GELLO R2 User Manual

Overview

Please note, this is a work in progress... The documents is being revised from an earlier version of GELLO.

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. Many of the semantics are defined by the data type definitions (eg ISO 21090 data types) and the Virtual Medical Record (vMR) definition. The data types and vMR are defined by the GELLO class definition which has been successfully balloted by HL7 as a Draft Standard for trial use (DSTU) in September 2011.

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.

Medical-Objects GELLO complies with the R2 release of the HL7/Ansi GELLO standard.

Why GELLO?

While GELLO has its roots in OCL 2.0, there are important distinctions between GELLO and OCL. You can read more about this **here**

Tokens

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

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 R2, identifiers are case sensitive, which means that identifiers with the same alphabetic letters but with a different case will be treated by the compiler as different names. 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"])* >
```

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.

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....

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 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 of two)
4.2E-30	4.2 divided by ten to the power of thirty
4.20.0e-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"])* )? >
```

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 omre than a line, and you can place any character in a string except for the surrounding quotation character.

If you wish to add control character to a string, they can be embedded with the standard XML quoting practice using the form "&ssss;". The following escape symbols are recognised.

"&nl;"	Add a new line (carriage return + line feed)
""""	Add a double quote to the string (")
"'"	Add a single quote to the sting (')

Here are some examples of strings.

```
'this is a string'
"this is also a string"
"this is a string with some quotations marks. "GELLO" 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])*
"\"" ) >
```

Symbols

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

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

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 R2, reserved words are case-sensitive, however some reserved words may have a capitalized variant.

Here is a list of them.

```
and or xor not div mod min max
iterate join
Let
If then else Endif
Context
Set Bag Sequence Tuple Enum
Integer String Real Boolean true false unknown null factory
Package EndPackage Class extends
```

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.

Single-line Comments[edit]
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"
```

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
```

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

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

Integer Type

The Integer type represents values which are whole numbers. They can be positive or negative numbers and also include the value zero. MO-GELLO-R2 Integer values are stored as Extended precision real numbers (80 bits) with an exponent of Zero. This means they have at most 64 bits of precision (from -9,223,372,036,854,775,807 up to +9,223,372,036,854,775,807)

Real Type

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

String Type

The String type represents values which are sequences of characters. Strings in MO-GELLO 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.

Boolean Type

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

Enumeration Types

Enumeration types are a specialization of the String type, however they are only permitted to have values that match the enumeration type. Any string value is permitted as an enumeration value.

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.

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
```

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: string,
city: string,
zipcode: integer
country: string)
= Tuple{
surname = "Smith",
givenname = "Fred",
streetnumber = 123,
streetname = "Lowdown St",
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
```

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.

Class Attributes

Enter topic text here.

Class Operations

Enter topic text here.

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
```

Operands

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

Literals

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

Integer Literals

Integer literals are values which are integer tokens. see Numbers

Real Literals

Real literals are values which are real tokens. see Numbers

String Literals

String literals are values which are string tokens. see Strings

Boolean Literals

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

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
```

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
```

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 this example,

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

there are several variable values.

observations code sodiums

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>
```

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 ")"
```

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
```

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
```

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
Unknown	Unknown	Unknown	True	Unknown	False
Unknown	False	False	Unknown	Unknown	Unknown
Unknown	True	Unknown	True	Unknown	Unknown
Unknown	Unknown	Unknown	Unknown	Unknown	Unknown

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.

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 Operator

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

Reject Operator

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

Collect Operator

```
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 Operator

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

Exists Operator

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

Iterate Operator

```
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 Operator

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

SortBy Operator

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

FirstN Operator

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

LastN Operator

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

ElemAt Operator

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

Size Operato

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

IsEmpty Operator

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

NotEmpty Operator

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

Sum Operator

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

Reverse Operator

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

Min Operator

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

Max Operator

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

Flatten Operator

```
flatten()observations->flatten()
First Operator[edit]
first()observations->first()
```

Last Operator

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

Average Operator

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

Variance Operator

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

Includes Operator

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

Including Operator

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

Excluding Operator

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

Intersection Operator

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

Union Operator

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

Join Operator

```
join(collections; joinedproperties; booleanExpression; orderbyExpression)
```

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 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>
```

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
```

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-R2, the reserved word "Let" is case-sensitive. Also, in MO-GELLO, 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 ::=
```

Final Expression

The **Final Expression** must be the last statement in a GELLO program. It is the result of executing the GELLO program. In MO-GELLO, this is currently optional, and when omitted, the GELLO program returns the null (or 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 ::=
```

Context Statement

The Context Statement is mainly used when GELLO is run as an query language within an existing class context. The <ClassName> parameter identifies the class context of the instance data for which GELLO query is being applied in an embedded GELLO environment. You can optionally assign a identifier to the class instance. If one is not given, the default name "self" is assigned.

For example, a messaging infrastructure might wish to execute a GELLO query on a message representing an instance from the data model. The instance root class should be the one specified in the Context statement (e.g. the SinglePatient class from the VMR model). Any Let statements or Final Expressions following the Context Statement can reference all the published attributes and operations of the underlying context class directly.

The syntax of a Context Statement is as follows.

```
ContextStatement ::= <Context> [ <Identifier> : ] <ClassName>
```

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....

NOTE - this only refers to the MO VMR model in MO-GELLO (GELLO R1)

```
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. (Only available in older MO-GELLO R1)

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

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
```

Using Physical Quantities

GELLO systems will use Health data types (Such as ISO21090 data types) in the Virtual Medical Record and generally the data types will include a special type of value called Physical Quantity (sometimes using the short name PQ). 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 SI 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
```

Example of PQ use

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
```

Appendices

GELLO R2 EBNF

The following notational conventions are used throughout GELLO BNF syntax:

The root symbol of the syntax is GELLOExpression
Non-terminal symbols are denoted by plain identifiers, 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:
(x)? denotes zero or one occurrences of x.
(x)* denotes zero or more occurrences of x.
(x)+ denotes one or more occurrences of x.
x | y means one of either x or y.

```
GELLOExpression::= OuterExpression
OuterExpression::= Declarative+ ( ExpressionNP? | <IN> Expression ) |
Expression
Declarative::= LetStatement
| ContextNavigationStatement
InnerExpression::= LetStatement+ (ExpressionNP | <IN> Expression) |
Expression
LetStatement::= <LET> <ID> ":" DataTypes <EQUAL> Expression
IfStatement::= <IF> Expression <THEN> InnerExpression <ELSE>
InnerExpression <ENDIF>
ContextNavigationStatement::= ContextStatement
| PackageStatement
ContextStatement::= <CONTEXT> ContextClass ContextBlock?
| <CONTEXT> Alias ":" ContextClass ContextBlock?
ContextClass::= ClassName | CollectionType "(" ContextClass ")"
ContextBlock::= DefinitionBody+
DefinitionBody::= <DEF> ":" <ID> ":" DataTypes <EQUAL> InnerExpression
| <DEF> ":" <ID> "(" FormalParams? ")" ":" DataTypes <EQUAL>
InnerExpression
FormalParams::= <ID> ":" DataTypes ("," FormalParams )?
Alias::= <ID>
PackageStatement::= <PACKAGE> PackageName
ContextStatement+
<ENDPACKAGE>
PackageName::= Name
Literal::= <STRING_LITERAL>
| <INTEGER_LITERAL>
| <REAL_LITERAL>
| <TRUE>
| <FALSE>
| <UNKNOWN>
| <NULL>
```

```
| CollectionLiteral
| TupleLiteral
| "#" <ID>
CollectionLiteral::= CollectionType? "{" ( CollectionLiteralElement ( ","
CollectionLiteralElement )* )? "}"
CollectionLiteralElement::= Expression ( ".." Expression )?
TupleLiteral::= <TUPLE> "{" TupleLiteralElement ( "," TupleLiteralElement )
* "}"
TupleLiteralElement::= <ID> (":" DataTypes)? <EQUAL> Expression
DataTypes::= GELLOType
| ModelTypes
GELLOType::= BasicType
| CollectionType ("(" DataTypes ")" )?
| TupleType
| EnumerationType
BasicType::= <INTEGER>
| <STRING>
| <REAL>
| <BOOLEAN>
ModelTypes::= ClassName
CollectionType::= <SET>
| <BAG>
| <SEQUENCE>
TupleType::= <TUPLE> "(" TupleTypeElement ( "," TupleTypeElement )* ")"
TupleTypeElement::= <ID> ":" DataTypes
EnumerationType::= <ENUM> "(" <ID> ( "," <ID> )* ")"
ClassName::= NameWithPath
NameWithPath::= Name ( "::" Name )*
Name::= <ID> ("." <ID>)*
Expression::= ConditionalExpression
ConditionalExpression::= OrExpression
OrExpression::= ConditionalAndExpression (<OR> ConditionalAndExpression |
<XOR> ConditionalAndExpression)*
ConditionalAndExpression::= ComparisonExpression (<AND>
ComparisonExpression)*
ComparisonExpression::= AddExpression (<EQUAL> AddExpression |
<NEQ> AddExpression | <LT> AddExpression |
<LEQ> AddExpression | <GT> AddExpression |
<GEQ> AddExpression)*
AddExpression::= MultiplyExpression (<MINUS> MultiplyExpression |
<PLUS> MultiplyExpression)*
MultiplyExpression::= UnaryExpression (<TIMES> UnaryExpression |
<DIVIDE> UnaryExpression | <MAX> UnaryExpression |
<MIN> UnaryExpression | <INTDIV> UnaryExpression |
<MOD> UnaryExpression )*
UnaryExpression::= PrimaryExpression
| <NOT> UnaryExpression
| <MINUS> UnaryExpression
| <PLUS> UnaryExpression
| PrimaryExpression
PrimaryExpression::= Literal
| Operand
| ReferenceToInstance
| IfStatement
| "(" Expression ")"
ExpressionNP::= ConditionalExpressionNP
ConditionalExpressionNP::= OrExpressionNP
OrExpressionNP::= ConditionalAndExpressionNP (<OR>
ConditionalAndExpression |
<XOR> ConditionalAndExpression)*
ConditionalAndExpressionNP::= ComparisonExpressionNP (<AND>
ComparisonExpression)*
ComparisonExpressionNP::= AddExpressionNP (<EQUAL> AddExpression |
<NEQ> AddExpression | <LT> AddExpression |
<LEQ> AddExpression | <GT> AddExpression |
<GEQ> AddExpression)*
AddExpressionNP::= MultiplyExpressionNP(<MINUS> MultiplyExpression |
<PLUS> MultiplyExpression)*
MultiplyExpressionNP::= UnaryExpressionNP (<TIMES> UnaryExpression |
<DIVIDE> UnaryExpression | <MAX> UnaryExpression |
<MIN> UnaryExpression | <INTDIV> UnaryExpression |
```

```
<MOD> UnaryExpression )*
UnaryExpressionNP::= PrimaryExpressionNP
| <NOT> UnaryExpression
PrimaryExpressionNP::= Literal
| Operand
| ReferenceToInstance
| IfStatement
Operand::= <ID>
| Operand "." <ID>
| Operand "." StringOperation
| Operand "." TupleOperation
| Operand "." StringOrTupleSize
| Operand "(" ParameterList ")"
| Operand "[" ExpressionList "]"
| Operand <ARROW> CollectionBody
| CollectionLiteral <ARROW> CollectionBody
| <SELF>
CollectionBody::= NonParamExp
| SelectionExp
| QuantifierExp
| SingleObjExp
| ListObjExp
| GetExp
| SetExp
| IterateExp
| JoinExp
SelectionExp::= <SELECT> "(" CExp ")"
| <REJECT> "(" CExp ")"
| <COLLECT> "(" CExp ")"
QuantifierExp::= <FORALL> "(" CExp ")"
| <EXISTS> "(" CExp ")"
CExp::= ConditionalExpression
| ConditionalExpressionWithIterator
| ConditionalExpressionWithIteratorType
ConditionalExpressionWithIterator::= <ID> "|" ConditionalExpression
ConditionalExpressionWithIteratorType::= <ID> ":" DataTypes "|"
ConditionalExpression
NonParamExp::= <SIZE> "(" ")"
| <ISEMPTY> "(" ")"
| <NOTEMPTY> "(" ")"
| <SUM> "(" ")"
| <REVERSE> "(" ")"
| <MIN> "(" ")"
| <MAX> "(" ")"
| <FLATTEN> "(" ")"
| <AVERAGE> "(" ")"
| <MEAN> "(" ")"
| <MEDIAN> "(" ")"
| <MODE> "(" ")"
| <STDEV> "(" ")"
| <VARIANCE> "(" ")"
| <DISTINCT> "(" ")"
SingleObjExp::= <COUNT> "(" Object ")"
| <INCLUDES> "(" Object ")"
| <INCLUDING> "(" Object ")"
| <EXCLUDING> "(" Object ")"
ListObjExp::= <INCLUDESALL> "(" ObjectList ")"
| <SORTBY> "(" PropertyList ")"
GetExp::= <FIRSTN> "(" Expression ")"
| <LASTN> "(" Expression ")"
| <ELEMAT> "(" Expression ")"
| <LIKE> "(" Expression ")"
| <NOTLIKE> "(" Expression ")"
| <BETWEEN> "(" Expression "," Expression ")"
SetExp::= <INTERSECTION> "(" Expression ")"
| <UNION> "(" Expression ")"
IterateExp::= <ITERATE> "(" IterateParameterList ")"
JoinExp::= <JOIN> "(" ParameterList ";" ParameterList ";"
ConditionalExpression ";" ParameterList ")"
StringOperation::= StrConcat
| StrToUpper
```

```
| StrToLower
| Substring
StringOrTupleSize::= <SIZE> "(" ")"
StrConcat::= <CONCAT> "(" Expression ")"
StrToUpper::= <TOUPPER> "(" ")"
StrToLower::= <TOLOWER> "(" ")"
Substring::= <SUBSTRING> "(" Expression "," Expression ")"
TupleOperation::= TupleGetValue
| TupleGetElemName
| TupleGetElemType
TupleGetValue::= <GETVALUE> "(" TupleElemName ")"
TupleGetElemName::= <GETELEMNAME> "(" Expression ")"
TupleGetElemType::= <GETELEMTYPE> "(" Expression ")"
IterateParameterList::= <ID> (":" DataTypes)? ";" <ID> ":" DataTypes
<EQUAL> Expression "|" Expression
ParameterList::= ExpressionList?
ExpressionList::= Expression ("," Expression)*
ObjectList::= Object ("," Object)*
Object::= Expression
PropertyList::= Property ("," Property)*
Property::= Name
TupleElemName::= Name
ReferenceToInstance::= <FACTORY>.ClassName(ParameterList )

<#DECIMAL_LITERAL: (["0"-"9"])+ >
<#EXPONENT: ["e", "E"] (["+","-"])? (["0"-"9"])+>
<INTEGER_LITERAL: <DECIMAL_LITERAL>>
<REAL_LITERAL: <DECIMAL_LITERAL> "." (["0"-"9"])* (<EXPONENT>)? | "." (["0"
-"9"])+ (<EXPONENT>)? >
<STRING_LITERAL: ('\"' (~[ '\"' , "\n", "\r"])* '\"' | "\'" (~[ "\'" ,
"\n", "\r"])* "\'" ) >
<ID: ["a"-"z","A"-"Z"] (["a"-"z","A"-"Z","0"-"9"] | "_"(["a"-"z","A"-"Z","
0"-"9"])+)* >
<BAG: "Bag">
<BOOLEAN: "Boolean">
<ENUM: "Enum">
<INTEGER: "Integer">
<NULL: "null">
<REAL: "Real">
<SEQUENCE: "Sequence">
<SET: "Set">
<STRING: "String">
<SELF: "Self">
<TUPLE: "Tuple" >
<UNKNOWN: "unknown" | "Unknown" >
<AND: "&" | "and" >
<ARROW: "->" | "?" >
<AVERAGE: "average" >
<BETWEEN: "between" >
<COLLECT: "collect" >
<CONCAT: "concat" >
<COUNT: "count" >
<DEF: "Def" | "def" >
<DISTINCT: "distinct" >
<DIVIDE: "/" >
<ELEMAT: "elemat" >
<EXCLUDING: "excluding" >
<EXISTS: "exists" >
<FACTORY: "factory">
<FALSE: "false" | "False" >
<FIRSTN: "firstN" >
<FLATTEN: "flatten" >
<FORALL: "forAll" >
<EQUAL: "=" >
<GEQ: ">=" >
<GETELEMNAME: "getElemName" >
<GETELEMTYPE: "getElemType" >
<GETVALUE: "getValue" >
<GT: ">" >
<INCLUDES: "includes" >
<INCLUDESALL: "includesAll" >
```

```
<INCLUDING: "including" >
<INTDIV: "div" >
<INTERSECTION: "intersection" >
<ISEMPTY: "isEmpty" >
<ITERATE: "iterate" >
<JOIN: "join" >
<LASTN: "lastN" >
<LEQ: "<=" >
<LIKE: "like" >
<LT: "<" >
<MAX: "max" >
<MEAN: "mean" >
<MEDIAN: "median" >
<MIN: "min" >
<MINUS: "-" >
<MOD: "mod" >
<MODE: "mode" >
<NEQ: "!=" | "<>" >
<NEW: "new" >
<NOT: "!" | "not" >
<NOTEMPTY: "notEmpty" >
<NOTLIKE: "notlike" >
<OR: "|" | |or" >
<PLUS: "+" >
<REJECT: "reject" >
<REVERSE: "reverse" >
<SELECT: "select" >
<SIZE: "size" >
<SORTBY: "sortBy" >
<SDEV: "stdev" >
<SUBSTRING: "substring" >
<SUM: "sum" >
<TIMES: "*" >
<TOLOWER: "toLower" >
<TOUPPER: "toUpper">
<TRUE: "true">
<UNION: "union" >
<VARIANCE: "variance" >
<XOR: "*|" | "xor" >
<CONTEXT: "context" | "Context">
<ELSE: "else" >
<ENDPACKAGE: "EndPackage" | "endPackage" | "endpackage" >
<ENDIF: "endif" >
<IF: "If" | "if" >
<IN: "in" >
<LET: "Let" | "let" >
<PACKAGE: "Package" | "package">
<THEN: "then" >
```

Values and Types[edit]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

Simple Types[edit]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

Integer Type[edit]The Integer type represents values which are whole numbers. They can be positive or negative numbers and also include the value zero. MO-GELLO-R2 Integer values are stored as Extended precision real numbers (80 bits) with an exponent of Zero. This means they have at most 64 bits of precision (from -9,223,372,036,854,775,807 up to +9, 223,372,036,854,775,807)

Real Type[edit]The Real type represents values which are numbers which are not necessarily integers. Integers are a subset of Real numbers. MO-GELLO Real values are stored as Double precision real numbers (64 bits).

String Type[edit]The String type represents values which are sequences of characters. Strings in MO-GELLO 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.
Boolean Type[edit]The Boolean type is used to represent GELLO truth values. Values of this type can only be true, false or unknown.
Enumeration Types[edit]Enumeration types are a specialization of the String type, however they are only permitted to have values that match the enumeration type. Any string value is permitted as an enumeration value. 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.
Collection Types[edit]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 differentBag{ "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 importantSequence{ "apple", "orange", "pear", "apple" } -- a list of strings with values corresponding to the names of fruit
Tuple Types[edit]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: string,   city: string,   zipcode: integer   country: string)
    = Tuple{   surname = "Smith",   givenname = "Fred",   streetnumber = 123,   streetname = "Lowdown St",   city = "MoTown",   zipcode = 998877,   country = "Republic of MoTownomia"}   The syntax of tuple type definitions is
TupleType ::= <Tuple> "(" TupleTypeList ")"
TupleTypeList ::= TupleTypeList "," TupleTypeElementTupleTypeList ::= TupleTypeElement
TupleTypeElement ::= <Identifier> ":" TypeClass Types[edit]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.

Class Attributes[edit]Enter topic text here.
Class Operations[edit]Enter topic text here.
Expressions[edit]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 OperandExpression ::= Operandand
Operand ::= LiteralOperand ::= VariableOperand ::= UnaryOperator
OperandOperand ::= "(" Expression ")"Operand ::=
ConditionalExpressionOperands[edit]operands can be either literals, variable values, or the results of attributes, operations or queries.
Literals[edit]Literal operands are operands which have fixed literal values, such as numbers, strings, or even complex literals like collection or tuple literals.

Integer Literals[edit]Integer literals are values which are integer tokens. see Numbers
Real Literals[edit]Real literals are values which are real tokens. see Numbers
String Literals[edit]String literals are values which are string tokens. see Strings
Boolean Literals[edit]Boolean literals are values which are boolean tokens. see Boolean Type
Collection Literals[edit]the syntax of Collection Literals is as follows
CollectionLiteralExp ::= CollectionType "{" CollectionLiteralParts "}"
CollectionLiteralExp ::= CollectionType "{" "}"

CollectionLiteralParts ::= CollectionLiteralParts ","
CollectionLiteralPartCollectionLiteralParts ::= CollectionLiteralPart
CollectionLiteralPart ::= ExpressionCollectionLiteralPart ::=
CollectionRange
CollectionRange ::= Expression ".." Expression
Tuple Literals[edit]the syntax of Tuple Literals is as follows
TupleLiteralExp ::= <Tuple> "{" TupleDefList "}"
TupleDefList ::= TupleDefList "," TupleDefTupleDefList ::= TupleDef
TupleDef ::= <Identifier> ":" Type "=" ExpressionTupleDef ::= <Identifier> "=" ExpressionVariable values[edit]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 = 1Let b: Integer = a + 25bthere 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 this example,
Let sodiums = observations->select(code.name = "Sodium")sodiums.valuethere are several variable values.
observations code sodiums
Class Attribute Values[edit]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.nameand
obs.ageThe operator "." may be repeatedly applied to the operand.
for example, one can write
 obs.name.surnameto represent the surname attribute of the name attribute of the Observation obs
The syntax of a Class Attribute is
Variable ::= Variable . <Identifier>Class Operation Values[edit]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): Integerto 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 ")"Operators[edit]
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 Operandor
 Operand1 BinaryOperator Operand2Arithmetic operators[edit]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 operandThe following operators work on Integer types only
div   integer division of two operandsmod   integer modulo division (remainder after division) of two operands
Boolean operators[edit]The following operators work on Boolean types. In GELLO, the Boolean operators also work for unknown values.
and   the logical and of two operandsor   the logical inclusive or of two operandsxor   the logical exclusive or of two operandsnot   the logical inverse of one operandHere is a table outlining the results of each Boolean operator
A   B   A and B   A or B   A xor B   not Afalse   false   false   false   false   true
truefalse   true   false   true   true   false
truetrue   false   false   true   true   false
falsetrue   true   true   true   false   true
falsefalse   unknown   false   unknown   unknown   truetrue   unknown   unknown   true
unknown   falseunknown   false   false   unknown   unknown   unknown   unknownunknown   true
unknown   true   unknown   unknownunknown   unknown   unknown   unknown   unknownClass operators[edit]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.
Collection Operators[edit]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 Operator[edit]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.code.name = 'Na')observations->select(obs: EncodedObservation | obs.encoded.code.value > 20 or obs.code.name = 'Na')
Reject Operator[edit]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.code.name = 'Na')observations->reject(obs: EncodedObservation | obs.encoded.code.value > 20 or obs.code.name = 'Na')
Collect Operator[edit]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 Operator[edit]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.code.name = 'Na')observations->forAll(obs:

```
EncodedObservation | obs.encoded.code.value > 20 or obs.code.name = 'Na')

Exists Operator[edit]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.code.name = 'Na')observations->exists(obs:
EncodedObservation | obs.encoded.code.value > 20 or obs.name = 'Na')

Iterate Operator[edit]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 Operator[edit]
includesAll(CollectionExpression)
observations->collect(code.name)->includesAll('Na')
SortBy Operator[edit]sortBy(ExpressionList)
observations->sortBy(code.name,date)
FirstN Operator[edit]firstN(IntegerExpression)
observations->firstN(10)
LastN Operator[edit]lastN(IntegerExpression)
observations->lastN(10)
ElemAt Operator[edit]elemAt(IntegerExpression)
observations->elemAt(5)
Size Operator[edit]size()
observations->size()
IsEmpty Operator[edit]isEmpty()
observations->isEmpty()
NotEmpty Operator[edit]notEmpty()
observations->notEmpty()
Sum Operator[edit]sum()
observations->collect(value)->sum()
Reverse Operator[edit]reverse()
observations->reverse()
Min Operator[edit]min()
observations->collect(value)->min()
Max Operator[edit]max()
observations->collect(value)->max()
Flatten Operator[edit]flatten()
observations->flatten()First Operator[edit]first()
observations->first()
Last Operator[edit]last()
observations->last()
Average Operator[edit]average()
observations->collect(value)->average()
stdev()
observations->collect(value)->stdev()
Variance Operator[edit]variance()
observations->collect(value)->variance()
count(object)
observations->collect(code.name)->count('Na')
Includes Operator[edit]includes(object)
observations->collect(code.name)->includes('Na')
Including Operator[edit]including(element)
observations->collect(code.name)->including('Na')
Excluding Operator[edit]excluding(element)
observations->collect(code.name)->excluding('Na')
Intersection Operator[edit]intersection(set)
all_allergies->intersection(airborne_allergies)
Union Operator[edit]union(set)
new_allergies->intersection(old_allergies)
Join Operator[edit]join(collections; joinedproperties; booleanExpression;
orderbyExpression)Conditional Expression[edit]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 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>Statements[edit]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 StatementStatementList ::=
FinalExpression ::= ExpressionFinalExpression ::=
Statement ::= LetStatementStatement ::= ContextStatement
Let Statement[edit]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-R2, the reserved word "Let" is case-sensitive. Also, in MO-GELLO, 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 ::= ":" TypeOptionalType ::=Final Expression[edit]The Final Expression must be the last statement in a GELLO program. It is the result of executing the GELLO program. In MO-GELLO, this is currently optional, and when omitted, the GELLO program returns the null (or undefined) expression value.
In the following GELLO program
Let a: Integer = 50Let b: Integer = (a*100 + 20) div 2a + bthe final expression is the last line
a + b
In typical use, most GELLO programs will contain a Final Expression.
The syntax is
FinalExpression ::= ExpressionFinalExpression ::=Context Statement[edit]
The Context Statement is mainly used when GELLO is run as an query language within an existing class context. The <ClassName> parameter identifies the class context of the instance data for which GELLO query is being applied in an embedded GELLO environment. You can optionally assign a identifier to the class instance. If one is not given, the default name "self" is assigned.
For example, a messaging infrastructure might wish to execute a GELLO query on a message representing an instance from the data model. The instance root class should be the one specified in the Context statement (e.g. the SinglePatient class from the VMR model). Any Let statements or Final Expressions following the Context Statement can reference all the published attributes and operations of the underlying context class

directly.
The syntax of a Context Statement is as follows.
ContextStatement ::= <Context> [<Identifier> :] <ClassName>The Data Model
[edit]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....
NOTE - this only refers to the MO VMR model in MO-GELLO (GELLO R1)
AbsoluteTimeAbsoluteTimeIntervalAllergyAllergiesArchetypesBooleanCodedValue ConceptRelationShipDurationDurationIntervalFactoryGLIFDecisionResultGTSInte gerIntegerIntervalLibMedicationMedicationsMLMModelModuleModulesObservationO bservationManagerObservationsObservationSequenceOutputPatientPhysicalQuanti tyPhysicalQuantityIntervalProviderRatioRealRealIntervalReportsSDSnomedAttri buteSnomedAttributeGroupStringStructuredNumericThere are also a number of predefined variables defined. (Only available in older MO-GELLO R1)
NameClassobservationObservationspatientPatientmodelModelUsing Collections
[edit]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.observationslet sodiums = observations->select(code.name = "Sodium")sodiums.valueThe 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[edit]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[edit]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.valueUsing Physical Quantities[edit]GELLO systems will use Health data types (Such as ISO21090 data types) in the Virtual Medical Record and generally the data types will include a special type of value called Physical Quantity (sometimes using the short name PQ). 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 SI units (e.g.

metres, seconds, kilograms, etc)
If A and B are physical quantities:
Addition and Subtraction[edit]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.unitA - B Result.value = A.value - B.convert(A.unit).value, Result.unit = A.unitMultiplication[edit]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.scaleDivision[edit]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.scaleExample of PQ use[edit]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 kgHEIGHT: PhysicalQuantity = 1.02 mBMI: PhysicalQuantity = 52.8642829680892 kgm^-2Result is BMI=52.864, units kgm^-2Appendices[edit]GELLO R2 EBNF[edit]The following notational conventions are used throughout GELLO BNF syntax:
The root symbol of the syntax is GELLOExpressionNon-terminal symbols are denoted by plain identifiers, e.g. expressionLeft-hand side terms in production rules are nonterminalTokens 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:(x)? denotes zero or one occurrences of x.(x)* denotes zero or more occurrences of x.(x)+ denotes one or more occurrences of x.x | y means one of either x or y.
GELLOExpression::= OuterExpression
OuterExpression::= Declarative+ (ExpressionNP? | <IN> Expression) | Expression
Declarative::
= LetStatement | ContextNavigationStatement
InnerExpression::= LetStatement+ (ExpressionNP | <IN> Expression) | Expression
LetStatement::= <LET> <ID> ":" DataTypes <EQUAL> Expression
IfStatement::= <IF> Expression <THEN> InnerExpression <ELSE> InnerExpression <ENDIF>
ContextNavigationStatement::
= ContextStatement | PackageStatement
ContextStatement::= <CONTEXT> ContextClass ContextBlock? | <CONTEXT> Alias ":" ContextClass ContextBlock?
ContextClass::= ClassName | CollectionType "(" ContextClass ")"
ContextBlock::= DefinitionBody+
DefinitionBody::= <DEF> ":" <ID> ":" DataTypes <EQUAL> InnerExpression | <DEF> ":" <ID> "(" FormalParams? ")" ":" DataTypes <EQUAL> InnerExpression
FormalParams::= <ID> ":" DataTypes ("," FormalParams)?
Alias::= <ID>
PackageStatement::= <PACKAGE> PackageName ContextStatement+ <ENDPACKAGE>

```
PackageName::=     Name
Literal::
=          <STRING_LITERAL>                                |   <INTEGER_LIT
ERAL>                                |   <REAL_LITERAL>
            |   <TRUE>                                |   <FALSE>
                        |   <UNKNOWN>                                |   <NUL
L>                                |   CollectionLiteral
              |   TupleLiteral                                |   "#" <ID>
CollectionLiteral::=  CollectionType? "{" ( CollectionLiteralElement ( ","
CollectionLiteralElement )* )? "}"
CollectionLiteralElement::= Expression ( ".." Expression )?
TupleLiteral::= <TUPLE> "{" TupleLiteralElement ( "," TupleLiteralElement )
* "}"
TupleLiteralElement::= <ID> (":" DataTypes)? <EQUAL> Expression
DataTypes::=      GELLOType                                |   ModelTypes
GELLOType::
=      BasicType                                |   CollectionType ("("
DataTypes ")"
)?                                |   TupleType
    |   EnumerationType
BasicType::
=         <INTEGER>                                |   <STRING>
                     |   <REAL>                                |   <BOOLEAN>
ModelTypes::=   ClassName
CollectionType::
=   <SET>                                |   <BAG>
        |   <SEQUENCE>
TupleType::=            <TUPLE> "(" TupleTypeElement ( ","
TupleTypeElement )* ")"
TupleTypeElement::=      <ID> ":" DataTypes
EnumerationType::=      <ENUM> "(" <ID> ( "," <ID> )* ")"
ClassName::=            NameWithPath
NameWithPath::=        Name ( "::" Name )*
Name::=        <ID> ("." <ID>)*
Expression::=     ConditionalExpression
ConditionalExpression::=     OrExpression
OrExpression::=     ConditionalAndExpression (<OR>
ConditionalAndExpression  |                                      <XOR>
ConditionalAndExpression)*
ConditionalAndExpression::=     ComparisonExpression (<AND>
ComparisonExpression)*
ComparisonExpression::=               AddExpression (<EQUAL>
AddExpression |                                  <NEQ> AddExpression |
<LT> AddExpression  |                                  <LEQ>
AddExpression |
<GT>  AddExpression  |                                   <GEQ>
AddExpression)*
AddExpression::=     MultiplyExpression (<MINUS> MultiplyExpression
|                                  <PLUS> MultiplyExpression)*
MultiplyExpression::=     UnaryExpression (<TIMES> UnaryExpression
|                                  <DIVIDE> UnaryExpression |
<MAX>  UnaryExpression |                                     <MIN>
UnaryExpression | <INTDIV> UnaryExpression
|                                  <MOD> UnaryExpression )*
UnaryExpression::=
PrimaryExpression                                          |   <NOT>
UnaryExpression                                |   <MINUS>
UnaryExpression                               |   <PLUS>
UnaryExpression                               |   PrimaryExpression
PrimaryExpression::
=     Literal                                |   Operand
           |   ReferenceToInstance                                |   IfSt
atement                               |   "(" Expression ")"
ExpressionNP::=     ConditionalExpressionNP
ConditionalExpressionNP::=     OrExpressionNP
OrExpressionNP::=     ConditionalAndExpressionNP (<OR>
ConditionalAndExpression  |                                      <XOR>
ConditionalAndExpression)*
ConditionalAndExpressionNP::=     ComparisonExpressionNP (<AND>
ComparisonExpression)*
ComparisonExpressionNP::=               AddExpressionNP (<EQUAL>
```

```
AddExpression |                                    <NEQ> AddExpression |
<LT> AddExpression  |                                 <LEQ>
AddExpression |
<GT>  AddExpression  |                                  <GEQ>
AddExpression)*
AddExpressionNP::=     MultiplyExpressionNP(<MINUS> MultiplyExpression
|                                 <PLUS> MultiplyExpression)*
MultiplyExpressionNP::=     UnaryExpressionNP (<TIMES> UnaryExpression
|                                 <DIVIDE> UnaryExpression |
<MAX>  UnaryExpression |                                   <MIN>
UnaryExpression | <INTDIV> UnaryExpression
|                                 <MOD> UnaryExpression )*
UnaryExpressionNP::
=       PrimaryExpressionNP                               |   <NOT>
UnaryExpression
PrimaryExpressionNP::
=    Literal                              |   Operand
            |   ReferenceToInstance                                |   IfSt
atement
Operand::=              <ID>                                | Operand
"."  <ID>                                | Operand "."
StringOperation                               | Operand "."
TupleOperation                              | Operand "."
StringOrTupleSize                              | Operand "("
ParameterList ")"                              | Operand "["
ExpressionList "]"                              | Operand <ARROW>
CollectionBody                              | CollectionLiteral <ARROW>
CollectionBody                              | <SELF>
CollectionBody::
=     NonParamExp                                 |   SelectionExp
                     |   QuantifierExp                                    |
SingleObjExp                                |   ListObjExp
             |   GetExp                                    |   SetExp
                      |   IterateExp                                |   Joi
nExp
SelectionExp::=     <SELECT> "(" CExp
")"                                 |   <REJECT> "(" CExp
")"                                 |   <COLLECT> "(" CExp ")"
QuantifierExp::=     <FORALL> "(" CExp
")"                                 |   <EXISTS>  "(" CExp ")"
CExp::
=     ConditionalExpression                                   |   ConditionalE
xpressionWithIterator                                |   ConditionalExpress
ionWithIteratorType
ConditionalExpressionWithIterator::=     <ID> "|" ConditionalExpression
ConditionalExpressionWithIteratorType::=      <ID> ":" DataTypes "|"
ConditionalExpression
NonParamExp::=     <SIZE> "("
")"                                 |   <ISEMPTY> "("
")"                                 |   <NOTEMPTY> "("
")"                                 |   <SUM> "("
")"                                 |   <REVERSE> "("
")"                                 |   <MIN> "("
")"                                 |   <MAX> "("
")"                                 |   <FLATTEN> "("
")"                                 |   <AVERAGE> "("
")"                                 |   <MEAN> "("
")"                                 |   <MEDIAN> "("
")"                                 |   <MODE>  "("
")"                                 |   <STDEV> "("
")"                                 |   <VARIANCE> "("
")"                                 |   <DISTINCT> "(" ")"
SingleObjExp::=            <COUNT> "(" Object
")"                                 |   <INCLUDES> "(" Object
")"                                 |   <INCLUDING> "(" Object
")"                                 |   <EXCLUDING> "(" Object ")"
ListObjExp::=     <INCLUDESALL> "(" ObjectList
")"                                 |   <SORTBY> "(" PropertyList ")"
GetExp::=     <FIRSTN> "(" Expression
")"                                 |   <LASTN> "(" Expression
")"                                 |   <ELEMAT> "(" Expression
```

```
")"                                |   <LIKE> "(" Expression
")"                                |   <NOTLIKE> "(" Expression
")"                                |   <BETWEEN> "(" Expression ","
Expression ")"
SetExp::=     <INTERSECTION> "(" Expression
")"                                |   <UNION> "(" Expression ")"
IterateExp::=     <ITERATE> "(" IterateParameterList ")"
JoinExp::=    <JOIN>  "(" ParameterList ";" ParameterList
";"                                    ConditionalExpression ";"
ParameterList ")"
StringOperation::
=     StrConcat                                |   StrToUpper
                  |   StrToLower                                  |   Substri
ng
StringOrTupleSize::=          <SIZE> "(" ")"
StrConcat::=     <CONCAT> "(" Expression ")"
StrToUpper::=     <TOUPPER> "(" ")"
StrToLower::=     <TOLOWER> "(" ")"
Substring::=     <SUBSTRING> "(" Expression "," Expression ")"
TupleOperation::
=     TupleGetValue                                    |   TupleGetElemName
                           |   TupleGetElemType
TupleGetValue::=     <GETVALUE> "(" TupleElemName ")"
TupleGetElemName::=     <GETELEMNAME> "(" Expression ")"
TupleGetElemType::=     <GETELEMTYPE> "(" Expression ")"
IterateParameterList::=     <ID> (":" DataTypes)? ";" <ID> ":"
DataTypes                                      <EQUAL> Expression "|"
Expression
ParameterList::=     ExpressionList?
ExpressionList::=     Expression ("," Expression)*
ObjectList::=    Object ("," Object)*
Object::=     Expression
PropertyList::=     Property ("," Property)*
Property::=     Name
TupleElemName::=     Name
ReferenceToInstance::=               <FACTORY>.ClassName(ParameterList )

<#DECIMAL_LITERAL: (["0"-"9"])+ >
<#EXPONENT: ["e", "E"] (["+","-"])? (["0"-"9"])+>
<INTEGER_LITERAL: <DECIMAL_LITERAL>>
<REAL_LITERAL: <DECIMAL_LITERAL> "." (["0"-"9"])* (<EXPONENT>)? | "." (["0"
-"9"])+ (<EXPONENT>)? >
<STRING_LITERAL: ('\"' (~[ '\"' , "\n", "\r"])* '\"' | "\'" (~[ "\'" ,
"\n", "\r"])* "\'" ) >
<ID: ["a"-"z","A"-"Z"] (["a"-"z","A"-"Z","0"-"9"] | "_"(["a"-"z","A"-"Z","
0"-"9"])+)* >
<BAG: "Bag">
<BOOLEAN: "Boolean">
<ENUM: "Enum">
<INTEGER: "Integer">
<NULL: "null">
<REAL: "Real">
<SEQUENCE: "Sequence">
<SET: "Set">
<STRING: "String">
<SELF: "Self">
<TUPLE: "Tuple" >
<UNKNOWN: "unknown" | "Unknown" >
<AND: "&" | "and" >
<ARROW: "->" | "?" >
<AVERAGE: "average" >
<BETWEEN: "between" >
<COLLECT: "collect" >
<CONCAT: "concat" >
<COUNT: "count" >
<DEF: "Def" | "def" >
<DISTINCT: "distinct" >
<DIVIDE: "/" >
<ELEMAT: "elemat" >
<EXCLUDING: "excluding" >
<EXISTS: "exists" >
```

```
<FACTORY: "factory">
<FALSE: "false" | "False" >
<FIRSTN: "firstN" >
<FLATTEN: "flatten" >
<FORALL: "forAll" >
<EQUAL: "=" >
<GEQ: ">=" >
<GETELEMNAME: "getElemName" >
<GETELEMTYPE: "getElemType" >
<GETVALUE: "getValue" >
<GT: ">" >
<INCLUDES: "includes" >
<INCLUDESALL: "includesAll" >
<INCLUDING: "including" >
<INTDIV: "div" >
<INTERSECTION: "intersection" >
<ISEMPTY: "isEmpty" >
<ITERATE: "iterate" >
<JOIN: "join" >
<LASTN: "lastN" >
<LEQ: "<=" >
<LIKE: "like" >
<LT: "<" >
<MAX: "max" >
<MEAN: "mean" >
<MEDIAN: "median" >
<MIN: "min" >
<MINUS: "-" >
<MOD: "mod" >
<MODE: "mode" >
<NEQ: "!=" | "<>" >
<NEW: "new" >
<NOT: "!" | "not" >
<NOTEMPTY: "notEmpty" >
<NOTLIKE: "notlike" >
<OR: "|" | |or" >
<PLUS: "+" >
<REJECT: "reject" >
<REVERSE: "reverse" >
<SELECT: "select" >
<SIZE: "size" >
<SORTBY: "sortBy" >
<SDEV: "stdev" >
<SUBSTRING: "substring" >
<SUM: "sum" >
<TIMES: "*" >
<TOLOWER: "toLower" >
<TOUPPER: "toUpper">
<TRUE: "true">
<UNION: "union" >
<VARIANCE: "variance" >
<XOR: "*|" | "xor" >
<CONTEXT: "context" | "Context">
<ELSE: "else" >
<ENDPACKAGE: "EndPackage" | "endPackage" | "endpackage" >
<ENDIF: "endif" >
<IF: "If" | "if" >
<IN: "in" >
<LET: "Let" | "let" >
<PACKAGE: "Package" | "package">
<THEN: "then" >
```