The OCL Datatypes
Real, Integer, String, and Boolean
together with their Operations

martin gogolla

University of Bremen

Summer semester 2020
Possible questions regarding the DVD cover

- What is the name of the dog?
- And the name of the boy?
- How old is the dog?
- And how old is the boy?
- The dog is a collie. Is that true or false?
- The boy is a collie. True or false?
- What is the weight of the dog?
- And the weight of the boy?
- Answers involve different kind of information

Construction of an object-oriented UML model with USE

- Representing elements of the DVD cover (of a snapshot, in general)
- Class diagram as a general frame for object diagrams (snapshots)
- Command list for building an object diagram
- Object diagram for representation of a snapshot
- OCL term evaluation for retrieving snapshot properties

Live: USE demo
- textual USE model file: model Lassie class Creature ... end
- Shell SOIL Actions: !new Creature ('lassie') OR !lassie.isCollie:=true
- Shell SOIL Queries: ?lassie.name
- Shell commands on multiple lines: \\
  ?lassie.isCollie and  
  jeff.isCollie

- Model browser, Class diagram, Command list, Object diagram, OCL expression evaluation, ...
class Creature

attributes
name : String
age : Integer
isCollie : Boolean
weight : Real

end
Command list for generating the object diagram

```plaintext
!new Creature('dog')  !new Creature('boy')
!dog.name:='Lassie'  !boy.name:='Jeff'        String values
!dog.age:=8          !boy.age:=12           Integer
!dog.isCollie:=true  !boy.isCollie:=false  Boolean
!dog.weight:=18.5    !boy.weight:=27.0    Real
```

Simple example terms querying for object attribute values

```plaintext
?dog.name
?boy.age
?boy.isCollie
?dog.weight
```

Example terms querying for values of combined properties

```plaintext
?dog.name.concat(' and ').concat(boy.name).concat(' are friends!')
?dog.weight + boy.weight
?(boy.age + dog.age) div 2
?boy.isCollie and dog.isCollie
```

First general observations on terms

- Constants: 'Lassie', 8, true, 18.5
- Operations: concatenation of strings 'concat'
- addition '+' on reals
- division 'div' on integers
- logical conjunction 'and' on booleans
OCL terms for *combined properties*

```
OCL expression: dog.name.concat(" and ").concat(boy.name).concat(" are friends!"))
Result: "Lassie and Jeff are friends!" : String
```

```
OCL expression: (boy.age + dog.age) div 2
Result: 10 : Integer
```

```
OCL expression: boy.isCollie and dog.isCollie
Result: false : Boolean
```

```
OCL expression: dog.weight + boy.weight
Result: 45.5 : Real
```
Example for USE shell protocol

use> ?dog.name
-> 'Lassie' : String
use> ?boy.age
-> 12 : Integer
use> ?boy.isCollie
-> false : Boolean
use> ?dog.weight
-> 18.5 : Real

use> ?dog.name.concat(' and ').concat(boy.name).concat(' are friends!')
-> 'Lassie and Jeff are friends!' : String
use> ?dog.weight + boy.weight
-> 45.5 : Real
use> ?(boy.age + dog.age) div 2
-> 10 : Integer
use> ?boy.isCollie and dog.isCollie
-> false : Boolean

use> ?dog.name.oclIsTypeOf(String)
-> true : Boolean
use> ?dog.age.oclIsTypeOf(Integer)
-> true : Boolean
use> ?dog.isCollie.oclIsTypeOf(Boolean)
-> true : Boolean
use> ?dog.weight.oclIsTypeOf(Real)
-> true : Boolean
Example terms checking the type of attributes

dog.name.oclIsTypeOf(String)
dog.age.oclIsTypeOf(Integer)
dog.isCollie.oclIsTypeOf(Boolean)
dog.weight.oclIsTypeOf(Real)

Datatypes and values

- Basic datatypes: Integer, Real, String, Boolean; constants available
- Integer values: 0, 1, 2, ..., -1, -2, ..., 365, 366, ...
- Real values: 0, 1, 1.1, 3.14, -3.141, ..., 9.81, ..., 299792.458, ...
- String values: '', 'a', ..., 'z', '0', ..., '9', '1.1', 'Lassie', ...
- Boolean values: false, true, ...

[Definition] Datatype

A datatype has a name and has associated operations that possess arguments and a result being also associated with a datatype. A datatype is connected to a set of values. The operations are associated with according functions on the value sets.

Example: Integer values: ... -2 -1 0 1 2 ...

    operation abs   ...  ↓  ↓  ↓  ↓  ↓ ...  Integer::abs():Integer

    ... 2 1 0 1 2 ...  -2.abs() = 2
OCL datatype class diagram - Classes and operations

Details (classes, operations, order) taken from OMG standard OCL 2.4.

Two kinds of notations for operations (notated without or with underscore):
- postfix operations as usual in object-orientation without using underscore
  example operation: Real::max(r:Real):Real; example term: 42.1.max(41.1)
- infix operations indicating operation parameters by underscore
  example operation: Real::_+(r:Real):Real; example term: 41.9 + 0.1

Postfix operations are here denoted with identifiers (letter, digits, ...).
Examples: abs, floor, mod, substring, at.
Infix operations are here written with symbols or identifiers (*, <, ...)
Examples: +, -, *, /, div, <, or, and.
OCL datatype class diagram – Example term per operation

We diverge the UML class diagram notation for showing OCL expressions that involve the corresponding operation from the class diagram. All OCL expressions evaluate to the Boolean value true.
OCL datatype class diagram —
Verbal explanation per operation
Generalization in the OCL datatype class diagram

Class Integer connected to class Real with unfilled arrowhead towards Real
alternative textual notation: Integer < Real
Real generalization of Integer; Integer specialization/subclass of Real

Real values: Set of Integer values is a subset of the set of Real values
Real operations: Operations from Real are also applicable on Integer

Examples: Real::_<(r:Real):Boolean allows to compute:
(2.5:Real < 3:Integer):Boolean = true
(2:Integer < 3:Integer):Boolean = true

Top-most type OclAny with Real < OclAny, String < OclAny, Boolean < OclAny

OclAny values: (Real values) U (String values) U (Boolean Values) ...
OclAny operations: equality OclAny::_=_ (o:OclAny):Boolean
inequality OclAny::_<>_(o:OclAny):Boolean

Examples: 42 = 84 div 2 : Boolean = true
3.14 <> 9.81 : Boolean = true
'Lassie'<>'Jeff' : Boolean = true
false = false : Boolean = true

Above the 2 operation parameters have identical datatypes
Below 2 parameters of different datatypes being also of type OclAny

42 <> 'fortytwo' : Boolean = true
42 <> '42' : Boolean = true
42 = 84 / 2 : Boolean = true
42 <> false : Boolean = true
Example terms with Real operations

3.14+3.14
6.28-3.14
3.14*3.14
-3.14
3.14/2
(-3.14).abs()
(3.14).floor()
(3.14+0.5).floor()
(3.14+0.35).round()
(3.14+0.36).round()
2.14.max(3.14)
3.14.min(4.14)
3.14<3.14
3.14>3.14
3.14<=3.14
3.14>=3.14
3.14.toString()
Example terms with Integer operations

-42
21+21
84-42
6*7
84/2
(-42).abs
84 div 2
42.mod(10)
41.max(42)
42.min(43)
42.toString()

[Exercise] Parentheses in terms. Evaluate the following OCL expressions on the USE shell by using the verbose evaluation with '??'.

18/6/3
(18/6)/3
18/(6/3)

3+3*7
(3+3)*7
3+(3*7)

'La'+'ss'+'ie'
('La'+'ss')+'ie'
'La'+('ss'+'ie')

false implies false implies false
(false implies false) implies false
false implies (false implies false)
use> ??18/6/3

Detailed results of subexpressions:

18 : Integer = 18
6 : Integer = 6

(18 / 6) : Real = 3.0

3 : Integer = 3

((18 / 6) / 3) : Real = 1.0

1.0 : Real

use> ??(18/6)/3

Detailed results of subexpressions:

18 : Integer = 18
6 : Integer = 6

(18 / 6) : Real = 3.0

3 : Integer = 3

((18 / 6) / 3) : Real = 1.0

1.0 : Real

use> ??18/(6/3)

Detailed results of subexpressions:

18 : Integer = 18
6 : Integer = 6
3 : Integer = 3

(6 / 3) : Real = 2.0

(18 / (6 / 3)) : Real = 9.0

9.0 : Real

use> ??3+3*7

Detailed results of subexpressions:

3 : Integer = 3
3 : Integer = 3
7 : Integer = 7

(3 * 7) : Integer = 21

(3 + (3 * 7)) : Integer = 24

24 : Integer

use> ??(3+3)*7

Detailed results of subexpressions:

3 : Integer = 3
3 : Integer = 3

(3 + 3) : Integer = 6

7 : Integer = 7

((3 + 3) * 7) : Integer = 42

42 : Integer

use> ??3+(3*7)

Detailed results of subexpressions:

3 : Integer = 3
3 : Integer = 3
7 : Integer = 7

(3 * 7) : Integer = 21

(3 + (3 * 7)) : Integer = 24

24 : Integer
use> ??'La'+'ss'+'ie'

Detailed results of subexpressions:

'La' : String = 'La'
'ss' : String = 'ss'
('La' + 'ss') : String = 'Lass'
'ie' : String = 'ie'
(('La' + 'ss') + 'ie') : String = 'Lassie'

use> ??('La'+'ss')+'ie'

Detailed results of subexpressions:

'La' : String = 'La'
'ss' : String = 'ss'
('La' + 'ss') : String = 'Lass'
'ie' : String = 'ie'
(('La' + 'ss') + 'ie') : String = 'Lassie'

use> ??false implies false implies false

Detailed results of subexpressions:

false : Boolean = false
false : Boolean = false
(false implies false) : Boolean = true
false : Boolean = false
((false implies false) implies false) : Boolean = false

use> ??(false implies false) implies false

Detailed results of subexpressions:

false : Boolean = false
false : Boolean = false
false : Boolean = false
((false implies false) implies false) : Boolean = false
false : Boolean = false

use> ??false implies (false implies false)

Detailed results of subexpressions:

false : Boolean = false
false : Boolean = false
false : Boolean = false
false : Boolean = false
((false implies false) implies false) : Boolean = false
true : Boolean
Example terms with String operations

'Lassie'+' is a '.concat('collie')
' is '.size()
'Lassie is a collie'.substring(7,10)
'42'.toInteger()
'3.14'.toReal()
'Lassie'.toUpperCase()
'Lassie'.toLowerCase()
'Lassie'.indexOf('s')
'Lassie'.equalsIgnoreCase('LASSIE')
'Lassie'.at(4)
'Lassie'.characters()
'false'.toBoolean()
'Jeff'<'Lassie'
'Jeff'>'Lassie'
'Jeff'='Lassie'
'Jeff'='Lassie'

Example terms with Boolean operations

boy.isCollie or dog.isCollie
boy.isCollie xor dog.isCollie
boy.isCollie and dog.isCollie
not boy.isCollie
boy.isCollie implies dog.isCollie
boy.isCollie.toString()
If we have in the class $C$ an operation $\text{op}$ returning the type $D$ and having parameter $p$ of type $D_1$, we denote this by: $C::\text{op}(p:D_1):D$. A term is a formation of constants and operations for building complex computations.

1. If $c$ is a constant for datatype $D$, then $c$ is a term of (data-)type $D$, notated as $c:D$.

   Examples: $42:\text{Integer}$, $3.14:\text{Real}$, 'Lassie':String, true:Boolean

2. If $t_1:D_1$, ..., $t_n:D_n$ are terms of type $D_i$ and $\text{op}:(D_1,...,D_n):D$ is an operation, then $\text{op}(t_1,...,t_n):D$ is a term of type $D$. Operations are notated in two forms: with postfix and with infix notation.

2a. If $C::\text{op}(p:D_1):D$ is a postfix operation in class $C$, $c$ is an instance of class $C$, and $t$ is a term for type $D_1$, then $c.\text{op}(t):D$ is a term in postfix notation.

   Example: Given 'Lassie':String, String::at(i:Integer):String and $3:\text{Integer}$, 'Lassie'.at(3):String is a postfix term of type String.

2b. If $C::\_\text{op}\_$(p:D1):D is an infix operation in class $C$, $c$ is an instance of class $C$, and $t$ is a term of type $D_1$, then $(c \text{ op } t):D$ is a term in infix notation. If the parentheses are ommitted, (complex) default rules apply.

   Example: Given $2.71:\text{Real}$, $3.14:\text{Real}$, $9.81:\text{Real}$ and Real::\_+\_$(r:\text{Real}):\text{Real}$, then $2.71+3.14*9.81:\text{Real}$ is a term of type Real using infix operations. Further example: $(2.71+3.14)*9.81$

Additionals matching parentheses may be introduced. Example: $(84)\text{div}(3-1)$
[Exercise] Syntactically valid and invalid terms

Consider the following list of pretended OCL terms. Check whether each term is syntactically valid or invalid w.r.t. the available predefined operations in the datatype class diagram.

3.14/0
(2.71+3.14).sqrt()
9.81/3.14/2.71

- - - - 42
40*41+42
42.max(41,42)

'Las'.concat('sie')
'Las'+_('sie')
'Las'_concat_('sie')

(false or false) and (false and false)
2.71 < 3.14 < 9.81
'Lassie'.isSubstringOf('Jeff and Lassie are friends')
Terms with let, terms with if, set membership and supersets

Variable names in terms

let VARIABLE=TERM in TERM-OPTIONALLY-WITH-VARIABLE

Conditionals in terms

if BOOLTERM then THEN-TERM else ELSE-TERM endif

In OCL there is an if-then-else-endif. There is no 'if-then-endif', because a value has to be calculated when the BOOLTERM is true and when it is false.

Sets and set membership

X=E1 or X=E2 or X=E3 <=> Set{E1,E2,E3}->{includes}(X)

Test for supersets and for subsets

S->includes(E1) and S->includes(E2) <=> S->includesAll(Set{E1,E2})

Good way of reading (and thinking of) an OCL term like 'S1->includesAll(S2)' is 'S1 is a superset of S2 (with equality making the term also true)'.

S1->includesAll(S2) <=> S1 isSupersetOf S2 <=> S1 ⊇ S2

<=> S2 isSubsetOf S1 <=> S2 ⊆ S1
Examples: Area of a circle

```
let pi=3.14 in let radius=3.7 in pi*radius*radius
3.14*3.7*3.7
```

Value of a linear function

```
let slope=4 in let yAxis=2 in let ARG=10 in slope*ARG+yAxis
4*10+2
```

OCL terms with 'let' have the advantage that variable names can make the purpose of the term and role that particular term parts play more clear. Structure of the term 'pi*radius*radius' is close to a mathematical textbook formula for the area of a circle. In contrast to the concrete term '3.14*3.7*3.7', the 'let' term can be re-used with different substitutions for the variable 'radius'. Manipulating the concrete term bears the danger that, if one wants to re-calculate the area with a different radius, say '3.8', only one of the needed two substitutions is made, e.g., '3.14*3.8*3.7'. 'let' gives the opportunity to build abstractions. The term 'slope*ARG+yAxis' is more abstract than the concrete evaluation '4*10+2'. It makes the purpose of the term more clear by using good explaining identifiers. It can also be re-used by employing different substitutions for the variables. Both 'let' terms are however more complex than the concrete evaluation terms.

```
let ARG=-42 in if ARG>=0 then ARG else -ARG endif
```

Absolute value of an argument: (-42).abs()

```
I=2 or I=4 or I=8 or I=16 or I=32 or I=64
Set{2,4,8,16,32,64}->includes(I)
Check whether an Integer value is equal to a potency of 2 up to 64

Set{1,2,3,4,5,6,7,8,9,10,11,12}->includesAll(Set{4,5,6,7,8,9})
The set of all months is a superset of the set of summer months.
Complex example using let, if, includes and includesAll

Check whether a String represents a date in the format 'YYYY-MM-DD'

Solution 1

```plaintext
let DATE='2000-11-31' in
let YEAR=DATE.substring(1,4) in
let MONTH=DATE.substring(6,7) in
let DAY=DATE.substring(9,10) in
let DIGIT=Set{'0','1','2','3','4','5','6','7','8','9'} in
'0000'<=YEAR and YEAR<='2020' and '01'<=MONTH and MONTH<='12' and
'01'<=DAY and DAY<='31' if MONTH='02' then '29' else if Set{'04','06',
'09','11'}->includes(MONTH) then '30' else '31' endif endif and
DIGIT->includesAll(Set{DATE.at(1),DATE.at(2),DATE.at(3),DATE.at(4),
DATE.at(6),DATE.at(7),DATE.at(9),DATE.at(10)}) and
DATE.at(5)= '-' and DATE.at(8)= '-' and DATE.size()=10
```

It may be the case that in the argument set of the 'includesAll' two different
terms (or even more) evaluate to the same String value. This is not a problem
for the evaluation:

```plaintext
let DATE='2000-12-31' in Set{DATE.at(1),DATE.at(2),DATE.at(3),
DATE.at(4),DATE.at(6),DATE.at(7),DATE.at(9),DATE.at(10)}
```

evaluates to

```
Set{'0','1','2','3'} : Set(String)
```
Instead of the restricting nested if conditions for the upper limit of DAY one can formulate this part with 3 implications that are connected through a conjunction.

'01'<=DAY and DAY<=if MONTH='02' then '29' else if Set{'04','06',
    '09','11'}->includes(MONTH) then '30' else '31' endif endif

<=>

'01'<=DAY and (MONTH='02' implies DAY<='29') and
(Set{'04','06','09','11'}->includes(MONTH) implies DAY<='30') and
(Set{'01','03','05','07','08','10','12'}->includes(MONTH)
    implies DAY<='31')

**Solution 2**

let DATE='2000-12-31' in
let YEAR=DATE.substring(1,4) in
let MONTH=DATE.substring(6,7) in
let DAY=DATE.substring(9,10) in
let DIGIT=Set{'0','1','2','3','4','5','6','7','8','9'} in
'0000'<=YEAR and YEAR<='2020' and '01'<=MONTH and MONTH<='12' and
'01'<=DAY and (MONTH='02' implies DAY<='29') and
(Set{'04','06','09','11'}->includes(MONTH) implies DAY<='30') and
(Set{'01','03','05','07','08','10','12'}->includes(MONTH)
    implies DAY<='31') and
DIGIT->includesAll(Set{DATE.at(1),DATE.at(2),DATE.at(3),DATE.at(4),
    DATE.at(6),DATE.at(7),DATE.at(9),DATE.at(10}) and
DATE.at(5)= '-' and DATE.at(8)= '-' and DATE.size()=10
Glimpse of collections in OCL

Two kinds of collections above: **sequences and sets**
- 'Lassie'.characters() = Sequence{'L','a','s','s','i','e'}
- Set{1,2,3,4,5,6,7,8,9,10,11,12}->includesAll(Set{4,5,6,7,8,9})

Features of sequences and sets
- Both are collections; both able to contain elements; both typed
- A sequence has an order for its elements; an element may appear twice
- A set does not have an element order; an element appears 'de facto' once
- Operations can be applied to collections (=, at, intersection, ...)
- Some operations can be applied to both sequences and sets (=, ...),
  some only to sequences (at, ...), some only to sets (intersection, ...)
- Collections essential OCL part; later discussed in detail

Example terms with collections and collection operations

Sequence{'J','e','f','f'} -> Sequence{'J','e','f','f'} : Sequence(String)
Set{'J','e','f','f'} -> Set{'J','e','f'} : Set(String)
Set{'L','a','s','s','i','e'} -> Set{'L','a','s','i','e'} : Set(String)
Set{'J','e','f','f'}=Set{'J','e','f'} -> true : Boolean
Set{'J','e','f','f'}=Set{'f','e','J'} -> true : Boolean
Sequence{'J','e','f','f'}=Sequence{'J','e','f'} -> false : Boolean

Sequence{'J','e','f','f'}->at(4) -> 'f' : String
Set{'J','e','f','f'}->intersection(Set{'L','a','s','s','i','e'}) -> Set{'e'} : Set(String)

Writing down sets more tricky than expected; good way of thinking about sets:
distinguish (in mind) a set representation and the set itself
Example: (Set{9,8,7} = Set{7,8,9}) = (Set{9,8,8,7,7,7} = Set{7,9,8}) -> true
Special cases and limitations requiring special attention

Arithmetic works correctly only up to certain bounds

42\neq 42.00000000000001 \rightarrow \text{false}
42\neq 42.000000000000001 \rightarrow \text{true}

Above USE evaluation; same behavior in Java; 13 zeros VS 14 zeros; relevant number of digits, not number of zeros

2147483647+1 \rightarrow -2147483648 : \text{Integer}

With markers for easier reading:
2.147.483.647+1 \rightarrow -2.147.483.648 : \text{Integer}

Typical Integer overflow in USE, as present in Java

Operation 'mod' in OCL needs getting used to

7 \text{ div } 3 \rightarrow 2 : \text{Integer}
7 \text{ / 3} \rightarrow 2.3333333333333335 : \text{Real}
7 \text{ mod } 3 \rightarrow \text{Error:line 1:2 missing EOF at 'mod'}
7.\text{mod}(3) \rightarrow 1 : \text{Integer}

ASCII ordering of String values with partly unexpected behavior

'0'<'9' and '9'<'A' and 'A'<'Z' and 'Z'<'a' and 'a'<'z' \rightarrow \text{true}
'a..z_A..Z_<'A..Z_a..z'
'9'<'10'

'ASCII order'<> 'Lexicographical order' \rightarrow \text{true}
'Lexicographical order'<> 'Numerical order' \rightarrow \text{true}

[End of 'Content Presentation' - Beginning of 'Exercises']
[Exercise] Build an OCL term that **converts** between Kilometer and English Miles

let K2M=true in let ARG=2 in
if K2M then ARG/1.609344 else ARG*1.609344 endif

Analogously build an OCL term to **convert** between degree Celsius and degree Fahrenheit

[Exercise] Build at least 3 syntactically different terms that **reverse** a string with fixed length 4. By 'syntactically different' we mean terms that, considered as sequences of characters, are different. Example: Syntactically different terms that evaluate to 42.0 are 42.0, 042.0, 21.0+21.0, 15.0+14+13, (15+14.0)+13, 15+(14+13.0)
[Exercise] Test whether a String with up to 3 characters represents a programming language identifier (first a letter or underscore, optionally followed by up to 2 characters being a letter, a digit or the underscore). A regular expression (i.e., a pattern over characters) for such a test is:

`^[a-zA-Z_][a-zA-Z_0-9]?[a-zA-Z_0-9]?$`

For the solution another operation on sets may be useful. The elements of a first set S1 and a second set S2 may be combined in that the result consists of all elements that are either in the first or the second set, the union of S1 and S2: S1->union(S2).

Example:
let ARG='Ab' in let abc=Set{'a','b','c'} in let ABC=Set{'A','B','C'} in
ABC->includes(ARG.at(1)) and ABC->union(abc)->includes(ARG.at(2))
The first letter of the String ARG is one of A, B, C and the second letter is one of A, B, C, a, b, c.
[Exercise] Precision in formulating descriptions

Consider the two following descriptions.

Description A:

let ARG=... in let abc=Set{'a','b','c'} in let ABC=Set{'A','B','C'} in
ABC->includes(ARG.at(1)) and ABC->union(abc)->includes(ARG.at(2))
The first letter of the String ARG is one of A, B, C and the second
letter is one of A, B, C, a, b, c.

Description B:

let ARG=... in let abc=Set{'a','b','c'} in let ABC=Set{'A','B','C'} in
ABC->includes(ARG.at(1)) and ABC->union(abc)->includes(ARG.at(2))
The first letter of the String ARG is one of 'A', 'B', 'C' and the second
letter is one of 'A', 'B', 'C', 'a', 'b', 'c'.

Discuss whether these two statements show a difference w.r.t. precision.
[Exercise] Construct an OCL term that checks a String for the time format 'HH:MM' with 00<=HH<=23 and 00<=MM<=59.

[Exercise] Modify the OCL term checking a date in order to cover leap years.
let DATE='1900-02-29' in
let YEAR=DATE.substring(1,4) in
let MONTH=DATE.substring(6,7) in
let DAY=DATE.substring(9,10) in
let DIGIT=Set{'0','1','2','3','4','5','6','7','8','9'} in
let year=YEAR.toInteger() in
let isLeap=if year.mod(4)<>0 then false else
  if year.mod(100)<>0 then true else
    if year.mod(400)<>0 then false else true endif endif endif in
'0000'<=YEAR and YEAR<='2020' and '01'<=MONTH and MONTH<='12' and
'01'<=DAY and ((MONTH='02' and not isLeap) implies DAY<='28') and
((MONTH='02' and isLeap) implies DAY<='29') and
(Set{'04','06','09','11'}->includes(MONTH) implies DAY<='30') and
(Set{'01','03','05','07','08','10','12'}->includes(MONTH) implies DAY<='31') and
DIGIT->includesAll(Set{DATE.at(1),DATE.at(2),DATE.at(3),DATE.at(4),
DATE.at(6),DATE.at(7),DATE.at(9),DATE.at(10)}) and
DATE.at(5)='-' and DATE.at(8)='-' and DATE.size()=10
[Exercise] Complete each of the following (partial) OCL terms showing placeholders ___ with missing items (values, terms or operation calls) so that a Boolean term evaluating to the Boolean value true arises.

Examples: OCL term (a): let I=___ in I+I = 42. The placeholder ___ can be substituted by 21 in order for the term to become true.

OCL term (b): let I=___ in I.___(2)=1. One placeholder substitution is: let I=3 in I.mod(2)=1. Another substitution is let I=5 in I.mod(2)=1. A third substitution is let I=1 in I.min(2)=1.

Two different placeholders may be substituted differently. In general, the solutions, i.e., the substitutions for the placeholders, are not unique. In other words, there may be more than one solution for each term.

___.___(3,5)='wxy'
___.size()=0
(__+___+___).size()=0
let V=___ in let W=___ in V.___(W).size=0
let V=___ in V.___(1,3)=V.___(3)+V.___(2)+V.___(1)
let V=___ in V.___()=Sequence{A',B',B',A'}
let V=___ in V.toUpperCase()=V.toLowerCase()
let V=___ in let W=___ in V+W=W*V and V div W = V/W
let \( V = \_ \_ \) in \( V . \text{round}() = V . \text{floor}() + 1 \)

let \( V = \_ \_ \) in \( V . \text{round}() = V . \text{floor}() \)

let \( V = \_ \_ \) in \( V . \text{abs}() = V . \text{abs}().\text{abs}() \)

let \( V = \_ \_ \) in let \( W = \_ \_ \) in \( V . \text{max}(W) = V . \text{min}(W) \)

let \( V = \_ \_ \) in let \( W = \_ \_ \) in \( (V \implies W) = (W \implies V) \)

let \( V = \_ \_ \) in let \( W = \_ \_ \) in \( (V \leftrightarrow W) \) and \( (V \text{ xor } W) \) and \( (W \text{ xor } V) \)

let \( V = \_ \_ \) in let \( W = \_ \_ \) in \( (V \implies W) = (\text{not false}) \)

let \( V = \_ \_ \) in let \( W = \_ \_ \) in \( (V \text{ and } W) = (V \text{ or } W) \)
let $V=2$ in let $W=2$ in $V+W=W*V$ and $V\div W=V/W$

let $V=0.5$ in $V\text{.round}()=V\text{.floor}()+1$

let $V=0.49$ in $V\text{.round}()=V\text{.floor}()$

let $V=0$ in $V\abs()=V\abs().\abs()$

let $V=0$ in let $W=0$ in $V\max(W)=V\min(W)$

let $V=true$ in let $W=true$ in $(V\implies W) = (W\implies V)$

let $V=false$ in let $W=true$ in $(V\neq W) \text{ and } (V\text{xor} W) \text{ and } (W\text{xor} V)$

let $V=false$ in let $W=true$ in $(V\implies W) = (\neg false)$

let $V=true$ in let $W=true$ in $(V\text{ and } W) = (V\text{ or } W)$
In OCL, the comparison operations less \(<\), greater \(>\), less-equal \(\leq\) and greater-equal \(\geq\) are not defined on the datatype Boolean. Define these operations either by (partly) filling in the following table or/and by (partly) defining these operations by other operations or terms like '\(V \ op \ W\) :\(\leftrightarrow\) 'TERM-OPTIONALLY-WITH-V-AND-W'.

<table>
<thead>
<tr>
<th>&lt;</th>
<th>&gt;</th>
<th>&lt;=</th>
<th>&gt;=</th>
</tr>
</thead>
<tbody>
<tr>
<td>false false</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>false true</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>true false</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>true true</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Give a justification for your decisions.

Alternative display of table:

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>A&lt;(B)</th>
<th>A&gt;(B)</th>
<th>A(\leq)B</th>
<th>A(\geq)B</th>
</tr>
</thead>
<tbody>
<tr>
<td>false false</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>false true</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>true false</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>true true</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>


A < B :<=> (A=false and B=true)
A > B  :<=> (B < A)
A <= B :<=> (A < B) or (A=B)
A >= B :<=> (A > B) or (A=B)

1. false is interpreted as 0, true as 1; similar to a bit value.
2. Operation less < should be symmetric to operation greater >.
3. Analogously to A <= B on numerical values: A < B or A = B.
4. Analogously to A >= B on numerical values: A > B or A = B.

In this solution, some general applicable fundamentals have been applied: Define as few as possible explicit rules for individual cases; define rules utilizing other already defined rules and utilizing universal principles, possibly from related situations.

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>A &lt; B</th>
<th>A &gt; B</th>
<th>A &lt;= B</th>
<th>A &gt;= B</th>
</tr>
</thead>
<tbody>
<tr>
<td>false</td>
<td>false</td>
<td>false</td>
<td>false</td>
<td>true</td>
<td>true</td>
</tr>
<tr>
<td>false</td>
<td>true</td>
<td>true</td>
<td>false</td>
<td>true</td>
<td>false</td>
</tr>
<tr>
<td>true</td>
<td>false</td>
<td>false</td>
<td>true</td>
<td>false</td>
<td>true</td>
</tr>
<tr>
<td>true</td>
<td>true</td>
<td>false</td>
<td>false</td>
<td>true</td>
<td>true</td>
</tr>
</tbody>
</table>
[Exercise] Construct 4 Real, 4 Integer, 4 String and 4 Boolean OCL terms (pairwise different) each containing at least 2 different operations (from all available operations). The 4 terms for each datatype have to yield the same result. Example: let I=42 in I, let I=42 in I+0, let I=42 in 0+I, let I=42 in 0+I+0.
[Exercise] In OCL the binary operations max and min are available on Real and Integer: `Real::max(r:Real):Real`, `Real::min(r:Real):Real`, `Integer::max(i:Integer):Integer`, `Integer::min(i:Integer):Integer`. Is there any advantage for defining max and min again on Integer? Or is this redundant and without any effect?

[Exercise] Construct an OCL term such that the term uses a Real, an Integer, a String and a Boolean operation or an operation with such a return type. Example: The used types in the term 'Lassie'.size() would be String and Integer.
**Exercise** Construct 12 pairs of syntactically different OCL terms (3 Real, 3 Integer, 3 String, 3 Boolean pairs) with appropriate variables such that the terms from the term pair are equivalent, i.e., evaluate to identical values for all substitutions for the variables.

Example pairs: I+0, 0+I for I:Integer; I+J, J+I for I,J:Integer
if $S \neq S\.toUpperCase()$ and $S\.size() = 1$ then $S\.toUpperCase().toLowerCase()$ else '' endif

B and C, C and B

B or C, C or B

B xor C, C xor B

B and (C and D), (B and C) and D

B and (C or D), (B and C) or (B and D)

false and B, false

true and B, B

false or B, B

true or B, true

false xor B, B

true xor B, \text{not} B

false implies B, true

true implies B, B

$S\.size() \geq 0$, true

$S\.size() = 2$ implies $S = S\.substring(1,1)+S\.substring(2,2)$, true

All Real, Integer, and String pairs (ONE, TWO) can be reformulated utilizing equality = as Boolean pairs (ONE=TWO, true). Example: (R+S, S+R) becomes (R+S=S+R, true). Sometimes the terms ONE and TWO can be even simplified and then be combined by equality in order to better show the essence of the equivalence. E.g.:

if $S\.size() = 1$ then $S\.substring(1,1)$ else '' endif,

if $S\.size() = 1$ then $S\.at(1)$ else '' endif

becomes:

$S\.size() = 1$ implies $S\.substring(1,1) = S\.at(1)$, true
**Exercise** Fill out the following matrixes evaluating the comparisons `<` and `=` between the given OCL terms. State whether the comparison is true with `T`, the comparison is false with `F` or the comparison yields a syntax error with `-.` Only the lower half and the diagonal have to be filled out.

<table>
<thead>
<tr>
<th></th>
<th>42</th>
<th>042</th>
<th>42.0</th>
<th>042.0</th>
<th>'42'</th>
<th>'042'</th>
<th>'42.0'</th>
<th>'042.0'</th>
</tr>
</thead>
<tbody>
<tr>
<td>42</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>042</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>42.0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>042.0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>'42'</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>'042'</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>'42.0'</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>'042.0'</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>42</th>
<th>042</th>
<th>42.0</th>
<th>042.0</th>
<th>'42'</th>
<th>'042'</th>
<th>'42.0'</th>
<th>'042.0'</th>
</tr>
</thead>
<tbody>
<tr>
<td>42</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>042</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>42.0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>042.0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>'42'</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>'042'</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>'42.0'</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>'042.0'</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
[Exercise] Consider the figure below and the table below that shows possible command lists for a construction of an object model in the USE class model discussed before. Each command list is trying to represent the circumstances from the figure.
1. Which of the command lists from (A)-(D) would be syntactically correct w.r.t. the USE class model? If errors during execution would occur, describe the nature of the error.

2. In your opinion, which elements of the command lists meet the circumstances presented in the previous figure most accurately?

3. How does one has to modify the USE class model in order to get the command list (E) getting accepted without syntax errors?
4. Define 6 additional, different Boolean attributes in the class Creature for the USE model, such that the attribute evaluations for the models { Lassie-Jeff, Tom-Jerry } show a wide spectrum of value combinations. For example:

<table>
<thead>
<tr>
<th></th>
<th>Lassie</th>
<th>Jeff</th>
<th>Tom</th>
<th>Jerry</th>
</tr>
</thead>
<tbody>
<tr>
<td>Att1</td>
<td>false</td>
<td>false</td>
<td>false</td>
<td>false</td>
</tr>
<tr>
<td>Att2</td>
<td>false</td>
<td>false</td>
<td>false</td>
<td>true</td>
</tr>
<tr>
<td>Att3</td>
<td>false</td>
<td>false</td>
<td>true</td>
<td>false</td>
</tr>
<tr>
<td>Att4</td>
<td>false</td>
<td>false</td>
<td>true</td>
<td>true</td>
</tr>
<tr>
<td>Att5</td>
<td>false</td>
<td>true</td>
<td>false</td>
<td>false</td>
</tr>
<tr>
<td>Att6</td>
<td>false</td>
<td>true</td>
<td>false</td>
<td>true</td>
</tr>
</tbody>
</table>

Ideally, the six lines with the false-true tuples show six different tuples, not necessarily the ones displayed.