Web documents, we did not consider OWL a suitable approach for a language focused on query and computation of complex clinical information. (A list of current W3C recommendations and technical documents can be found at the W3C website).

We also considered Java as an option for an expression language. Java meets most of the requirements in §72. Java is platform-independent, and object-oriented; it is relatively easy to read and write, and is flexible and extensible. However, Java is a full-fledged programming language with side-effects, and it is controlled by a single vendor.

As noted above, OCL has arisen as a consensus effort at creating a standard approach to object-oriented modeling and design. OCL meets all our requirements and, importantly, OCL is already been used by various TCs and SIGs within HL7.

A complete description of the OCL language can be found at the OMG website.

Features in GELLO differentiating it from OCL 2.0 are listed here:

- extended operators for standard types
- extended boolean value "unknown"
- specialized syntax for join() and sort() operators
- extended collection operators
- implicit factory support to generate instances of complex types for use within queries
- A GELLO expression implicitly defines a query operation, whilst OCL 2.0 defines queries as additional operations within a given context

In summary, GELLO removes some OCL unneeded capabilities as a language to be used as an expression language, plus the inclusion of particular UML data types and the HL7 types to enable it to reference instances of patient-specific data rather than just model types.

5 GELLO

GELLO was conceived as a pure, declarative, strongly-typed expression language. GELLO is free of side effects; it provides...
the mechanisms to access medical data through an OO data model compatible with the v.3 RIM. Several features of OCL have been incorporated into GELLO to make it a robust and flexible platform-independent language.

GELLO follows the same conventions as OCL in relationship with the UML metamodel. In other words, every GELLO expression is written in the context of a specific type. Although GELLO is a subset of OCL, not all GELLO features are OCL features. We have preserved consistency as much as possible. The majority of GELLO operators are part of OCL, with the exception of some collection operators (firstN §?5.10.18, lastN §?5.10.19, and join §?5.10.27) which have been added to support the requested functionality. As for data types, all GELLO built-in data types are part of OCL including the Tuple data type §?5.1.4 (the abstract specification for OCL data types can be found at the OMG website). In this section, the use of HL7 classes as a data model is introduced. All basic data types are provided by GELLO (§?5.1), while model types –or classes- are defined in the underlying HL7 data model (§?5.1.2). Variables (§?5.4) are strongly typed named GELLO expressions that can be used at any place a GELLO expression ca be used (§?5.5.2). In order to preserve GELLO as a side-effect free language, the creation of objects as instances of HL7 classes is delegated to the HL7 RIM. This will allow any UML based framework with a full reference implementation of RIM and data types to perform these functions as required.

In summary,

GELLO incorporates most of OCL functionality with the exception of some unneeded capabilities have been removed a)pre and post conditions for constraints -not used in GELLO and b) invariants -also for constraints. The UML ITS implements the semantics of the HL7 Abstract Data Types in such a way that HL7 data types are mapped into the core UML and OCL kernel data types where such mappings are appropriate. Such functionality compatible with the HL7 RIM is used by GELLO, hence enabling it to reference instances of patient-specific data independent of platform, vendor, and data model.

5.1 GELLO Types

GELLO is a strongly-typed language. This means that every expression must have a known type. There are two type categories: predefined types and model (user-defined) types.

Predefined types include basic types, collection types, tuple type and enumeration type.

5.1.1 Basic Types

A primitive data type is named by its reserved word. They are: Boolean, integer, real, string.

Having basic data types allows us to create literals of the form:

- let aNum : integer = 5
- let xNum: real = 5.6
- let aString: string = 'abc' (note single quotes)
- let aBoolean: Boolean = true

5.1.1.1 Boolean

The Boolean type in GELLO is a three-valued type. A GELLO Boolean type can have one of three possible values: true, false, and unknown. GELLO Boolean operators are described in §?5.9.22.

5.1.1.2 Integer

Integer represents the mathematical natural numbers. Integer is a subtype of real. Arithmetic operators are described in §?5.9.

5.1.1.3 Real

Real represents the mathematical concept of real values. Arithmetic operators are described in §?5.9.

5.1.1.4 String

Strings are sequences of characters. Literal strings are enclosed with single quotes. The available operations for strings are described in §?5.9.23 and §?5.9.24.

5.1.1.5 Unknown and Null as third Boolean value

In the previous versions of this specification there was a discrepancy between GELLO unknown and HL7 RIM Null as third
Boolean value. This discrepancy has been resolved by the UML ITS data types, which map GELLO unknown into HL7 RIM Null and vice versa.

5.1.1.6 Basic Type Hierarchy and Type Conformance Rules

Integer is a subtype of real. Hence, integer conforms to real.

5.1.2 Model Types

Model types refer to user-defined classes in the underlying data model. References to such types include either a full description name, e.g. PhysicalQuantity or its alias, e.g. PQ. Both examples refer to the model type PhysicalQuantity as defined in the HL7 RIM.

5.1.2.1 Model Type Hierarchy

For any given class in the data model, if class B is a subtype of class A, then class B conforms to class A.

5.1.3 Collection Types

A GELLO collection is an abstract type with concrete collection types as subtypes. As in OCL, a GELLO collection has three types: Set, Bag and Sequence.

- A Set is the mathematical set. It does not contain any duplicate elements. All elements have the same type.
- A Bag is a collection of elements. Duplicates are allowed. All elements in a Bag have the same type.
- A Sequence is a collection with ordered elements. Duplicates are allowed. All elements in a Sequence must have the same type.

Notation for collections is as follows:

\[\text{typeOfCollection \{element1, ..., elementn\}},\]

Where

- typeOfCollection is one of Set, Bag, Sequence; and \{element1, ..., elementn\} is a list of the elements –all with the same type- separated by commas.

When creating a sequence of integers, the list of elements can be replaced by an interval specification consisting of two literals of type integer intLiteral1 and intLiteral2 separated by `..': Sequence\{1..5\} is equivalent to Sequence\{1,2,3,4,5\}.

Collections can be specified by a literal –as defined above- or as a result of an operation over a collection. See §5.10 for collection operators.

5.1.3.1 Collections of Collections

Collections of collections are not flattened automatically. Flattening is carried out by means of an explicit function making the effect of the flattening process clear. Flattening in OCL applies to all collection types. The indicated rules for flattening can be applied recursively until the element type of the result is a non-collection type.

5.1.3.2 Collection Type Hierarchy and Type Conformance Rules

Set, Bag and Sequence are all subtypes of Collection.

5.1.4 Tuple Type

The tuple type is part of OCL and hence of the GELLO grammar. A tuple combines elements with different types into an aggregate type. Each tuple part has a name, a type and a value. A tuple part can be a single element or a collection. The type of a tuple part can be of a basic or a model type.

The syntax of a tuple is as follows: Tuple\{ label1: type1= value1, ..., labeln: typen = valuen\}, Where labeli is the label of the element ith, typei and valuei are the valid type and value respectively.

Tuples can be used as a return type from expressions that retrieve information from more than one source such as joins (§5.10.27). GELLO provides some tuple operators to access the information returned by an expression. These operators are described in §5.11.
5.1.5 Enumeration Types

Enumeration are datatypes in UML. They define a number of literals as possible values of the enumeration.

5.2 Names

Names may be used in expressions to bind domain elements to values. If the name of an attribute or an operation appears in an expression, the class it belongs to must be explicitly referred to.

5.3 Properties

A property of a class can be an attribute (§5.3.1) or an operation (§5.3.2). The syntax for referring to a property of an instance of class is: InstanceOfClass.property, which is consistent with the ‘dot’ notation of an OO data model.

5.3.1 Attributes

Attributes are named properties of class that describe the characteristics that instances of such class can have. For example, aPerson is an instance of the class Person with an attribute FirstName, we refer to a person’s name by writing aPerson.FirstName, where aPerson is an instance of the Person class and FirstName is an attribute of that class.

5.3.2 Operations

An operation is the specification of a service that can be requested from an object of the class. The name of an operation may appear in a GELLO expression only as a part of a full method invocation expression. That is, along with the name of the operation and its arguments, the class or object it belongs to must be explicitly referred to. Operations are side effect-free, they do not change the state of any object.

The HL7 data model provides classes and associated operations. These operations provide the functionality for handling the clinical data stored in the data model.

5.3.2.1 Operation Parameters

Parameters are name argument values passed to an operation or method in a method call. If required, arguments must be included in the invocation separated by commas. The type of each value must match the type of the expected argument in each position in the argument list.

5.4 Context and References to Contextual Instances

5.4.1 Context, Self and Implicit References to Contextual Instances

Evaluation of expressions requires a contextual instance of a specific type. The context of an expression is explicitly defined using the context expression. The syntax is as follows: context [alias:] Class. Where class indicates the ‘primary’ class in the data model from which the following expression will be evaluated. Context Statements cannot be nested. The notation is as follows:

Example 1.

Context [alias:] Class
GELLO expressions with a reference to either alias or class name

The ‘primary’ class is the class that defines the context where the given expressions are to be evaluated. For example (using the data model depicted in Annex D):

Example 2.

Context Patient
... GELLO expressions with Patient

Sets up the context as all the instances of the ‘primary’ class Patient in the data model.

The following context definition:
Example 3.

Context p: Patients
Let PatientsOverFifty: Sequence(Patient) = p ? select(p.age> 50)

sets the context over which later expressions will be evaluated against as all patients older than 50 years of age.

Example 4.

Context Patient
Let Patient: Patient = Self ? (ID = "123abc")

sets the context of a single patient with a patient ID = "123abc" over which later expressions will be evaluated against.

Note the use of self as an implicit reference to Patient.

5.4.2 Package Context and Pathname

Package context can be used to group classes, while Pathnames can be used to refer to the classes contained in them. The notation is as follows:

Example 5.

Package packageName ::
  Context typeName :
  GELLOexpressions
EndPackage

For PathName:

Example 6.

packageName :: [packageNamei ::]* TypeName

5.4.3 Navigation through Associations

Access to objects and their properties can be performed by navigating through associations of objects. The notation is as follows:

Example 7.

context Object
  self.associatedObject

For example:

Example 8.

context Patient
  self.Medications

returns all the medications a patient is taking, while:

Example 9.

context Patient
  self.Medications["aspirin"]

returns all the instances of medication = aspirin for a given patient.

5.5 Variable Declaration
A variable declaration declares a variable name and binds it to a type. A variable can only hold a value of the same type. A declaration defines the name for a variable inside a program. Each variable declaration must have a scope (§5.5.2).

Variables are declared using the let OCL expression. The type of the variable can be either one of the basic built-in GELLO data types §5.1.1, GELLO collection types §5.1.3, GELLO tuple type §5.1.4, or a class from the underlying data model §5.1.2. The type of the return value of an expression must match the type of the variable to which such a value is to be assigned. Once an expression is bound to a variable, it cannot be changed. The syntax for let expressions is as follows:

```
let varName: type = Expression.
```

Where varName is a string that is the name of the variable with type type, and Expression is a valid GELLO expression.

An example of a variable of basic type:

```
Example 10.
let threshold_for_osteodystrophy : integer = 70                                   ...(1)
```

If a variable references an instance of a model type, the Expression must include the Factory static method as in: Factory.class("argument list"), where class is a class in model class. Factory takes only one string literal as argument list. As a result, it binds a name varName to an object of type class. The full notation:

```
Example 12.
let variableName: type = Factory.class("argument list")
An example:  let potassium: PhysicalQuantity = Factory.PhysicalQuantity("76 kg")   ...(2)
```

An example when a variable references a collection type:

```
Example 13.
let Myobservations: set = observation ?select(coded_concept= 'abc')             ...(3)
```

An example when the variable is a tuple:

```
Example 14.
let variousPatientData : tuple = patient ? join(paramList1; paramList2; CondExp; ParamList3)    ...(4)
```

In (1) threshold_for_osteodystrophy is a GELLO built-in basic type and hence we create the variable and assign the value of 70 in the same operation.

In (2), potassium has a type PhysicalQuantity which is a class in the data model, and we need to create a reference to an instance of that class before assigning it to the variable.

In (3) Myobservations is a set, which is a GELLO collection type. Observations contains all the instances of the class observation with a coded concept = 'abc'.

In (4) variousPatientData is a tuple containing information related to the current patient. Such information is from different sources and has different types, hence it is grouped into an aggregated type tuple. See §5.10.27 for the full notation of the join operator.

In summary, to create a variable with a basic data type, the syntax is:

```
Example 15.
let variable : type = expression
```
5.5.1 Declaring References to Instances of Classes

Declaring a new reference to class is an operation implemented from an external factory. Although the functionality of this approach is beyond the scope of this document, below we describe the syntax of the operation. This approach satisfies both the HL7 RIM abstract specification and the UML ITS specification, and is similar to that approved by the Java-SIG.

A static factory method "Factory" takes only one string literal as argument list. As a result it binds a name (variableName) to an object of type class, where class is a model class. The notation is as follows:

Example 16.

```
let  variableName: type = Factory.class("argument list")
```

An example:

Example 17.

```
let aQuantity: PhysicalQuantity = Factory.PhysicalQuantity("76 kg")
```

Where "76 kg" is the literal form of an instance of PhysicalQuantity, with 76 being the value and "kg" being the code for kilogram in the Unified Code for Units of Measure (UCUM).

5.5.2 Scope of Declarations

The scope of a declaration is the portion of a program where the declared entity is valid and can be referred to. The scoping rules are consistent with those of OCL.

5.6 <<definition>> Constraint

<<definition>> constraints must be attached to a classifier and may only contain let definitions. All variables and operations defined in the <<definition>> constraint are known in the same context as if they were properties of the classifier, and can be used in the same manner. In a way, these <<definition>> constraints become pseudo-attributes and -operations of the classifier, and can be used in a GELLO expression in the same way attributes of a classifier are used. The notation for <<definition>> is as follows. Note the use of 'def' keyword:

Example 18.

```
context Patient def:
  let varName: type = Expression.
```

5.7 Reflection

Within OCL, there are properties that apply to all objects in the underlying data model. Reflection properties can be used to determine:

- The direct type of an object/variable
- The supertype of an object

The following operations from OCL have been incorporated into GELLO:

- object.oclTypeOf(t: OclType): Boolean
  The evaluation of oclTypeOf returns true if the direct type of the object and t are the same: aPerson.oclTypeOf(t: Person) returns true if Person is the direct type of aPerson.
- object.oclKindOf(t: OclType): Boolean
  Similarly, the evaluation of oclKindOf returns true if the t is either the direct type or one of the supertypes of an object: aPatient.oclKindOf(t: Person) returns true if Person is either a direct type or a supertype of aPatient.

5.8 Casting
Every expression written in GELLO must have a type that is matched to a built-in or a model type. It is possible however, to change the type of an expression into another type depending on the context in which such an expression occurs. A specific conversion from type A to type B allows an expression of type A to be treated as type B. This is called casting and OCL provides an operation that has been incorporated into GELLO: oclAsType.

5.9 Built-in Operators

Infix operators are allowed in GELLO. Operator precedence is as defined in §5.13 and the OCL specification (OCL). The following arithmetic operators are supported by GELLO. Other infix operators are allowed if and only if the used model type defines those operators with the correct signature.

5.9.1 Arithmetic Operators "+", ",-", "**"

This section presents the evaluation function for "+" and the allowed types. The types for ",-", and "**" are the same as those for "+", and so is the evaluation function. The evaluation function for "+" is defined as follows:

Example 19.

<table>
<thead>
<tr>
<th>F+(V1,V2)</th>
<th>V1 + V2 If V1 and V2 are both integers or reals</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>= undefined otherwise</td>
</tr>
</tbody>
</table>

Types for "+":
- (integer x integer) ? integer
- (integer x real) ? real
- (real x integer) ? real
- (real x real) ? real

5.9.2 Arithmetic Operator "/"

The result of a division is always a real number even if the arguments are integers.

Example 20.

<table>
<thead>
<tr>
<th>F/ (V1,V2)</th>
<th>V1 / V2 If V1 and V2 are either integers or reals, and V2 &lt;&gt; 0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>= undefined otherwise</td>
</tr>
</tbody>
</table>

Types for "/":
- (integer x integer) ? real
- (integer x real) ? real
- (real x integer) ? real
- (real x real) ? real

5.9.3 Arithmetic Operators "div" and "mod"

The result of integer division "div", and modulus "mod" operations is always an integer number. The arguments must both be integers. The following is the evaluation function and allowed types for integer division. The same applies for modulus.

Example 21.

<table>
<thead>
<tr>
<th>Fdiv(V1,V2)</th>
<th>V1 div V2 If V1 and V2 are both integers and V2 &lt;&gt; 0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>= undefined otherwise</td>
</tr>
</tbody>
</table>

Type "div" and "mod":
- (integer x integer) ? integer

5.9.4 Arithmetic Operator unary minus "-"

The following are the evaluation function and allowed types for unary minus.

Example 22.

<table>
<thead>
<tr>
<th>F-(V1)</th>
<th>-V1 If V1 is either an integer or a real</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>= undefined otherwise</td>
</tr>
</tbody>
</table>

Types for unary minus "-":
- (integer) ? integer
- (real) ? real


The allowed types and evaluation rule for the operators ",=", ",>", ",<", ",>=", ",<=", ",<>" are as follows. All these operators return undefined if one or both of the comparands is undefined.
Example 23.

Types for `=`, `>`, `<`, `>=`, `<=`, `<>`:
- `(real x real)? truth_value`
- `(real x integer)? truth_value`
- `(integer x real)? truth_value`
- `(integer x integer)? truth_value`
- `(string x string)? truth_value` Only valid for `=``<>`
- `(boolean x boolean)? truth_value` Only valid for `=``<>`

Definition of the evaluation function \(F>\) (\(V1, V2\)). The evaluation function is the same for the other operators:

\[
F>(V1,V2) = \begin{cases} 
    \text{true} & \text{If } V1 \text{ and } V2 \text{ are both either integers or reals and } V1 > V2 \\ 
    \text{false} & \text{Else if } V1 \text{ and } V2 \text{ are both either integers or reals and } V1 = V2 \\ 
    \text{undefined} & \text{otherwise} 
\end{cases}
\]

Note: for the cases `(real x integer)` and `(integer x real)` there is an implicit casting of integer to real, hence both cases are evaluated as `(real x real)` after casting.

Definition of the evaluation function \(F=\) (\(V1, V2\)). The evaluation function is the same for \(F<>\) (\(V1, V2\)) when the operators are both strings of Booleans:

Example 25.

\[
F=(V1,V2) = \begin{cases} 
    \text{true} & \text{If } V1 \text{ and } V2 \text{ are both strings or Booleans and } V1 = V2 \\ 
    \text{false} & \text{Else if } V1 \text{ and } V2 \text{ are both strings or Booleans and } V1 <> V2 \\ 
    \text{undefined} & \text{otherwise} 
\end{cases}
\]

5.9.6 Mathematical Operator "abs"

The mathematical operator "abs" returns the absolute value of a number.

Example 26.

\[
\text{Fabs}(V1) = \begin{cases} 
    V1 & \text{If } V1 \text{ is a positive integer or positive real number} \\ 
    -V1 & \text{Else if } V1 \text{ is a negative integer or negative real number} \\ 
    \text{undefined} & \text{otherwise} 
\end{cases}
\]

Type "abs"
- `(integer)` ? integer
- `(real)` ? real

5.9.7 Mathematical Operator "acos"

The mathematical operator "acos" returns the arc cosine of an angle in the range of 0.0 through \(\pi\). If the argument is not a number, is undefined or its absolute value is ">" 1, the the returning value is undefined

Example 27.

\[
\text{Facos}(V1) = \begin{cases} 
    \text{acos}(V1) & \text{If } V1 \text{ is an integer or real number and } \text{abs}(V1) = 1 \\ 
    \text{undefined} & \text{otherwise} 
\end{cases}
\]

Type "acos"
- `(integer)` ? real
- `(real)` ? real

5.9.8 Mathematical Operator "asin"

The mathematical operator "asin" returns the arc sine of an angle, in the range of \(-\pi/2\) through \(\pi/2\).

Example 28.

\[
\text{Fasin}(V1) = \begin{cases} 
    \text{asin}(V1) & \text{If } V1 \text{ is an integer or real number and } \text{abs}(V1) = 1 \\ 
    0 & \text{Else if } V1 = 0 \\ 
    \text{undefined} & \text{otherwise} 
\end{cases}
\]

Type "asin"
- `(integer)` ? real
- `(real)` ? real

5.9.9 Mathematical Operator "atan"

The mathematical operator "atan" returns the arc tangent of an angle, in the range of \(-\pi/2\) through \(\pi/2\).
Example 29.
\[
\text{Fatan}(V1) = \begin{cases} 
\text{atan}(V1) & \text{if } V1 \text{ is an integer or real number and } |V1| = 1 \\
0 & \text{if } V1 = 0 \\
\text{undefined} & \text{otherwise}
\end{cases}
\]
Type "atan"
- (integer) -> real
- (real) -> real

5.9.10 Mathematical Operator "ceiling"

The mathematical operator "ceiling" returns the smallest integer value that is not less than the argument.

Example 30.
\[
\text{Fceiling}(V1) = \begin{cases} 
\text{ceiling}(V1) & \text{if } V1 \text{ is a real} \\
V1 & \text{else if } V1 \text{ an integer} \\
0 & \text{else if } V1 < 0 \text{ and } V1 > -1.0 \\
\text{undefined} & \text{otherwise}
\end{cases}
\]
Type "ceiling"
- (integer) -> integer
- (real) -> integer

5.9.11 Mathematical Operator "cos"

The mathematical operator "cos" returns the trigonometric cosine of an angle.

Example 31.
\[
\text{Fcos}(V1) = \begin{cases} 
\cos(V1) & \text{if } V1 \text{ is an integer or real number and } V1 \text{ is the value of an angle in radians} \\
\text{undefined} & \text{otherwise}
\end{cases}
\]
Type "cos"
- (integer) -> real
- (real) -> real

5.9.12 Mathematical Operator "exp"

The mathematical operator "exp" returns the value e^a, where e is the base of the natural logarithms.

Example 32.
\[
\text{Fexp}(V1) = \begin{cases} 
e^{V1} & \text{if } V1 \text{ is an integer or real number} \\
+\infty & \text{if } V1 \text{ is positive infinity} \\
0 & \text{if } V1 \text{ is negative infinity} \\
\text{undefined} & \text{otherwise}
\end{cases}
\]
Type "exp"
- (integer) -> real
- (real) -> real

5.9.13 Mathematical Operator "floor"

The mathematical operator "floor" returns the largest integer value that is not greater than the argument.

Example 33.
\[
\text{Ffloor}(V1) = \begin{cases} 
\text{floor}(V1) & \text{if } V1 \text{ is a real number} \\
V1 & \text{else if } V1 \text{ an integer number} \\
V1 & \text{else if } V1 = 0 \text{ or } V1 = \infty \\
\text{undefined} & \text{otherwise}
\end{cases}
\]
Type "floor"
- (integer) -> integer
- (real) -> integer

5.9.14 Mathematical Operator "log"

The mathematical operator "log" returns the natural logarithm (base e) of a number.

Example 34.
\[
\text{Flog}(V1) = \begin{cases} 
\log(V1) & \text{if } V1 \text{ is an integer or real number} \\
+\infty & \text{if } V1 \text{ is positive infinity} \\
-\infty & \text{if } V1 = 0 \\
\text{undefined} & \text{otherwise}
\end{cases}
\]
Type "log"
- (integer) -> real
- (real) -> real

5.9.15 Mathematical Operator "max"

Example 35.
The mathematical operator "max" returns the greater of two numbers.

Example 35.

\[
\text{Fmax}(V1,V2) = \begin{cases} 
  V1 & \text{if } V1 > V2 \\
  V2 & \text{if } V1 = V2 \\
  \text{undefined} & \text{otherwise}
\end{cases}
\]

Type "max"
- (integer x integer) ? integer
- (integer x real) ? real
- (real x integer) ? real
- (real x real) ? real

5.9.16 Mathematical Operator "min"

The mathematical operator "min" returns the smaller of two numbers.

Example 36.

\[
\text{Fmin}(V1,V2) = \begin{cases} 
  V1 & \text{if } V1 < V2 \\
  V2 & \text{if } V1 = V2 \\
  \text{undefined} & \text{otherwise}
\end{cases}
\]

Type "min"
- (integer x integer) ? integer
- (integer x real) ? real
- (real x integer) ? real
- (real x real) ? real

5.9.17 Mathematical Operator "power"

The mathematical operator "power" returns the value of the first argument raised to the power of the value of the second argument.

Example 37.

\[
\text{Fpower}(V1,V2) = \begin{cases} 
  V1^V2 & \text{if } V1 \text{ and } V2 \text{ are valid arguments} \\
  1 & \text{if } V2 = 0 \\
  V1 & \text{if } V2 = 1 \\
  \text{abs}(V1)^V2 & \text{else if } V1 < 0 \text{ and } V2 \text{ is a finite even integer} \\
  \text{undefined} & \text{else if } V1 < 0 \text{ and } V2 \text{ finite and not an integer} \\
  \text{undefined} & \text{else if } V2 \text{ is undefined} \\
  \text{+ infinity} & \text{else if } \text{abs}(V1) > 1 \text{ and } V2 = +\text{infinity} \\
  \text{+ infinity} & \text{else if } V1 = 0 \text{, } V2 < 0 \text{ or } V1 = +\text{infinity} \text{ and } V2 > 0 \\
  \text{+ infinity} & \text{else if } V1 = -0 \text{ and } V2 < 0 \text{ and } V2 = \text{negative finite odd integer or } V1 = -\text{infinity} \text{ and } V2 > 0 \text{ and } V2 = \text{finite odd integer} \\
  0 & \text{else if } V1 < 0 \text{ and } V2 > 0 \text{ is a positive finite odd integer or } V1 = -\text{infinity} \text{ and } V2 \text{ is a negative finite odd integer} \\
  0 & \text{else if } \text{abs}(V1) < 1 \text{ and } V2 = +\text{infinity} \\
  0 & \text{else if } V1 = 0 \text{ and } V2 > 0 \text{ or } V1 = \text{infinity} \text{ and } V2 < 0 \\
  \text{+ infinity} & \text{else if } V1 = -0 \text{ and } V2 = 0 \text{ and } V2 = \text{finite odd number or } V1 = -\text{infinity} \text{ and } V2 < 0 \text{ and } V2 = \text{finite odd integer} \\
  0 & \text{else if } V1 < 0 \text{ and } V2 = \text{positive odd number or } V1 = -\text{infinity} \text{ and } V2 \text{ is a negative odd integer} \\
  V2 & \text{else if } V2 < V1 \\
  \text{undefined} & \text{otherwise}
\end{cases}
\]

Type "power"
- (integer x integer) ? integer
- (integer x real) ? real
- (real x integer) ? real
- (real x real) ? real

5.9.18 Mathematical Operator "rand"

The mathematical operator "rand" returns a positive real number in the range [0.0, 1.0).

Example 38.

\[
\text{Frand}() = V1 \text{ where } 0.0 \leq V1 < 1.0
\]

Type "rand"
- () ? real

5.9.19 Mathematical Operator "sin"

The mathematical operator "sin" returns the trigonometric sine of an angle.

Example 39.

\[
\text{Fsin}(V1) = \sin(V1) \text{ if } V1 \text{ is an integer or real number representing the value of an angle in radians}
\]

= 0 \text{ else if } V1 = 0

= \text{undefined} \text{ otherwise}

= \text{+ infinity} \text{ else if } \text{abs}(V1) > 1 \text{ and } V2 = +\text{infinity} \\
= \text{+ infinity} \text{ else if } V1 = 0 \text{ and } V2 > 0 \text{ or } V1 = +\text{infinity} \text{ and } V2 < 0 \\
= \text{+ infinity} \text{ else if } V1 = -0 \text{ and } V2 = 0 \text{ and } V2 = \text{finite odd number or } V1 = -\text{infinity} \text{ and } V2 < 0 \text{ and } V2 = \text{finite odd integer} \\
= 0 \text{ else if } V1 < 0 \text{ and } V2 = \text{positive odd number or } V1 = -\text{infinity} \text{ and } V2 \text{ is a negative odd integer} \\
= V2 \text{ else if } V2 < V1 \\
= \text{undefined} \text{ otherwise
Type "sin"
• (integer) ? real
• (real) ? real

5.9.20 Mathematical Operator "sqrt"

The mathematical operator "sqrt" returns the square root of a number.

Example 40.

Fsqrt(V1) = +sqrt(V1) If V1 is an integer or real number and V1 ≥ 0
  = 0 Else if V1 = 0
  = undefined Else if V1 < 0
  = undefined otherwise

Type "sqrt"
• (integer) ? real
• (integer) ? integer
• (real) ? real

5.9.21 Mathematical Operator "tan"

The mathematical operator "tan" returns the trigonometric tangent of an angle.

Example 41.

Ftan(V1) = tan(V1) If V1 is an integer or real number representing the value of an angle in radians
  = 0 Else if V1 = 0
  = undefined otherwise

Type "tan"
• (integer) ? real
• (real) ? real

5.9.22 Boolean Operators

The valid types and evaluation functions for Boolean operators "and", "or", "xor" and "not" are given as follows:

Example 42.

Types for "and", "or", "xor" and "implies":
(truth_value x truth_value) ? truth_value
Types for "not":
(truth_value) ? truth_value

Values of the evaluation functions (as in OCL p5.12):

<table>
<thead>
<tr>
<th>V1</th>
<th>V2</th>
<th>V1 and V2</th>
<th>V1 or V2</th>
<th>V1 xor V2</th>
<th>not V1</th>
<th>V1 implies V2</th>
</tr>
</thead>
<tbody>
<tr>
<td>false</td>
<td>false</td>
<td>false</td>
<td>false</td>
<td>false</td>
<td>true</td>
<td>true</td>
</tr>
<tr>
<td>false</td>
<td>true</td>
<td>false</td>
<td>true</td>
<td>true</td>
<td>true</td>
<td>true</td>
</tr>
<tr>
<td>true</td>
<td>false</td>
<td>false</td>
<td>true</td>
<td>true</td>
<td>false</td>
<td>false</td>
</tr>
<tr>
<td>true</td>
<td>true</td>
<td>true</td>
<td>true</td>
<td>false</td>
<td>false</td>
<td>true</td>
</tr>
<tr>
<td>false</td>
<td>unknown</td>
<td>false</td>
<td>unknown</td>
<td>unknown</td>
<td>true</td>
<td>true</td>
</tr>
<tr>
<td>true</td>
<td>unknown</td>
<td>unknown</td>
<td>true</td>
<td>unknown</td>
<td>false</td>
<td>unknown</td>
</tr>
<tr>
<td>unknown</td>
<td>false</td>
<td>false</td>
<td>unknown</td>
<td>unknown</td>
<td>unknown</td>
<td>unknown</td>
</tr>
<tr>
<td>unknown</td>
<td>true</td>
<td>unknown</td>
<td>true</td>
<td>unknown</td>
<td>unknown</td>
<td>true</td>
</tr>
<tr>
<td>unknown</td>
<td>unknown</td>
<td>unknown</td>
<td>unknown</td>
<td>unknown</td>
<td>unknown</td>
<td>unknown</td>
</tr>
</tbody>
</table>

A note on truth values and Boolean operators: GELLO is intended to support extended truth values: true, false and unknown, while HL7 Boolean values in the HL7 RIM data model are true, false and NULL. The discrepancy between unknown and NULL is resolved by the UML ITS.

5.9.23 String Operators "size", "concat", "toUpper", "toLower" "substring", "=" and "<>"

The types and evaluation functions for string operators are given as follows:
**Example 43.**

Type for "size"
• (string) ? integer

Definition of evaluation function for Fsize(V)
Fsize(V) = integer
= undefined otherwise

Type for "concat"
• (string x string) ? string

Definition of evaluation function for Fconcat(V1, V2)
Fconcat(V1, V2) = string
= undefined otherwise

Type for "toUpper", "toLower"
• (string) ? string

Definition of evaluation function for FtoUpper(V). The same applied for "toLower" operator.
FtoUpper(V) = string all in upper characters according to upper case mapping as defined by the Unicode Standard
= undefined otherwise

Type for "substring"
• (string x integer x integer) ? string

Definition of evaluation function for Fsubstring(V1,V2,V3)
Fsubstring(V1,V2,V3) = returns a substring of length V3, at a given starting point V2.
= undefined otherwise

Type for "=", "<>"
• (string x string) ? truth_value

Definition of the evaluation function F= (V1, V2). The evaluation function is the same for F<> (V1, V2) when the operands are both strings.
F=(V1,V2) = true
= false
= undefined otherwise

5.9.24 String Operators "tochar", "lpad", "rpad", "ltrim", "rtrim" and "replace"

The types and evaluation functions for string operators are given as follows:

**Example 44.**

Type for "tochar"
• (integer) ? string
• (real) ? string

Definition of evaluation function for Ftochar(V)
Ftochar(V) = string
= undefined otherwise

Type for "lpad"
• (string x integer x string) ? string

Definition of evaluation function for Flpad(V).
Flpad(V1, V2, V3) = string
= undefined otherwise

Type for "rpad"
• (string x integer x string) ? string

Definition of evaluation function for Frpad(V).
Frpad(V1, V2, V3) = string
= undefined otherwise

Type for "ltrim"
• (string x string) ? string

Definition of evaluation function for Fltrim(V1,V2).
Fltrim(V1,V2) = string
= undefined otherwise

Type for "rtrim"
• (string x string) ? string

Definition of evaluation function for Frtrim(V1,V2).
Frtrim(V1,V2) = string
= undefined otherwise

Type for "replace"
• (string x string x string) ? string

Definition of the evaluation function Freplace (V1, V2,V3).
Freplace(V1, V2,V3) = string
= undefined otherwise
5.10 Collection Operators

GELLO incorporates from OCL standard operations to handle elements in collections. These operations take the elements in a collection and evaluate a GELLO expression on each of them. These operators are described in the following sections: select (§?5.10.3), reject (§?5.10.4), collect (§?5.10.5), forAll (§?5.10.6), iterate (§?5.10.7), exists (§?5.10.8) and flatten (§?5.10.9).

5.10.1 The ‘Arrow’ Notation

All operations on collections in GELLO are denoted with the ‘arrow’ ? notation. The ‘arrow’ notation distinguishes a collection operation from a model type operation. Hence the notation is as follows:

Example 45.

```
collection ? collectionOperator(parameters)
```

5.10.2 Single Instances as Collections

GELLO treats a single instance as a collection with only one element. This allows us to apply collection operators to instances. The type definition for collection operators will treat a single instance as a collection with one element, as specified in this section. The notation is the same as in (§?5.10.1):

Example 46.

```
singleInstance ? collectionOperator(parameters)
```

5.10.3 Operator Select

Select is an operator to specify a selection from a specific collection. Select returns all the elements in a collection that satisfy a criterion. There are three different forms, of which the simplest one is:

Example 47.

```
collection ? select( boolean-expression )
```

The result of the select operation, in the context of a complete repository of patient records, is a collection of problem list instances with problem code = "123":

Example 48.

```
patient.problemList ? select(code = "123")
```

This results in a collection that contains all the elements from collection for which the boolean-expression evaluates to true. For each element in collection the expression boolean-expression evaluates over a property of the elements in the collection.

A more general syntax for the select expression:

Example 49.

```
collection ? select( v | boolean-expression-with-v )
```

The variable v, called the iterator, is a reference to an object in the collection. v iterates over the collection and the boolean-expression-with-v is evaluated for each v. The third form is an extension of the latest, where the type of the variable v can be specified. The notation is:

Example 50.

```
collection ? select( v : Type | boolean-expression-with-v)
```

The notation for all variants is:

```
collection(select x BooleanExpression)
collection ? select( v | boolean-expression-with-v)
collection ?select( v : Type | boolean-expression-with-v)
```

Type for "select"

```
(collection x Boolean Expression) ? collection
```

Definition of evaluation function for Fselect(V,E)

```
Fselect(V,E)= collection     If V is a collection  and E is a valid GELLO Boolean expression.
```

The resulting collection contains all the elements from V for which the...
5.10.4 Operator Reject

The Boolean expression evaluates over a property of the elements in the collection, returning all the elements that do not satisfy such condition (all elements that evaluate to False).

The notation is:

```
collection? reject(booleanExpression)
collection? reject( v:Type | boolean-expression-with-v )
collection? reject( v | boolean-expression-with-v )
```

The result of reject in the following example is a collection of patients who do not have problem code = "123" in their problem list:

```
patient.problemList? reject(code = "123")
```

5.10.5 Operator Collect

Collect iterates over a collection, computes a value for each element of the collection, and gathers the evaluated values into a new collection. The type of the elements of the resulting collection is usually different from the type of the elements in the original collection over which the operator is applied. The collect operation uses the same syntax as the select and reject operators.
The notation is:

\[
\text{collection? collect(Expression)}
\]

\[
\text{collection? collect( v | expression-with-v )}
\]

\[
\text{collection? collect( v : Type | expression-with-v )}
\]

Type for "collect"
(set x Expression)? bag
(bag x Expression)? bag
(sequence x Expression)? sequence

Definition of evaluation function for Fcollect(V,E)
Fcollect(V,E) = collection  If V is a collection and E is an expression that is evaluated for each element in to collection.
= undefined  otherwise

Type for "collect"
(set x Iterator x Type x Expression)? bag
(bag x Iterator x Type x Expression)? bag
(sequence x Iterator x Type x Expression)? sequence

Definition of evaluation function for Fcollect(V,I,E)
Fcollect(V,I,E) = collection  If V is a collection, I is the iterator referring to the object from the collection and E is a valid GELLO expression. The resulting collection contains the results of evaluating E for each element of V.
= undefined  otherwise

collection? collect(v:Type | Expression-with-v)

Type for "collect"
(set x Iterator x Type x Expression)? bag
(bag x Iterator x Type x Expression)? bag
(sequence x Iterator x Type x Expression)? sequence

Definition of evaluation function for Fcollect(V,I,T,E)
Fcollect(V,I,E) = collection  If V is a collection, I is the iterator with Type T referring to the object from the collection and E is a valid GELLO Expression. The resulting collection contains the result of evaluating E for each element of V.
= undefined  otherwise

The following example returns a collection lab result values for creatinine:

Example 54.

LabResult? select(code = "CRE")? collect(value)

5.10.6 Operator ForAll

ForAll is used to specify a Boolean expression that evaluates the value of a property over all the elements of a collection. The result of the Boolean expression is true if all the elements of the collection evaluate to true. If the Boolean expression is false for one or more elements of the collection, then the complete expression evaluates to false.

The notation is:

\[
\text{collection? forAll(Boolean Expression)}
\]

\[
\text{collection? forAll( v | boolean-expression-with-v )}
\]

\[
\text{collection? forAll( v : Type | boolean-expression-with-v )}
\]

Type for "forAll"
(collection x Boolean Expression)? truth_value

Definition of evaluation function for FforAll(V,E)
FforAll(V,E) = true  If V is a collection and E is a valid GELLO Boolean expression containing a property of the elements, and this expression evaluates to true for all elements in the collection
= false  If V is a collection and E is a valid GELLO Boolean expression containing a property of the elements, and there is at least one element for which the expression E evaluates to false.
= undefined  otherwise

collection? forAll(v | BooleanExpression-with-v)

Type for "forAll"
(collection x Iterator x Boolean Expression)? truth_value

Definition of evaluation function for FforAll(V,I,E)
FforAll(V,I,E) = true  If V is a collection, I is the iterator referring to the object from the collection and E is a valid GELLO Boolean expression containing a property of the elements, and this expression evaluates to true for all elements in the collection.
= false  If V is a collection and I is the iterator referring to the object from the collection and E is a valid GELLO Boolean expression containing a property of the elements, and there is at least one element for which the expression E evaluates to false.
= undefined  otherwise
The following example returns true if all the lab results are for creatinine:

Example 56.

\[
\text{LabResult ? forAll(code = "CRE")}
\]

5.10.7 Operator Iterate

Iterate is a generic operator. It iterates over a collection of elements elem evaluating each element against a valid GELLO expression-with-elem-and-result. The result of the expression-with-elem-and-result is stored in result. The initial value of result is defined by expression. Once iterate goes through the whole collection, it returns result.

The notation is:

Example 57.

\[
\text{collection ? iterate(elem: Type; result: Type = expression | expression-with-elem-and-result)}
\]

The following example returns the number of times creatinine occurs in the collection LabResult:

Example 58.

\[
\text{LabResult ? iterate(code; acc: Integer = 0 | if code = "CRE" then acc + 1 else acc endif)}
\]

5.10.8 Operator Exists

Exists returns true if there is at least one element in the collection that satisfies the Boolean expression.

The notation is:

Example 59.

\[
\text{collection ? exists(BooleanExpression)}
\]
Definition of evaluation function for Fexists(V, I, E)

\[
F_{\text{exists}}(V, I, E) =
\begin{cases}
  \text{true} & \text{If } V \text{ is a collection, } I \text{ is the iterator referring to the object from the collection and } E \text{ is a valid GELLO Boolean expression containing a property of the elements and there is at least one element in the collection } V \text{ for which } E \text{ evaluates to true.} \\
  \text{false} & \text{If } V \text{ is a collection } I \text{ is the iterator referring to the object from the collection and } E \text{ is a valid GELLO and there is no element in the collection } V \text{ such that } E \text{ evaluates to true for that element} \\
  \text{undefined} & \text{otherwise}
\end{cases}
\]

\[
collection ? \text{existsl}(v: Type | \text{BooleanExpression-with-v})
\]

Type for "exists"

\[
(\text{collection Iterator} \times \text{Type} \times \text{Boolean Expression}) ? \text{truth_value}
\]

The following example returns true if there is at least one lab result for creatinine:

Example 60.

\[
\text{LabResult} ? \text{exists(code = "CRE")}
\]

5.10.9 Operator Flatten

Flatten returns a collection without any nested elements. If the resulting type is a collection, the operator is applied recursively until the return type is a collection without nested collections.

The notation is:

Example 61.

\[
\text{collection} ? \text{flatten()}
\]

Type for "flatten"

\[
(\text{collection}) ? \text{collection}
\]

Definition of evaluation function for Fflatten(V)

\[
F_{\text{flatten}}(V) =
\begin{cases}
  \text{collection} & \text{If } V \text{ is a collection, } \text{the result is a collection containing all the elements of } V. \text{ If } V \text{ is a set, bag, or sequence, then } F_{\text{flatten}}(V) \text{ is a set, bag, or sequence respectively.} \\
  \text{element} & \text{If } V \text{ is not a collection.}
\end{cases}
\]

The result of using flatten in the following example is the collection \{1, 2, 3, 4, 5\}:

Example 62.

\[
\{1, 2, (3), \{(4),(5)\} \} ? \text{flatten()}
\]

5.10.10 Operator Size

The operator size returns the number of elements in a collection.

The notation is:

Example 63.

\[
\text{collection} ? \text{size()}
\]

Types for "size"

\[
(\text{collection}) ? \text{integer}
\]

Definition of evaluation function for Fsize(V)

\[
F_{\text{size}}(V) =
\begin{cases}
  \text{integer} & \text{If } V \text{ is a collection} \\
  \text{undefined} & \text{otherwise}
\end{cases}
\]
The following example returns the number problems the patient has in his problem list:

Example 64.

```
patient.problemList ? size()
```

### 5.10.11 Operator Count

Count returns the number of occurrences of object in a collection.

The notation is:

```
collection ? count(object)
```

**Type for "count"

(collection x object) ? integer

**Definition of evaluation function for \( F_{\text{count}}(V,O) \)

\[ F_{\text{count}}(V,O) = \begin{cases} 
\text{integer} & \text{If } V \text{ is a collection and } O \text{ is a defined object} \\
\text{undefined} & \text{otherwise}
\end{cases} \]

The following example returns the number of times the patient has had fever:

Example 66.

```
patient.problemList ? collect(code) ? count("fever")
```

### 5.10.12 Operators "Max" and "Min"

Max and min return the biggest and smallest value respectively in a collection. The collection must contain numbers. The following also applies for the min operator.

The notation is:

```
collection ? max()
```

**Type for "max"

(collection) ? number

**Definition of evaluation function for \( F_{\text{max}}(V) \)

\[ F_{\text{max}}(V) = \begin{cases} 
\text{number} & \text{If } V \text{ is a collection of numbers (integers or reals)} \\
\text{undefined} & \text{otherwise}
\end{cases} \]

Example 68.

```
{2,5,1} ? max()        returns 5
{2,5,1} ? min()       returns 1
```

### 5.10.13 Operator Includes

Includes operator returns a true if the object is an element of the collection.

The notation is:

```
collection ? includes(object)
```

**Type for "includes"

(collection x object) ? truth_value

**Definition of evaluation function for \( F_{\text{includes}}(V,O) \)

\[ F_{\text{includes}}(V,O) = \begin{cases} 
\text{true} & \text{If } V \text{ is a collection and } O \text{ is an element in the collection.} \\
\text{false} & \text{Else if } V \text{ is a collection and } O \text{ is not an element } V.
\end{cases} \]
The following example returns true if the patient has "fever" in problem list:

Example 70.

patient.problemList ? collect(code) ? includes("fever")

5.10.14 Operator IncludesAll

IncludesAll returns true if all the elements in the parameter collection are in the current collection.

The notation is:

Example 71.

collection ? includesAll(parameterCollection)

Types for "includesAll"
(collection x singleInstance) ? truth_value
(collection x collection ) ? truth_value

Definition of evaluation function for FincludesAll(V,C)
FincludesAll(V,C)= true         If both V and C are collections and all the elements in C
                                 appear in V.
= false         Else if V is a collection and there is at least one element in C
                                 that does not appear in V.
= undefined     otherwise

The following example returns true if the patient has "fever" and "rash" in problem list:

Example 72.

patient.problemList ? collect(code) ? includesAll( Set{"fever", "rash"})

5.10.15 Operator IsEmpty

isEmpty returns true if the collection contains no elements.

The notation is:

Example 73.

collection ? isEmpty()

Type for "isEmpty"
(collection) ? truth_value

Definition of evaluation function for FisEmpty(V)
FisEmpty(V) = true          If V is a collection with no elements
= false         Else if V is a collection with one or more elements
= undefined     otherwise

The following example returns true if the patient has no problems in problem list:

Example 74.

patient.problemList ? isEmpty()

5.10.16 Operator notEmpty

notEmpty returns true if the collection contains one or more elements.

The notation is:
Example 75.

```
collection ? notEmpty()
```

Type for "notEmpty"
(collection) ? truth_value

Definition of evaluation function for FnotEmpty(V)
FnotEmpty(V) = true          If V is a collection with one or more elements
 = false         Else if V is a collection with no elements
 = undefined     otherwise

The following example returns true if the patient has at least one problem in problem list:

Example 76.

```
patient.problemList ? notEmpty()
```

5.10.17 Operator Sum

Sum adds up all the elements in a collection. The elements must be of type integer or real.

The notation is:

Example 77.

```
collection  ? sum()
```

Types for "sum"
(collection_of_integers)  ? integer
(collection_of_reals)  ? real

Definition of evaluation function for Fsum(V)
Fsum(V) = V1+…+Vn           If V is a non-empty collection of n integer or
real values < V1 , … , Vn > with 1= i = n)
 = 0                 Else if V is an empty collection
 = undefined         otherwise

The following example returns 15, the sum of all the values in the collection:

Example 78.

```
{1, 2, 3, 4, 5}  ? sum()
```

5.10.18 Operator FirstN

firstN returns a sequence with the first n elements from the current sequence (a collection with ordered elements). firstN returns the first n elements.

The notation is:

Example 79.

```
sequence ? firstN(numberOfElements)
```

Type for "firstN"
(sequence x integer)  ? sequence

Definition of evaluation function for FfirstN(V,N)
FfirstN(V,N)= sequence        If V is an non-empty sequence and N is an integer
such that 1= N = size of V. The resulting sequence
 = undefined       otherwise

The following example returns the first 3 elements in a sequence:

Example 80.

```
{1,2,3,4,5}  ? firstN(3)      returns {1,2,3}
```

5.10.19 Operator LastN
Returns the last n elements from the current sequence. The elements are returned as a sequence of n elements.

The notation is:

```gello
sequence ? lastN(numberOfElements)
Type for "lastN"
(sequence x integer) ? sequence
Definition of evaluation function for FlastN(V,N)
FlastN(V,N) = sequence If V is an non-empty sequence and N is an integer
such that 1 <= N = size of V. The resulting sequence
= undefined otherwise
```

The following example returns the last 3 elements in a collection:

```gello
{1,2,3,4,5} ? lastN(3) returns {3,4,5}
```

### 5.10.20 Operator ElemAt

Returns the element at the Nth position from the current sequence.

The notation is:

```gello
sequence ? elemAt(ElementPosition)
Type for "elemAt"
(sequence x integer) ? element
Definition of evaluation function for FelemAt(V,N)
FelemAt(V,N) = element If V is an non-empty sequence and N is an integer
such that 1 = N = size of V. The result is the
element at the Nth position.
= undefined otherwise
```

The following example returns the element in the third position in the sequence. The positions go from 1 to size of sequence -1:

```gello
{a, f, g, k, z} ? elemAt(3) returns g
```

### 5.10.21 Operator Reverse

Reverse returns a sequence in reversed order. E.g. the first element of the current sequence is returned as the last and so on.

The notation is:

```gello
sequence ? reverse()
Type for "reverse"
(sequence) ? sequence
Definition of evaluation function for Freverse(V)
Freverse(V) = sequence If V is a single instance or a sequence.
The resulting sequence is of the same
= undefined otherwise
```
The following example returns a sequence in a reversed order:

Example 86.

\{(a, f, g, k, z) \rightarrow \text{reverse}() \rightarrow \{z, k, g, f, a\}\}

5.10.22 Operator SortBy

The notation is:

Example 87.

\(\text{collection} \rightarrow \text{sortBy(orderByExpression)}\)

Types for "sortBy"

- \((\text{collection} \times \text{ListOfProperties}) \rightarrow \text{sequence}\)

Definition of evaluation function for \(\text{sortBy}(V,E)\)

\(\text{sortBy}(V,E) = \text{sequence}\)

If \(V\) is a single instance or a collection and \(E\) is a GELLO expression specifying the properties by which the current collection should be ordered by. The result is a sequence.

= undefined otherwise

The following example returns a collection of medications sorted in ascending order by effective time:

Example 88.

\(\text{Patient.Medication} \rightarrow \text{sortBy(effectiveTime.high)}\)

5.10.23 Operator Intersection

The intersection operator returns a collection with the elements that appear in both the current collection and the parameter collection. This operation is valid for any combination set and bag, but is not defined for sequences. The return type is a set. Set does not allow duplicates.

The notation is:

Example 89.

\(\text{collection} \rightarrow \text{intersection(parameterCollection)}\)

Types for "intersection"

- \((\text{set} \times \text{set}) \rightarrow \text{set}\)
- \((\text{bag} \times \text{bag}) \rightarrow \text{set}\)
- \((\text{set} \times \text{bag}) \rightarrow \text{set}\)
- \((\text{bag} \times \text{set}) \rightarrow \text{set}\)

Definition of evaluation function for \(\text{intersection}(V1,V2)\)

\(\text{intersection}(V1,V2) = \text{set}\)

If \(V1\) and \(V2\) are either sets or bags with elements \(<V1,1, \ldots, V1,n>\) and \(<V2,1, \ldots, V2,n>\) respectively. The resulting set contains all the elements from \(V1\) that also are elements of \(V2\) \((V1,i = V2,j)\).

If there are not common elements in \(V1\) and \(V2\) the intersection returns an empty set.

= undefined otherwise

Example 90.

\(\text{TDrug: a variable holding a collection of drugs, each of which has a "compellingIndication" property. aPatient is a variable referencing a particular patient’s medical record:}\)

\(\text{Drug} \rightarrow \exists(aDrug: \text{SubstanceAdministration} \ | \ \neg(aDrug.compellingIndication \ ? \ \text{intersection}(aPerson.problemList) \ ? \ \text{isEmpty}())\)

5.10.24 Operator Union

The union operation combines two collections into a new one. The union operation can combine a set and a bag, but a sequence only can be combined with another sequence.

The notation is:
Example 91.

```
Example 92.
```

The following example returns a collection times when a patient has taken any medication and has had problems:

Example 93.

```
Example 94.
```

5.10.25 Operator Including

The operator including returns a collection containing all the elements of the current collection plus an element (which is added at the end if the collection is a sequence).

The notation is:

Example 95.

```
Example 96.
```

The following example returns a set with the appended element:

Example 97.

```
Example 98.
```

5.10.26 Operator Excluding

The operator excluding returns a collection containing all the elements of the current collection minus all the occurrences of element.

The notation is:
The following example returns a bag with the deleted element:

**Example 96.**

\[
\text{Bag}(7, 2, 5, 8, 4, 5) \ ? \ \text{excluding}(5) \quad \text{returns} \quad \text{Bag}(7, 2, 8, 4)
\]

### 5.10.27 Operator Join

The join operator brings together data from two or more collections. The result is a collection of tuples. Each tuple contains data from each element in the specified collections where the values of the specified conditions match.

The notation is:

\[
\text{currentCollection} \ ? \ \text{join}(\text{namesOfCollections}; \ \text{namesOfProperties}; \ \text{booleanExpression}; \orderByExpression)
\]

Where:
- **namesOfCollections** is a list of strings separated by commas, where each string represents the name of a collection from where data is retrieved.
- The name of the current collection `currentCollection` must appear in the list.

**Notation for namesOfCollections:**
- `Collection1, collection2,…collectionn`
- `Alias1 in collection1, alias2 in collection2, …, aliasn in collectionn`

**E.g.**
- `patient, labTest, …, treatment`
- `p in patient, lt in labTest, …, t in treatment`

- **namesOfProperties** is a list of strings separated by commas, where each string is the full description of the properties from the objects in the collections we want to get in the result.

**Notation for namesOfProperties:**
- `object.property`
  - E.g. `[patient.ID, labTest.ID, labTest.result, labTest.date, treatment.ID, treatment.description]`.
  - Or using aliases: `p.ID, lt.ID, etc.`

- **booleanExpression** is a valid GELLO Boolean expression containing the conditions the elements from the collections defined in `listOfCollections` must satisfy in order to be included in the result. For each pair of collections there must be at least one condition related to these collections in the booleanExpression. In general, the number of conditions must be at least equal to the number of collections in `listOfCollections-1`.

- **orderByExpression** is a valid GELLO expression specifying the properties by which the result should be ordered. E.g. `patient.ID, treatmentID` or using aliases `p.ID, t.ID`, will sort the result by patientID and treatment ID.

**Types for "join"**

- `bag_of_tuples`: for any type of collection if `orderByExpression` is not specified.
- `sequence_of_tuples`: for any type of collection if `orderByExpression` is specified.

**Definition of evaluation function for Fjoin(V,S1,S2,E1,E2)**

If \( V \) is collection, \( S_1 \) is a parameter list of strings with the names of the collections from where data will be retrieved, \( S_2 \) is a parameter list of strings with the full names of the properties to be included in the result, \( E_1 \) is a boolean expression containing the conditions \( C \ < C_1 \ booleanOP \ C_2 \ ... \ booleanOP \ C_n \ > \) the returning elements must satisfy. The number of conditions \( C_i \) in \( E_1 \ = \ |S_1| size()-1 \). \( E_2 \) is an optional parameter that specifies the criteria for ordering the resulting elements. If \( E_2 \) is not specified, the result is a bag.

\[
\text{Fjoin}(V,S_1,S_2,E_1,E_2) = \text{bag of tuples}
\]

If \( V \) is collection, \( S_1 \) is a parameter list of strings with the names of the collections from where data will be retrieved, \( S_2 \) is a parameter list of strings with the full names of the properties to be included in the result, \( E_1 \) is a boolean expression containing the conditions \( C_i \ < C_1 \ booleanOP \ C_2 \ ... \ booleanOP \ C_n \ > \) the returning elements must satisfy. The number of conditions \( C_i \) in \( E_1 \ = \ |S_1| size()-1 \). \( E_2 \) is an optional parameter that specifies the criteria for ordering the resulting elements. If \( E_2 \) is not specified, the result is a bag.

\[
\text{Fjoin}(V,S_1,S_2,E_1,E_2) = \text{sequence of tuples}
\]
The following example returns a collection of tuples containing Medication code, effective time and value, and Lab Results code, effective time and value, for all the Lab Results performed (effective time) while a patient was taking any Medication effective time):

Example 98.

LabResults join(Medications; Medications.code, Medications.effectiveTime, Medications.value, LabResults.code, LabResults.effectiveTime, LabResults.value; Medications.effectiveTime.contains(LabResults.effectiveTime)

5.10.28 Operator Average

Average returns the average (arithmetic mean) of the numerical elements in a collection.

The notation is:

Example 99.

collection ? average()

Type for "average"
(colLECTION) ? real

Definition of evaluation function for $F_{average}(V)$

$F_{average}(V) = real$  If $V$ is a collection and all the elements in the collection are either real or integer numbers  
= undefined  otherwise

The following example returns the average value of the elements in a collection:

Example 100.

recorded_temperatures is a collection containing the recorded temperatures of a patient: {97, 98, 98.5, 99, 99, 97, 97}.

So, recorded_temperatures ? average()          returns: 97.92 F

5.10.29 Operator Stdev

Stdev returns the standard deviation of the numerical elements in a collection.

The notation is:

Example 101.

collection ? stdev()

Type for "stdev"
(colLECTION) ? real

Definition of evaluation function for $F_{stdev}(V)$

$F_{stdev}(V) = real$  If $V$ is a collection and all the elements in the collection are either real or integer numbers  
= undefined  otherwise

The following example returns the standard deviation of the elements in a collection:

Example 102.

recorded_temperatures is a collection containing the recorded temperatures of a patient: {97, 98, 98.5, 99, 99, 97, 97}.

So, recorded_temperatures ? stdev()          returns: 0.9322

5.10.30 Operator Variance
Variance returns the variance of the numerical elements in a collection.

The notation is:

Example 103.

```gello
collection ? variance()
```

Type for "variance"
(collection) ? real

Definition of evaluation function for Fvariance(V)
Fvariance(V) = real  
| If V is a collection and all the elements in the collection are 
| either real or integer numbers 
| = undefined  
| otherwise

The following example returns the variance of the elements in a collection:

Example 104.

```
recorded_temperatures is a collection containing the recorded temperatures of a patient: {97, 98, 98.5, 99, 99, 97, 97}.
So, recorded_temperatures ? variance() returns: 0.8690
```

5.10.31 Operator Median

Median returns the median of the numerical elements in a collection. The median is the number in the middle of a set of numbers; that is, half the numbers have values that are greater than the median, and half have values that are less. If the number of elements is even, then the median is the average value of the two numbers in the middle.

The notation is:

Example 105.

```gello
collection ? median()
```

Type for "median"
(collection) ? real

Definition of evaluation function for Fmedian(V)
Fmedian(V) = integer  
| If V is a collection and all the elements in the collection are 
| integer numbers 
| = real  
| Else if V is a collection and all the elements in the collection are 
| either real or integer numbers 
| = undefined  
| otherwise

The following example returns the median of the elements in a collection:

Example 106.

```
recorded_temperatures is a collection containing the recorded temperatures of a patient: {97, 98, 98.5, 99, 99, 97, 97}.
So, recorded_temperatures ? median() returns: 98
```

5.10.32 Operator Mode

Mode returns the most frequently occurring value in a collection.

The notation is:

Example 107.

```gello
collection ? mode()
```

Type for "mode"
(collection) ? real

Definition of evaluation function for Fmode(V)
Fmode(V) = integer  
| If V is a collection and all the elements in the collection are 
| integer numbers 
| = real  
| Else if V is a collection and all the elements in the collection are 
| either real or integer numbers 
| = undefined  
| otherwise

```
recorded_temperatures is a collection containing the recorded temperatures of a patient: {97, 98, 98.5, 99, 99, 97, 97}.
So, recorded_temperatures ? mode() returns: 98
```
The following example returns the median of the elements in a collection:

Example 108.

```
recorded_temperatures is a collection containing the recorded temperatures of a patient: [97, 98, 98.5, 99, 99, 97, 97].
So, recorded_temperatures ? mode() returns: 97
```

5.10.33 Operator Like

Like operator searches a collection of strings and returns those that match a given pattern.

The notation is:

Example 109.

```
collection ? like(String)
Type for "like"
(collection x string) ? collection
Definition of evaluation function for Flike(V1,V2)
Flike(V1,V2) = collection If V1 is a collection of strings and V2 is string pattern. The result is a collection of all strings that matched the given pattern. If there are no matches, the returning collection is empty.
= undefined otherwise
```

The following example returns a collection with problems that are like "ast" (asthma, astigmatism...):

Example 110.

```
patient.problemList ? collect(code) ? like("ast")
```

5.10.34 Operator NotLike

NotLike operator searches a collection of strings and returns those that do not match a given pattern.

The notation is:

Example 111.

```
collection ? notlike(String)
Type for "notlike"
(collection x string) ? collection
Definition of evaluation function for Fnotlike(V1,V2)
Fnotlike(V1,V2) = collection If V1 is a collection of strings and V2 is string pattern. The result is a collection of all strings that did not match the given pattern. If there are no matches, the returning collection is empty.
= undefined otherwise
```

The following example returns a collection with problems that are not like "ast" (e.g. diabetes, COPD):

Example 112.

```
patient.problemList ? collect(code) ? notlike("ast")
```

5.10.35 Operator Between

Between operator searches a collection of strings and returns those strings that are between a given range.
The notation is:

Example 113.

```
collection ? between(String1, String2)
```

Type for "between"

```
(collection x string x string) ? collection
```

Definition of evaluation function for Fbetween(V1,V2,V3)

```
Fbetween(V1,V2,V3) = collection   If V1 is a collection of strings and V2 and V3 are both strings defining a range. The result is a collection of all strings that are between the given range. If there are no matches, the returning collection is empty.
= undefined   otherwise
```

The following example returns a collection with problems that are between diabetes and reflux:

Example 114.

```
{asthma, copd, diabetes, IRS, meningitis, reflux, UTI} ? between(diabetes, reflux)
returns: {diabetes, IRS, meningitis, reflux}
```

5.10.36 Operator Distinct

Distinct operator returns a collection (set) with no duplicate elements. Basically is a casting operation, that converts a bag or sequence into a set, hence eliminating duplicates.

The notation is:

Example 115.

```
collection ? distinct()
```

Type for "distinct"

```
(collection) ? set
```

Definition of evaluation function for Fdistinct(V1)

```
Fdistinct(V1) = set   If V1 is a collection the result is a set
= undefined   otherwise
```

The following example returns a set with problems:

Example 116.

```
{asthma, copd, diabetes, copd, UTI, IRS, reflux, UTI} ? distinct()
returns: {asthma, copd, diabetes, UTI, IRS, reflux}
```

5.11 Tuple Operators

A GELLO tuple is an aggregated data type formed by one or more elements with different types. As described in §5.1.4, each tuple part has a name and a type. GELLO provides the following operations to handle and access tuple elements. Since all elements in a tuple have unique values, we use the ‘dot’ notation to access them in the same manner as we access attributes in a class (§5.3.1), e.g. tupleName.elemName.

5.11.1 Operator Size

The operator size returns the number of elements in a tuple.

The notation is:

Example 117.

```
Tuple.size()
```

Types for "size"

```
(tuple) ? integer
```
Definition of evaluation function for Fsize(T)

\[ Fsize(T) = \begin{cases} \text{integer} & \text{if } T \text{ is a tuple} \\ \text{undefined} & \text{otherwise} \end{cases} \]

For the Tuple `personalData{name: String = 'John Smith', nickname: String = 'Johnny', age: Integer = 10}` the following example returns 3:

Example 118.

`personalData.size()` returns 3

### 5.11.2 Operator getValue

The operator `getValue` returns the value of an element with name = `elemName` in a tuple.

The notation is:

Example 119.

Tuple.getValue(elemName)

Types for "getValue"

- `(tuple(string) ? PredefinedDataTypeValue` 
- `(tuple(string) ? ModelDataTypeObject` 

Definition of evaluation function for FgetValue(T,S)

\[ FgetValue(T,S) = \begin{cases} \text{PredefinedDataTypeValue} & \text{if } T \text{ is a tuple and } S \text{ is a string with the name of an element in } T, \text{ and the type of the returning value is one of the predefined data types: integer, real, boolean, string or one of the collection types.} \\ \text{ModelDataTypeObject} & \text{else if } T \text{ is a tuple and } S \text{ is a string with the name of an element in } T, \text{ and the type of the returning value is one of the model data types.} \\ \text{undefined} & \text{otherwise} \end{cases} \]

For the Tuple `personalData{name: String = 'John Smith', nickname: String = 'Johnny', age: Integer = 10}` the following example returns Johnny:

Example 120.

`personalData.getValue(nickname)` returns 'Johnny'

this is equivalent to: `personalData.nickname`

### 5.11.3 Operator getElemName

The operator `getElemName` returns a string with the name of the element i in the ith position in the tuple.

The notation is:

Example 121.

Tuple.getElemName(position)

Type for "getElemName"

- `(tuple x integer) ? string` 

Definition of evaluation function for FgetElemName(T,I)

\[ FgetElemName(T,I) = \begin{cases} \text{string} & \text{if } T \text{ is a tuple and } I \text{ is an integer (} 1 = T.size()) \text{.} \\ \text{undefined} & \text{otherwise} \end{cases} \]

For the Tuple `personalData{name: String = 'John Smith', nickname: String = 'Johnny', age: Integer = 10}` the following example returns nickname:

Example 122.

`personalData.getElemName(2)` returns 'nickname'
5.11.4 Operator getElemType

The operator getElemType returns a string which represents the basic or model data type associated to an element in a tuple. GetElemType can be used by giving the position or the name of an element in the tuple.

The notation is:

```
Tuple.getElemType(position)
Tuple.getElemType(elemName)
```

Types for "getElemType"

- (tuple x integer) ? string
- (tuple x string) ? string

Definition of evaluation function for FgetElemType(T,I)

```
FgetElemType(T,I) = PredefinedDataType     If T is a tuple  and I is an integer (1 = T.size()).
                    The returning value is a string referring to a predefined data type: integer, real, Boolean, string or one of the collection types.
= ModelDataType          Else if T is a tuple  and I is an integer (1 = T.size()).
                    The returning value is a string referring to a model data type.
= undefined              otherwise
```

Definition of evaluation function for FgetElemType(T,S)

```
FgetElemType(T,S)= PredefinedDataType     If T is a tuple  and S is a string referring to the name of an element in the tuple. The returning value is a string referring to a basic data type: integer, real, Boolean, string, or one of the collection types.
= ModelDataType          Else if T is a tuple and S is a string referring to the name of an element in the tuple. The returning value is a string referring to a model data type.
= undefined              otherwise
```

For the Tuple `personalData{name: String = 'John Smith', nickname: String = 'Johnny', age: Integer = 10}` the following example returns String:

```
personalData.getElemType(2)     returns String
this is equivalent to:    personalData.getElemType('nickname')
```

5.12 Date/Time Operators

The following operators handle date and time objects.

5.12.1 Operator ToDate

The operator ToDate takes a string and returns a PointInTime object.

The notation is:

```
Type for "todate"
* (string) ?  PointInTime    (a RIM object)
```

Definition of evaluation function for Ftodate(V)

```
Ftodate(V)= PointInTime    If V is a valid string. The result is a PointInTime object
= undefined      otherwise
```

The notation for using this operator is as follows. It requires the Factory method because the returning object is an instance of a RIM Class

```
The following expression creates a PointInTime object with the argument string:
Let aDate: PointInTime = Factory.PointInTime(string)
```
5.12.2 Operator AddMonths, AddDate and NextDay

These operations are supported by the RIM PoinInTime Class using the operator plus.

5.12.3 Operator LastDay

This operation is fully supported by the RIM (time) Interval Class using the operator high.

5.13 Precedence Rules

The precedence order for operations in GELLO, starting with the highest precedence, is as follows:

- "dot" ("."), and "arrow" ("? ") operations
- unary "not" and unary "minus"
- "*", "/", "div" and "mod"
- "+" and binary "-"
- "if-then-else-endif"
- '<', '>', '<=', '>='
- '!=', '<>'
- 'and', 'or' and 'xor'
- 'implies'
- Parentheses "(" and ")" can be used to change precedence.

5.14 If Expression

An If Expression evaluates a condition and depending on the resulting truth value, the result is one of two possible expressions. Both expressions are mandatory. The If Expression is not intended for control flow, but as a conditional for the returning value of an expression. The syntax of an If Expression is as follows:

Example 127.

```plaintext
if condition then
  expression1
else
  expression2
endif
```

Example 128.

Definition of the evaluation function Fif(V1, V2, V3), where V1 is the condition, a GELLO expression which its evaluation returns a truth value; and V2 and V3 are expression1 and expression2 respectively, both valid GELLO expressions.

Fif(V1, V2, V3) = V2            If V1 = true
= V3            Else if V1 = false
= undefined     otherwise

Example 129.

```plaintext
let renal_failure :Boolean =
  if lastCreatinine.oclIsDefined() and
    lastCreatinine.value.greaterThan(renal_failure_threshold)  then
      true
    else
      false
  endif
```

6 GELLO Syntax

This section describes the grammar used in this specification to define the lexical and syntactic structure of GELLO expressions. A context-free grammar consists of a number of productions. Each production is formed by two parts: the left-hand side consisting of a nonterminal symbol and a right-hand side formed by a sequence of one or more nonterminal and terminal symbols.

Starting from a sentence consisting of a single nonterminal, and a set of production rules, the complete grammar is derived by means of a set of possible sequences of terminal symbols that can result from repeatedly replacing any nonterminal symbol in a sequence with its associated right-hand side sequence of a production rule.

Section §?6.2 describes the grammar used in this specification to define the lexical and syntactic structure of GELLO expressions. For an expression to be syntactically correct it must conform to:

The BNF and lexical grammar defined in this section of the document (§?6.3).
The context sensitive constraints:

Every expression must be type-correct. It must comply with the type definitions in §5.1.

### 6.1 Inferring Type Rules for Expressions

Let $E$ be a valid GELLO expression. The type of $E$ is either a basic (Integer, Real, Boolean or String), a model type or one of the collection or tuple types. The type of $E$ can be inferred by using the rules defined in GELLO lexical grammar §6.2 and GELLO BNF §6.3.

### 6.2 GELLO Lexical Grammar

GELLO BNF syntax is defined in terms of the following lexical tokens. GELLO grammar is based on the grammar for [OCL expressions](https://www.omg.org/spec/OCL/).

A reserved word is any string that appears in double quotes in the BNF. Reserved words are case sensitive.

An atom consists of any sequence of alphanumeric characters, which begins with a letter and can contain one or more underscores. Atoms are case sensitive.

A number could be either an integer or a real.

An integer is represented by one or more digits

A real is represented by a sequence of one or more digits followed by "." followed by zero or more digits, optionally followed by "e" or "E" a sign "+" or "-" one or more digits and "d" or "D".

A single quoted string is a pair of single quote characters enclosing a sequence of zero or more characters other than comments, tabs, newlines and carriage returns.

A comment is any sequence of characters other than newlines, or carriage returns following two successive dashes --, e.g.

```
-- this is a comment.
```

### 6.3 GELLO BNF Syntax

The Backus-Naur Form (BNF) syntax of GELLO assumes that text defining a GELLO expression has been converted into lexical tokens by the lexical analyzer defined in the previous section.

The following notational conventions are used throughout GELLO BNF syntax:

- The root symbol of the syntax is `<GELLOExpression>`
- Non-terminal symbols are denoted with underlined text strings, e.g. `<expression>`
- Left-hand side terms in production rules are nonterminal
- Tokens are represented with text strings enclosed in angle brackets, e.g. `<atom>`.
- Reserved words are represented by text strings enclosed in double quotes.
- The grammar below uses the following conventions:
  1. `(x)` denotes zero or one occurrences of $x$.
  2. `(x)*` denotes zero or more occurrences of $x$.
  3. `(x)+` denotes one or more occurrences of $x$.
  4. $x \mid y$ means one of either $x$ or $y$.

#### 6.3.1 Root Symbol

```
GELLOExpression ::= OuterExpression
```

#### 6.3.2 Literals
Literal ::= <STRING_LITERAL>
 |   <INTEGER_LITERAL>
 |   <REAL_LITERAL>
 |   <TRUE>
 |   <FALSE>
 |   <UNKNOWN>
 |   <NULL>
 |   CollectionLiteral
 |   TupleLiteral
 |   "#" <ID>

CollectionLiteral ::= CollectionType? "{" ( CollectionLiteralElement ( "," CollectionLiteralElement )* )? "}"

CollectionLiteralElement ::= Expression ( ".." Expression )?

TupleLiteral ::= <TUPLE> "{" TupleLiteralElement ( "," TupleLiteralElement )* "}"

TupleLiteralElement ::= <ID> (":" DataTypes)? <EQUAL> Expression

6.3.3 Data Types

DataTypes ::= GELLOType
 |   ModelTypes

GELLOType ::= BasicType
 |   CollectionType ("(" DataTypes ")" )?
 |   TupleType
 |   EnumerationType

BasicType ::= <INTEGER>
 |   <STRING>
 |   <REAL>
 |   <BOOLEAN>

ModelTypes ::= ClassName

CollectionType ::= <SET>
 |   <BAG>
 |   <SEQUENCE>

TupleType ::= <TUPLE> "(" TupleTypeElement ( "," TupleTypeElement )* ")"

TupleTypeElement ::= <ID> (":" DataTypes

EnumerationType ::= <ENUM> "(" <ID> ( "," <ID> )* ")"
6.3.4 Names

NameWithPath ::= Name ( ":-" Name )*  

Name ::= <ID> ("." <ID>)*

6.3.5 Expressions

Expression ::= ConditionalExpression  

ConditionalExpression ::= OrExpression  

OrExpression ::= ConditionalAndExpression (<OR> ConditionalAndExpression | <XOR> ConditionalAndExpression)*  

ConditionalAndExpression ::= ComparisonExpression (<AND> ComparisonExpression)*  

ComparisonExpression ::= AddExpression (<EQUAL> AddExpression | <NEQ> AddExpression | <LT> AddExpression | <LEQ> AddExpression | <GT> AddExpression | <GEQ> AddExpression)*  

AddExpression ::= MultiplyExpression (<MINUS> MultiplyExpression | <PLUS> MultiplyExpression)*  

MultiplyExpression ::= UnaryExpression (<TIMES> UnaryExpression | <DIVIDE> UnaryExpression | <MAX> UnaryExpression | <MIN> UnaryExpression | <INTDIV> UnaryExpression | <MOD> UnaryExpression )*  

UnaryExpression ::= PrimaryExpression  

| <NOT> UnaryExpression  

| <MINUS> UnaryExpression  

| <PLUS> UnaryExpression  

| PrimaryExpression  

PrimaryExpression ::= Literal  

| Operand  

| ReferenceToInstance  

| IfStatement  

| "(" Expression ")"  

ExpressionNP ::= ConditionalExpressionNP  

ConditionalExpressionNP ::= OrExpressionNP
OrExpressionNP::=   ConditionalAndExpressionNP (<OR> ConditionalAndExpression | <XOR> ConditionalAndExpression)*

ConditionalAndExpressionNP::=   ComparisonExpressionNP (<AND> ComparisonExpression)*

ComparisonExpressionNP::=   AddExpressionNP (<EQUAL> AddExpression | <NEQ> AddExpression | <LT> AddExpression | <LEQ> AddExpression | <GT> AddExpression | <GEQ> AddExpression)*

AddExpressionNP::=   MultiplyExpressionNP(<MINUS> MultiplyExpression | <PLUS> MultiplyExpression)*

MultiplyExpressionNP::=   UnaryExpressionNP (<TIMES> UnaryExpression | <DIVIDE> UnaryExpression | <MAX> UnaryExpression | <MIN> UnaryExpression | <INTDIV> UnaryExpression | <MOD> UnaryExpression )*

UnaryExpressionNP::=   PrimaryExpressionNP | <NOT> UnaryExpression

PrimaryExpressionNP::=   Literal | Operand | ReferenceToInstance | IfStatement

Operand::=   <ID> | Operand "." <ID> | Operand "." StringOperation | Operand "." TupleOperation | Operand "." StringOrTupleSize | Operand "." ( ParameterList "") | Operand [" ExpressionList "] | Operand <ARROW> CollectionBody | CollectionLiteral <ARROW> CollectionBody | <SELF>


SelectionExp::=   <SELECT> "(" CExp ")" | <REJECT> "(" CExp ")" | <COLLECT> "(" CExp ")"
QuantifierExp::=  
  <FORALL> "(" CExp ")"
  |  <EXISTS> "(" CExp ")"

CExp::= ConditionalExpression
  |  ConditionalExpressionWithIterator
  |  ConditionalExpressionWithIteratorType

ConditionalExpressionWithIterator::=  <ID> "|" ConditionalExpression

ConditionalExpressionWithIteratorType::=  <ID> ":" DataTypes "|" ConditionalExpression

NonParamExp::=  <SIZE> "(" ")"
  |  <ISEMPTY> "(" ")"
  |  <NOTEMPTY> "(" ")"
  |  <SUM> "(" ")"
  |  <REVERSE> "(" ")"
  |  <MIN> "(" ")"
  |  <MAX> "(" ")"
  |  <FLATTEN> "(" ")"
  |  <AVERAGE> "(" ")"
  |  <MEAN> "(" ")"
  |  <MEDIAN> "(" ")"
  |  <MODE> "(" ")"
  |  <STDEV> "(" ")"
  |  <VARIANCE> "(" ")"
  |  <DISTINCT> "(" ")"

SingleObjExp::=  <COUNT> "(" Object ")"
  |  <INCLUDES> "(" Object ")"
  |  <INCLUDING> "(" Object ")"
  |  <EXCLUDING> "(" Object ")"

ListObjExp::=  <INCLUDESALL> "(" ObjectList ")"
  |  <SORTBY> "(" PropertyList ")"

GetExp::=  <FIRSTN> "(" Expression ")"
  |  <LASTN> "(" Expression ")"
  |  <ELEMAT> "(" Expression ")"
  |  <LIKE> "(" Expression ")"
  |  <NOTLIKE> "(" Expression ")"
  |  <BETWEEN> "(" Expression "," Expression ")"

SetExp::=  <INTERSECTION> "(" Expression ")"
  |  <UNION> "(" Expression ")"

IterateExp::=  <ITERATE> "(" IterateParameterList ")"

JoinExp::=  <JOIN> "(" ParameterList ";" ParameterList ";"
             ConditionalExpression ";" ParameterList ")"
StringOperation ::= StrConcat
|   StrToUpper
|   StrToLower
|   Substring

StringOrTupleSize ::= <SIZE> "(" ")"

StrConcat ::= <CONCAT> "(" Expression ")"

StrToUpper ::= <TOUPPER> "(" ")"

StrToLower ::= <TOLOWER> "(" ")"

Substring ::= <SUBSTRING> "(" Expression "," Expression ")"

TupleOperation ::= TupleGetValue
|   TupleGetElemName
|   TupleGetElemType

TupleGetValue ::= <GETVALUE> "(" TupleElemName ")"

TupleGetElemName ::= <GETELEMNAME> "(" Expression ")"

TupleGetElemType ::= <GETELEMTYPE> "(" Expression ")"

IterateParameterList ::= <ID> "(" DataTypes)? ";" <ID> ";" DataTypes
|   <EQUAL> Expression "|" Expression

ParameterList ::= ExpressionList?

ExpressionList ::= Expression ("," Expression)*

ObjectList ::= Object ("," Object)*

Object ::= Expression

PropertyList ::= Property ("," Property)*
6.3.6 Statements

OuterExpression ::= Declarative+ (ExpressionNP? | <IN> Expression) | Expression

Declarative ::= LetStatement | ContextNavigationStatement

InnerExpression ::= LetStatement+ (ExpressionNP | <IN> Expression) | Expression

LetStatement ::= <LET> <ID> ":" DataTypes <EQUAL> Expression

IfStatement ::= <IF> Expression <THEN> InnerExpression <ELSE> InnerExpression <ENDIF>

ContextNavigationStatement ::= ContextStatement | PackageStatement

ContextStatement ::= <CONTEXT> ContextClass ContextBlock? | <CONTEXT> Alias ":" ContextClass ContextBlock?

ContextClass ::= ClassName | CollectionType "(" ContextClass ")"

ContextBlock ::= DefinitionBody+

DefinitionBody ::= <DEF> ":" <ID> ":" DataTypes <EQUAL> InnerExpression | <DEF> ":" <ID> "(" FormalParams? ")" ":" DataTypes <EQUAL> InnerExpression

FormalParams ::= <ID> ":" DataTypes "," FormalParams?

Alias ::= <ID>

PackageStatement ::= <PACKAGE> PackageName ContextStatement+ <ENDPACKAGE>

PackageName ::= Name

6.3.7 Reference to an Instance of a Model Class
6.3.8 Literals and Identifiers

\(<\#DECIMAL\_LITERAL:\ (["0"-"9"])+ >\)
\(<\#EXPONENT:\ ["e","E"] (["+","-"])? (["0"-"9"])+ >\)
\(<\INTEGER\_LITERAL: <DECIMAL\_LITERAL>>\)
\(<\REAL\_LITERAL: <DECIMAL\_LITERAL> \." (["0"-"9"])* (<EXPONENT>)? | \." (["0"-"9"])+ (<EXPONENT>)? >\)
\(<\STRING\_LITERAL: (\" | \"\n", \"\r")\~ (\" | \"\n", \"\r")\~ \" | \" | \"\n", \"\r")\~ \"\)\)
\(<ID: ["a"-"z","A"-"Z"] (["a"-"z","A"-"Z",0-9]) | _(["a"-"z","A"-"Z",0-9])*>\)

6.3.9 Reserved Words

\(<BAG: \"Bag\">\)
\(<BOOLEAN: \"Boolean\">\)
\(<ENUM: \"Enum\">\)
\(<INTEGER: \"Integer\">\)
\(<NULL: \"null\">\)
\(<REAL: \"Real\">\)
\(<SEQUENCE: \"Sequence\">\)
\(<SET: \"Set\">\)
\(<STRING: \"String\">\)
\(<SELF: \"Self\">\)
\(<TUPLE: \"Tuple\">\)
\(<UNKNOWN: \"unknown\" | \"Unknown\">\)

6.3.10 Operators

\(<AND: \& | \"and\">\)
\(<ARROW: \"->\" | \"?\">\)
\(<AVERAGE: \"average\">\)
\(<BETWEEN: \"between\">\)
\(<COLLECT: \"collect\">\)
\(<CONCAT: \"concat\">\)
\(<COUNT: \"count\">\)
<DEF: "Def" | "def" >

<DISTINCT: "distinct" >

<DIVIDE: "/" >

<ELEMAT: "elemat" >

<EXCLUDING: "excluding" >

<EXISTS: "exists" >

<FACTORY: "factory" >

<FALSE: "false" | "False" >

<FIRSTN: "firstN" >

<FLATTEN: "flatten" >

<FORALL: "forAll" >

<EQUAL: "=" >

<GEQ: ">=" >

<GETELEMNAME: "getElemName" >

<GETELEMTYPE: "getElemType" >

<GETVALUE: "getValue" >

<GT: ">" >

<INCLUDES: "includes" >

<INCLUDESALL: "includesAll" >

<INCLUDING: "including" >

<INTDIV: "div" >

<INTERSECTION: "intersection" >

<IEMPTY: "isEmpty" >

<ITERATE: "iterate" >

<JOIN: "join" >

<LASTN: "lastN" >

<LEQ: "<=" >

<LIKE: "like" >

<LT: "<" >
<MAX: "max" />
<MEAN: "mean" />
<MEDIAN: "median" />
<MIN: "min" />
<MINUS: "-" />
<MOD: "mod" />
<MODE: "mode" />
<NEQ: "!=" | "<>" />
<NEW: "new" />
<NOT: "!" | "not" />
<NOTEMPTY: "notEmpty" />
<NOTLIKE: "notlike" />
<OR: "|" | |or" />
<PLUS: "+" />
<REJECT: "reject" />
<REVERSE: "reverse" />
<SELECT: "select" />
.SIZE: "size" />
<SUBSTRING: "substring" />
<SUM: "sum" />
<TIMES: "*" />
<TOLOWER: "toLowerCase" />
<TOUPPER: "toUpperCase" />
<TRUE: "true" />
<UNION: "union" />
<VARIANCE: "variance" >
6.4 GELLO Expressions

A GELLO expression is any text string conforming to the definition of an expression in the GELLO language specification. GELLO expressions can be used to:

- Access information from a repository
- Build decision criteria
- Abstract or derive summary values

When an expression is evaluated, the result of such evaluation is a value. The type of the result is the type of the expression.

Evaluation of an expression does not produce any side effects, although the returning value can be bound to a variable name and used by the guideline to make decisions, control execution flow, etc. If an expression can be embedded in a conditional statement, the returning value is interpreted by the application to which the conditional statement belongs.

6.5 Type of an Expression

If an expression denotes a variable or a value, then such expression has a type that must be checked for compatibility. Such variable or value must match any of GELLO predefined §5.1.1, collection §5.1.3 or tuple data types §5.1.4, or classes defined in the underlying data model §5.1.2.

If a value is bound to a variable name, both the returning value and the variable to which it is assigned must be of the same type.

6.6 Normal and Abrupt Completion of Evaluation

Expressions are evaluated by following a series of steps. Normal completion signifies that all steps can be carried out without an exception being thrown. If, however, evaluation of an expression throws an exception, the expression is said to complete abruptly. GELLO provides basic error checking described in the following section.
operand, the return type of the function is undefined.

6.6.2 Handling Exceptions

Although GELLO provides basic type checking, it does not provide any mechanisms for handling exceptions as a result of a type mismatch. The applications into which GELLO is embedded should provide the necessary error handling mechanisms.

6.7 Evaluation of Expressions

Expressions are evaluated from left to right. In the case of infix operators, the evaluation order is determined by the precedence of the operators.

6.7.1 Argument List

Argument lists included in method invocations are evaluated left-to-right.

6.8 Example of Expressions

When the following expressions are evaluated, they return a value of type Boolean. Expressions like these can be used to build decision criteria:

Example 130.

- calcium ? notEmpty() and phosphate ? notEmpty()
- renal_failure and calcium_phosphate_product > threshold_for_osteodystrophy
- Observation ? select(coded_concept="C0428279")

The expression above returns a collection of observations with a coded concept equal to "C0428279". The result of an expression can be bound to a variable name using the let operator:

- let CreatinineReadings: set = Observation ? select(coded_concept="C0428279")

7 Examples in GELLO

In this section we present some examples written in GELLO.

7.1 An MLM into GELLO

From an MLM:

Example 131.

Example in GELLO:

Example 132.

let lastCreatinine : Observation = Observation ? select(code=Factory.CodedValue("SNOMED-CT", "xxxxxx")).sortedBy(effectiveTime.high).last()
let lastCalcium : Observation = Observation ? select(code=Factory.CodedValue("SNOMED-CT", "yyyyy")).sortedBy(effectiveTime.high).last()
let lastPhosphate : Observation = Observation ? select(code=Factory.CodedValue("SNOMED-CT", "zzzzz")).sortedBy(effectiveTime.high).last()
let renal_failure :Boolean = if lastCreatinine <> null and lastCreatine.value.greaterThan(renal_failure_threshold) then true else false endif
let calcium_phosphate_product : real = if lastCalcium...
7.2 Example of an iteration over more than one collection at a time

This example shows how collection operators can be nested in expressions as long as they comply with the notation.

Statement in English (many thanks to Samson Tu):

"There exists (for a patient) an anti-hypertensive prescription (?drug) such that there exists (for the patient) a problem (?problem) such that ?problem is a compelling indication for ?drug". Where:

- 'a patient' is the current patient
- drug is any drug in the drug database
- ?problem is a patient's problem

Statement in English (many thanks to Samson Tu):

Example 133.

Presence of Azotemia Observation within last three months: Assumptions:
1. The data model has as code a generic term such as SNOMED "finding" ("246188002") and the value slot has the code for Azotemia.
2. For a diagnosis such as azotemia, the effective time is the time interval during which the disease is thought to be present.
3. A PointInTime.NOW() function returns the current time

Example in GELLO:
Let month : CodedValue = Factory.CodedValue("SNOMED-CT", "258706009")
Let finding : CodedValue = Factory.CodedValue("SNOMED-CT", "246188002")
Let azotemia : CodedValue = Factory.CodedValue("SNOMED-CT", "371019009")
Observation ? exists(code.equal(finding) and value.implies(azotemia) and effective_time.intersect(ThreeMonthsAgo, PointInTime.NOW()))

7.3 Example: Number of current anti-hypertensive Medications > 1

Statement in English (many thanks to Samson Tu):

Number of current anti-hypertensive Medications > 1

Example 134.

Example in GELLO:
Let hypotensive_agents : CodedValue = Factory.CodedValue("SNOMED-CT", "1182007")
MedicationOrder?select(code.implies(hypotensive_agents) and effectiveTime.high = null)?size() > 1

7.4 3rd Td dose before 12 months of age

Statement in English (many thanks to Samson Tu):

3rd Td dose before 12 months of age

Example 135.

Example in GELLO:
Let month : CodedValue = Factory.CodedValue("SNOMED-CT", "258706009")
Let DOBcode : CodedValue = Factory.CodedValue("SNOMED-CT", "184099003")
Let DateOfBirth : Observation= Factory.Observation? select(code.equal(DOBCode)).sortedBy(effectiveTime.high).last()
### 8 Grouping GELLO expressions into Model Processes

GELLO expressions can be grouped into ‘Model processes’ to perform user-defined operations upon given classes of the data model for a specific purpose.

A model process must be defined as an attribute of a stereotype class which could be:

1. a user-defined class independent of the data model,
2. a class dependent of a class in the data model. The latter being the ‘metaclass’ to which the user-defined stereotype class relates to. In this case the stereotype class must follow the UML class hierarchy definition (UML-OMG v1.5 p2-75)

In both cases the names of new stereotypes must not clash with the names of predefined classes in the data model. The stereotype class should not affect any of the properties of the classes in the data model, but rather, to extend such properties adding extra functionality.

Stereotypes may be assembled into ‘profiles’ –or libraries. A profile is a stereotyped package containing user-defined stereotype model elements customized for a specific domain or purpose (OCL/UML). In the following example, Package P1 is the profile where we assembled the stereotypes S1 and S2 dependent on C1 and C2 respectively. Both C1 and C2 are classes from the data model and hence S1 and S2 add extended functionality to those classes.

Example 136.

```
Package P1
  Context S1<<stereotype>> : C1
  Context S2<<stereotype>> : C2
EndPackage
```

Example 137.

```
Package P1
  Context Azotemia3months : Observation
    Def: checkAzotemia() : Boolean =
      Let month:CodedValue = Factory.CodedValue("SNOMED-CT", "258706009")
      Let finding: CodedValue = Factory.CodedValue("SNOMED-CT", "246188002")
      Let azotemia: CodedValue = Factory.CodedValue("SNOMED-CT", "371019009")
      Observation->exists(code.equal(finding) and value.implies(azotemia) and effective_time.intersect(ThreeMonthsAgo, PointInTime.Now()))
EndPackage
```

In the following example, S3 is a stereotype independent of the data model located in Package P2:

Example 138.

```
Package P2
  Context S3 <<stereotype>>
EndPackage
```

As with all GELLO expressions, all model processes must be side-effect free, that is, they should not change the state of the modeled system.

The exhibit below is the UML representation for a user-defined stereotype class as an extension of a Model Class.
The exhibit below is an example of the user-defined Azotemia3months as a stereotype class related to the model class Observation. CheckAzotemia is the model process containing the GELLO expressions that evaluate whether a patient has suffered from azotemia within the past 3 months.

```plaintext
Let month: CodeValue = Factory.CodeValue("SNOMED-CT", "258706009")
Let finding: CodeValue = Factory.CodeValue("SNOMED-CT", "246188002")
Let azotemia: CodeValue = Factory.CodeValue("SNOMED-CT", "371019009")
Observation -> exists(code.equal(finding) and value.implies(azotemia) and effective_time.intersect(ThreeMonthsAgo, PointInTime.NOW()))
```
9 Acknowledgements

We would like to thank Gunther Schadow, Grahame Grieve, Dale Nelson, Bob Dolin, Anthony Malia, Mor Peleg, and Eclipsys Corporation for their valuable comments.

Support for this project has been provided by the CKBP grant and Partners Information Systems.

10 References

• HL7 RIM

• HL7 RIM information model

• UML-OMG v1.5

• Combining OCL and Programming Languages for UML Model Processing

• A virtual medical record for guideline-based decision support.

• The Virtual Medical Record (vMR)

• Minute from HL7 CDS TC San Diego, CA, January 2002.

• [DSG02-01] BNF and built-in classes for object-oriented guideline expression language (GELLO).

• [TC1] Harmonization of Arden’s expression syntax and the RIM

• [TC2] Expression and query languages

• [TC3] Progress on Guideline Expression Language also here

• [TC4] Progress on Object-Oriented Guideline Expression Language (GELLO) Also here

• [TC5] The GELLO Report. 2002 Spring WGM

• [TC6] GELLO Update. HL7 Meeting, Baltimore, October 2002

• UML 2.0 OCL

• Guideline Execution Model

• OWL latest work in progress report

• Example of an OWL ontology.

• Current W3C recommendations and technical documents

• UML ITS data model


A Annex A: HL7v3DataTypes
B  Annex B: The HL7 Reference Information Model

THIS IS THE SECTION WHERE THE DIAGRAM IS---- RIM FoundationClasses Subject Area

C  Annex C: The core UML OCL kernel declarations
D Annex D: A Simplified Data Model

The following is a simplified data model included in this specification so examples of GELLO operators and operations can be illustrated and easily followed. The model upon which the simplified data model is based is the result of a feasibility study which reviewed various decision support systems within Brigham & Women's Hospital and Massachusetts General Hospital [GR04]. The simplified data model consists of 5 model classes, each of which holds some properties that can be mapped into the HL7 RIM. These 5 classes and their equivalent in the RIM are showed in Table 1.
Classes in the Data Model | Equivalent Classes in the HL7 RIM
--- | ---
Patient | Person in the role of Patient
LabResult | Observation
ProblemList | Observation
Allergy | Observation
Medication | Substance Administration

Figure below depicts the data model. Each class includes some properties equivalent to those found in the RIM. The data model is by no means extensive. This data model will be used along the document to exemplify the use of GELLO operators in expressions in a simple manner.

When referencing RIM classes directly more complex expressions can be written. Such expressions may include references to mood, and class code, in accordance with the HL7 RIM specification.

E Annex E: Temporal Relations and Temporal Intervals

It is assumed that the HL7 v3 Data Types provides all 13 operators for handling Allen relations on temporal intervals. Such operators can be used in GELLO expressions referring to the appropriate model class and method. The 13 operators are described below for clarity.

A time interval is a set of consecutive time-stamps values during which the given information is expected to be valid. As defined in HL7 V3 Data Types (p 100), a time interval can be open or closed, infinite or undefined on either side. Graphic examples of temporal relations are depicted in Figure 2. There are thirteen fundamental relations, known as the Allen primitives, between pairs of time intervals.

E.1 Before

The notation is:

```
before(interval1, interval2).
```

Types of "before"

(timeInterval x timeInterval) ? Boolean
E.2 After

The notation is:

Example 140.

```
after(interval1, interval2).
```

Types of "after"
(timeInterval x timeInterval) ? Boolean

Definition of evaluation function for Fafter(IVL1, IVL2)
Fafter(IVL, IVL) = true \[ If IVL1 and IVL2 are both time intervals and the start-point of interval1 occurs after the end-point of interval2.
\]

= false \[ Else if IVL1 and IVL2 are both time intervals and the start-point of interval1 does not occur after the end-point of interval2.
\]

= undefined otherwise

E.3 Meets

The notation is:

Example 141.

```
meets(interval1, interval2).
```

Types of "meets"
(timeInterval x timeInterval) ? Boolean

Definition of evaluation function for Fmeets(IVL1, IVL2)
Fmeets(IVL, IVL) = true \[ If IVL1 and IVL2 are both time intervals and the end-point of interval1 is simultaneous with the start-point of interval2.
\]

= false \[ Else if IVL1 and IVL2 are both time intervals and end-point of interval1 is not simultaneous with the start-point of interval2.
\]

= undefined otherwise

E.4 Met-By

The notation is:

Example 142.

```
met-by(interval1, interval2).
```

Types of "met-by"
(timeInterval x timeInterval) ? Boolean

Definition of evaluation function for Fmet-by(IVL1, IVL2)
Fmet-by(IVL, IVL) = true \[ If IVL1 and IVL2 are both time intervals and the start-point of interval1 is earlier than the end-point of interval2.
\]

= false \[ Else if IVL1 and IVL2 are both time intervals and the start-point of interval1 is earlier than the end-point of interval2.
\]

= undefined otherwise

E.5 Overlaps

The notation is:

Example 143.

```
overlaps(interval1, interval2).
```

Types of "overlaps"
(timeInterval x timeInterval) ? Boolean

Definition of evaluation function for Foverlaps(IVL1, IVL2)
Foverlaps(IVL, IVL) = true \[ If IVL1 and IVL2 are both time intervals and the start-point of interval1 occurs strictly between the start- and end-points of interval2.
\]

= false \[ Else if IVL1 and IVL2 are both time intervals
\]

= undefined otherwise
E.6 Overlapped-by

The notation is:

\[
\text{overlapped-by(interval1, interval2)}.
\]

Types of "overlapped-by"
(timeInterval x timeInterval) ? Boolean

Definition of evaluation function for Foverlapped-by(IVL1, IVL2)
\[F_{\text{overlapped-by}}(IVL1, IVL2) = \begin{cases} 
\text{true} & \text{if IVL1 and IVL2 are both time intervals and the start-point of interval1 is not earlier than the start-point of interval2, or the end-point of interval1 does not occur strictly between the start- and end-points of interval2.} \\
\text{false} & \text{else if IVL1 and IVL2 are both time intervals and the start-point of interval1 does not occur between the start- and end-points of interval2, or the end-point of interval1 occurs before the end-point of interval2.} \\
\text{undefined} & \text{otherwise}
\end{cases}\]

E.7 Starts

The notation is:

\[
\text{starts(interval1, interval2)}.
\]

Types of "starts"
(timeInterval x timeInterval) ? Boolean

Definition of evaluation function for Fstarts(IVL1, IVL2)
\[F_{\text{starts}}(IVL1, IVL2) = \begin{cases} 
\text{true} & \text{if IVL1 and IVL2 are both time intervals and the start-point of interval1 occurs simultaneously with the start-point of interval2, but the end-point of interval1 occurs after the end-point of interval2.} \\
\text{false} & \text{else if IVL1 and IVL2 are both time intervals and the start-point of interval1 is not simultaneous with the start-point of interval2, or the end-point of interval1 occurs before the end-point of interval2.} \\
\text{undefined} & \text{otherwise}
\end{cases}\]

E.8 Started-by

The notation is:

\[
\text{started-by(interval1, interval2)}.
\]

Types of "started-by"
(timeInterval x timeInterval) ? Boolean

Definition of evaluation function for Fstarted-by(IVL1, IVL2)
\[F_{\text{started-by}}(IVL1, IVL2) = \begin{cases} 
\text{true} & \text{if IVL1 and IVL2 are both time intervals and the start-point of interval1 occurs simultaneously with the start-point of interval2, but the end-point of interval1 occurs later than the end-point of interval2.} \\
\text{false} & \text{else if IVL1 and IVL2 are both time intervals and the start-point of interval1 is not simultaneous with the start-point of interval2, or the end-point of interval1 occurs before the end-point of interval2.} \\
\text{undefined} & \text{otherwise}
\end{cases}\]

E.9 During

The notation is:

\[
\text{during(interval1, interval2)}.
\]
Types of "during"
(timeInterval x timeInterval) ? Boolean

Definition of evaluation function for Fduring(IVL1,IVL2)
Fduring(IVL,IVL) = true         If IVL1 and IVL2 are both time intervals and the start-point of interval1 occurs after the start-point of interval2 and the end-point of interval1 occurs earlier than the end-point of interval2.
= false        Else if IVL1 and IVL2 are both time intervals but the start-point of interval1 occurs before the start-point of interval2 or the end-point of interval1 occurs later than the end-point of interval2.
= undefined    otherwise

E.10 Contains
The notation is:

contains(interval1,interval2).

Example 148.

Types of "contains"
(timeInterval x timeInterval) ? Boolean

Definition of evaluation function for Fcontains(IVL1,IVL2)
Fcontains(IVL,IVL) = true         If IVL1 and IVL2 are both time intervals and the start-point of interval1 is earlier than the start-point of interval2, and the end of interval1 occurs later than the end-point of interval2.
= false        Else if IVL1 and IVL2 are both time intervals but the start-point of interval1 is later than the start-point of interval2, or the end of interval1 occurs earlier than the end-point of interval2.
= undefined    otherwise

E.11 Finishes
The notation is:

finishes(interval1,interval2).

Example 149.

Types of "finishes"
(timeInterval x timeInterval) ? Boolean

Definition of evaluation function for Ffinishes(IVL1,IVL2)
Ffinishes(IVL,IVL) = true         If IVL1 and IVL2 are both time intervals and the end-point of interval1 is simultaneous with the end-point of interval2, but the start-point of interval1 is later than the start-point of interval2.
= false        Else if IVL1 and IVL2 are both time intervals but the end-point of interval1 is not simultaneous with the end-point of interval2, or the start-point of interval1 is earlier than the start-point of interval2.
= undefined    otherwise

E.12 Finished-by
The notation is:

finished-by(interval1,interval2).

Example 150.

Types of "finished-by"
(timeInterval x timeInterval) ? Boolean

Definition of evaluation function for Ffinished-by(IVL1,IVL2)
Ffinished-by(IVL,IVL) = true         If IVL1 and IVL2 are both time intervals and the end-point of interval1 is simultaneous with the end-point of interval2, but the start-point of interval1 is earlier than the start-point of interval2.
= false        Else if IVL1 and IVL2 are both time intervals but the end-point of interval1 is not simultaneous with the end-point of interval2, or the start-point of interval1 is later than the start-point of interval2.
= undefined    otherwise

E.13 Equals
The notation is:

equals(interval1, interval2).

Types of "equals"
(timeInterval x timeInterval) ? Boolean

Definition of evaluation function for Fequals(IVL1, IVL2)
Fequals(IVL, IVL) = true  If IVL1 and IVL2 are both time intervals and the
    -- and end-points of both interval1 and
    start- and end-points of both interval1 and
    interval2 are respectively simultaneous.
    = false  Else if IVL1 and IVL2 are both time intervals
    but the start- and end-points of both interval1
    and interval2 respectively are not simultaneous.
    = undefined  otherwise

Return to top of page