Medical-Objects Documentation

Medical-Objects GELLO User Guide

Table of Contents

1 Introduction

The Medical-Objects implementation of GELLO was written as a response to the need for decision support within the range of HL7 products produced by Medical-Objects Pty Ltd. It was initially developed in 2006 and has been successfully integrated into Medical-Objects Explorer and other research related products. For more information on Medical-Objects products visit www.medical-objects.com.au

1.1 GELLO v.1.x

GELLO is formally described as "An Object-Oriented Query and Expression Language for Clinical Decision Support". In a nutshell, it is a language to write expressions and make queries on medical data. GELLO expressions are somewhat like mathematical formulas but include additional functions to query and organize the data that the expressions refer to.

The GELLO language has been targeted at a wide range of IT Health professionals, from IT savvy doctors to experienced IT specialists working with in the IT Health industry. It has been based in part on the OCL (Object Contraint Language) version 2.0 language developed by OMG (Object Management Group) GELLO v.1.x is an extended version of GELLO which has been developed by Medical-Objects. While based on the official specification of GELLO, it provides some extensions to the language which have been required when using GELLO in a real world situation, and also to address any ambiguities and omissions within the specification. These changes to the BNF of the existing HL7 standard are in the preballot stage.

2 Getting Started

Let's learn about GELLO by working through some small examples.

2.1 Example 1 - very simple program

Here's an example of a very simple GELLO program.

```
let a:integer = 1
let b:integer = a + 1
b
```

This program is composed of 3 lines. The first two lines comprise of Statements and the last line is a final GELLO expression.

GELLO statements are of several forms. Our example only contains Let statements. A Let statement defines a variable (in this example's first statement, the variable 'a' and assigns a value to that variable. The final GELLO expression signifies that the GELLO program should return whatever value the expression evaluates to.

In our example, the first let statement "let a:integer = 1" defines the variable 'a' to be of type 'integer' and gives it the value '1'. (an integer is a whole number like 1, 2, 3, 4 and so forth and also includes negative numbers like -1, -5 and so forth). A type in GELLO refers to the kind of data that a variable can have. In this example, the type of the variable is "Integer".

The second let statement defines the variable 'b' and assigns it the value of 'a + 1' (which turns out to be 1 + 1 = 2).

The final line of the program, the GELLO expression simply returns the value of ‘b’ which in this case happens to be 2.

Here is the output from running such a program…

```
A: Integer = 1
B: Integer = 2
Result is 2
```

2.2 Example 2 - some expressions

Here is a slightly more complicated GELLO example.

```
let a:integer = 50
let b:integer = (a*100 + 20) div 2
a + b
```

And for which the output is…

```
A: Integer = 50
B: Integer = 2510
Result is 2560
```

2.3 Example 3 - using strings

We can also work with expressions which are not numbers but rather are strings, which are simply a group of letters like words.
Here is an example using strings.

```
Let surname:String = ‘Smith’
Let givenname:String = ‘Fred’
givenname.concat(‘ ‘).concat(surname)
```

And the results are

```
SURNAME: String = Smith
GIVENNAME: String = Fred
Result is Fred Smith
```

This example produces a result by concatenating (or joining) the three strings givenname, ‘ ‘ and surname with the concat() operator producing the string ‘Fred Smith’.

Note that strings are not simply words, but can contain any character including spaces, however string values must always be surrounded by the ‘ character.

2.4 Example 4 - some real GELLO

Let’s look at a more realistic example.

```
context observations: Observations from Model.observations
let sodiums = observations->select(code.name = "Sodium")
sodiums.value
```

This example introduces a new kind of statement, the context statement. This statement defines a variable much like the let statement, however, it is different in that instead of assigning a simple value, it assigns data to the variable from an external source. In this case, it defines a context variable of type Observation which will be populated by the caller with all the patient's observations. The identifier "Model" refers to the containing Data Model that the GELLO program runs within.

The type Observation is predefined to be a collection (or list) of individual observations. Each of these observations will have predefined properties depending on the data model, which in our case will have at least the properties "code" and "value". The codes of an observation will in turn have properties of "code" of type "CodedValue" (not to be confused with the observation's "code") and "name" which are both strings (groups of letters like words).

The next statement is somewhat familiar in that it is like the Let statement we saw earlier, however this time the type of the variable has been omitted. When written this way, the type of the variable is inferred from the expression used to create it. The expression assigned to the variable also looks a little unusual. You will notice after the variable name "Observation" is the symbol "->". This is the query operator and it is used to process the variable in a particular way, and in this example, we are specifying that we want to select from the observation list all observations that have a code with name "Sodium". We then finally assign the resulting new collection to the variable "sodiums"
The final line in the program returns only the value properties of the variable "sodiums".

You may have noticed that we have introduced two new concepts, properties and collections.

Variables can be defined with simple types like Integer, String, Real or Boolean (we'll discuss Real and Boolean later), or they can be defined with more complex types which are aggregates of other variables called "properties". We refer to the properties of a variable by using the "." operator. In the example, there two places where we use ".". The first is at the place "code.name" which is referring to the property name of the variable "code" (of type CodedValue). The second place is the reference "sodiums.value" which is a slightly more complicated use of the "." operator.

The other new concept is that of collections. A collection is a group or list of values each representing the same type of data. In GELLO, these can be a "Set", "Bag" or "Sequence", but for this example, it does not matter greatly which kind of collection it is, just that the variable "observation" is a collection of individual observations and that "sodiums" is the resulting collection formed by filtering out just the items which have as their code the coded name of "Sodium".

Let's run the program to see what the results might look like. For this example we'll assume that the test data has been loaded. If you are running this yourself, make sure your test data is loaded first, or you won't see any results.

```
OBSERVATIONS: Bag(Observation) = 496 Elements
SODIUMS: Bag = Bag{137, 138, 142, 131, 140, 137, 139, 135, 136, 140}
Result is Sequence{137, 138, 142, 131, 140, 137, 139, 135, 136, 140}
```

2.5 Wrap up

We've seen a few brief examples of how GELLO might look and feel. The following chapters will go into more detail on how to use GELLO.

3 User Manual

Here we will discuss the basics of what comprises a GELLO program.

3.1 GELLO Programs

A GELLO program is actually an expression (value) which is evaluated by the GELLO compiler.
The complete program must comply with a set of rules called a grammar, and must also also comply with another set of rules called the semantics. The combination of the grammar and the semantics defines the GELLO language. The grammar is defined formally by what is called a BNF grammar which is provided in the Appendices to this document.

GELLO programs are comprised of one or more lines of text which are further broken down into tokens and comments. The tokens are the items which have meaning to the compiler, while the comments are readable annotations for documenting the program and are ignored by the compiler.

GELLO v.1.x has extended the standard GELLO specification. Where there are differences between standard GELLO and GELLO v.1.x, they will be noted.

3.1.1 Tokens

Tokens are the individual words which make up a GELLO program. Tokens can be identifiers, numbers and strings and symbols.

3.1.1.1 Identifiers

Identifiers are used in GELLO to represent variable names, type names, property names and method names.

Identifiers can be 1 or more characters in length, and must start with "A-Z", "a-z" or "_". Subsequent characters must be "A-Z", "a-z", "0-9", or "_".

In GELLO v.1.x, identifiers are not case sensitive, which means that identifiers with the same alphabetic letters but with a different case will be treated by the compiler as meaning the same thing. There is no limit to the number of characters in an identifier.

Here are some examples of identifiers.

```
X
x
a1
A20
observation
Test_for_Creatinine
X201
_This_is_also_an_identifier
```

The BNF syntax for a GELLO variable is

```
<Identifier:
   ["A"-"Z","a"-"z","_"](["A"-"Z","a"-"z","_","0"-"9"])* >
```

3.1.1.2 Numbers

Numbers are used in GELLO to represent Integer or Real values.

All numbers must start with a digit ("0-9") followed by digits ("0-9"), optionally a period ("."), more digits ("0-9") and an optional exponent.

An exponent is represented by the letter "E" or "e" followed by a signed integer. It means that the first part (the mantissa) is multiplied by 10 to the power of the exponent. i.e. 1.1e3 means 1.1 times 10 to the power 3 which is 1100.

Integer numbers do not have a period or exponent (i.e. it is a string of digits only). In practice there is an upper and lower limit of what can be represented in computable form.

Real numbers are distinguished from Integer numbers by having a "." and an exponent. In practice there are limits to the size of the exponent and also the number of digits in the mantissa which can be represented in computable form.

Here are some examples of Integer numbers.

```
0 // the value zero
1 // the value one
15 // the value fifteen
2345 // the value two thousand, three hundred and forty five
812838482 // and so forth....
```

Here are some examples of Real numbers.

```
0.0 // the value zero as a real number
1.0 // the value one as a real number
1.0e0 // the same value as above
0.15 // the value 15 divided by one hundred (or three twentieths)
20.201 // twenty plus two hundred and one thousands
3.141596254 // an approximation of Pi
1.0E2 // the value one hundred (one times ten to the power two)
4.2E-30 // 4.2 divided by ten to the power thirty.
420.0e-32 // the same value as above
```

The BNF syntax for Integer and Real numbers is as follows

```
<IntegerConstant: (["0"-"9"])+ >
<RealConstant: (["0"-"9"])+ "."(["0"-"9"])* (["E"|"e"](["+","-"])?
   (["0"-"9"])* )? >
```

3.1.1.3 Strings

Strings are used to represent textual values in GELLO expressions.
They are started with a quotation character (either " or ') and are terminated by the same character. Strings are not allowed to continue over more than a line, and you can place any character in a string except for the surrounding quotation character.

If you wish to add control characters to a string, they can be embedded with the standard XML quoting practice using the form "&ssss;". The following escape symbols are recognized.
"&nl;" Add a new line (carriage return + line feed)
"&quot;" Add a double quote to the string (")
"&apos;" Add a single quote to the string (')

Here are some examples of strings.

```
'this is a string'
"this is also a string"
"this is a string with some quotations marks. &quot;GELLO&quot; is the
best language!!!"
"multi line string.&nl;another line.&nl;and another."
```

The syntax of strings is defined as follows.

```
<StringConstant: ("\'"(~["\'",\n,\r])* "\'" | "\"" (~["\"","\n,\r])*
   "\"" ) >
```

3.1.1.4 Symbols

Symbols are characters or combinations of characters which have special meaning in GELLO. Here is a list of them.

```
+
-
*
/
=
.
..
(
)
[
]
{
}
,
;
:
::
<>
!=
<
>
<=
>=
->
|
```

3.1.1.5 Reserved Words

Reserved words are identifiers which are special in GELLO. They cannot be used as identifiers since they have special meaning within the grammar of GELLO. In GELLO v.1.x, reserved words are not case sensitive.
Here is a list of them.

```
AND OR XOR NOT DIV MOD
SELECT REJECT COLLECT FORALL EXISTS
SIZE ISEMPTY NOTEMPTY SUM REVERSE MIN MAX FLATTEN
FIRST LAST AVERAGE STDEV VARIANCE
COUNT INCLUDES INCLUDING EXCLUDING INCLUDESALL SORTBY
ISTYPEOF ASTYPE HASINTERFACE
FIRSTN LASTN ELEMAT
INTERSECTION UNION ITERATE JOIN
LET
IF THEN ELSE ENDIF
CONCLUDE CONTINUE
CONTEXT FROM
SET BAG SEQUENCE TUPLE ENUM
```

```
INTEGER STRING REAL BOOLEAN TRUE FALSE UNKNOWN
MODULE TYPE CLASS METHOD EXTENDS VOID PROPERTY
```

3.1.2 Comments

Comments can be either single-line comments or multi-line comments. Comments are used to annotate the GELLO program without affecting how the program functions, and are a GELLO V.1.X extension to the specification.

3.1.2.1 Single-line Comments

A single line comment starts with the "//" characters and can appear anywhere on a line, even on lines with GELLO source. All characters from the "//" up until the end of the line are treated as comments. Some examples of single line comments are...

```
// this is a single line comment.
// All characters are ignored up to the
end of the line.
// the following is a line of commented GELLO source.
// let a:integer = 52
// the following is a comment after some GELLO source.
let b:string = 'some string' // our temporary variable "b" has the
value "some string"
```

3.1.2.2 Multi-line Comments

A multi-line comment starts with the character sequence "/*" and finishes with the character sequence "*/". They can span more than one line or simple be embedded within an existing line.
Here are some examples of multi-line comments.

```
/* this GELLO comment spans many lines.
All the text here is commented and will be ignored by the GELLO
compiler.
We can put anything we like in here, including strings, numbers symbols
and so forth as long
as it doesn't contain the multi-line comment end string.
This comment will end after here */
/*
Another comment
*/
/****** this too ********/
let a: integer = /* an embedded comment */ 20
/* we can put one here too */ if a = 20 then /* and here */ 50 else /*
blah... */ endif
```

3.2 Values and Types

All expressions in GELLO have a Value and a Type. The Value is the actual representation of the Type. Types can be either Simple Types, Collection Types, Tuple Types or Model Types

3.2.1 Simple Types

Simple types represent the most fundamental pieces of data that a GELLO program can work with. GELLO has several simple data types available. Integer, Real, String and Boolean

3.2.1.1 Integer Type

The Integer type represents values which are whole numbers. They can be positive or negative numbers and also include the value zero. GELLO v.1.x Integer values are stored as Double precision real numbers (64 bits) with an exponent of Zero. This means they have at most 54 bits of precision (from -(2_{53}-1) up to (2_{53}-1))

3.2.1.2 Real Type

The Real type represents values which are numbers which are not necessarily integers. Integers are a subset of Real numbers. GELLO v.1.x Real values are stored as Double precision real numbers (64 bits).

3.2.1.3 String Type

The String type represents values which are sequences of characters. Strings in GELLO v.1.x can have any length within the constraints of the available memory in the executing environment, and the characters are taken from the extended ASCII character set.

3.2.1.4 Boolean Type

The Boolean type is used to represent GELLO truth values. Values of this type can only be **true**, **false** or **unknown**.

3.2.2 Enumeration Types

Enumeration types are similar to simple types except that the values are defined using symbols.
For example a variable with enumeration type Colour could be defined as follows.
```
let colour:enum(red,blue,yellow,green,violet) = blue
```
At the moment, the GELLO specification is incomplete with regard to using enumeration types, however if the underlying data model uses enumeration types, these may be imported by the GELLO program.
Implementation Note: This feature is not activated in the current version of GELLO v.1.x.

3.2.3 Collection Types

A **collection** is a list of data values, each with the same data type. The data values can be either of a simple type like integer or string, or can be complex data types like collections, tuples or classes. The components or items of a collection are formerly known as **collection elements**.

There are three kinds of collections, **Sets**, **Bags** and **Sequences**.

A **Set** is a list of items which are all distinct (i.e. there can be no identical items). They may be in any order, however the ordering is unimportant when comparing two sets.

A **Bag** is similar to a Set with the exception that more than one item of the same value is allowed, and ordering is unimportant when comparing two bags.

A **Sequence** is much the same as a bag, except that the order of the items is important.

Here are examples of **Sets**.

```
Set{ 1, 2, 3, 4, 5 } // a set of the first 5 integers.
Set{ "apple", "orange", "pear" } // a set of strings with values
// corresponding to the names of fruit
```

Here are examples of **Bags**

```
Bag{ 1, 2, 2, 3, 4, 4, 4, 5 } // a list of integer values
// (with duplicates)
Bag{ 1, 2, 3, 2, 4, 5, 4, 4 } // same as above even though
// the ordering is different
Bag{ "apple", "orange", "pear", "apple" } // a list of strings with
values
// corresponding to the names of fruit
```

Here are examples of **Sequences**

```
Sequence{ 1, 2, 2, 3, 4, 4, 4, 5 } // a list of integer values
// (with duplicates)
Sequence{ 1, 2, 3, 2, 4, 5, 4, 4 } // different to above since the
// ordering is important
Sequence{ "apple", "orange", "pear", "apple" } // a list of strings
with values
// corresponding to the names of fruit
```

3.2.4 Tuple Types

Tuple types are similar to Collections in that they represent a group of related data items. However, unlike collections where the collection elements must be all of the same type, in tuples they may be of different types. Each element of a tuple is accessed by its name, and has its own distinct element type.

Here is an example of a let statement with a tuple representing the contact details and of a patient. The tuple definition starts with "tuple(" and ends with ")". Tuple types may be nested within other complex types, and other types can be nested within a tuple type.

```
let patient_contact:
      tuple(      surname: string,
                  givenname: string,
                  streetnumber: integer,
                  streetname
                  city: string,
                  zipcode: integer
                  country: string)
      = tuple{
                  surname = "Smith",
                  givenname = "Fred",
                  streetnumber = 123,
                  city = "MoTown",
                  zipcode = 998877,
                  country = "Republic of MoTownomia"}
```

The syntax of tuple type definitions is

```
TupleType ::= <Tuple> "(" TupleTypeList ")"
TupleTypeList ::= TupleTypeList "," TupleTypeElement
TupleTypeList ::= TupleTypeElement
TupleTypeElement ::= <Identifier> ":" Type
```

3.2.5 Class Types

Class types are similar to tuple types in that that have named elements which are called Attributes. Classes also have Operations which can be performed on the class. In general, the data model supplied to the GELLO program will have a number of classes which represent components of the data model.

3.2.5.1 Class Attributes

Enter topic text here.

3.2.5.2 Class Operations

Enter topic text here.

3.3 Expressions

Expressions form the foundation of GELLO programs and are made up of operands and operators. Generally an expression is written as a list of operands separated by operators. Operators have precedence, which means the order in which the operators will be applied when evaluating an expression with more than one operator. The precedence of operators may be overridden by the use of "(" and ")" to group sub-expressions.

conceptually in grammatical form

```
Expression ::= Expression Operator Operand
Expression ::= Operand
```

and

```
Operand ::= Literal
Operand ::= Variable
Operand ::= UnaryOperator Operand
Operand ::= "(" Expression ")"
Operand ::= ConditionalExpression
```

3.3.1 Operands

operands can be either literals, variable values, or the results of attributes, operations or queries.

3.3.1.1 Literals

Literal operands are operands which have fixed literal values, such as numbers, strings, or even complex literals like collection or tuple literals.

3.3.1.1.1 Integer Literals

Integer literals are values which are integer tokens. see Numbers

3.3.1.1.2 Real Literals

Real literals are values which are real tokens. see Numbers

3.3.1.1.3 String Literals

String literals are values which are string tokens. see Strings

3.3.1.1.4 Boolean Literals

Boolean literals are values which are boolean tokens. see Boolean Type

3.3.1.1.5 Collection Literals

the syntax of Collection Literals is as follows

```
CollectionLiteralExp ::= CollectionType "{" CollectionLiteralParts "}"
CollectionLiteralExp ::= CollectionType "{" "}"

CollectionLiteralParts ::= CollectionLiteralParts ","
      CollectionLiteralPart
CollectionLiteralParts ::= CollectionLiteralPart

CollectionLiteralPart ::= Expression
CollectionLiteralPart ::= CollectionRange

CollectionRange ::= Expression ".." Expression
```

3.3.1.1.6 Tuple Literals

the syntax of Tuple Literals is as follows

```
TupleLiteralExp ::= <Tuple> "{" TupleDefList "}"

TupleDefList ::= TupleDefList "," TupleDef
TupleDefList ::= TupleDef
TupleDef ::= <Identifier> ":" Type "=" Expression
TupleDef ::= <Identifier> "=" Expression
```

3.3.1.2 Variable values

A variable operand is represented by a variable name, and can be modified by any number of attributes or operations.

for example, in the program

```
let a:integer = 1
let b:integer = a + 25
b
```

there are variables **a** and **b**.

Whenever a variable is referred to, it is replaced in the expression by its value (in this example a has the value 1 and b has the value 26.

In the this example,

```
let sodiums = observations->select(code.name = "Sodium")
sodiums.value
```

there are several variable values.

```
observations
code
sodiums
```

3.3.1.3 Class Attribute Values

A class attribute value is specified by following an expression operand by a "." and an identifier

representing the attribute name. The resulting operand can be used as operand. An attribute means the same as the property of a class in other object oriented languages.
for example,
if a variable named **obs** of type **Observation** has the attributes **name** and **age**, these attributes can be written as

```
obs.name
```

and

```
obs.age
```

The operator may be repeatedly applied to the operand.

for example, one can write

```
obs.name.surname
```

to represent the **surname** attribute of the **name** attribute of the Observation **obs**

The syntax of a Class Attribute is

```
Variable ::= Variable . <Identifier>
```

3.3.1.4 Class Operation Values

Class Operation values are similar to attributes, but instead return the result of an operation applied to a variable operand. An operation means the same as a method of a class in other object oriented languages.

for example, if there is a variable **patient** of class **Patient**, and it has an operation

prescription_count_for_recent_years(num_years: Integer): Integer

to count the number of prescriptions in the last N years, one could get the number of prescriptions in the last year by writing...

```
patient.prescription_count_for_recent_years(1)
```

the syntax of a Class Operation is

```
Variable ::= Variable . <Identifier> "(" Params ")"
```

3.3.2 Operators

Expression Operators represent an operation which can be performed on one or two values or Operands of an expression. An operation on a single operand is called a Unary Operator and an operation on two operands is called a Binary Operator. Generally the form is either

```
UnaryOperator Operand
```

Or

```
Operand1 BinaryOperator Operand2
```

3.3.2.1 Simple Type Operators

Arithmetic operators
Boolean operators

3.3.2.1.1 Arithmetic operators

The following operators work on Reals and Integer types.

+ addition of two operands
- subtraction of two operands
* multiplication of two operands
/ division of two operands

- negation of a single operand

The following operators work on Integer types only

div	integer division of two operands
mod	integer modulo division (remainder after division) of two operands

3.3.2.1.2 Boolean operators

The following operators work on Boolean types. In GELLO, the Boolean operators also work for unknown values.

and	the logical **and** of two operands
or	the logical **inclusive or** of two operands
xor	the logical **exclusive or** of two operands
not	the logical **inverse** of one operand

Here is a table outlining the results of each Boolean operator

A	**B**	**A and B**	**A or B**	**A xor B**	**not A**
false	false	false	false	false	true
false	true	false	true	true	true
true	false	false	true	true	false
true	true	true	true	false	false
false	unknown	false	unknown	unknown	true
true	unknown	unknown	true	unknown	false
unknown	false	false	unknown	unknown	unknown
unknown	true	unknown	true	unknown	unknown
unknown	unknown	unknown	unknown	unknown	unknown

3.3.2.2 Class operators

There are several class operators available.

value.isUndefined()	returns true if the value is null or undefined.
value.isDefined()	returns true if the value is not null or undefined.
value.isTypeof(name)	returns true if the class of value is name.

3.3.2.3 Collection Operators

A collection operator is an operator that operates on collection classes only. To understand how collection operators are used, see Using Collections.

It takes the form <collection> "->" <collection operator> "(" <parameters> ")"

Here is an example using a collection operator.

```
      let sodiums = observations -> select(code.name = "Sodium")
```

This means **select** from the collection **observations** only observations which have the attribute **code** with a **name** of **"Sodium"**.

Some collection operators may take one or more conditions as parameters, while others may take a number, and some no parameters at all.
There are many predefined collection operators in GELLO. Here is a list of them with some examples of usage.

select(BooleanExpression)
select(v | boolean-expression-with-v)
select(v:Type | boolean-expression-with-v)

```
      observations->select(code.name = 'sodium')
      observations->select(obs | obs.code.value > 20 or obs.name = 'Na')
      observations->select(obs:EncodedObservation | obs.encoded.code.value >
      20 or obs.name = 'Na')
```

reject(BooleanExpression)
reject(v | boolean-expression-with-v)
reject(v:Type | boolean-expression-with-v)

```
      observations->reject(code.name = 'sodium')
      observations->reject(obs | obs.code.value > 20 or obs.name = 'Na')
      observations->reject(obs:EncodedObservation | obs.encoded.code.value >
      20 or obs.name = 'Na')
```

collect(Expression)
collect(v | expression-with-v)
collect(v:Type | expression-with-v)

```
      observations->collect(code.name)
      observations->collect(obs | obs.code.value)
      observations->collect(obs:EncodedObservation | obs.encoded.code.value)
```

forAll(BooleanExpression)
forAll(v | boolean-expression-with-v)
forAll(v:Type | boolean-expression-with-v)

```
      observations->forAll(code.name = 'sodium')
      observations->forAll(obs | obs.code.value > 20 or obs.name = 'Na')
      observations->forAll(obs:EncodedObservation | obs.encoded.code.value >
      20 or obs.name = 'Na')
```

exists(BooleanExpression)
exists(v | boolean-expression-with-v)
exists(v:Type | boolean-expression-with-v)

```
      observations->exists(code.name = 'sodium')
      observations->exists(obs | obs.code.value > 20 or obs.name = 'Na')
      observations->exists(obs:EncodedObservation | obs.encoded.code.value >
20 or obs.name = 'Na')
```

iterate(elem:Type; result:Type = expression | expression-with-elem-and-result)

```
        observations->iterate(  i:integer;
                                r:integer = 0 |
                                if code.name = 'Na' then r + 1 else r endif
                                )
```

includesAll(CollectionExpression)

```
        observations->collect(code.name)->includesAll('Na')
```

sortBy(ExpressionList)

```
        observations->sortBy(code.name,date)
```

firstN(IntegerExpression)

```
        observations->firstN(10)
```

lastN(IntegerExpression)

```
        observations->lastN(10)
```

elemAt(IntegerExpression)

```
        observations->elemAt(5)
```

size()

```
        observations->size()
```

isEmpty()

```
        observations->isEmpty()
```

notEmpty()

```
        observations->notEmpty()
```

sum()

```
        observations->collect(value)->sum()
```

reverse()

```
        observations->reverse()
```

min()

```
        observations->collect(value)->min()
```

max()

```
        observations->collect(value)->max()
```

flatten()

```
observations->flatten()
```

first()

```
observations->first()
```

last()

```
observations->last()
```

average()

```
observations->collect(value)->average()
```

stdev()

```
observations->collect(value)->stdev()
```

variance()

```
observations->collect(value)->variance()
```

count(object)

```
observations->collect(code.name)->count('Na')
```

includes(object)

```
observations->collect(code.name)->includes('Na')
```

including(element)

```
observations->collect(code.name)->including('Na')
```

excluding(element)

```
observations->collect(code.name)->excluding('Na')
```

intersection(set)

```
all_allergies->intersection(airborne_allergies)
```

union(set)

```
new_allergies->intersection(old_allergies)
```

join(collections; joinedproperties; booleanExpression; orderbyExpression)

3.3.3 Conditional Expression

A **Conditional Expression** is an expression which is determined by a boolean value. If the expression between **If** and **Then** evaluates to true, the resulting expression is that between the **Then** and **Else** symbols, otherwise it is the expression between the **Else** and **Endif** symbols.. An important aspect of

Conditional Expressions is that the two expressions alternatives are actually Expression Blocks which means one can have additional Let or Context statements inside the Conditional Expression. It is important to remember that any variables defined in an expression block are only local to that block.

The syntax is as follows.

```
ConditionalExpression ::= <If> Expression <Then> ExpressionBlock <Else>
ExpressionBlock <Endif>
```

3.4 Statements

A GELLO program typically consists of a number of statements one after the other. Usually a GELLO program contains a number of Let statements followed by a final Expression. It is completely valid to have a final Expression without any Let statements. The combination of statements and final expression is called an Expression Block. Expression Blocks can also appear inside Conditional Expressions.

There are several kinds of statements, Let Statements, Context Statements and Final Expression.

Statements are joined together into statement lists. Since GELLO is a declarative language, the order of statements should not affect the end result, however variables must be defined before they are used so any let statements defining them will need to be placed earlier in the statement lists before those variable are used.

The syntax of the statement section of a GELLO program is as follows

```
GELLO_Program ::= ExpressionBlock
ExpressionBlock ::= StatementList FinalExpression
StatementList ::= StatementList Statement
StatementList ::=
FinalExpression ::= Expression
FinalExpression ::=
Statement ::= LetStatement
Statement ::= ContextStatement
```

3.4.1 Let Statement

The Let Statement allows a GELLO expression to be assigned to a variable name. It is a very useful concept in that it allows GELLO expressions to be broken down into meaningful pieces, and also allows frequently used values to be reused within the GELLO program. A Let Statement can also be referred to as a Variable Declaration.

GELLO variables differ to those in typical computer languages in that they may only be assigned a value once which means that GELLO variables are effectively constant for their lifetime. The reason for this is that GELLO is derived from OCL which belongs to the family of functional languages.

If you are a programmer of commonly used programming languages like C or Pascal, it requires some rethinking to grasp the ways in which GELLO expressions are written. However, with a little practice complex GELLO programs can be effectively structured through the use of GELLO variables.

Some examples of Let statements are...

```
Let a: Integer = 25

let j = observation->select((code.code="2951-2") or (code.code = "2823-
      3"))

let alt_ok:boolean =
      if alt_obs.isdefined() then
            alt_obs.value < alt_obs.reference_range.upper_limit * 2
      else
```

```
            unknown
      endif
```

In GELLO v.1.x, the reserved word "Let" is not case-sensitive. Also, in GELLO v.1.x, the type of the variable is optional and may be omitted. The type of the variable can be inferred from the expression which is used to create it. If the type is specified, the assigned expression must be compatible with that type.

The syntax of Let statements is as follows.

```
LetStatement ::= <Let> <Identifier> OptionalType "=" Expression
OptionalType ::= ":" Type
OptionalType ::=
```

3.4.2 Final Expression

The Final Expression must be the last statement in a GELLO program. It is the result of executing the GELLO program. In GELLO v.1.x, this is currently optional, and when omitted, the GELLO program returns the **undefined** expression value.

In the following GELLO program

```
let a:integer = 50
let b:integer = (a*100 + 20) div 2
a + b
```

the final expression is the last line

```
a + b
```

In typical use, most GELLO programs will contain a Final Expression.

The syntax is

```
FinalExpression ::= Expression
FinalExpression ::=
```

3.4.3 Context Statement

The Context statement is still the subject of further research. It is currently implemented in a similar way to the let statement.

The syntax of a Context Statement is as follows.

```
ContextStatement ::= <Context> <Identifier> OptionalType <From>
<Expression>
```

3.5 The Data Model

All GELLO program will have a predefined environment which is available to it. This environment will contain a list of predefined classes which can be used. When using the GELLO Interactive Debugging Environment (IDE) you can use the Class Explorer tab to explore the classes available with their properties and methods (attributes and operations).

Here is a list of some of the classes available....

```
AbsoluteTime
AbsoluteTimeInterval
Allergy
Allergies
Archetypes
Boolean
CodedValue
ConceptRelationShip
Duration
DurationInterval
Factory
GLIFDecisionResult
GTS
Integer
IntegerInterval
Lib
Medication
Medications
MLM
Model
Module
Modules
Observation
ObservationManager
Observations
ObservationSequence
Output
Patient
PhysicalQuantity
PhysicalQuantityInterval
Provider
Ratio
Real
RealInterval
Reports
SD
SnomedAttribute
SnomedAttributeGroup
String
StructuredNumeric
```

There are also a number of predefined variables defined.

Name	**Class**
observation	Observations
patient	Patient
model	Model

3.6 Using Collections

One of the most powerful features of GELLO is its ability to work with collections. Typically a data model has many different collections which can be queried with the collection operators.

It is important to remember that when a collection operator is used, the attributes of the element type of the collection become available automatically as variables inside the query. This can be seen with one of our demonstration examples.

```
context observations: Observations from Model.observations
let sodiums = observations->select(code.name = "Sodium")
sodiums.value
```

The collection **observations** from the supplied model has as its elements, values of class **Observation**. One of the attributes of an observation is **code** which of class **CodedValue**. Inside the select query, this attribute is made available in the same way as a variable and can be referred to directly. In this example, we compare the value of each element's attribute **code** and if its name matches the string "Sodium", that element is selected and placed into the new collection. The same principle applies to several collection operators.

Some of the frequently used ones are...

Select Operator

You can create a subset of a collection by using the **select** operator

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

This will create a new collection of the same type as collection, but only containing elements of the original collection which match the boolean expression or condition. In our example, the following is the collection operation select.

```
observations->select(code.name = "Sodium")
```

Collect Operator

You can create a new collection based on elements of the collection by using the **collect** operator.

```
collection -> collect(sub-expression)
```

This will create a new collection based on the original collection, but each new element will only be the value as determined by sub-expression. A short hand for a collect operation when sub-expression is a single attribute of the element type of the expression is

```
collection.sub-expression
```

In our example, the following expressions are identical

```
sodiums->collect(value)
sodiums.value
```

3.7 Using Physical Quantities

GELLO v.1.x has included a special type of value called Physical Quantity. These are similar to the numeric Real type in that one can perform arithmetic on them (+, -, *, / and so forth), however they have the added property that each Physical Quantity has a units attached to it. This units property is fully managed by the GELLO framework when performing arithmetic on Physical Quantities. Many predefined units are included as standard, including most SI units. In the vEMR data model supplied by the Medical-Objects framework, Physical Quantities are used wherever possible in observations, medications and so forth.

It is important to remember the following rules when calculating with physical quantities. Units are compatible if their unit exponents are the same. For example, units of length (metres, feet, inches etc) are all compatible. As long as there is a conversion from one unit to another, units are also compatible. All units are formed from base units (e.g. metres, seconds, kilograms, etc)

If A and B are physical quantities:

Addition and Subtraction

Units of A and B must be compatible. If the units of A and B not identical, a units conversion operation will be made before the calculation. If they are not compatible, a GELLO exception (run time error) will be produced.

A + B	Result.value = A.value + B.convert(A.unit).value, Result.unit = A.unit
A - B	Result.value = A.value - B.convert(A.unit).value, Result.unit = A.unit

Multiplication

The unit indices of B are added to those of A. For example if A is in metres (m) and B is in square metres (m^2) then the result will be in cubic metres(m^3). If there are any conversion factors from the base units they will be multiplied together. The units do not need to be compatible - however this means that the result of the multiplication will need to be meaningful to what you are planning to do. If the units of A and B are not compatible, a derived unit will be formed with the combination of both units.

A * B	Result.value = A.value * B.value,
	Result.unit.exponent = A.unit.exponent + B.unit.exponent,
	Result.unit.scale = A.unit.scale + B.unit.scale

Division

The unit indices of B are subtracted from those of A. For example if A is in cubic metres (m^3) and B is in metres (m) then the result will be in square metres(m^2). If there are any conversion factors from the base units they will be conversion(A) / conversion(B).

A / B	Result.value = A.value / B.value,
	Result.unit.exponent = A.unit.exponent - B.unit.exponent,
	Result.unit.scale = A.unit.scale / B.unit.scale

An example of using physical quantities would be Body Mass Index. BMI = W/ H^2

```
let weight: PhysicalQuantity = factory.physicalquantity(55,'kg')
let height: PhysicalQuantity = factory.physicalquantity(1.02,'m')
let BMI: PhysicalQuantity = weight / (height * height)
"BMI=" + bmi.value.format(1,3) + ', units '+bmi.unit
```

The results are...

WEIGHT: PhysicalQuantity = 55 kg
HEIGHT: PhysicalQuantity = 1.02 m
BMI: PhysicalQuantity = 52.8642829680892 kgm^-2
Result is BMI=52.864, units kgm^-2

4 Appendices

4.1 GELLO BNF

```
      GELLO_Program ::= ExpressionBlock

      ExpressionBlock ::= StatementList FinalExpression

      StatementList ::= StatementList Statement

      StatementList ::=
      FinalExpression ::= Expression
      FinalExpression ::=

      Statement ::= LetStatement
      Statement ::= ContextStatement

LetStatement ::= <Let> <Identifier> OptionalType "=" Expression

IfStatement ::= <If> Expression <Then> StatementList <Else>
      StatementList <Endif>

ContextStatement ::= <Context> <Identifier> OptionalType <From>
      <Expression>

Type ::= TypeName
Type ::= BasicType
Type ::= CollectionType "(" Type ")"
Type ::= TupleType

BasicType ::= <Integer>
BasicType ::= <String>
BasicType ::= <Real>
BasicType ::= <Boolean>

CollectionType ::= <Set>
CollectionType ::= <Bag>
CollectionType ::= <Sequence>

TupleType ::= <Tuple> "(" TupleTypeList ")"

TupleTypeList ::= TupleTypeList "," TupleTypeElement
TupleTypeList ::= TupleTypeElement

TupleTypeElement ::= <Identifier> ":" Type

TypeName ::= Name

Name ::= Name "." <Identifier>
Name ::= <Identifier>

Expression ::= Expression "and" Expression
Expression ::= Expression "or" Expression
Expression ::= Expression "xor" Expression
Expression ::= Expression "=" Expression
Expression ::= Expression "<>" Expression
Expression ::= Expression "<" Expression
Expression ::= Expression ">" Expression
Expression ::= Expression "<=" Expression
Expression ::= Expression ">=" Expression
Expression ::= Expression "+" Expression
Expression ::= Expression "-" Expression
Expression ::= Expression "*" Expression
Expression ::= Expression "/" Expression
Expression ::= Expression "div" Expression
Expression ::= Expression "mod" Expression
Expression ::= "-" Expression
Expression ::= "not" Expression
Expression ::= Operand
Expression ::= ConditionalExpression

ConditionalExpression ::= <If> Expression <Then> ExpressionBlock <Else>
      ExpressionBlock <Endif>

Operand ::= Variable
Operand ::= "(" ExpressionList ")"
Operand ::= LiteralExp

Variable ::= <Identifier>
Variable ::= Variable "." <Identifier>
Variable ::= Variable "(" Params ")"
Variable ::= Query
Variable ::= IterateQuery
Variable ::= Variable "." "size" "(" ")"

IterateQuery ::= QueryVarOp "iterate" "(" iteratorID OptionalType ","
            initializerID OptionalType "=" Expression "|"
      Expression ")"

QueryVarOp ::= Variable "->"

OptionalType ::= ":" Type
OptionalType ::=

iteratorID ::= <Identifier>
initializerID ::= <Identifier>

Query ::= SelectionQuery
Query ::= ListObjQuery
Query ::= GetQuery
Query ::= NonParamQuery
Query ::= SingleObjQuery
Query ::= SetQuery
Query ::= JoinQuery

SelectionQuery ::= SelectionQueryHead "(" Expression ")"
SelectionQuery ::= SelectionQueryHead "(" <Identifier> OptionalType "|"
      Expression ")"

SelectionQueryHead ::= QueryVarOp SelectionQueryName

SelectionQueryName ::= <Identifier>
SelectionQueryName ::= "select"
SelectionQueryName ::= "reject"
SelectionQueryName ::= "collect"
SelectionQueryName ::= "forAll"
SelectionQueryName ::= "exists"
```

```
ListObjQuery ::= QueryVarOp "includesAll" "(" ExpressionList ")"
ListObjQuery ::= QueryVarOp "sortBy" "(" ExpressionList ")"

GetQuery ::= QueryVarOp "firstN" "(" Expression ")"
GetQuery ::= QueryVarOp "lastN" "(" Expression ")"
GetQuery ::= QueryVarOp "elemAt" "(" Expression ")"

NonParamQuery ::= QueryVarOp "size" "(" ")"
NonParamQuery ::= QueryVarOp "isEmpty" "(" ")"
NonParamQuery ::= QueryVarOp "notEmpty" "(" ")"
NonParamQuery ::= QueryVarOp "sum" "(" ")"
NonParamQuery ::= QueryVarOp "reverse" "(" ")"
NonParamQuery ::= QueryVarOp "min" "(" ")"
NonParamQuery ::= QueryVarOp "max" "(" ")"
NonParamQuery ::= QueryVarOp "flatten" "(" ")"
NonParamQuery ::= QueryVarOp "first" "(" ")"
NonParamQuery ::= QueryVarOp "last" "(" ")"
NonParamQuery ::= QueryVarOp "average" "(" ")"
NonParamQuery ::= QueryVarOp "stdev" "(" ")"
NonParamQuery ::= QueryVarOp "variance" "(" ")"

SingleObjQuery ::= QueryVarOp "count" "(" Expression ")"
SingleObjQuery ::= QueryVarOp "includes" "(" Expression ")"
SingleObjQuery ::= QueryVarOp "including" "(" Expression ")"
SingleObjQuery ::= QueryVarOp "excluding" "(" Expression ")"

SetQuery ::= QueryVarOp "intersection" "(" Expression ")"
SetQuery ::= QueryVarOp "union" "(" Expression ")"

JoinQuery ::= QueryVarOp "join" "(" ExpressionList ";" ExpressionList
      ";" Expression ";" ExpressionList ")"

Params ::= ExpressionList
Params ::=
ExpressionList ::= ExpressionList "," Expression
ExpressionList ::= Expression

LiteralExp ::= CollectionLiteralExp
LiteralExp ::= TupleLiteralExp
LiteralExp ::= PrimitiveLiteralExp

CollectionLiteralExp ::= CollectionType "{" CollectionLiteralParts "}"
CollectionLiteralExp ::= CollectionType "{" "}"

CollectionLiteralParts ::= CollectionLiteralParts ","
      CollectionLiteralPart
CollectionLiteralParts ::= CollectionLiteralPart

CollectionLiteralPart ::= Expression
CollectionLiteralPart ::= CollectionRange

CollectionRange ::= Expression ".." Expression

TupleLiteralExp ::= <Tuple> "{" TupleDefList "}"

TupleDefList ::= TupleDefList "," TupleDef
TupleDefList ::= TupleDef
```

```
TupleDef ::= <Identifier> ":" Type "=" Expression
TupleDef ::= <Identifier> "=" Expression

PrimitiveLiteralExp ::= <IntegerConstant>
PrimitiveLiteralExp ::= <RealConstant>
PrimitiveLiteralExp ::= <SringConstant>
PrimitiveLiteralExp ::= <True>
PrimitiveLiteralExp ::= <False>

<Let: "Let"|"let">
<If: "If"|"if">
<Then: "Then"|"then">
<Else: "Else"|"else">
<Endif: "Endif"|"endif">
<Conclude: "Conclude"|"conclude">
<Continue: "Continue"|"continue">
<Integer: "Integer"|"integer">
<Real: "Real"|"real">
<String: "String"|"string">
<Boolean: "Boolean"|"boolean">
<Set: "Set"|"set">
<Bag: "Bag"|"bag">
<Sequence: "Sequence"|"sequence">
<Tuple: "Tuple"|"tuple">
<True: "True"|"true">
<False: "False"|"false">

<IntegerConstant: (["0"-"9"])+ >

<RealConstant: (["0"-"9"])+ "."(["0"-"9"])* (["E"|"e"](["+","-"])?
      (["0"-"9"])* )? >
<StringConstant: ("\'"(~["\'",\n,\r])* "\'" | "\"" (~["\"","\n,\r])*
      "\"" ) >
<Identifier:
      ["A"-"Z","a"-"z","_"](["A"-"Z","a"-"z","_","0"-"9"])* >
```