WHITESTEIN
Technologies

Living Systems® Process Suite

Expression Language

Living Systems Process Suite Documentation

3.1
Tue Jan 12 2021

Whitestein Technologies AG | Hinterbergstrasse 20 | CH-6330 Cham
Tel +41 44-256-5000 | Fax +41 44-256-5001 | http://www.whitestein.com

Contents

Chapter 1

Expression Language

The LSPS Expression Language is a statically typed, functional language used in models of the Living Systems® Process Suite to compute and process values.

It is not a full-fledged programming language and it is not possible to define business models exclusively in the LSPS Expression Language; however the language supports your modeling efforts: you will typically define properties of GO-BPMN elements as expressions in the LSPS Expression Language.

For example, you design a BPMN process with a Log task (Log tasks log messages into the logs and possibly to the console). You define the log message as an expression that results in a String in the Log property:

```
"Process " + processID + " started."
```

This expression uses String literals "Process " and " started", and the variable processID. On runtime, the system fetches the value of processID and concatenates the strings. The resolved expression is then used as the log message. Note that the processID variable is not defined in the expression: the variable is a global variable, which is a model element that exists in a variable resource file (refer to Model Elements).

Chapter 2

Expressions

An expression is a combination of literals, data types, keywords, variables, function calls, operators, and calls to model elements that results in a single value; for example, `"Hello " + getWorld()` is a String expression that concatenates two strings: "Hello " and the string returned by the getWorld() function call. The getWorld() function is a model element and cannot be defined in the LSPS Expression Language; we assume, it returns the string "world!". The result of the expression is the string value `"Hello world!"`. The interpretation is ruled by the precedence rules and associations.

Every expression exists in its own namespace and hence context; for example, the message property of a Log task exists in its own namespace, which exists in the immediate parent namespace. The expression namespace is a local namespace. More on namespaces in GO-BPMN is available in GO-BPMN Modeling Language Specification.

An expression namespace can define its local variables that can be used within the expression namespace. Note that you cannot define entities like functions, or custom data types in the language directly. However, you can make use of functions, model variables, custom data types, etc. defined in the model.

2.1 Block

An expression block represents its own namespace (implicitly if, then, else represent their own expression blocks): if you create a variables in an expression block, you cannot access from outside of it.

To create an expression block explicitly, start the block with the `begin` keyword and finish it with the `end` keyword.

```
//declaration of a local visibility variable
def Boolean visibility :=
   //beginning a codeblock
   begin
     //declaration of a visibility variable in the codeblock (not available out of the block):
     def Boolean visibility := false;
   end;
```

Language constructs such as `if`, `switch`, `foreach`, `while`, `for` implicitly represent a block.

2.2 Literals

Literals represent fixed values. The notation of literals depends on the data type they represent. The notations are documented in Data Types.

Chapter 3

Data Types

The LSPS Expression Language is a statically typed language: all values must define their data type before they can be used on runtime. The data type determines what value the object can hold.

When you declare a local string variable (`def String s`), an object that can hold a reference to a string is created in the memory.

Data types can inherit properties from each other: one type becomes the supertype of another type: An object of a data type can be used anywhere where you can use its super type, for example, when assigning values, sending function arguments, etc. For example, you can assign a variable of the type Decimal also a value of the type Integer, since Integer is a subtype of Decimal. These relationships apply to built-in as well as custom data types in a hierarchy.

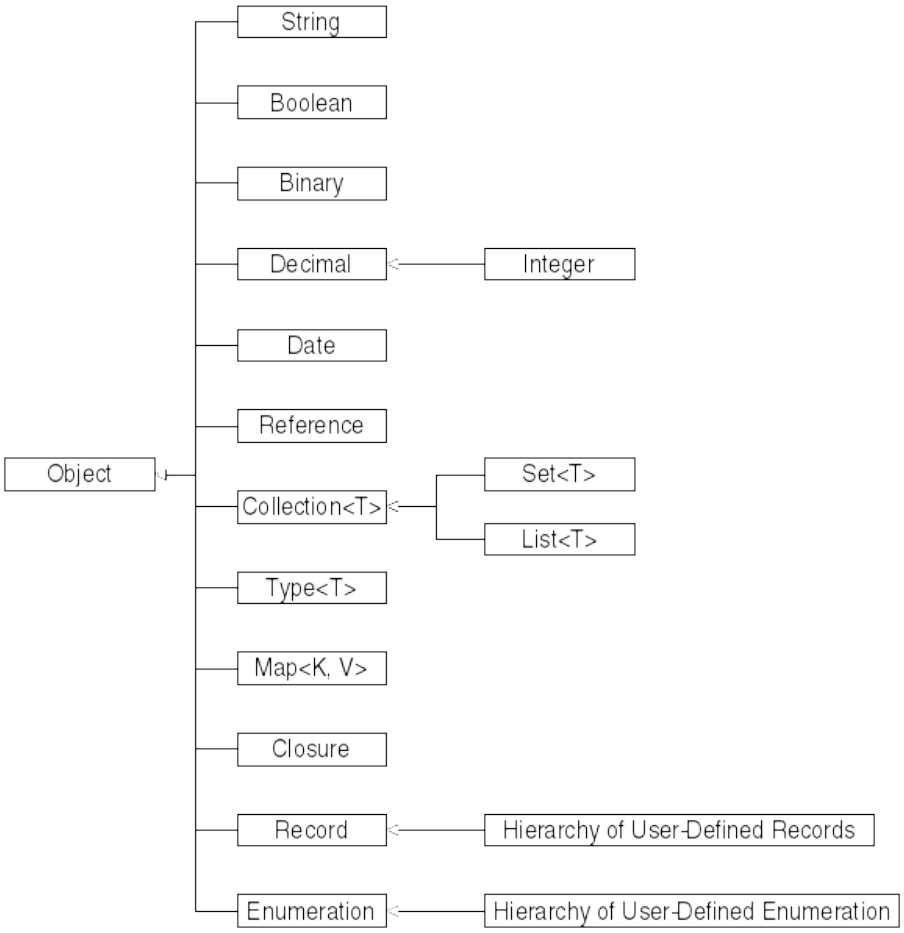

Figure 3.1 Built-in Data Type Hierarchy

All data types are part of the data type hierarchy with the Object data type as the root of the tree.

3.1 Data Type Mapping

The data types are implemented as Java or LSPS classes. This information is useful when you need to pass them to your Java code.

Expression Language Type	Class
String	java.lang.String
Boolean	java.lang.Boolean
Binary	com.whitestein.lsps.lang.exec.BinaryHolder
Decimal	com.whitestein.lsps.lang.Decimal
Integer	com.whitestein.lsps.lang.Decimal
Date	java.util.Date
Reference	com.whitestein.lsps.lang.exec.ReferenceHolder
Collection	com.whitestein.lsps.lang.exec.CollectionHolder
List	com.whitestein.lsps.lang.exec.ListHolder
Set	com.whitestein.lsps.lang.exec.SetHolder
Type	com.whitestein.lsps.lang.type.Type
Map	com.whitestein.lsps.lang.exec.MapHolder
Closure	com.whitestein.lsps.lang.exec.ClosureHolder
Record	com.whitestein.lsps.lang.exec.RecordHolder
Enumeration	com.whitestein.lsps.lang.exec.EnumerationImpl
Property	com.whitestein.lsps.lang.exec.Property
Object	java.lang.Object

3.2 Casting

Casting takes place when you change the type of a value to another type. When "widening" a type, that is changing a value of a subtype to its supertype, the type change occurs automatically. When "narrowing" a type, you need to cast the type explicitly:

```
<objectName> as <newObjectType>
```

```
person as NaturalPerson
```

> **Note:** Alternatively, you can use the `cast()` function of the Standard Library, for example, `cast(o, type(D))`

3.3 Object

The *Object* data type is the super type of all data types and therefore the only data type without a supertype. It is represented by *java.lang.Object*. Therefore if you want to allow any data type, for example, as a parameter, use the Object data type. It is represented by the *java.lang.Object* class. All objects can hold the `null` value.

3.4 Simple Data Types

Simple data types contain values without further data type structuring.

3.4.1 Binary

Objects of binary data type are typically used to hold binary data when downloading and uploading files, or working over binary database data.

It is represented by *com.whitestein.lsps.lang.exec.BinaryHolder*.

The Binary literals are not supported.

To define the type:

```
Binary
```

The Binary data type serves to work with binary data typically from another resource, typically, pictures, videos, etc.: You could define a Field of a Record as being of the Binary type and populate it with picture data.

3.4.2 String

A **String** holds a sequence of Unicode characters. The data type is implemented by the `java.lang.String` class.

To define the type:

```
String
```

To create an instance:

```
"Sequence of Unicode characters"
```

A String can contain special characters defined using their ASCII codes. For example, you can use the ASCII tab code (`#9`) to have a tab in your String, line feed (`#10`) to make a multi-line String, etc., for example:

```
"This" + #10 + "is" + #10 + "a" + #10 + "multi-line" + #10 + "string with "+
"multiple" + #10 + " new lines " + #10 + " which represent " + #10 + "line breaks."
```

To escape characters, use the double-quote (`"`) character:

```
"This is all one string: ""Welcome to String escaping!"""
```

To annotate a string that is not to be localized, add the hashtag # sign in front of the string.

```
#"This is a string value which will not be localized."
```

The hashtag # sign signalizes that the String is to remain unlocalized and that this was the intention of the developer. Such Strings are excluded from checks of unlocalized Strings.

To create a local variable:

```
def String s := "My String"
```

3.4.3 Boolean

Boolean objects hold the values `null`, `true` or `false`. It is implemented by the `java.lang.Boolean` class.

To define the type:

```
Boolean
```

To create an instance:

```
true
```

To create a local variable:

```
def Boolean s := true
```

The expression `def Boolean boolVar := true` declares and defines a local variable of the type.

3.4.4 Integer

Integer objects hold natural numbers and their negatives. The data type is a subtype of the Decimal type. It is represented by *com.whitestein.lsps.lang.Decimal*.

Note that the underlying Java type is BigDecimal; hence no relevant maximum limit applies to the value.

To define the type:

```
Integer
```

To create an instance:

```
-100
```

To create a local variable:

```
def Integer i := 42
```

3.4.5 Decimal

Decimal objects hold numerical fixed-point values. It is represented by *com.whitestein.lsps.lang.Decimal*.

> **Note:** A Decimal type is internally represented by two integer values: an unscaled value and a scale. The value is hence given as `<UNSCALED_VALUE>*10**<SCALE_VALUE>`. The integer scale defines where the decimal point is placed on the unscaled value. The scale is a 32-bit integer. If zero or positive, the scale is the number of digits to the right of the decimal point. If negative, the unscaled value is multiplied by ten to the power of the negation of the scale. Decimal values are, for example, 2e+12, -1.2342e0, 1.0.

When assigning a value of the type Decimal, you need to define the scale and rounding mode. The following rounding modes can be used on decimals:

- **UP**: rounds away from zero
- **DOWN**: rounds towards zero
- **CEILING**: rounds towards positive infinity
- **FLOOR**: rounds towards negative infinity
- **HALF_UP**: rounds towards the nearest neighbor unless both neighbors are equidistant, in which case it rounds up
- **HALF_DOWN**: rounds towards nearest neighbor unless both neighbors are equidistant, in which case it rounds down
- **HALF_EVEN**: rounds towards the nearest neighbor unless both neighbors are equidistant, in which case, it rounds towards the even neighbor
- **UNNECESSARY**: asserts that no rounding is necessary

The rounding mode is defined for decimal variables or for a record field of the type Decimal.

To define the type:

```
Decimal
```

To define a Decimal type with Scale 100 and Rounding Model UP:

```
Decimal(100, UP)
```

To create an instance:

```
-10.0;
6.63E-34
```

To create a local variable:

```
def Decimal intVar := 100
```

> **Important:** Decimal values are normalized if they contain 0 digits after the decimal point: decimal value 1.0 and integer value 1 have the same numerical value and therefore `1.0 == 1`.

3.4.6 Date

The **Date** object holds a specific time. It is represented by *java.util.Date*.

> **Important:** Only dates since the introduction of the Gregorian calendar, that is, the year 1582 are supported: for Dates that occurred before, a shift in days can occur rendering the date incorrect.

To define the type:

```
Date
```

To create an instance:

```
d'yyyy-MM-dd HH:mm:ss.SSS'
```

To create a local variable:

```
def Date myDate := d'2015-12-24 20:00:00.000'
```

> **Important:** When constructing a date, consider using the date functions from the Standard Library.

3.5 Complex Data Types

Complex data types are composite data types based on other data types.

3.5.1 Collections

A *Collection* is a groupings of items of a particular data type. It is represented by *com.whitestein.lsps.lang.exec.↩ CollectionHolder*.

Collections are ordered and immutable: once a collection is created, it cannot be changed, while you can change individual collection items. Each item of the collection represents an expression

To access an element of a Collection, you need to specify the position of the element in the List. Note that the first element of a List is on position zero. For example, `[10,20,30][1]` returns 20 as position 0 of this list points to the value 10, the first element of the list.

Hierarchy of collections follows the hierarchy of their elements: List<TA> is a subtype of List<TB>, if TA is a subtype of TB. Set<TA> is a subtype of Set<TB>, if TA is a subtype of TB.

To define the type:

```
Collection<T>
```

3.5.1.1 List

A **List** represents an ordered collection of items of a type with possible duplicate values. It is represented by the *com.whitestein.lsps.lang.exec.ListHolder* class.

To define the type:

```
List<T>
```

To create an instance:

```
["a", "b", "c", "d"]
```

List of lists

```
[["a", "b"], ["b", "c", "a"]]
```

To access an item in a List: Specify the position of the item in the List starting from 0: For example, `[10,20,30][1]` returns 20 as position 0 points to the value 10, the first element.

Unlike on Sets, accessing list items is performance-wise efficient.

3.5.1.1.1 Creating a List of Integers with the Range Operator

To create a List of Integers, you can use the `..` operator, the range operator:

```
1..5
//is equivalent to [1, 2, 3, 4, 5]

3..1
//is equivalent to [3, 2, 1]
```

3.5.1.2 Set

A **Set** represents an ordered collection of items of a type with no duplicate values. It is represented by the *com.↩ whitestein.lsps.lang.exec.SetHolder* class.

To define the type:

```
Set<T>
```

To create an instance:

```
//Set<Integer>:
{1,2,3};
```

To access an item in a Set: Specify the position of the item in the Set starting from 0: For example, `{10,20,30}[1]` returns 20 as position 0 points to the value 10, the first item.

Unlike in Lists, accessing items is performance-wise inefficient.

3.5.2 Map

Maps hold keys with their values and are immutable: once a map is created, it cannot be changed. You can change its key-value pairs, however, the map itself cannot be modified.

It is represented by *com.whitestein.lsps.lang.exec.MapHolder*.

To define the type:

```
//type map with the K type of its keys and the V type of its values:
Map<K, V>
```

Note that hierarchy of maps follows the hierarchy of its key and value types: Map<KA, VA> is a subtype of Map<KB, VB>, if KA is a subtype of KB and VA is a subtype of VB.

To create an instance:

```
[1->"a", 2->"b"]
```

To initialize an empty map, use the empty-map operator `[->]`:

```
def Map<Object, Object> myEmptyMap := [->];
```

To get a value of a key, specify the key for the appropriate value in square brackets, for example, `["firstKey"->"a", "anotherKey"->"b"]["anotherKey"]` returns the string "b".

3.5.3 Reference

A **Reference** holds a reference expression that resolves to a variable or a record field (slot), not their value.

A Reference is conceptually similar to pointers in other languages. However, a Reference points to a variable or a record field, not to memory slot.

- A Reference to a variable resolves to the variable object
- A Reference to a record field resolves to the record instance and the association path.

The data type is represented by *com.whitestein.lsps.lang.exec.ReferenceHolder*.

To define the Reference type:

```
Reference<T>
```

To create a Reference instance, use the & reference operator:

```
&<TARGET>
```

**To get the value in the referenced slot, use the * dereference operator.

Example Use of Reference

```
//creates new Patient record instance with the diagnosis "flu":
def Patient r := new Patient(diagnosis -> "flu");
~
//creates the local variable status that holds the reference to the diagnosis field of the Patient
def Reference<String> status := &(r.diagnosis);
~
//function sets the status to cured on the patient:
setStatusToCured(&r.status);
~
//implementation of the setStatusToCured() function:
//def Reference<String> status:= statusReference;
//*status:="cured";
```

3.5.4 Closure

A closure is an anonymous function that can declare and make use of local variables in its own namespace and use variables from its parent expression context as opposed to lambdas. It is represented by *com.whitestein.lsps.lang.exec.ClosureHolder*.

To define the type:

```
//Syntax: { <INPUT_TYPE_1>, <INPUT_TYPE_2> : <OUTPUT_TYPE> }
{ String : Integer}
//Closure that that has no parameters and returns an Object:
{ : Object}
```

Subtyping in closures is governed by their parameters and return type: Closure A is a subtype of closure B when:

- the return type of closure A is a subtype of the return type of closure B
- and parameter types of closure B are subtypes of parameter types of closure A.

{ S1,S2,... : S } is subtype of { T1, T2,... : T } when T1 is subtype of S1, T2 is subtype of S2, etc. and S is subtype of T.

To create an instance:

```
//Syntax: { <PARAMETERS> -> <CLOSURE_BODY> }
{s:String -> length(s)}
```

Parameter types can be omitted:

```
{s -> length(s)}
```

The system attempts to infer the type of closure arguments and its return value. Note that the types might be resolved incorrectly. To prevent such an event, consider defining the argument type explicitly as in the example.

To evaluate a closure use the parentheses `()` operator with the closure arguments:

```
def {Integer:String} closureVar := {x:Integer -> toString(x)};
def String closureResult := closureVar(3);
```

3.5.5 User-Defined Record

A user-defined **Record** is the subtype of the Record type. It serves to create custom structured data types. A Record can define `relationships` with another Records.

It is not possible to declare a Record type in the Expression Language: Records are modeled in the `data-type editor` of the Living Systems® Process Design Suite. However, you can create instances of Records. When you pass a Record, for example, as a function argument, it is passed by reference.

It is represented by *com.whitestein.lsps.lang.exec.RecordHolder*.

To create an instance:

```
new <RECORD>(<NAME_OF_FIELD> -> <FIELD_VALUE>, <NAME_OF_FIELD> -> <VALUE>, ...))
```

For example:

```
new Book(genre -> Genre.FICTION, title -> "Catch 22", author -> new Author(name -> "Heller, Joseph"))
```

Use the Dot operator to access Record fields or related records and fields.

Example: *def* declares a new variable of the MyRecord type. *new* creates a new MyRecord instance, which is assigned to the variable (the instance is the proper memory space with the record) . Variable r points to the My↩ Record instance and it is returned by the expression.

```
def MyRecord r := new MyRecord(recordField ->  "value")
```

Function call with a new Record instance as its argument:

```
acquireApproval(new Approval(outcome -> "permitted"))
```

To access fields of a Record, use the access operator `.`:

```
book.title
```

Note that the dot operator `.` fails with an exception if the `<EXPRESSION>` with the access operator is `null`. Use the safe-dot operator to prevent the exception.

When accessing a field or record that is of the type Reference, the property is automatically dereferenced. Therefore the expression `(*ref).fieldName` and `ref.fieldName` are identical.

3.5.6 Property

The **Property** data type holds the type of a Field in a user-defined Record.

Example

```
def Property authorNameField := Author.firstName;
```

3.5.7 Type

The **Type** data type holds data types, for example, a Type can hold the value String data type, a particular Map data type, a Record data type, etc. It is represented by *com.whitestein.lsps.lang.type.Type*.

The Type data type can be used to check if object are of a particular data type. The output can be then further used when Casting the object.

To define the type:

```
Type<T>
```

To create an instance:

```
//type is a keyword:
type(<TYPE>)
```

Example Usage

```
switch typeOf(person)
  case type(NaturalPerson) -> getFullName(person as NaturalPerson)
  case type(LegalPerson) -> getFullName(person as LegalPerson)
  case type(PersonGroup) -> getFullName(person as PersonGroup)
  default -> person.code + #" <Unknown type>"
end
```

3.5.8 Enumeration

An *Enumeration* is a special data type that holds literals. It is represented by *com.whitestein.lsps.lang.exec.EnumerationImpl*.

You cannot create an Enumeration type in the Expression Language directly: It is modeled as a special Record type.

To define the type:

```
<ENUMERATION>
```

To create an instance:

```
<enumeration_name>.<literal_name>
```

Example:

```
//Weekday is an enumeration with values MONDAY, TUESDAY, etc.
def Weekday wd := Weekday.WEDNESDAY
```

You can compare enumeration literals with the comparison operators =, !=, <, >, <=, >= for enumerations of the same enumeration type. Enumeration literals are arranged as a list of values; hence the comparison is based on comparing their indexes: the order depends on the order of the literal as modeled in the Enumeration.

3.5.9 Null

The Null data type signalizes an unspecified value: its only value is `null`.

The data type is the subtype of every other data type, so that any object can take the `null` value.

To define the type:

```
Null
```

Chapter 4

Operators

Operators are symbols that cause a particular action, for example, comparison of values, summing up, assignment, etc.

4.1 Arithmetic Operators

Arithmetic operators are used on Integer and Decimal values similarly to their use in algebra.

Operator	Description	Example	Result	Note
+	addition	`2 + 2`	4	
−	subtraction	`3 - 1`	2	
++	increment by 1	`intVar++` or `++intVar`		postfix: returns value; then increments by one and assigns it to the referenced object (`2++`); prefix: increments value by 1 and reassigns it to the referenced object
−−	decrements by 1	`intVar++` or `++intVar`		postfix: returns value; then decrements by one and assigns it to the referenced object (`2++`); prefix: decrements value by 1 and reassigns it to the referenced object
*	multiplication	`3 * 3`	9	
/	division	`9 / 3`	3	
%	modulo	`10 % 3`	1	
**	exponentiation	`3 ** 3`	27	Since expressions are evaluated primarily in the left-to-right manner unless precedence or association rules take over, the expression `2**3**4` is evaluated as 2^3^4^^; the exponent follows the ** operator

4.2 Assignment Operator

To assign a value to use the assignment operator `:=`:

```
//assign value 1 to x:
int x := 1
```

4.3 Logical Operators

Logical operators are used to combine multiple expressions that each return a boolean value. Combination of expressions with logical operators return a single boolean value.

Operators and their return values

`and`, `&&` (conjunction) true if both operands are true; otherwise false

`or`, || (inclusive disjunction) true when at least one of the operands is true; otherwise false

`xor`, exclusive disjunction true when the operands value is not identical, that is one operand is true and the other operand is false; otherwise false

`not` or `!` (negation) true if the operand is false; false if the operand is true (unary operator)

Truth Table

Operand A	Operand B	A and B	A or B	A xor B	not A
true	true	true	true	false	false
true	false	false	true	true	false
false	true	false	true	true	true
false	false	false	false	false	true

Since logical expressions are evaluated from left to right, the short-circuit evaluation is applied on `and` and `or` expressions:

- for `and` expressions:`<false> and <not_evaluated>` evaluates to `false`
- for `or` expression `<true> or <not_evaluated>` evaluates to `false`

4.4 Comparison Operators

Comparison operators serve to compare two values. Comparing returns a boolean value.

To compare two values, you can use the following operators:

- = or == (equal) and != (not equal, alternatively noted as <>) check if the values of any data type are equal.

```
def String varA:="value"
def String varB:="value"
varA=varB
//returns true
```

> **Note:** When comparing records, it is the object identity that is compared, *not* the record value. Analogously, on shared records, the record IDs are compared. However, on non-shared records, you can define fields or relationships so that the values of these are used when comparing records (refer to the Record Fields in the GO-BPMN Modeling Language Guide).

- < (lesser), > (greater), <= (lesser or equal), and >= (greater or equal) check if values of *decimals, integers, and strings* are lesser, lesser or equal, greater, or greater or equal and returns `true` or `false`.

  When comparing Strings, they are compared lexicographically as per Java lexicographic ordering of strings, for example, `"Čapek" > "Hemingway"` and `"Čapek" > "Asimov"` are true.

- `like` searches for an occurrence of a pattern in a string

  The operator supports the wildcards `?` for one character and * for one or multiple characters

  ```
  "matching exactly THIS word" like "* ??????? THIS *"
  ```

- <=> (the spaceship operator): checks if values are lesser, equal, or greater and returns `-1` if the left operand is lesser than the right operand, `0` if the left operand and right operand are equal, and `1` if the left operand is greater than the right operand.

  Applicable to the String, Decimal, Integers, and Date types (Date is a complex data type defined in the Standard Library)

  ```
  person1.name <=> person2.name
  ```

4.5 Concatenation

To concatenate two Strings, use the + concatenation operator.

```
varStringA+"String literal"
```

The String concatenation operator + can concatenate also a String and any subtype of the Object type: the Object type is converted to the appropriate String value and concatenated with the String; for example, a Date value is converted to a human readable date representation when concatenated with a string. Note that the String object must come first in such expressions.

String concatenation example If the first operands of the + operator is a string, the + operator is considered automatically the operand of concatenation and the non-string operand is automatically converted to a string.

```
"Timestamp: " + date("2014-07-31", yyyy-MM-dd)
//The date() function uses the DateTimeFormat implementation of the joda library
       (http://www.joda.org/joda-time/apidocs/org/joda/time/format/DateTimeFormat.html) and returns a Date object;
//the function is evaluated as if wrapper in toString: toString(date("2014-07-31", yyyy-MM-dd))
```

Resulting string:

```
Timestamp: Thu Jul 31 00:00:00 CEST 2014
```

> **Note:** If you want to keep a non-string operand before the string operand in concatenation, start your expression with the empty String literal `""`, for example, `"" + author.surname + ", " + author.firstname`.

4.6 Reference and Dereference Operators

The reference operator `&` returns Reference to a variable or a property path from a variable in the context of the variable. The dereference operator `*` takes a Reference value and returns the value currently stored in the referenced variable or property.

```
//instantiates record c1 with c1.name "Walter White":
def Partner c1 := new Partner(name -> "Walter White");
~
//reference variable ref with reference c1.name (current value "Walter White"):
def Reference<String> ref := &c1.name;
~
//variable x assigned dereferenced ref, that is "Walter White":
def String x := *ref;
~
//c1 assigned a new record instance with the name "Jesse Pinkman":
c1 := new Partner(name -> "Jesse Pinkman");
~
//new variable y assigned the dereferenced ref value (that resolves to "Jesse Pinkman"):
def String y := *ref;
~
//note that the value held by x remains "Walter White"
```

In the example the variable part is `c1` and the property part is `.name`. To acquire the value of a reference, use the dereference operator `*`.

References hold the referenced expression and the associated context. In our example the referenced expression is `c1.name`.

4.7 Inclusion Operator

The inclusion operator `in` checks if an element is in a set or list.

The check returns the String "1 is in mySet" if 1 is in mySet.

```
if 1 in mySet
     then "1 is in mySet"
end
```

4.8 Namespace Operator

The module namespace operator (`::`) is used to refer to elements of other module namespaces.

Consider ModuleA with a variable var imported into ModuleB. You can access var from ModuleB as follows:

```
ModuleA::var
```

The mechanism of module import is described in the *GO-BPMN Modeling Language Specification*.

4.9 Selector Operator

To access items in collections or maps, use the selector operator `[]` to specify the element to be returned:

on sets

```
name_of_set[element_position]
```

on lists

```
name_of_list[element_position]
```

Note that the first element of a List is on position zero. For example, `[10,20,30][1]` returns 20.

on maps

```
name_of_map[element_key]
```

4.10 Ternary Conditional

The ternary conditional operator `?:` enables you to define a condition and two expressions. If the condition evaluates to `true` the first expression is returned. If the condition evaluates to `false`, the second expression is returned.

With the operator you can write the expression

```
if <CONDITION> then <EXPRESSION_1> else <EXPRESSION_2> end
```

as

```
<CONDITION> ? <TRUE_EXPRESSION> : <FALSE_EXPRESSION>
```

4.11 Null-Coalescing Operator

The null-coalescing operator `??` is a more effective version of `if <expression_1> != null then <expression_2>` with `<expression_1>` evaluated only once.

```
<expression_1> ?? <expression_2>;
//is equivalent to:
if <expression_1> != null then
   <expression_1>
else
   <expression_2>
end
```

Example:

```
def String title := getTitle() ?? "Default Title";
```

4.12 Access Operator

The dot operator `.` serves to access fields of a Record, possibly via relationships. **Example:**

```
<EXPRESSION>.<FIELD>
```

For example:

```
book.title
```

Note that the dot operator `.` fails with an exception if the `<EXPRESSION>` with the access operator is `null`. Use the safe-dot operator to prevent the exception.

4.13 Safe-Dot Operator

To prevent the system from raising an exception when it attempts to access a record field of a record which is `null`, use the safe-dot operator `?.`.

Similarly to the dot operator, the `?.` operator serves to access Record Fields, possibly of related Records. Unlike the dot operator, no exception is raised when the record is `null`. The expression simply returns `null`.

```
<expression>?.field
```

Example:

```
def Person person := null;
//returns null without an error:
person?.email;
//you can also chain the access requests:
person?.contact?.operator?.operatorCallCode
```

4.14 Operator and Chaining Precedence

Generally, expression are evaluated from left to right.

Mathematical operators, logical, and relational operators follow their natural operator precedence:

1. unary before multiplicative before additive and
2. negation before relational before equality before exclusive disjunction before conjunction before disjunction.

Parentheses override operator precedence and the expression in parenthesis is evaluated as a whole.

Operator precedence order

1. `::` (scope operator)
2. `[]` (selector), `()` (function/closure call), `.` (dot operator), `?.` (safe dot operator)

3. +, −, & (reference), * (dereference)
4. ?? (ifnull)
5. * (multiplication), / (division), % (modulo), ** (exponentiation)
6. + (addition), − (subtraction)
7. <, >, <=, >=, <=>
8. = and == (equal), != (not equal), <>, `like`, `in` (inclusion)
9. `cast`, `as`
10. `not` and ! (negation)
11. `and` and && (conjunction)
12. `or` and ||, `xor` (disjunctions)
13. `?:` (ternary if)
14. := (assignment)
15. ; (chaining operator)

4.15 Evaluation Order

The order of expression evaluation at runtime is generally from left to right, that is, first the left-hand operand is evaluated and only then the right-hand operand is evaluated. Note that operator precedence can influence the evaluation order.

4.15.1 Chaining Expressions

To chain multiple expression, divide them with a semi-colon or a new line. A chained expression returns the return value of the last expression in the chain. Intermediary return values of the other chained expressions are ignored.

```
def String varString; varString := "Hello" //two expressions chained by a semi-colon (;)
varString:=varString + "World" //an expression chained by a new line
```

Chapter 5

Comments

Characters in code marked as comments are not interpreted on runtime and serve to provide information about the code.

To comment out a single like use the // characters: anything following the // characters until the end of the line is considered a comment.

```
uiCreateBook::createBook() //This is a comment.
//The book is defined as a shared record.
```

To comment out multiple lines, mark the start of the comment with the /* symbols and finish it with */

```
/* Multiline
   code
   comment */
```

Chapter 6

Local Variables

Local variables are created as part of an expression or an expression block and cannot be accessed from outside of it. However, from within an expression you can refer to any variable that exists in the scope of the expression and its parent scopes.

```
def String upperVar := "1";
begin
 def String lowerVar := "2";
 upperVar := "3";
end;
//this is not correct:
//lowerVar := "4";
```

To create a local variable use the `def` keyword in an expression. Note that `def` only declares the variable:

```
def <VARIABLE_TYPE><VARIABLE_NAME>
```

The variable value when declared is `null`, which is returned as its value.

To assign a value to a variable, use the assignment (`:=`) operator: such an expression *returns the right-hand-side value* of the assignment.

```
def String varString := "This is my variable value."
//returns "This is my variable value."
```

> **Note:** In models, you can define also global and local variables. Global variables are accessible from the entire Model; local variables within the given resource, such as, a process or form. Mind you cannot create global or local variables in the Expression Language; these are created in dedicated `model resources`.

Chapter 7

Function Calls

Function calls are calls to function definitions which are special kinds of closures defined in a function definition: Function definitions are model elements which cannot be created directly in the Expression Language; however, you can call functions and use their return value in your expressions.

A function call follows the syntax

```
<FUNCTION_NAME>(<COMMA_SEPARATED_ARGUMENTS>)
```

or alternatively

```
<FUNCTION_NAME>(<PARAMETER_NAME_1> -> <ARGUMENT_1>, <PARAMETER_NAME_2> -> <ARGUMENT_2>)
```

Example function call:

```
getModel("Delivery", 1.4)
//alternatively:
getModel(name -> "Delivery", version -> "1.4")
```

If a function uses `type parameters`, their types are inferred. However, you can define the types explicitly if required:

```
<FUNCTION> | <COMMA_SEPARATED_TYPES_PARAMETER_TYPES> | (<ARGUMENTS>)
```

The list of types in <COMMA_SEPARATED_TYPES> is used in the same order as the type parameters are defined. Note that you need to define the types for all type parameters.

A function call is resolved into the function based on the call arguments: overloading is supported.

Call to a Standard Library function with the types of type parameters:

```
//collect has the E and T type parameters:
//E will be handled as Employee and T as Decimal:
sum(collect|Employee, Decimal|(e, {e -> e.salary}))
```

Chapter 8

Controlling Flow

8.1 Branching

Branching serves to accommodate different reactions depending on a particular condition.

You can perform branching using the appropriate `if` construct or a `switch`. Note that the constructs represent an expression block.

8.1.1 if-then-end

The *if-then-end* returns the value returned by the `<expression>` if the Boolean_expression is true and `null` if the Boolean_expression is false.

```
if <boolean_expression> then
  <expression>
end
```

Example

```
//sendInfo is a Boolean variable.
if sendInfo then
  "Do send the newsletter."
end
//if sendInfo is false, the expression returns null (consider exception handling).
```

8.1.2 if-then-else-end

The *if-then-else-end* returns the value returned by `<expression_1>` if the Boolean_expression is true and value returned by `<expression_2>` if the Boolean_expression is false.

```
if <boolean_expression> then
  <expression_1>
else
  <expression_2>
end
```

Example:

```
//passedTest is a Boolean variable.
if passedTest then
  "Passed"
else
  "Failed"
end
```

8.1.3 if-then-elsif-end

- If the `boolean_expression_1` evaluates to false, `boolean_expression_2` is checked.
- If `boolean_expression_2` is true, `expression_2` is evaluated, and the evaluation leaves the if construct.
- If `boolean_expression_2` is false, the next *elsif* expression is checked, etc. If none of the *elsif* Boolean expression is true, `expression_N` is evaluated, and the evaluation leaves the if construct.

```
if <boolean_expression_1> then
  <expression_1>
  elsif <boolean_expression_2> then
    <expression_2>
end
```

Example

```
//passedTest is a String variable.
if passedTest=="yes" then
  "Passed"
  elsif passedTest=="no" then
    "Failed"
end
```

8.1.4 if-then-elsif-then-else-end

```
if <boolean_expression_1> then
  <expression_1>
  elsif <boolean_expression_2> then
    <expression_2>
  else
    <else_expression>
end
```

Example:

```
//passedTest is a String variable.
if passedTest=="yes" then
  "Passed"
  elsif passedTest=="no" then
    "Failed"
  else
    "Did not attend"
end
```

8.1.5 switch

The `switch` construct branches the execution flow based on condition value: it compares the argument expression against multiple possible values. If the value of the argument expression matches the value of the case, the expression defined for that case is executed and the `switch` returns the value of the expression. Unlike in Java, every case expression has an implicit `break`.

You can define a `default` expression that is executed if none of the cases matches is executed.

```
switch month
    case "January" -> 1
    case "February" -> 2
    default -> "Not January nor February"
end
```

8.2 Looping

Looping serves to repeat the same or similar action.

Note that the looping constructs represent an expression block block.

8.2.1 for

The *for* loop, server to loop through a block of expressions, until a condition becomes true.

```
for(init; condition; update) do
   expression
end
```

Example

```
def Integer i := 0;
for ( i; i < 10; i++) do
     debugLog({-> toString(i)}, 1000)
end
```

> **Important:** Collecting results of `foreach`, `for`, and `while` in a collection so that you create a new collection on each iteration as below, is inefficient and can cause performance issues (consider that collections are immutable):
>
> ```
> def Integer i := 0;
> def Set<Integer> varSet := {};
> for ( i; i < 10; i++) do
>   //creates a new set with the i added and assigns it to varSet:
>   varSet := add(varSet, i);
> end
> //varSet will contain { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 }
> ```
>
> Use `collect()`, `fold()`, `exist()`, `forAll()`, etc. of the Standard Library instead. For example:
>
> ```
> collect(1..10, { x:Integer -> new Option(label -> "Option " + x, value -> "Value " + x) })
> //instead of:
> //def List<Option> options := [];
> //foreach Integer x in 1..10  do
> //   options := add(options, new Option(label -> "Option " + x, value -> "Value " + x) )
> //end;
> //options;
> ```

8.2.2 foreach

To iterate through items in a collection, use `foreach`:

```
foreach <type> <iterator_name> in <collection> do
    <expression>
end
```

Example:

```
def Set<Person> persons := { ... };
~
foreach Person person in persons do
  sendEmail("Important Notification", "", {} , {person.email}, {}, {}, "UTF-8");
end
```

8.2.3 while

To loop code while an expression is true, use the `while` construct:

```
while <boolean_expression> do
    <expression>
end
```

8.2.4 break

In *while*, *for*, and *foreach* loops, you can use the *break* keyword to finish the looping immediately and continue with the next expression.

```
def Integer i := 0;
for ( i; i < 10; i++) do
  if i = 3 then
    break;
  end
end
```

8.2.5 continue

In *while*, *for*, and *foreach* loops, you can skip the current loop with the *continue* keyword.

```
def Set<Person> persons := {};
~
foreach Person person in persons do
  if isEmpty(person.email) then
    continue;
  end;
  sendEmail("Important Notification", "", {} , {person.email}, {}, {}, "UTF-8");
end
```

Chapter 9

Exception Handling

On runtime, code can cause an error that halts the execution and potentially terminate the execution unexpectedly. Typically, this can occur on user input, when the user input is unexpected (for example, while the code expects a Decimal value and the user enters a value with letters). Exception handling enables you to deal with such situations and handle the thrown exception gracefully.

To handle an exception, use the `try-catch` construct on the code.

Also, you can decide that a particular expression should produce an exception. To throw an exception, use the `error()` function from the Standard Library.

9.1 Throwing Exceptions

The `error()` function throws an error with an error code. The error code is a String parameter of the construct.

```
error(<errorcode>)
```

Example:

```
error("InvalidISBNFormat")
```

If an error exception is not caught and handled, the execution terminates. The error can be caught and handled by a try-catch block.

> **Important:** Error throwing functions are part of the Standard Library. Refer to the Standard Library Reference for further information.

9.2 Catching Exceptions

To catch and handle error exceptions without interrupting the execution, use the try-catch block on the code which might cause an exception:

```
try <expression>
    catch <error_code>, ... -> <handle_expression_1>;
    handle_expression_2;
    ...
end
```

Example:

```
try getCode()
  catch "Invalid ISBN format", "Invalid ISSN format" -> "Code value is not valid."
end
```

If `catch` takes `null`, any error is caught. Note that the block returns an object and you might need to cast it as appropriate:

```
try val.toDecimal()
catch null -> "not decimal" as String
```

9.3 Built-in Errors

The *Expression Language* makes use of the errors with the following error codes:

Error	Description (occurrence circumstances)
∗NullParameterError∗	A mandatory parameter has the `null` value.
∗IncompatibleTypeError∗	The type of processed value is incorrect (typically on casting or assigning).
∗ArithmeticError∗	An operand in an arithmetic operation cannot be processed.
∗OutOfBoundsError∗	The collection element does not exist.
∗FormatError∗	The format of an argument is incorrect (for example, on casting of a String).
∗WrongSizeError∗	The size of a value is incorrect (for example, a String being cast to a map).
∗NoSuchPropertyError∗	The record property does not exist.
∗DoesNotExistError∗	The entity does not exist.
∗RecordNotFound∗	The record instance does not exist.
∗ReferenceNotFound∗	Dereferencing failure (Referenced value was not found.)
∗ModelInstantiationError∗	Instantiation of a model failed.
∗ModelInterpretationError∗	Model cannot be interpreted.
∗SendingError∗	Error sending failed.
∗SendingSignalError∗	Signal sending failed.
∗MergeEvaluationError∗	Evaluation context cannot be merged (refer to Forms User Guide).
∗BinaryDataError∗	Binary data cannot be retrieved (for example, from database).
∗ReadOnlyAccessError∗	The system attempted to write to a read-only object.
∗IncorrectPathname∗	The string with the path in invalid.
∗NoExternalRecordProvider∗	The resource with the requested external record is not available.
∗AmbiguousNameError∗	The provided name cannot be resolved to a unique entity.

Chapter 10

Model Elements

Expressions can access named elements defined in the model, which cannot be defined directly in the Expression Language.

Such model elements include the following:

- **Modules** represent a structuring unit similar to a package and contain all the resources with model elements. A module can use resources of another module only if it imports the module–similarly to packages in Java. The module importing mechanism is described in the GO-BPMN Modeling Language guide.

  If you want to reference an element from another module, the name of the entity must be preceded by the module name and the `::` operator.

  ```
  <MODULE_NAME>::<ELEMENT_NAME>
  ```

- **Functions** are defined in a function definition file in a module.

  To call a function from an expression, use the following syntax:

  ```
  <FUNCTION_NAME>(<PARAMETER_1>, <PARAMETER_2>)
  ```

  Note that parameters can be themselves expressions that return the data type defined as the parameter type. If a function requires a parameter of the String data type, the parameter can be an expression that results in a String object.

  ```
  getName(getProcessName()+"#"+getId())
  ```

  > **Note:** Other elements are implemented as functions and are called in the same way. This includes queries and forms.

- **Variables from parent namespaces** are defined in a variable definition file, or directly on the elements representing the parent namespace, for example, a sub-process, a form, etc.

  Variables from parent namespaces are referred to by their name with no special notation.

  > **Note:** When referencing variables from imported modules, use the namespace operator (`::`). Further information on variables is available in the GO-BPMN Modeling Language Specification.

- **Records and their fields** are defined in a data type definition file.

  Records are referred to by their name with no special notation. To access record fields, use the dot operator: `<RECORD_NAME>.<RECORD_FIELD>`

- **Constants** are named values of a basic data type, the enumeration data type, or maps of the these data types. After their value has been initialized, it remains unchanged during the rest of the runtime. Initialization expressions of constants can use other constants.

  Constants cannot be defined in the Expression Language directly. To call a constant in an expression, use its name, for example, `"This is the current date format:  " + DATE_FORMAT`.

Chapter 11

Reserved Words

The following words are reserved words intended for future use:

- `public`
- `private`
- `repeat`
- `until`
- `protected`
- `this`
- `super`
- `final`
- `abstract`
- `return`
- `static`
- `void`