

**Technical Report  
DSG-TR-2003-02**



**Software Specifications for GELLO:  
An Object-Oriented Query and Expression Language for  
Clinical Decision Support**

Margarita Sordo, Ph.D., Omolola Ogunyemi, Ph.D.,  
Aziz A. Boxwala, M.B.B.S., Ph.D., Robert A. Greenes, M.D., Ph.D.

Decision Systems Group, Brigham & Women's Hospital, Harvard Medical  
School, Boston, MA

**[msordo@dsg.harvard.edu](mailto:msordo@dsg.harvard.edu)**

December 2003

©

Copyright, Decision Systems Group, 2002, 2003

# Contents

1.	Introduction.....	6
2.	Requirements for Query and Expression Languages in the Clinical Context .....	8
2.1.	Underlying Data Model .....	9
2.2.	Queries, Expressions and Clinical Guidelines .....	9
3.	GELLO: Goals and Properties .....	9
4.	OCL.....	10
4.1.	Why OCL? .....	11
4.2.	OCL Language Description .....	12
4.2.1.	Relation to the UML Metamodel.....	12
4.2.1.1.	Self .....	12
4.2.1.2.	Specifying the UML context.....	12
4.2.1.3.	Invariants.....	12
4.2.1.4.	Pre- and Post- conditions .....	12
4.2.1.5.	Package Context.....	13
4.2.2.	Basic Values and types .....	13
4.2.2.1.	Types from the UML Model.....	13
4.2.2.2.	Enumeration Types .....	13
4.2.2.3.	Let Expressions and «definition» Constraints .....	13
4.2.2.4.	Type Conformance.....	13
4.2.2.5.	Casting .....	13
4.2.2.6.	Precedence Rules .....	13
4.2.2.7.	Infix Operators .....	13
4.2.2.8.	Keywords .....	13
4.2.2.9.	Undefined Values.....	14
4.2.3.	Objects and Properties .....	14
4.2.3.1.	Reference to Object Properties .....	14
4.2.3.2.	Combining Properties .....	15
4.2.3.3.	Pathnames for Packages.....	15
4.2.3.4.	Predefined Properties on all Objects.....	15
4.2.3.5.	Features on Classes Themselves.....	15
4.2.4.	Collections and Collection Operators .....	15
4.2.4.1.	Collections .....	15
4.2.4.2.	Collections of Collections.....	15
4.2.4.3.	Collection Operators .....	16
5.	OCL Features Not Included in GELLO.....	16
6.	GELLO .....	17
6.1.	GELLO Types.....	17
6.1.1.	Basic Types.....	17
6.1.1.1.	Boolean .....	17
6.1.1.2.	Integer .....	18
6.1.1.3.	Real .....	18

6.1.1.4.	String.....	18
6.1.1.5.	Unknown and Null as third Boolean value .....	18
6.1.1.6.	Basic Type Hierarchy and Type Conformance Rules.....	18
6.1.2.	Model Types .....	18
6.1.2.1.	Model Type Hierarchy.....	18
6.1.3.	Collection Types .....	18
6.1.3.1.	Collections of Collections.....	19
6.1.3.2.	Collection Type Hierarchy and Type Conformance Rules.....	19
6.1.4.	Tuple Type.....	19
6.2.	Names .....	19
6.3.	Properties .....	20
6.3.1.	Attributes.....	20
6.3.2.	Methods.....	20
6.3.2.1.	Method Parameters .....	20
6.4.	Variable Declaration .....	20
6.4.1.	Mechanisms for Declarations: Creating a Variable .....	20
6.4.2.	Assigning Values to Temporary Variables: the let Operator.....	21
6.4.3.	Scope of Declarations .....	21
6.4.4.	Creating New Instances of Classes .....	22
6.5.	Reflection.....	22
6.6.	Casting .....	22
6.7.	Built-in Operators .....	22
6.7.1.	Arithmetic Operators “+”, “-”, “*” .....	22
6.7.2.	Arithmetic Operator “/” .....	23
6.7.3.	Arithmetic Operators “div” and “mod” .....	23
6.7.4.	Arithmetic Operator unary minus “-” .....	23
6.7.5.	Comparison Operators “=”, “>”, “<”, “>=”, “<=”, “<>” .....	23
6.7.6.	Boolean Operators .....	24
6.7.7.	String Operators “size”, “concat”, “toUpper”, “toLower” “substring”, “=” and “<>” .....	25
6.8.	Collection Operators .....	26
6.8.1.	The ‘Arrow’ Notation .....	26
6.8.2.	Single Instances as Collections.....	26
6.8.3.	Operator Select.....	26
6.8.4.	Operator Reject .....	27
6.8.5.	Operator Collect.....	28
6.8.6.	Operator ForAll.....	29
6.8.7.	Operator Iterate .....	31
6.8.8.	Operator Exists.....	31
6.8.9.	Operator Flatten .....	32
6.8.10.	Operator Size .....	33
6.8.11.	Operator Count.....	33
6.8.12.	Operators “Max” and “Min” .....	33
6.8.13.	Operator Includes.....	34
6.8.14.	Operator IncludesAll.....	34
6.8.15.	Operator isEmpty .....	34

6.8.16.	Operator notEmpty.....	35
6.8.17.	Operator Sum.....	35
6.8.18.	Operator FirstN.....	35
6.8.19.	Operator LastN.....	36
6.8.20.	Operator Reverse.....	36
6.8.21.	Operator SortBy.....	36
6.8.22.	Operator Intersection.....	37
6.8.23.	Operator Union.....	37
6.8.24.	Operator Including.....	38
6.8.25.	Operator Excluding.....	38
6.8.26.	Operator Join.....	39
6.9.	Tuple Operators.....	41
6.9.1.	Operator Size.....	41
6.9.2.	Operator getValue.....	41
6.9.3.	Operator getElemName.....	41
6.9.4.	Operator getElemType.....	42
6.10.	Temporal Operators.....	42
6.11.	Precedence Rules.....	43
6.12.	If Expression.....	43
7.	GELLO Syntax.....	43
7.1.	Inferring Type Rules for Queries and Expressions.....	44
7.2.	GELLO Lexical Grammar.....	44
7.3.	GELLO BNF Syntax.....	44
7.3.1.	Root Symbol.....	45
7.3.2.	Literals.....	45
7.3.3.	Data Types.....	45
7.3.4.	Names.....	45
7.3.5.	Expressions.....	46
7.3.6.	Statements.....	46
7.3.7.	Creating a NEW Instance of a Model Class.....	47
7.3.8.	Queries.....	47
7.3.9.	Literals and Identifiers.....	49
7.3.10.	Reserved Words.....	49
7.3.11.	Operators.....	49
7.3.12.	Statements.....	50
8.	GELLO Queries.....	50
8.1.	Return Type of a Query.....	51
8.2.	Evaluation of Queries.....	51
8.3.	Examples of Queries.....	51
9.	GELLO Expressions.....	52
9.1.	Type of an Expression.....	52
9.2.	Normal and Abrupt Completion of Evaluation.....	52
9.2.1.	Type Checking.....	52
9.2.2.	Handling Exceptions.....	52
9.3.	Evaluation of Expressions.....	52
9.3.1.	Argument Lists.....	53

9.4.	Example of Expressions.....	53
10.	Examples in GELLO.....	53
10.1.	An MLM into GELLO.....	53
10.2.	Example of an iteration over more than one collection at a time .....	54
10.3.	Example: Presence of Azotemia Observation within last three months .....	54
10.4.	Example: Number of current anti-hypertensive Medications > 1.....	55
10.5.	3rd Td dose before 12 months of age.....	55
11.	The HL7 RIM data model.....	56
12.	Acknowledgements.....	57
13.	References.....	57

# 1. Introduction

GELLO is intended to be a standard query and 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.

This document presents the full specification of GELLO query and expression language, for consideration for balloting as a standard. The document addresses issues discussed at previous HL7 WG meeting, through the September, 2003, meeting in Memphis, TN. See prior documents and presentations: [[TC1](#), [TC2](#), [TC3](#), [TC4](#), [TC5](#), [TC6](#)]. An earlier BNF specification of the language is available at [[DSG02-01](#)].

:

Specific issues raised in the September, 2003, meeting relating to the following are addressed in this document:

- Built-in basic data types
- Built-in collection types
- Syntax of “.” and “->” operators
- Collection operators
- “->” notation for collection operators
- Declaration of temporary variables (including the “new()” operator)
- Assignment of values to variables
- Scope of variables
- “Xor” and “implies” operators
- Tuples as a built-in data type for aggregated data
- Tuple operators
- Need for a full specification

The syntax of the GELLO language depends on the use of an object-oriented data model. We refer to this as a “virtual medical record” (or vMR), as it is referred to in the CDSTC. The vMR is an R-MIM view of the HL7 RIM. 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. There can be many possible vMRs, and in fact the vMR model adopted by the CDSTC has not yet been decided upon.

Based on the premise that any properly defined vMR is a view of the HL7 RIM, it has been possible for us to independently develop GELLO regardless of any particular specification of the vMR. It is thus only necessary, when producing a set of decision support applications using GELLO to specify the particular vMR 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. It also defines a query mechanism, but this part of OCL is less well-developed than the constraint language features, and is still in evolution.

Since GELLO is intended to provide both query and expression support, and inasmuch as it is intended to use only standard features wherever possible, GELLO's query capabilities, being dependent on OCL's, are also therefore somewhat more limited than its expression language capabilities. Because of this, we had considered only balloting the expression language part of GELLO initially, but we believe that an expression language by itself would be too limited to be useful, in that it would not allow accessing data from a data source. However, it is now possible to iterate over more than one collection. Given the availability of this approach, we have decided to include the query definition as part of the initial specification document. Examples of GELLO use, including query with iteration over more than one collection, are provided in this document.

### **1.1. What is GELLO**

GELLO is a class-based, object-oriented (OO) language that is built on existing standards. GELLO includes both query and expression sublanguages, which we refer to collectively as the GELLO language (for succinctness we will also refer to the two sublanguages simply as languages). GELLO 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 query and expression languages 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 queries 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, and queries for other applications.

The query and expression languages share a common OO data model since the expressions must operate on data supplied by queries.

The query language (§8) has been designed in the context of a guideline execution model proposed in the HL7 CDSTC. This model proposes the use of a vMR that provides a standard interface to heterogeneous medical record systems. While the query language does not depend on the specific classes or tables in the vMR, it does rely on the general framework of the vMR.

The expression language (§9) can be used for 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 queries and expressions, GELLO includes basic built-in data types (§6.1), while providing the necessary mechanisms to access an underlying data model with all its 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 is the full software specification for GELLO expression and query languages and is organized as follows:

Section 2 describes the requirements for query and expression languages 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 lists OCL features not included in the GELLO language specification

Section 6 describes OCL features included in GELLO query and expression languages, including basic data types in §6.1.1, model types in §6.1.2 (classes in the data model), collection types in §6.1.3, properties of model types (attributes and methods) in §6.3, and variables in §6.4. Variables are created as instances of classes defined in the HL7 data model. Variables are strongly typed temporal storage locations (see §6.4.2) with a predefined, limited scope (see §6.4.3). In order to preserve GELLO as a side-effect-free language, the creation of variables as instances of classes is delegated to the underlying data model; hence such classes should provide the appropriate mechanisms. This section also describes GELLO built-in operators (see §6.7), and their syntax and semantics, and discusses the tuple type as an aggregation type (see §6.1.4) and tuple operators (see §6.9) supported by GELLO.

Section 7 describes the syntax of the GELLO grammar. Section 8 describes queries in GELLO, and how they are used to retrieve information from a vMR. Similarly, section 9 describes GELLO expressions used to build decision criteria, perform abstraction or derive summary values. Section 10 presents examples of GELLO queries and expressions. Finally, section 11 contains a diagram of the HL7 RIM data types for reference.

## **2. Requirements for Query and Expression Languages in the Clinical Context**

A major obstacle to sharing clinical knowledge is the lack of a common format for data encoding and manipulation. Although the Arden Syntax addressed this problem by isolating references to local data in curly braces [{" "}] in MLMs, it still does not provide the mechanisms for accessing data in a truly format-independent way.

## **2.1. Underlying Data Model**

The “[virtual medical record](#)” (vMR), an object-oriented approach compatible with the [HL7 RIM](#) (see also section 11), uses a standard data model as an intermediary to heterogeneous medical record systems. The concept of a [vMR](#) has been proposed as an underlying model for handling patient data in the context of decision-support systems. A vMR is a refinement of the Reference Information Model (RIM).

Although this approach represents a paradigm shift in data representation, moving from time-stamped atomic data types to an object-oriented data model, it provides a first approach towards a standard for exchange, management and integration of clinical data. However, the OO approach of the vMR is incompatible with the Arden Syntax, since the latter can only handle atomic data types.

## **2.2. Queries, Expressions and Clinical Guidelines**

The need for a language to formulate queries and expressions to extract and manipulate clinical data is clear. Ideally such a language should be:

- vendor-independent<sup>1</sup>
- platform-independent<sup>1</sup>
- object-oriented and compatible with the vMR
- easy to read/write<sup>1</sup>
- side-effect free<sup>1</sup>
- flexible
- extensible

The following section describes how GELLO complies with the above requirements and provides the mechanisms for handling OO clinical data stored in a standardized data model such as the vMR.

## **3. GELLO: Goals and Properties**

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

- GELLO is targeted to clinical applications that need to use queries and expression languages for decision support.
- GELLO is vendor-independent by relying on a language specification that is not vendor-specific.
- GELLO is platform-independent in that the language can be implemented on various platforms.

---

<sup>1</sup> As discussed in HL7 Clinical Decision Support Technical Committee and Clinical Guideline SIG Working Group meeting in San Diego, CA, January 9-11, 2002.

- GELLO provides the mechanisms to access data through an OO data model, with strongly-typed expressions, via general purpose query and expression languages.
- GELLO is a declarative language. Its queries and expressions have no side effects.
- GELLO is extensible by adding new user-defined classes to the underlying OO data model.
- All data manipulation methods must be explicitly defined in the OO data model. The purpose of GELLO is to provide a robust syntax for queries and expressions so data can be easily handled.
- By using a specified OO data model, such as the vMR, each decision rule or guideline need not provide a separate mechanism for translation of data elements to/from host environments (e.g., the “curly braces” needed in the Arden Syntax data section).
- The object-oriented approach allows modularity, encapsulation and extensibility.

Thus use of GELLO:

- provides platform-independent support for mapping to the OO data model used (e.g., the vMR). Therefore it eliminates the need for curly braces or other implementation-specific encoding methods for information retrieval as part of knowledge content (guidelines, alerts, etc.).
- simplifies the creation and updating of clinical data objects, and their evaluation.
- facilitates sharing of decision logic and other computer expressions.

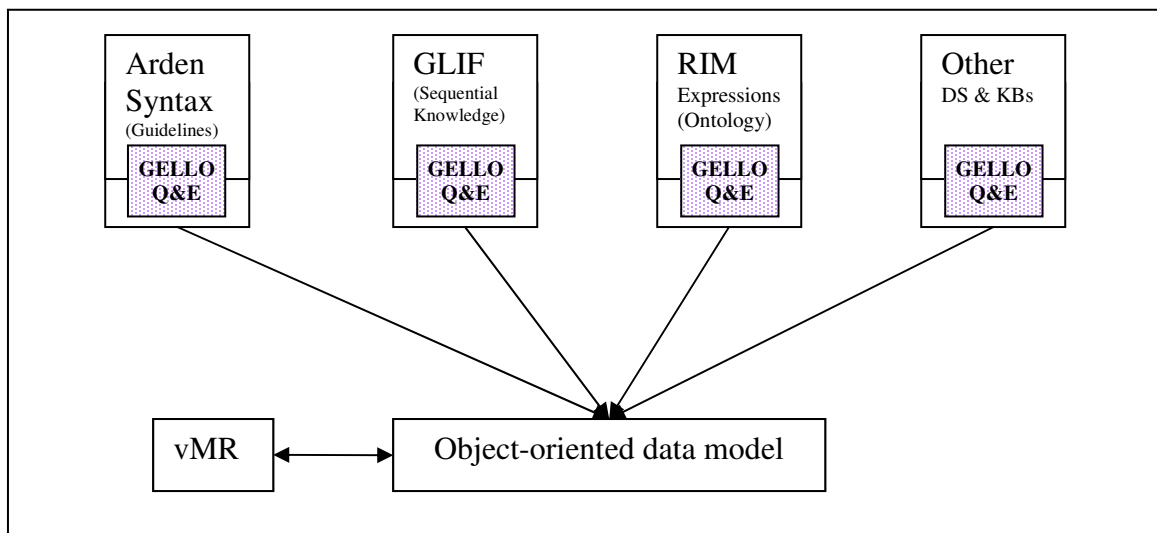


Figure 1: GELLO and its relation to Arden Syntax, GLIF, RIM and other DSs and KBs. GELLO query and expression languages can be embedded into various tools to provide the mechanisms for access and manipulation of OO data.

## 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<sup>1</sup>
- platform-independent<sup>1</sup>
- object-oriented and compatible with the vMR
- easy to read/write<sup>1</sup>
- side-effect free<sup>1</sup>
- 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 data 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 implement a language to publish and share sets of terms called ontologies. Still under development, OWL is intended to provide a language that can be used to describe the classes and inherent relations between them in Web documents and applications. OWL can be used in applications that need to understand the content of information rather than providing a human readable representation of the content. In other words, OWL is more focused on the semantics of the language than on the readability of the syntax. This fact makes writing and reading an OWL document an arduous task ([click here to see an OWL](#)

---

<sup>1</sup> As discussed in HL7 Clinical Decision Support Technical Committee and Clinical Guideline SIG Working Group meeting in San Diego, CA, January 9 -11, 2002.

[ontology](#)). A list of current W3C recommendations and technical documents can be found [at the W3C website](#). Given that OWL has focused on Web applications and the semantics of Web documents, and its syntax lacks readability, we did not consider OWL as an appropriate approach for an ‘easy to read and write’ object-oriented query and expression language.

We also considered Java as an option for a query and expression language. Java meets most of the requirements in §2. Java is vendor-independent and platform-independent, is object-oriented and compatible with the vMR, is relatively easy to read and write, and is flexible and extensible. However, Java is a full-fledged procedural programming language with side-effects.

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.

## **4.2. OCL Language Description**

This section briefly describes the elements of the OCL language. A complete description can be found [at the OMG website](#).

### **4.2.1. Relation to the UML Metamodel**

OCL has been developed to add precision to the diagrammatic elements of UML. OCL allows modelers to express all the relevant aspects of a specification that just UML diagrams cannot represent.

#### **4.2.1.1. Self**

The reserved word *Self* is used to refer to a contextual instance of a specific type. It assumes there is a previously specified context.

#### **4.2.1.2. Specifying the UML context**

The context of an OCL expression can be specified through a context declaration at the beginning of an OCL expression.

#### **4.2.1.3. Invariants**

An invariant is a constraint associated to a classifier (a class in the data model). An invariant must be true for all the instances of the class the invariant refers to. Invariants are defined in the context of the classifier to which they apply.

#### **4.2.1.4. Pre- and Post- conditions**

Pre- and post- conditions are OCL constraints associated with an operation or method. As with invariants, pre- and postconditions are defined in the context of the classifier to which they refer to. Pre- and post- conditions evaluate an object in the predefined context.

#### 4.2.1.5. Package Context

An UML data model can be divided into packages containing classes that need to be grouped together.

#### 4.2.2. Basic Values and types

OCL basic data types are predefined and independent of any data model. OCL basic data types are: Boolean, integer, real and string.

##### 4.2.2.1. Types from the UML Model

All classifiers (classes) in the UML are data types in OCL.

##### 4.2.2.2. Enumeration Types

An enumeration is a datatype that defines a number of possible values such an enumeration can have.

##### 4.2.2.3. Let Expressions and «definition» Constraints

In OCL, the let expression allows the definition of an attribute or operation that can be used in a constraint. Let expressions are only known within the specification of the constraint. However, if a let variable or operation is defined within a «definition» constraint attached to a classifier (a class in the data model), it will be known in the same context where a property of the classifier can be used.

##### 4.2.2.4. Type Conformance

OCL is a typed language. All types in an OCL expression must conform to either a basic type or a classifier in the UML model.

##### 4.2.2.5. Casting

OCL objects with a type *type1* can be re-typed into *type2* if *type2* is a subtype of *type1*.

##### 4.2.2.6. Precedence Rules

OCL has a predefined precedence order for operators (see page 2-8 in [OCL](#)).

##### 4.2.2.7. Infix Operators

The use of infix operators is allowed in OCL. The operators '+', '-', '\*', '/', '<', '>', '<>', '<=' '>=' can be used as infix operators. If a type defines one of those operators with the correct signature, they will be used as infix operators. The expression: a + b is conceptually equal to the expression: a.+(b) that is, invoking the '+' operation on a with b as the parameter to the operation. The infix operators defined for a type must have exactly one parameter. For the infix operators '<', '>', '<=', '>=', '<>', 'and', 'or', and 'xor' the return type must be Boolean.

##### 4.2.2.8. Keywords

Keywords in OCL are treated as reserved words. Reserved words cannot occur anywhere in an OCL expression.

#### 4.2.2.9. Undefined Values

Some expressions will, when evaluated, have an undefined value. For instance, typecasting with `oclAsType()` to a type that the object does not support or getting the first element of an empty collection will result in undefined. In general, an expression where one of the parts is undefined will itself be undefined. There are some important exceptions to this rule, however. First, there are the logical operators:

- True OR-ed with anything is True
- False AND-ed with anything is False
- False IMPLIES anything is True
- anything IMPLIES True is True

The rules for OR and AND are valid irrespective of the order of the arguments and they are valid whether the value of the other sub-expression is known or not.

The IF-expression is another exception. It will be valid as long as the chosen branch is valid, irrespective of the value of the other branch.

Finally, there is an explicit operation for testing if the value of an expression is undefined. `oclIsUndefined()` is an operation on `OclAny` that results in True if its argument is undefined and False otherwise (see OCL 2.0 Section 7).

#### 4.2.3. Objects and Properties

OCL expressions can refer to classifiers and datatypes. Also, all attributes, association-ends, methods and side-effect free operations attached to such classifiers can be used. A property of an object under OCL can be:

- an attribute
- an association end
- a side-effect free operation
- a side-effect free method

##### 4.2.3.1. Reference to Object Properties

The notation to access a property of an object is the name of the object followed by a dot and the name of the property: *object.property*

- **Attributes.** The value and type of an expression that refers to an attribute of an object are the type and value of the attribute such an expression refers to.
- **Operations and Methods.** Operations and methods are operations associated to a class. Such operations may have parameters that must be included in the operation or method call. For example, *aPerson.income(aDate)* *aDate* is passed as parameter to the method *income* associated to the object *aPerson*. The type of the result is the return type of the operation.
- **Association-ends and Navigation.** Starting from a specific object, it is possible to refer to other associated objects and their properties using association-ends.

### 4.2.3.2. Combining Properties

Properties can be combined into complex expressions. All OCL expressions evaluate to a specific type, so after obtaining a result, such result can be used in another property. OCL evaluates expressions from left to right.

### 4.2.3.3. Pathnames for Packages

OCL provides the notation for referring to types organized in packages by using a package-pathname prefix.

### 4.2.3.4. Predefined Properties on all Objects

OCL defines several properties that apply to all objects:

- `oclIsTypeOf(t: oclType): Boolean`
- `oclIsKindOf(t: oclType): Boolean`
- `oclInState(s: oclState): Boolean`
- `oclIsNew(): Boolean`
- `oclAsType(t: oclType): Boolean`
- `oclIsUndefined(t: oclType): Boolean`

### 4.2.3.5. Features on Classes Themselves

In OCL is possible to use features defined on the classes and datatypes themselves defined in the class model.

## 4.2.4. Collections and Collection Operators

### 4.2.4.1. Collections

Collection is a predefined type in OCL with a large number of operations to manipulate them. Operations applied on collections never change the original collection; the result of an operation is another collection. Collection is an abstract type with three concrete subtypes: Set, Sequence and Bag.

- A Set is a mathematical set with no duplicates elements.
- A Bag is like a Set with duplicates.
- A Sequence is like a Bag but the elements are ordered.

### 4.2.4.2. 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. Table 1 shows all possibilities for combining *Set*, *Bag*, and *Sequence* into a nested collection type. For each of the different cases, the collection type resulting from flattening is shown in the right column. The element type *t* can be any type. In particular, if *t* is also a collection type the indicated rules for flattening can be applied recursively until the element type of the result is a non-collection type.

Nested collection type	Type after flattening
<i>Set(Sequence(t))</i>	<i>Set(t)</i>
<i>Set(Set(t))</i>	<i>Set(t)</i>
<i>Set(Bag(t))</i>	<i>Set(t)</i>
<i>Bag(Sequence(t))</i>	<i>Bag(t)</i>
<i>Bag(Set(t))</i>	<i>Bag(t)</i>
<i>Bag(Bag(t))</i>	<i>Bag(t)</i>
<i>Sequence(Sequence(t))</i>	<i>Sequence(t)</i>
<i>Sequence(Set(t))</i>	<i>Sequence(t)</i>
<i>Sequence(Bag(t))</i>	<i>Sequence(t)</i>

**Table 1: Flattening of nested collections**

#### 4.2.4.3. Collection Operators

OCL provides operations to project new collections over existing ones.

- **Select and reject** are constructs to specify a selection from a specific collection. *Select* returns all the elements in a collection that satisfy a criterion (all elements that evaluate to *True*) whereas *reject* returns all the elements that do not satisfy such conditions (all elements that evaluate to *False*).
- **Collect** is a construct used to specify a collection derived from other collection, with elements that are different from the original collection.
- **ForAll** is used to specify a Boolean expression that must hold true for all the elements of the resulting collection.
- **Exists** is used to specify a Boolean expression with must be true for at least one element in a collection.
- **Iterate** is a generic operation that checks each element in a collection to see whether such element satisfies an expression.
- **Flatten** converts nested collections into a single ‘flat’ collection.
- **Resolving Properties.** References to properties of objects must be defined using a full notation including the object itself. If *self* is used, then the reference to such an object is implicit.

## 5. OCL Features Not Included in GELLO

As discussed above, GELLO is a subset of OCL. The following OCL features are not included in GELLO:

- Self and implicit references to objects
- Context declaration
- Invariants
- Pre- and postconditions
- Package context
- Enumeration types
- «definition» constraints (although let is included)

- Pathnames for packages
- Associations and aggregations

## 6. GELLO

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

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 §6.8.18, lastN §6.8.19, and join §6.8.26). As for data types, all GELLO built-in data types are part of OCL including the Tuple data type §6.1.4. In this section, the use of HL7 classes as a data model is discussed. All basic data types are provided by GELLO (§6.1), while model types –or classes- are defined in the underlying HL7 data model (§6.1.2) (the abstract specification for OCL data types can be found [at the OMG website](#)). Variables are declared as instances of classes defined in the HL7 data model. Variables (§6.4) are strongly typed temporal storage locations with a predefined, limited scope (§6.4.3). In order to preserve GELLO as a side-effect free language, the creation of variables as instances of HL7 classes is delegated to the HL7 data model, hence classes in the data model should provide the appropriate mechanisms for creating instances.

### 6.1. GELLO Types

GELLO is a strongly-typed language. This means that every expression must have a known type. There are two type categories: basic types and model types. Each and every GELLO query and expression has a type that must conform to either the basic or model types.

#### 6.1.1. Basic Types

OCL basic or primitive data types are part of the GELLO grammar. 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

##### 6.1.1.1. 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 §6.7.6.

### 6.1.1.2. Integer

Integer represents the mathematical natural numbers. Integer is a subtype of real. Arithmetic operators are described in §6.7.

### 6.1.1.3. Real

Real represents the mathematical concept of real values. Arithmetic operators are described in §6.7.

### 6.1.1.4. String

Strings are sequences of characters. Literal strings are enclosed with single quotes. The available operations for strings are described in §6.7.7.

### 6.1.1.5. Unknown and Null as third Boolean value

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

### 6.1.1.6. Basic Type Hierarchy and Type Conformance Rules

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

## 6.1.2. Model Types

Model types refer to user-defined classes in the underlying data model (e.g. vMR) or any other data model. References to such types should include a full description name, e.g. *HL7.PhysicalQuantity* refers to the model type PhysicalQuantity defined as an HL7 data type.

### 6.1.2.1. 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.

## 6.1.3. 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:

- `typeOfCollection {element1, ..., elementn}`,

where

- `typeOfCollection` is one of Set, Bag, Sequence; and  $\{\text{element}_1, \dots, \text{element}_n\}$  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 §6.8 for collection operators.

### **6.1.3.1. 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.

### **6.1.3.2. Collection Type Hierarchy and Type Conformance Rules**

Set, Bag and Sequence are all subtypes of Collection.

### **6.1.4. 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 and a type. 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::value1, ..., labeln: valuen}`, Where label<sub>i</sub> is the label of the element *ith* and value<sub>i</sub> is a valid value –there is a valid data type (basic or model) that can hold such value.

Tuples can be used as a return type from queries that retrieve information from more than one source such as joins (§6.8.26). GELLO provides some tuple operators to access the information returned by a query. These operators are described in §6.9.

## **6.2. Names**

Names refer to declared entities in an expression. A declared entity is a local variable, a class type, a basic type, or a parameter in a method call. Names can consist of a single identifier, or a sequence of identifiers separated by “.”.

Names may be used in expressions and queries. If the name of an attribute or a method appears in an expression, it requires full invocation –the class it belongs to must be explicitly referred to.

## 6.3. Properties

A property of a class can be an attribute (§6.3.1) or a method (§6.3.2). The syntax for referring to a property of a class is: *Class.property*, which is consistent with the ‘dot’ notation of an OO data model.

### 6.3.1. Attributes

Attributes are properties of an object defined in a class of the data model. For example, a class *Person* can have an attribute *FirstName*. Hence, we refer to a person’s name by writing *Person.FirstName*, where *Person* is the class in the data model and *FirstName* is an attribute of that class.

### 6.3.2. Methods

A method declares executable code that can be invoked by passing a fixed number of values as arguments. The name of a method may appear in a GELLO expression only as a part of a full method invocation expression. That is, along with the name of the method and its arguments, the class or object it belongs to must be explicitly referred to. Methods are side effect-free, they do not change the state of any object.

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

#### 6.3.2.1. Method Parameters

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

## 6.4. Variable Declaration

A variable declaration declares a variable name and binds it to a type. The type of the variable can be a basic type (§6.1.1) or a model type (§6.1.2). In either case, 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 (§6.4.3).

### 6.4.1. Mechanisms for Declarations: Creating a Variable

In GELLO, variables can be declared using the *let* OCL expression. The syntax is as follows: *let varName: type = initExpression*. Where *initExpression* is the initial value of the variable. If the variable is of model type, the *initExpression* must include the *new()* operator as in: *Class.new(parameters)*, where *Class* is a class in the data model, *new* is the associated method for creating a new instance of the class *Class*, and *parameters* are the values for the class attributes, separated by commas.

### 6.4.2. Assigning Values to Temporary Variables: the let Operator

In OCL, the *let* expression allows the definition of an attribute or operation that can be used in a constraint. GELLO does not include constraints. Instead, the *let* operator assigns a value to a variable. The type of the variable can be either one of the basic built-in GELLO data types §6.1.1, GELLO collection types §6.1.3, GELLO tuple type §6.1.4, or a class from the underlying data model §6.1.2. If the type of the return value is a class in the data model, an instance of such a class must be created. See (§6.4.1 and §6.4.4). 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. The syntax for let expressions is as follows:

*let variable : type = value*

Where *variable* is a variable with type *type*, and *value* is the returning value of a GELLO expression. For example:

```
let threshold_for_osteodystrophy : integer = 70 ... (1)
let potassium : PhysicalQuantity = PhysicalQuantity.new(70, 'dl') ... (2)
let observations: set = observation→select(coded_concept= 'abc') ... (3)
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 an instance of that class before assigning it to the variable.

In (3) *observations* 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 §6.8.26 for the full notation of the join operator.

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

*let variable : type = value returned by an expression or query*

Values assigned to variables cannot be changed. Once the value is assigned it remains the same for the scope of the variable.

### 6.4.3. 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 must be defined separately by each standard. For example, Arden Syntax can define the scope of the variable declared in the data slot to be the entire MLM, while a variable declared in the logic slot, only has scope within that slot.



- (integer×real) → real
- (real×integer) → real
- (real×real) → real

### 6.7.2. Arithmetic Operator “/”

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

$$F_{/}(V_1, V_2) = V_1 / V_2 \quad \text{If } V_1 \text{ and } V_2 \text{ are either integers or reals, and } V_2 \neq 0$$

$$= \text{undefined} \quad \text{otherwise}$$

Types for “/”:

- (integer×integer) → real
- (integer×real) → real
- (real×integer) → real
- (real×real) → real

### 6.7.3. 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.

$$F_{\text{div}}(V_1, V_2) = V_1 \text{ div } V_2 \quad \text{If } V_1 \text{ and } V_2 \text{ are both integers and } V_2 \neq 0$$

$$= \text{undefined} \quad \text{otherwise}$$

Type “div” and “mod”:

- (integer×integer) → integer

### 6.7.4. Arithmetic Operator unary minus “-”

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

$$F_{-}(V_1) = -V_1 \quad \text{If } V_1 \text{ is either an integer or a real}$$

$$= \text{undefined} \quad \text{otherwise}$$

Types for unary minus “-”:

- (integer) → integer
- (real) → real

### 6.7.5. Comparison Operators “=”, “>”, “<”, “>=”, “<=”, “<>”

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*.

Types for “=”, “>”, “<”, “>=”, “<=”, “<>”:

- (real×real) → truth\_value
- (real×integer) → truth\_value
- (integer×real) → truth\_value
- (integer×integer) → truth\_value

- (string×string)→truth\_value Only valid for “=” and “<>”
- (boolean×boolean)→truth\_value Only valid for “=” and “<>”

Definition of the evaluation function  $F_{>}(V_1, V_2)$ . The evaluation function is the same for the other operators:

$F_{>}(V_1, V_2) = \text{true}$                       If  $V_1$  and  $V_2$  are both either integers or reals and  $V_1 > V_2$   
                                                   = false                                      Else if  $V_1$  and  $V_2$  are both either integers or reals and  $V_1 \leq V_2$   
                                                   = undefined                                  otherwise

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

Definition of the evaluation function  $F_{=}(V_1, V_2)$ . The evaluation function is the same for  $F_{<>}(V_1, V_2)$  when the operators are both strings of Booleans.

$F_{=}(V_1, V_2) = \text{true}$                       If  $V_1$  and  $V_2$  are both strings or Booleans and  $V_1 = V_2$   
                                                   = false                                      Else if  $V_1$  and  $V_2$  are both strings or Booleans and  $V_1 \neq V_2$   
                                                   = undefined                                  otherwise

### 6.7.6. Boolean Operators

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

Types for “and”, “or”, “xor” and “implies”:

- (truth\_value×truth\_value) → truth\_value

Types for “not”:

- (truth\_value) → truth\_value

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

$V_1$	$V_2$	$V_1$ and $V_2$	$V_1$ or $V_2$	$V_1$ xor $V_2$	not $V_1$	$V_1$ implies $V_2$
false	false	false	false	false	true	true
false	true	false	true	true	true	true
true	false	false	true	true	false	false
true	true	true	true	false	false	true
false	unknown	false	unknown	unknown	true	true
true	unknown	unknown	true	unknown	false	unknown
unknown	false	false	unknown	unknown	unknown	unknown
unknown	true	unknown	true	unknown	unknown	true
unknown	unknown	unknown	unknown	unknown	unknown	unknown

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.

### 6.7.7. String Operators “size”, “concat”, “toUpper”, “toLower”, “substring”, “=” and “<>”

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

#### Type for “size”

- (string)→integer

Definition of evaluation function for  $F_{\text{size}}(V)$

$F_{\text{size}}(V) = \text{integer}$       If  $V$  is a string  
 $= \text{undefined}$       otherwise

#### Type for “concat”

- (string×string)→string

Definition of evaluation function for  $F_{\text{concat}}(V_1, V_2)$

$F_{\text{concat}}(V_1, V_2) = \text{string}$       If  $V_1$  and  $V_2$  are both strings  
 $= \text{undefined}$       otherwise

#### Type for “toUpper”, “toLower”

- (string)→string

Definition of evaluation function for  $F_{\text{toUpper}}(V)$ . The same applied for “toLower” operator.

$F_{\text{toUpper}}(V) = \text{string all in upper characters}$       If  $V$  is a string  
 $= \text{undefined}$       otherwise

#### Type for “substring”

- (string×integer×integer)→string

Definition of evaluation function for  $F_{\text{substring}}(V_1, V_2, V_3)$

$F_{\text{substring}}(V_1, V_2, V_3) = \text{returns a substring of length } V_3, \text{ at a given starting point } V_2.$

- $V_1$  is a string and  $V_2$  and  $V_3$  are integers and
- $0 \leq V_2 < \text{size}(V_1)$  and
- $0 \leq V_3 \leq \text{size}(V_1)$  and
- $V_2 + V_3 \leq \text{size}(V_1)$

$= \text{undefined}$       otherwise

#### Type for “=” and “<>”

- (string×string)→truth\_value

Definition of the evaluation function  $F_{=} (V_1, V_2)$ . The evaluation function is the same for  $F_{<>} (V_1, V_2)$  when the operands are both strings.

$F=(V_1,V_2)= \text{true}$	If $V_1$ and $V_2$ are both strings and $V_1 = V_2$
$= \text{false}$	Else if $V_1$ and $V_2$ are both strings and $V_1 \neq V_2$
$= \text{undefined}$	otherwise

## 6.8. 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* (§6.8.3), *reject* (§6.8.4), *collect* (§6.8.5), *forAll* (§6.8.6), *iterate* (§6.8.7), *exists* (§6.8.8) and *flatten*(§6.8.9).

### 6.8.1. The ‘Arrow’ Notation

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

$$\text{collection} \rightarrow \text{collectionOperator}(\text{parameters})$$

### 6.8.2. 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 (§6.8.1):

$$\text{singleInstance} \rightarrow \text{collectionOperator}(\text{parameters})$$

### 6.8.3. Operator Select

*Select* is a construct 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:

$$\text{collection} \rightarrow \text{select}(\text{boolean-expression})$$

This results in a collection that contains all the elements from *collection* for which the *boolean-expression* evaluates to true. To find the result of this expression, for each element in *collection* the expression *boolean-expression* is evaluated. The Boolean expression evaluates over a property of the elements in the collection. A more general syntax for the select expression:

$$\text{collection} \rightarrow \text{select}(v \mid \text{boolean-expression-with-}v)$$

The variable  $v$  is called the iterator. When the select is evaluated,  $v$  iterates over the *collection* and the *boolean-expression-with- $v$*  is evaluated for each  $v$ . The  $v$  is a reference to the object from the collection and can be used to refer to the objects themselves from the *collection*. The third form is an extension of the latest, where the type of the variable

$v$  can be specified. Hence, requiring the objects in *collection* to be of type *Type*. The notation is:

collection->select(  $v : \text{Type} \mid \text{boolean-expression-with-}v$  )

The notation is:

- collection→select(BooleanExpression)
- collection->select(  $v \mid \text{boolean-expression-with-}v$  )
- collection->select(  $v : \text{Type} \mid \text{boolean-expression-with-}v$  )

Type for “select”

- (collection×Boolean Expression)→collection

Definition of evaluation function for  $F_{\text{select}}(V,E)$

$F_{\text{select}}(V,E) = \text{collection}$     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 Boolean expression  $E$  evaluates to true  
 = undefined    otherwise

- collection→select( $v \mid \text{BooleanExpression-with-}v$ )

Type for “select”

- (collection×Iterator×Boolean Expression)→collection

Definition of evaluation function for  $F_{\text{select}}(V,I,E)$

$F_{\text{select}}(V,I,E) = \text{collection}$     If  $V$  is a collection,  $I$  is the iterator referring to the object from the collection and  $E$  is a valid GELLO Boolean expression. The resulting collection contains all the elements from  $V$  for which the Boolean expression  $E$  evaluates to true  
 = undefined    otherwise

- collection→select( $v:\text{Type} \mid \text{BooleanExpression-with-}v$ )

Type for “select”

- (collection×Iterator×Type×Boolean Expression)→collection

Definition of evaluation function for  $F_{\text{select}}(V,I,T,E)$

$F_{\text{select}}(V,I,E) = \text{collection}$     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 Boolean expression. The resulting collection contains all the elements from  $V$  for which the Boolean expression  $E$  evaluates to true  
 = undefined    otherwise

#### 6.8.4. Operator Reject

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

The notation is:

- $\text{collection} \rightarrow \text{reject}(\text{BooleanExpression})$
- $\text{collection} \rightarrow \text{reject}(v : \text{Type} \mid \text{boolean-expression-with-}v)$
- $\text{collection} \rightarrow \text{reject}(v \mid \text{boolean-expression-with-}v)$

Type for “reject”

- $(\text{collection} \times \text{Boolean Expression}) \rightarrow \text{collection}$

Definition of evaluation function for  $F_{\text{reject}}(V, E)$

$F_{\text{reject}}(V, E) = \text{collection}$      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 Boolean expression  $E$  evaluates to false  
= undefined     otherwise

- $\text{collection} \rightarrow \text{reject}(v \mid \text{BooleanExpression-with-}v)$

Type for “reject”

- $(\text{collection} \times \text{Iterator} \times \text{Boolean Expression}) \rightarrow \text{collection}$

Definition of evaluation function for  $F_{\text{reject}}(V, I, E)$

$F_{\text{reject}}(V, I, E) = \text{collection}$      If  $V$  is a collection,  $I$  is the iterator referring to the object from the collection and  $E$  is a valid GELLO Boolean expression. The resulting collection contains all the elements from  $V$  for which the Boolean expression  $E$  evaluates to false  
= undefined     otherwise

- $\text{collection} \rightarrow \text{select}(v : \text{Type} \mid \text{BooleanExpression-with-}v)$

Type for “reject”

- $(\text{collection} \times \text{Iterator} \times \text{Type} \times \text{Boolean Expression}) \rightarrow \text{collection}$

Definition of evaluation function for  $F_{\text{reject}}(V, I, T, E)$

$F_{\text{reject}}(V, I, T, E) = \text{collection}$      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 Boolean expression. The resulting collection contains all the elements from  $V$  for which the Boolean expression  $E$  evaluates to false  
= undefined     otherwise

### 6.8.5. Operator Collect

*Collect* is a construct to derive a collection from other collection, containing objects different from the original collection. In other words, collect returns a collection of elements. These elements are a collection of a particular property of the current collection of objects. The collect operation uses the same syntax as the select and reject operators.

The notation is:

- $\text{Collection} \rightarrow \text{collect}(\text{ExpressionWithPropertyName})$

- collection->collect( v | expression-with-v )
- collection->collect( v : Type | expression-with-v )

Type for “collect”

- (set×ExpressionWithPropertyName)→bag
- (bag×ExpressionWithPropertyName)→bag
- (sequence×ExpressionWithPropertyName)→sequence

Definition of evaluation function for  $F_{\text{collect}}(V,E)$

$F_{\text{collect}}(V,E)$ = collection    If  $V$  is a collection and  $E$  is a conditional expression with the name of a property. The returning collection contains the elements whose property satisfied the condition specified in  $E$ .  
 = undefined    otherwise

Type for “collect”

- (set×Iterator×Boolean Expression)→bag
- (bag×Iterator×Boolean Expression)→bag
- (sequence×Iterator×Boolean Expression)→sequence

Definition of evaluation function for  $F_{\text{collect}}(V,I,E)$

$F_{\text{collect}}(V,I,E)$ = collection    If  $V$  is a collection,  $I$  is the iterator referring to the object from the collection and  $E$  is a valid GELLO Boolean expression. The resulting collection contains the elements whose property satisfied the condition specified in  $E$   
 = undefined    otherwise

- collection→collect(v:Type | BooleanExpression-with-v)

Type for “collect”

- (set×Iterator×Type×Boolean Expression)→bag
- (bag×Iterator×Type×Boolean Expression)→bag
- (sequence×Iterator×Type×Boolean Expression)→sequence

Definition of evaluation function for  $F_{\text{collect}}(V,I,T,E)$

$F_{\text{collect}}(V,I,E)$ = 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 Boolean expression. The resulting collection contains the elements whose property satisfied the condition specified in  $E$   
 = undefined    otherwise

### 6.8.6. 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:

- $\text{collection} \rightarrow \text{forall}(\text{Boolean Expression})$
- $\text{collection} \rightarrow \text{forall}(v \mid \text{boolean-expression-with-}v)$
- $\text{collection} \rightarrow \text{forall}(v : \text{Type} \mid \text{boolean-expression-with-}v)$

Type for “forall”

- $(\text{collection} \times \text{Boolean Expression}) \rightarrow \text{truth\_value}$

Definition of evaluation function for  $F_{\text{forall}}(V, E)$

$F_{\text{forall}}(V, E) = \text{true}$       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

$= \text{false}$                       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 that does not satisfy this expression

$= \text{undefined}$                 otherwise

- $\text{collection} \rightarrow \text{forall}(v \mid \text{BooleanExpression-with-}v)$

Type for “forall”

- $(\text{collection} \times \text{Iterator} \times \text{Boolean Expression}) \rightarrow \text{truth\_value}$

Definition of evaluation function for  $F_{\text{forall}}(V, I, E)$

$F_{\text{forall}}(V, I, E) = \text{true}$       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.

$= \text{false}$                       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 there is at least one element that does not satisfy this expression

$= \text{undefined}$                 otherwise

- $\text{collection} \rightarrow \text{forall}(v : \text{Type} \mid \text{BooleanExpression-with-}v)$

Type for “forall”

- $(\text{collection} \times \text{Iterator} \times \text{Type} \times \text{Boolean Expression}) \rightarrow \text{truth\_value}$

Definition of evaluation function for  $F_{\text{forall}}(V, I, T, E)$

$F_{\text{forall}}(V, I, T, E) = \text{true}$       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 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  $I$  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 that does not satisfy this expression  
 = undefined otherwise

### 6.8.7. Operator Iterate

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

The notation is:

- $\text{collection} \rightarrow \text{iterate}(\text{elem: modelType; result: collectionType} = \text{expression} \mid \text{expression-with-  
elem-and-result})$

Type for “iterate”

- $(\text{collection} \times \text{elementName} \times \text{elementType} \times \text{resultName} \times \text{collectionType} \times \text{expression} \times \text{expression-with-  
elem-and-result}) \rightarrow \text{collection}$

Definition of evaluation function for  $F_{\text{iterate}}(V,S,T1,R,T2,E1,E2)$

$F_{\text{iterate}}(V,S,T1,R,T2,E1,E2) = \text{collection}$  If  $V$  is a collection,  $S$  is a string with the name of an element with model type  $T1$ ,  $R$  is a string with the name of the returning result with a collection type  $T2$ ,  $E1$  is a valid GELLO expression defining the initial value of  $R$  and  $E2$  is a valid GELLO expression that evaluates all the elements  $E1$  in the collection. The return value of  $E2$  is stored in  $R$ .  
 = undefined otherwise

### 6.8.8. Operator Exists

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

The notation is:

- $\text{collection} \rightarrow \text{exists}(\text{BooleanExpression})$
- $\text{collection} \rightarrow \text{exists}(v \mid \text{boolean-expression-with-  
v})$
- $\text{collection} \rightarrow \text{exists}(v : \text{Type} \mid \text{boolean-expression-with-  
v})$

Type for “exists”

- $(\text{collection} \times \text{BooleanExpression}) \rightarrow \text{truth\_value}$

Definition of evaluation function for  $F_{\text{exists}}(V,E)$

$F_{\text{exists}}(V,E) = \text{true}$       If  $V$  is a collection  $E$  is a valid GELLO Boolean expression containing a property of the elements, and there is at least one element in the collection  $V$  that satisfies  $E$

$= \text{false}$               If  $V$  is a collection  $E$  is a valid GELLO Boolean expression containing a property of the elements, and none of the elements in the collection  $V$  satisfies  $E$

$= \text{undefined}$         otherwise

Type for “exists”

- $(\text{collection} \times \text{Iterator} \times \text{Boolean Expression}) \rightarrow \text{truth\_value}$

Definition of evaluation function for  $F_{\text{exists}}(V,I,E)$

$F_{\text{exists}}(V,I,E) = \text{true}$       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 there is at least one element in the collection  $V$  that satisfies  $E$

$= \text{false}$                   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 none of the elements in the collection  $V$  satisfies  $E$

$= \text{undefined}$         otherwise

- $\text{collection} \rightarrow \text{forall}(v:\text{Type} \mid \text{BooleanExpression-with-}v)$

Type for “exists”

- $(\text{collection} \times \text{Iterator} \times \text{Type} \times \text{Boolean Expression}) \rightarrow \text{truth\_value}$

Definition of evaluation function for  $F_{\text{exists}}(V,I,T,E)$

$F_{\text{exists}}(V,I,E) = \text{true}$       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 Boolean expression containing a property of the elements and there is at least one element in the collection  $V$  that satisfies  $E$

$= \text{false}$                   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 none of the elements in the collection  $V$  satisfies  $E$

$= \text{undefined}$         otherwise

### 6.8.9. 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:

- $\text{collection} \rightarrow \text{flatten}()$

Type for “flatten”

- (collection) → collection

Definition of evaluation function for  $F_{\text{flatten}}(V)$

$F_{\text{flatten}}(V) = \text{set}$                       If  $V$  is a collection, the result is a set containing all the elements of  $V$ .  
= element                              If  $V$  is not a collection.

### 6.8.10. Operator Size

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

The notation is:

- collection → size()

Types for “size”

- (collection) → integer

Definition of evaluation function for  $F_{\text{size}}(V)$

$F_{\text{size}}(V) = \text{integer}$                       If  $V$  is a collection  
= undefined                              otherwise

### 6.8.11. Operator Count

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

The notation is:

- collection → count(object)

Type for “count”

- (collection × object) → integer

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

$F_{\text{count}}(V, O) = \text{integer}$                       If  $V$  is a collection and  $O$  is a defined object  
= undefined                              otherwise

### 6.8.12. 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  $F_{\text{max}}(V)$

$F_{\text{max}}(V) = \text{number}$                       If  $V$  is a collection of numbers (integers or reals)

= undefined                      otherwise

### 6.8.13. 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 × object) → truth\_value

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

$F_{\text{includes}}(V, O) = \text{true}$	If $V$ is a collection and $O$ is an element in the collection.
$= \text{false}$	Else if $V$ is a collection and $O$ is not an element $V$ .
$= \text{undefined}$	otherwise

### 6.8.14. Operator IncludesAll

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

The notation is:

- collection → includesAll(parameterCollection)

Types for “includesAll”

- (collection × singleInstance) → truth\_value
- (collection × collection) → truth\_value

Definition of evaluation function for  $F_{\text{includesAll}}(V, C)$

$F_{\text{includesAll}}(V, C) = \text{true}$	If $V$ is a collection and $C$ is either a single instance or a collection, and all the elements in $C$ appear in $V$ .
$= \text{false}$	Else if $V$ is a collection and there is at least one element in $C$ that does not appear in $V$ .
$= \text{undefined}$	otherwise

### 6.8.15. Operator isEmpty

*isEmpty* returns true if the collection contains no elements.

The notation is:

- collection → isEmpty()

Type for “isEmpty”

- (collection) → truth\_value

Definition of evaluation function for  $F_{\text{isEmpty}}(V)$

$F_{\text{isEmpty}}(V) = \text{true}$	If $V$ is a collection with no elements
$= \text{false}$	Else if $V$ is a collection with one or more elements

= undefined            otherwise

### 6.8.16. Operator notEmpty

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

The notation is:

- collection → notEmpty()

Type for “notEmpty”

- (collection) → truth\_value

Definition of evaluation function for  $F_{\text{notEmpty}}(V)$

$F_{\text{notEmpty}}(V) = \text{true}$	If $V$ is a collection with one or more elements
$= \text{false}$	Else if $V$ is a collection with no elements
$= \text{undefined}$	otherwise

### 6.8.17. Operator Sum

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

The notation is:

- collection → sum()

Types for “sum”

- (collection\_of\_integers) → integer
- (collection\_of\_reals) → real

Definition of evaluation function for  $F_{\text{sum}}(V)$

$F_{\text{sum}}(V) = V_1 + \dots + V_n$	If $V$ is a non-empty collection of $n$ integer or real values values $\langle V_1, \dots, V_n \rangle$ with $(1 \leq i \leq n)$
$= 0$	Else if $V$ is an empty collection
$= \text{undefined}$	otherwise

### 6.8.18. Operator FirstN

*firstN* returns a *collection* with the first  $n$  elements from the current collection. This operator does not assume any order in the elements of the collection. If the collection is a bag or a set, elements are not ordered, whereas if it is a sequence, its elements are ordered (see 6.1.3 for a definition on Collections). In other words, *firstN* does not order the elements of the current collection, it simply returns the first  $n$  elements.

The notation is:

- collection → firstN(numberOfElements)

Type for “firstN”

- (collection × integer) → collection

Definition of evaluation function for  $F_{\text{firstN}}(V,N)$

$F_{\text{firstN}}(V,N) = \text{collection}$       If  $V$  is a non-empty collection (set, bag or sequence) and  $N$  is an integer such that  $1 \leq N \leq \text{size of } V$ . The resulting collection is of the same type as  $V$   
= undefined      otherwise

### 6.8.19. Operator LastN

Returns the *last*  $n$  elements from the current collection. The elements are returned as a collection of  $n$  elements. This operator does not assume any order in the elements of the collection. If the collection is a bag or a set, elements are not ordered, whereas if it is a sequence, its elements are ordered (see 6.1.3 for a definition on Collections). In other words, *lastN* does not order the elements of the current collection, it simply returns the last  $n$  elements.

The notation is:

- $\text{collection} \rightarrow \text{lastN}(\text{numberOfElements})$

Type for “last”

- $(\text{collection} \times \text{integer}) \rightarrow \text{collection}$

Definition of evaluation function for  $F_{\text{lastN}}(V,N)$

$F_{\text{lastN}}(V,N) = \text{collection}$       If  $V$  is a non-empty collection (set, bag or sequence) and  $N$  is an integer such that  $1 \leq N \leq \text{size of } V$ . The resulting collection is of the same type as  $V$   
= undefined      otherwise

### 6.8.20. Operator Reverse

*Reverse* returns a collection in reversed order. E.g. the first element of the current collection is returned as the last and so on. This operator does not sort the elements in the collection.

The notation is:

- $\text{collection} \rightarrow \text{reverse}()$

Type for “reverse”

- $(\text{collection}) \rightarrow \text{collection}$

Definition of evaluation function for  $F_{\text{reverse}}(V)$

$F_{\text{reverse}}(V) = \text{collection}$       If  $V$  is a single instance or a collection (the collection can be a set, bag or sequence). The resulting collection is of the same type as  $V$ .  
= undefined      otherwise

### 6.8.21. Operator SortBy

The notation is:



- (set×set) →bag
- (bag×bag)→bag
- (sequence×sequence) →sequence

Definition of evaluation function for  $F_{\text{union}}(V1, V2)$

$F_{\text{union}}(V1, V2) = \text{bag}$	If $V1$ and $V2$ are either sets or bags. The resulting bag contains the values $\langle V1,1, \dots, V1,n \rangle$ from $V1$ and $\langle V2,1, \dots, V2,n \rangle$ from $V2$ .
$= \text{sequence}$	Else if $V1$ and $V2$ are both sequences. The resulting sequence contains the values $\langle V1,1, \dots, V1,n \rangle$ from $V1$ and $\langle V2,1, \dots, V2,n \rangle$ from $V2$ .
$= \text{undefined}$	otherwise

### 6.8.24. Operator Including

The operator *including* returns a collection containing all the elements of the current collection plus an *element* added as the last element. If the returning collection is a set or bag, the elements in the resulting collection are not ordered. The element is appended to the current collection. However, if the returning type is a sequence, the element is inserted in the appropriate position.

The notation is:

- collection→including(element)

Types for “including”

- (set×element)→set
- (bag×element)→bag
- (sequence×element)→sequence

Definition of evaluation function for  $F_{\text{including}}(V, E)$

$F_{\text{including}}(V, E) = \text{set}$	If $V$ is a set and $E$ is an element that does not exist in $V$ . If $E$ already exists in $V$ , it is added, but the returning collection is bag, not a set. $E$ must be of the same type as the elements in $V$ .
-------------------------------------------	----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

$F_{\text{including}}(V, E) = \text{bag}$	If $V$ is a bag, the returning type of the collection is a bag. Also, if $V$ is a set and $E$ already exists in $V$ , it is added and the resulting collection is a bag. $E$ must be of the same type as the elements in $V$ .
-------------------------------------------	--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

$F_{\text{including}}(V, E) = \text{sequence}$	If $V$ is a sequence. $E$ must be of the same type as the elements in $V$ .
$= \text{undefined}$	otherwise

### 6.8.25. 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:



- *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”

- (collection × parameterList × parameterList × booleanExpression × OrderExpression) → bag\_of\_tuples  
\*: for any type of collection if OrderExpression is not specified.
- (collection × parameterList × parameterList × booleanExpression × OrderExpression) → sequence\_of\_tuples  
+: for any type of collection if OrderExpression is specified

#### Definition of evaluation function for $F_{\text{join}}(V, S1, S2, E1, E2)$

$F_{\text{join}}(V, S1, S2, E1, E2) =$  bag of tuples

If  $V$  is collection,  $S1$  is a parameter list of strings with the names of the collections from where data will be retrieved,  $S2$  is a parameter list of strings with the full names of the properties to be included in the result,  $E1$  is a boolean expression containing the conditions  $C_i \langle C_1 \text{ booleanOP } C_2 \dots \text{ booleanOP } C_n \rangle$  the returning elements must satisfy. The number of conditions  $C_i$  in  $E1 \geq [S1 \rightarrow \text{size}() - 1]$ .  $E2$  is an optional parameter that specifies the criteria for ordering the resulting elements. If  $E2$  is not specified, the result is a bag.

$F_{\text{join}}(V, S1, S2, E1, E2) =$  sequence of tuples

If  $V$  is collection,  $S1$  is a parameter list of strings with the names of the collections from where data will be retrieved,  $S2$  is a parameter list of strings with the full names of the properties to be included in the result,  $E1$  is a boolean expression containing the conditions  $C_i \langle C_1 \text{ booleanOP } C_2 \dots \text{ booleanOP } C_n \rangle$  the returning elements must satisfy. The number of conditions  $C_i$  in  $E1 \geq [S1 \rightarrow \text{size}() - 1]$ .  $E2$  is an optional parameter that specifies the criteria for ordering the resulting elements. If ordering is required, then a GELLO expression specifying the properties by which the result should be ordered by must be defined. The resulting collection is a sequence.

= undefined

otherwise

## 6.9. Tuple Operators

A GELLO tuple is an aggregated data type formed by one or more elements with different types. As described in §6.1.4, each tuple part has a name and a type. GELLO provides the following operations to handle and access tuple elements. We use the ‘dot’ notation to access elements in a tuple in the same manner as we access attributes in a class (§6.3.1).

### 6.9.1. Operator Size

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

The notation is:

- `Tuple.size()`

Types for “size”

- $(\text{tuple}) \rightarrow \text{integer}$

Definition of evaluation function for  $F_{\text{size}}(T)$

$F_{\text{size}}(T) = \text{integer}$       If  $T$  is a tuple  
= undefined      otherwise

### 6.9.2. Operator getValue

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

The notation is:

- `Tuple.getValue(elemName)`

Types for “getValue”

- $(\text{tuple} \times \text{string}) \rightarrow \text{BasicDataTypeValue}$
- $(\text{tuple} \times \text{string}) \rightarrow \text{ModelDataTypeObject}$

Definition of evaluation function for  $F_{\text{getValue}}(T, S)$

$F_{\text{getValue}}(T, S) = \text{BasicDataTypeValue}$       If  $T$  is a tuple and  $S$  is a string with the name of an element in  $T$ , and the type of the returning value is one of the basic data types: integer, real, boolean or string.  
=  $\text{ModelDataTypeObject}$       Else if  $T$  is a tuple and  $S$  is a string with the name of an element in  $T$ , and the type of the returning value is one of the model data types.  
= undefined      otherwise

### 6.9.3. 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:



Act objects act1, act2 one can say that act1 occurred before act2 if `act1.effectiveTime.lessThan(act2.effectiveTime)`.

## 6.11. 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.

## 6.12. If Expression

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

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

Definition of the evaluation function  $F_{if}(V_1, V_2, V_3)$ , where  $V_1$  is the condition, a GELLO expression which its evaluation returns a truth value; and  $V_2$  and  $V_3$  are *expression1* and *expression2* respectively, both valid GELLO expressions.

$$\begin{aligned} F_{if}(V_1, V_2, V_3) &= V_2 && \text{If } V_1 = \text{true} \\ &= V_3 && \text{Else if } V_1 = \text{false} \\ &= \text{undefined} && \text{otherwise} \end{aligned}$$

## 7. GELLO Syntax

This section describes the grammar used in this specification to define the lexical and syntactic structure of GELLO queries and 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 §7.2 describes the grammar used in this specification to define the lexical and syntactic structure of GELLO queries and expressions. For a query or expression to be syntactically correct it must conform to:

- The BNF and lexical grammar defined in this section of the document (§7.3).
- The context sensitive constraints:
  - Every query and expression must be type-correct. It must comply with the type definitions in §6.1.

### 7.1. Inferring Type Rules for Queries and Expressions

Let  $E$  be a valid GELLO query or an expression. The type of  $E$  is either a basic (Integer, Real, Boolean or String) or a model type. The type of  $E$  can be inferred by using the rules defined in GELLO lexical grammar §7.2 and GELLO BNF §7.3.

### 7.2. 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](#):

- A *reserved word* is any string that appears in double quotes in the BNF.
- An *atom* consists of any sequence of alphanumeric characters, which begins with a letter and can contain one or more underscores.
- 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 enclosed by the strings `/*` (marks the start of a comment) and `*/` (marks the end of a comment).

### 7.3. GELLO BNF Syntax

The Backus-Naur Form (BNF) syntax of GELLO assumes that text defining a GELLO query or 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 `<ExpressionOrQuery>`

- 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  $y$ .

### 7.3.1. Root Symbol

ExpressionOrQuery ::= Query  
 | Expression  
 | LetAssignment  
 | IfStatement

### 7.3.2. Literals

Literal ::= <STRING\_LITERAL>  
 | <INTEGER\_LITERAL>  
 | <REAL\_LITERAL>  
 | <TRUE>  
 | <FALSE>  
 | <UNKNOWN>

### 7.3.3. Data Types

DataTypes ::= GELLOTTypes  
 | ModelTypes

GELLOTTypes ::= BasicType  
 | CollectionType  
 | TupleType

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

ModelTypes ::= ClassName

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

TupleType ::= <TUPLE>

ClassName ::= Name

### 7.3.4. Names

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

### 7.3.5. Expressions

Expression::=	<u>ConditionalExpression</u>   <u>NewInstanceOfClass</u>   <u>FunctorExpression</u>
ConditionalExpression::=	<u>OrExpression</u>
OrExpression::=	<u>ConditionalAndExpression</u> (<OR> <u>ConditionalAndExpression</u>   <XOR> <u>ConditionalAndExpression</u> )*
ConditionalAndExpression::=	<u>ComparisonExpression</u> (<AND> <u>ComparisonExpression</u> )*
ComparisonExpression::=	<u>AddExpression</u> (<EQUAL> <u>AddExpression</u>   <NEQ> <u>AddExpression</u>   <LT> <u>AddExpression</u>   <LEQ> <u>AddExpression</u>   <GT> <u>AddExpression</u>   <GEQ> <u>AddExpression</u> )*
AddExpression::=	<u>MultiplyExpression</u> (<MINUS> <u>MultiplyExpression</u>   <PLUS> <u>MultiplyExpression</u> )*
MultiplyExpression::=	<u>UnaryExpression</u> (<TIMES> <u>UnaryExpression</u>   <DIVIDE> <u>UnaryExpression</u>   <MAX> <u>UnaryExpression</u>   <MIN> <u>UnaryExpression</u>   <INTDIV> <u>UnaryExpression</u>   <MOD> <u>UnaryExpression</u> )*
UnaryExpression::=	<u>UnaryNumber</u>   <u>UnaryModDivNum</u>   <u>UnaryMinus</u>   <u>MinusModDivNum</u>   <NOT> <u>UnaryExpression</u>   <u>PrimaryExpression</u>
UnaryNumber::=	<u>Number</u>
UnaryMinus::=	- <u>Number</u>
UnaryModDivNum::=	<INTEGER_LITERAL>
MinusModDivNum::=	- <INTEGER_LITERAL>
PrimaryExpression::=	<u>Literal</u>   <u>Name</u>   "(" <u>Expression</u> ")"
FunctorExpression::=	<u>FunctorName</u> "(" <u>ExpressionList</u> ")"
FunctorName::=	<u>Name</u>
ExpressionList::=	<u>Expression</u> ? (<COMMA> <u>ExpressionList</u> )*

### 7.3.6. Statements

Statement::= LetAssignment

	<u>IfStatement</u>
	<u>Query</u>
LetAssignment::=	<LET> <ID> “.” ( ( <u>BasicType</u>   <u>GELLOTtype</u> ) <EQUAL> ( <u>Expression</u>   <u>Query</u> ) )   ( <u>ClassName</u> <EQUAL> <u>NewInstanceOfClass</u> )
IfStatement::=	<IF> <u>Expression</u> <THEN> <u>Statement</u> <ELSE> <u>Statement</u> <ENDIF>

### 7.3.7. Creating a NEW Instance of a Model Class

NewInstanceOfClass::=	<u>ClassName</u> “.” <NEW> “(“ <u>ParameterList</u> “)”
-----------------------	---------------------------------------------------------

### 7.3.8. Queries

Query::=	<u>CollectionQuery</u>   <u>StringOperation</u>   <u>TupleQuery</u>
CollectionQuery::=	<u>CollectionName</u> “->” <u>QueryBody</u>
CollectionName::=	<ID>
QueryBody::=	<u>NonParamQuery</u>   <u>SelectionQuery</u>   <u>SingleObjQuery</u>   <u>ListObjQuery</u>   <u>GetQuery</u>   <u>SetQuery</u>   <u>IterateQuery</u>   <u>JoinQuery</u>
SelectionQuery::=	<SELECT> “(“ <u>CExp</u> ”)”   <REJECT> “(“ <u>CExp</u> ”)”   <COLLECT> “(“ <u>CExp</u> ”)”   <FORALL> “(“ <u>CExp</u> ”)”   <EXISTS> “(“ <u>CExp</u> ”)”
CExp::=	<u>ConditionalExpression</u>   <u>ConditionalExpressionWithIterator</u>   <u>ConditionalExpressionWithIteratorType</u>
ConditionalExpressionWithIterator::=	<u>Name</u> “ ” <u>ConditionalExpression</u>
ConditionalExpressionWithIteratorType::=	<u>Name</u> “.” <u>DataTypes</u> “ ” <u>ConditionalExpression</u>
NonParamQuery::=	<SIZE> “(“ “)”   <ISEMPTY> “(“ “)”   <NOTEMPTY> “(“ “)”   <SUM> “(“ “)”   <REVERSE> “(“ “)”   <MIN> “(“ “)”

	<MAX> “(“ “)”
	<FLATTEN> “(“ “)”
SingleObjQuery::=	<COUNT> “(“ <u>Object</u> “)”
	<INCLUDES> “(“ <u>Object</u> “)”
	<INCLUDING> “(“ <u>Object</u> “)”
	<EXCLUDING> “(“ <u>Object</u> “)”
ListObjQuery::=	<INCLUDESALL> “(“ <u>ObjectList</u> “)”
	<SORTBY> “(“ <u>PropertyList</u> “)”
GetQuery::=	<FIRSTN> “(“ <INTEGER_LITERAL> “)”
	<LASTN> “(“ <INTEGER_LITERAL> “)”
SetQuery::=	<INTERSECTION> “(“ <u>CollectionName</u> “)”
	<UNION> “(“ <u>CollectionName</u> “)”
IterateQuery::=	<ITERATE> “(“ <u>IterateParameterList</u> “)”
JoinQuery::=	<JOIN> “(“ <u>ParameterList</u> “;” <u>ParameterList</u> “;”
	<u>ConditionalExpression</u> “;” <u>ParameterList</u> “)”
StringOperation::=	<ID> “.” ( <u>StrSize</u>
	<u>StrConcat</u>
	<u>StrToUpper</u>
	<u>StrToLower</u>
	<u>Substring</u> )
StrSize::=	<SIZE> “(“ “)”
StrConcat::=	<CONCAT> “(“ <u>Expression</u> “)”
StrToUpper::=	<TOUPPER> “(“ “)”
StrToLower::=	<TOLOWER> “(“ “)”
Substring::=	<SUBSTRING> “(“ <INTEGER>, <INTEGER> “)”
TupleQuery::=	<u>TupleName</u> “.” ( <u>TupleSize</u>
	<u>TupleGetValue</u>
	<u>TupleGetElemName</u>
	<u>TupleGetElemType</u> )
TupleSize::=	<SIZE> “(“ “)”
TupleGetValue::=	<GETVALUE> “(“ <u>TupleElemName</u> “)”
TupleGetElemName::=	<GETELEMNAME> “(“ <INTEGER> “)”
TupleGetElemType::=	<GETELEMTYPE> ( “(“ <INTEGER> “)”
	“(“ <STRING> “)” )
TupleName::=	<ID>
ElementList::=	<u>Element</u> (<COMMA> <u>ElementList</u> )*

Element::=	<u>Name</u> “.” <u>Expression</u>
IterateParameterList::=	<u>Name</u> “.” <u>ClassName</u> “,” <u>Name</u> “.” <u>CollectionName</u> <EQUAL> <u>Expression</u> “ ” <u>Expression</u>
ParameterList::=	<u>Expression</u> (<COMMA> <u>ParameterList</u> )*
ObjectList::=	<u>Object</u> (<COMMA> <u>ObjectList</u> )*
Object::=	<u>Name</u>
PropertyList::=	<u>Property</u> (<COMMA> <u>PropertyList</u> )*
Property::=	<u>Name</u>
CollectionName::=	<u>Name</u>
TupleElemName::=	<u>Name</u>

### 7.3.9. Literals and Identifiers

<NUMBER:	<INTEGER_LITERAL>   <REAL_LITERAL >>
<INTEGER_LITERAL:	<DECIMAL_LITERAL >>
<#DECIMAL_LITERAL:	[“1”-“9”] ([“1”-“9”])*>
<REAL_LITERAL:	[“0”-“9”]+ “.” ([“0”-“9”])* (<EXPONENT>)?   “.” ([“0”-“9”])+ (<EXPONENT>)?   ([“0”-“9”])+ >
<#EXPONENT:	[“e”, “E”] ([“+”, “-”])? ([“0”-“9”])+>
<STRING_LITERAL:	“\”(~[“\”, “\n”, “\r”])*“\” >
<ID:	[“a”-“z”, “A”-“Z”] ([“a”-“z”, “A”-“Z”, “0”-“9”]   “_”([“a”-“z”, “A”-“Z”, “0”-“9”])*)* >

### 7.3.10. Reserved Words

<BAG: ‘Bag’>
<BOOLEAN: “Boolean”>
<INTEGER: “Integer”>
<REAL: “Real”>
<SEQUENCE: “Sequence”>
<SET: “Set”>
<STRING: “String”>
<TUPLE: “Tuple” >

### 7.3.11. Operators

<AND: “&”   “and” >
<ARROW: “->” >
<COLLECT: “collect” >
<COMMA: “,” >
<CONCAT: “concat” >
<COUNT: “count” >
<DIVIDE: “/” >
<EXCLUDING: “excluding” >
<EXISTS: “exists” >
<FIRSTN: “firstN” >
<FORALL: “forAll” >
<EQUAL: “=” >
<GEQ: “>=” >

<GETELEMNAME: "getElemName" >  
 <GETELEMTYPE: "getElemType" >  
 <GETVALUE: "getValue" >  
 <GT: ">" >  
 <IMPLIES: "implies" >  
 <INCLUDES: "includes" >  
 <INCLUDESALL: "includesAll" >  
 <INCLUDING: "including" >  
 <INTDIV: "div" >  
 <INTERSECTION: "intersection" >  
 <ISEMPTY: "isEmpty" >  
 <ITERATE: "iterate" >  
 <JOIN: "join" >  
 <LASTN: "lastN" >  
 <LEQ: "<=" >  
 <LT: "<" >  
 <MAX: "max" >  
 <MIN: "min" >  
 <MINUS: "-" >  
 <MOD: "mod" >  
 <NEQ: "!=" | "<>" >  
 <NEW: "new" >  
 <NOT: "!" | "not" >  
 <NOTEMPTY: "notEmpty" >  
 <OR: "|" | "or" >  
 <REJECT: "reject" >  
 <REVERSE: "reverse" >  
 <SELECT: "select" >  
 <SIZE: "size" >  
 <SORTBY: "sortBy" >  
 <SUBSTRING: "substring" >  
 <SUM: "sum" >  
 <TIMES: "\*" >  
 <TOLOWER: "toLower" >  
 <TOUPPER: "toUpper">  
 <UNION: "union" >  
 <XOR: "\*|" | "xor" >

### 7.3.12. Statements

<ELSE: "else" >  
 <ENDIF: "endif" >  
 <IF: "If" | "if" >  
 <LET: "Let" | "let" >  
 <THEN: "then" >

## 8. GELLO Queries

A GELLO query is any text string conforming to the definition of a query in the GELLO language specification §7.3.8. GELLO queries can be used to retrieve information from a data model such as the vMR.

When a query is executed, it returns a value. The type of the returning value is one of a basic §6.1.1, collection §6.1.3 or tuple type §6.1.4, or model type §6.1.2 defined in the

underlying data model. E.g. it could be an object of class *Class*, where *Class* is a class in the data model.

As with expressions, executing a query does not produce any side effects. However, the returning value can be bound to a variable using the *let* operator. See (§6.4.2).

### 8.1. Return Type of a Query

All GELLO queries return a value, and such value must have a type. The type of a return value must match any of the basic data types or classes defined in the underlying data model. The return type of a query can be one of the following:

- Single: basic type (§6.1.1) or model type (§6.1.2)
- Collection (§6.1.3): set, bag or sequence of single or tuples.
- Tuple (§6.1.4): an aggregation of different types

If such a value is assigned to a variable, both the value and the variable must have the same type.

### 8.2. Evaluation of Queries

As in OCL, GELLO queries always evaluate to a specific object of a specific type. Complex queries can be constructed by using the result of a query as a parameter of another query, both in the same expression. Queries are evaluated from left to right. In the case of infix operators, the evaluation order is determined by the precedence of the operators §6.10.

### 8.3. Examples of Queries

Queries interrogate a data repository (e.g. vMR) to retrieve data that match certain conditions specified in the query criteria. For purposes of the query language, the medical data repository is a set of objects representing medical data. Objects may have associations with each other (e.g., a lab test result may be associated with the order for the test).

```
Observation→select(coded_concept="C0428279")
```

The query above returns a collection of observations with a coded concept equal to "C0428279". As mentioned in (§6.4.2) the return value of a query can be assigned to a variable. For example, such value could be assigned to a *creatinineReadings* variable using the *let* operator. *CreatinineReadings* is a variable of type set:

```
let CreatinineReadings: set = Observation→select(coded_concept="C0428279")
```

The following example returns the last 3 creatinine readings:

```
CreatinineReadings→lastN(3)
```

## 9. 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:

- 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 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.

### 9.1. 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 basic §6.1.1, collection §6.1.3 or tuple data types §6.1.4, or classes defined in the underlying data model §6.1.2.

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

### 9.2. 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.

#### 9.2.1. Type Checking

Since GELLO is a strongly-typed language, it checks that the types of all expressions and queries are valid and match one of GELLO or model data types. Similarly, GELLO checks that the operands match the required types for any given operator. In other words, if an operator is applied to an incompatible operand, the return type of the function is *undefined*.

#### 9.2.2. 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.

### 9.3. 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.

### 9.3.1. Argument Lists

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

### 9.4. 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:

```
calcium.notEmpty() and phosphate.notEmpty()  
renal_failure and calcium_phosphate_product > threshold_for_osteodystrophy
```

## 10. Examples in GELLO

### 10.1. An MLM into GELLO

**From a MLM:**

maintenance:

title: Screening for elevated calcium-phosphate product;;

library:

purpose: provide an alert if the product of the blood calcium and phosphorus exceeds a certain threshold in the setting of renal failure;;

explanation: An elevated Ca-PO4 product suggests a tendency toward renal osteodystrophy and predisposes to soft-tissue calcification;;

**In GELLO:**

```
Let lastCreatinine : Observation = Observation->select(code=  
CodedValue.New("SNOMED-CT", "xxxxxx")).sortedBy(effectiveTime.high).last() Let  
lastCalcium : Observation = Observation->select(code =  
CodedValue.New("SNOMED-CT", "yyyyy")).sortedBy(effectiveTime.high).last()
```

```
Let lastPhosphate : Observation = Observation->select(code=  
codedValue.New("SNOMED-CT", "zzzzz")).sortedBy(effectiveTime.high).last()
```

```
let renal_failure_threshold : PhysicalQuantity = PhysicalQuantity.new(2.0, "mg/dl")  
let threshold_for_osteodystrophy : int = 70
```

```
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 <> null and lastPhosphate <> null then  
  lastCalcium.value * lastPhosphate..value  
else  
-1  
endif
```

```

if renal_failure and calcium_phosphate_product > threshold_for_osteodystrophy
then
    whatever action or message
else
    whatever action or message
endif

```

## 10.2. Example of an iteration over more than one collection at a time

This example shows how collection operators can be nested in queries 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

Query in GELLO (thanks to Conrad Bock):

```

patient.problem ->exist(problem: drug.AllInstances()->
exist(not(drug.compellingIndication -> intersection(problem) -> isEmpty))

```

## 10.3. Example: Presence of Azotemia Observation within last three months

Statement in English (many thanks to Samson Tu):

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
4. `IVL<TS>` has Allen's set of interval-based temporal operators

Query in GELLO:

```

Let month : CodedValue = CodedValue.new("SNOMED-CT", "258706009")
Let ThreeMonthsAgo : PointInTime =
    PointInTime.NOW().minus(PhysicalQuantity.new(3,month))

```

```
Let finding : CodedValue = CodedValue.new("SNOMED-CT", "246188002")
Let azotemia : CodedValue = CodedValue.new("SNOMED-CT", "371019009")
```

```
Observation->exists(code.equal(finding) and value.equal(azotemia) and
    effective_time.intersect(
        IVL<TS>.new(ThreeMonthsAgo, PointInTime.NOW()))))
```

#### **10.4. Example: Number of current anti-hypertensive Medications > 1**

Statement in English (many thanks to Samson Tu):

```
Number of current anti-hypertensive Medications > 1
```

Query in GELLO:

```
Let hypotensive_agents : CodedValue = CodedValue.new("SNOMED-CT", "1182007")
MedicationOrder->select(code.subclassOf(hypotensive_agents) and
    effectiveTime.high = null
)->size() > 1
```

#### **10.5. 3rd Td dose before 12 months of age**

Statement in English (many thanks to Samson Tu):

```
3rd Td dose before 12 months of age
```

Query in GELLO:

```
Let month : CodedValue = CodedValue.new("SNOMED-CT", "258706009")
Let DOBcode : CodedValue = CodedValue.new("SNOMED-CT", "184099003")
Let DateOfBirth : Observation = Observation-
>select(code.equal(DOBCode)).sortedBy(effectiveTime.high).last()
Let TwelveMonthsOfAge : PointInTime =
    DateOfBirth.effectiveTime.high.plus(PhysicalQuantity.new(12, month))
Let Td : CodedValue = CodedValue.new("SNOMED-CT", "59999009")
Let ThirdTdDose : SubstanceAdministration = SubstanceAdministration-
>select(code.equal(Td)).sortedBy(effectiveTime.high).third()
ThirdTdDose.effectiveTime.high.before(TwelveMonthsOfAge)
```

# 11. The HL7 RIM data model

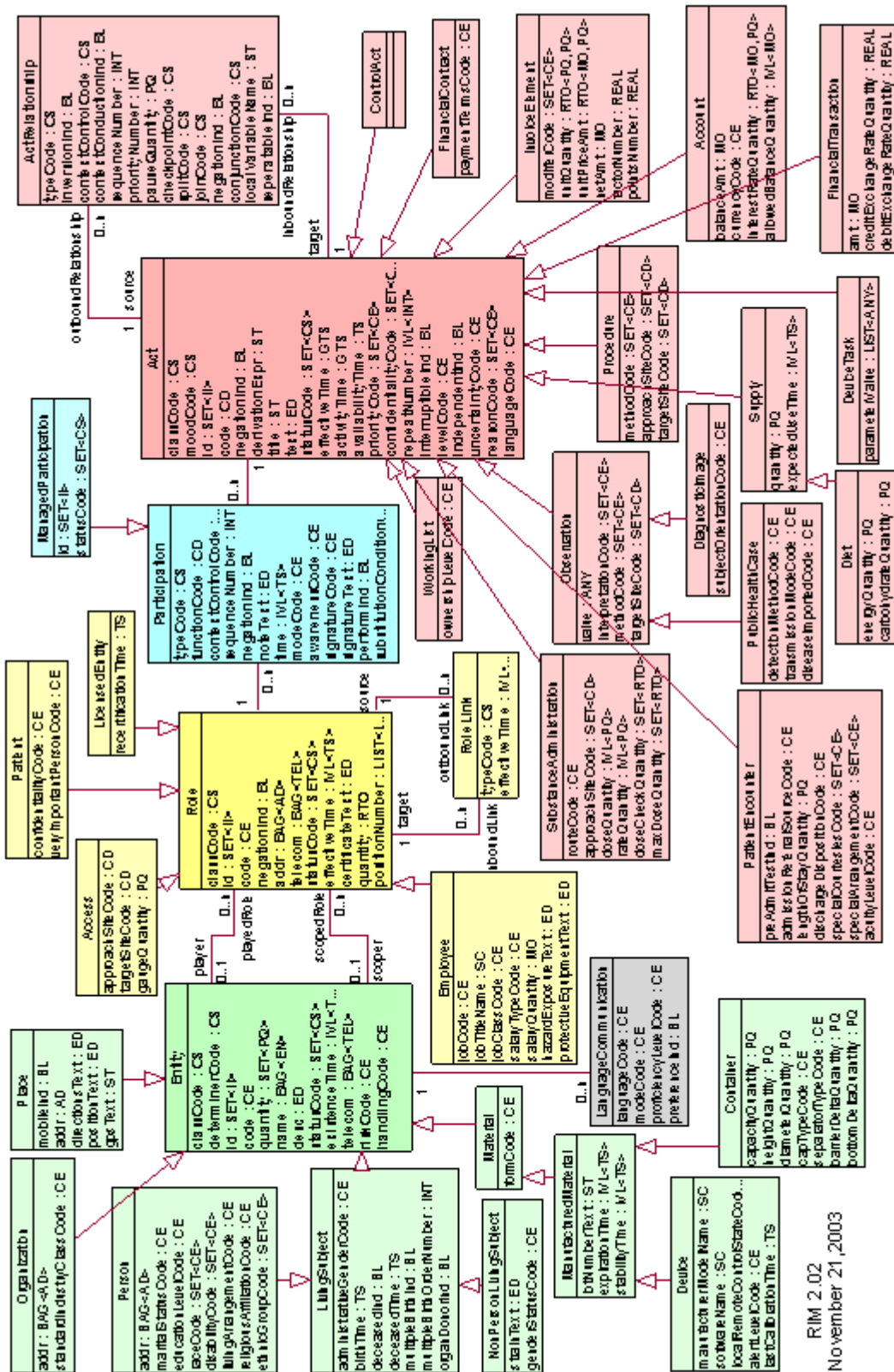


Figure 2: HL7 RIM 2.02 data model

## 12. Acknowledgements

We would like to thank Samson Tu, Gunther Schadow, Grahame Grieve, Dale Nelson, Bob Dolin, Anthony Malia, and Mor Peleg for their valuable comments. Support for this project has been provided by the CKBP grant and Partners Information Systems.

## 13. References

- HL7 RIM  
[http://www.hl7.org/Library/data-model/RIM/modelpage\\_mem.htm](http://www.hl7.org/Library/data-model/RIM/modelpage_mem.htm)  
HL7 RIM data model  
<http://www.hl7.org/Library/data-model/RIM/C30202/rim.htm>
- A virtual medical record for guideline-based decision support.  
[http://www.ncbi.nlm.nih.gov/entrez/query.fcgi?cmd=Retrieve&db=PubMed&list\\_uids=11825198&dopt=Abstract](http://www.ncbi.nlm.nih.gov/entrez/query.fcgi?cmd=Retrieve&db=PubMed&list_uids=11825198&dopt=Abstract)
- The Virtual Medical Record (vMR)  
[http://www-smi.stanford.edu/pubs/SMI\\_Reports/SMI-2001-0876.pdf](http://www-smi.stanford.edu/pubs/SMI_Reports/SMI-2001-0876.pdf)
- Minute from HL7 Clinical Decision Support Technical Committee and Clinical Guideline SIG Working Group meeting in San Diego, CA, January 9 -11, 2002.  
<http://smi-web.stanford.edu/people/tu/HL7/HL7SanDiegoJan2002.txt>
- [DSG02-01]. Omolola Ogunyemi, Qing Zeng, Aziz Boxwala. BNF and built-in classes for object-oriented guideline expression language (GELLO).  
<http://dsg.bwh.harvard.edu/gello/GELLClassesBNF.rtf>
- [TC1]. [http://dsg.bwh.harvard.edu/gello/Arden\\_GLIF\\_May\\_2001\\_AB.ppt](http://dsg.bwh.harvard.edu/gello/Arden_GLIF_May_2001_AB.ppt)
- [TC2]. <http://dsg.bwh.harvard.edu/gello/gello.ppt>
- [TC3]. <http://www.hl7.org/library/committees/dss/minutes/expr-lang-boxwala-10-2001.ppt>  
Also: <http://dsg.bwh.harvard.edu/gello/slc.ppt>
- [TC4]. <http://www.hl7.org/library/committees/dss/minutes/gelloupdate2-W2002.ppt>  
Also: <http://dsg.bwh.harvard.edu/gello/gelloupdate.ppt>
- [TC5]. [http://dsg.bwh.harvard.edu/gello/GELLO\\_may02.ppt](http://dsg.bwh.harvard.edu/gello/GELLO_may02.ppt)
- [TC6]. [GELLO update HL7 meeting. September 2002](http://dsg.bwh.harvard.edu/gello/GELLO_update_HL7_meeting_September_2002.ppt)
- UML 2.0 OCL <http://www.omg.org/cgi-bin/doc?ptc/03-10-14.pdf>
- OWL latest work in progress report:  
<http://www.w3.org/TR/2003/WD-owl-ref-20030221/>
- Example of an OWL ontology. <http://www.w3.org/TR/owl-guide/wine.owl>
- Current W3C recommendations and technical documents  
<http://www.w3.org/TR/>
- UML ITS data model.  
<http://www.hl7.org/v3ballot/html/foundationdocuments/itsuml/datatypes-its-uml.htm>