SMART Notation 3.0
Language Reference Manual

1 Introduction

1.1 About the language

SMART Notation 3.0 (or SN3) is a requirements specification language that can be used to specify the functional requirements of information processes and systems in an efficient, easily understandable and unambiguous way.

SN3 is part of the SMART Requirements 3.0 method for requirements analysis and specification, which also includes a method and tool for automatically/FF transforming these specifications into a set of business, user and system processes. These in turn can be transformed into a fully operational system that can execute these processes and thus achieve the specified functionality.

This document defines the formal requirements language SMART Notation. It specifies the syntax and semantics of the language, and includes usage notes about what syntactic constructs are and aren't allowed, and how they should be used to achieve the desired effect in a process specification.

1.2 About this document

This document is a reference manual, in other words a complete and rigorous description of all aspects of the language SN3, intended as the definitive specification of the language and structured for quick look-up by knowledgeable readers. It is not intended to be a learner's guide. The reader is expected to have a basic understanding of the role of dependencies and of GEARS admin features.

Each language construct is explained in a section that starts with a specification of the syntax using EBNF. The SN3 syntax is by design overly permissive (for reasons of efficiency) and allows various constructs that are not actually allowed in the language. Additional notes and examples are provided to further restrict and clarify the constructs, and to explain what they mean and how they should be used.

Syntax sections have the following format.

```
    requirementsSpecification =
        processSpecification | reusableSpecification ;

#   processNameLiteral = [a-zA-Z0-9 -]+ // enclosed in single quotes
```

Since the syntax is divided over multiple sections, each section may make use of constructs that are defined in other sections. Trivial definitions however (e.g. of separators or literals) are provided on the spot (and, therefore, redundantly), marked by a # symbol. These redundant definitions are given for reading convenience; the formal definitions are grouped in chapter 7 "Generic constructs".

In the **Semantics and usage notes** subsections, wherever the word *must* is used in italics, this implies that if this condition is not met, the construct is grammatically incorrect. When analysed by the process generation engine, such a construct would result in an error.

SMART Notation **Examples** are given in the following format, with SN3 specification snippets in the grey area above the horizontal line and explanations in italics below.

```
for each CUSTOMER in "customers who have placed an order in the past month" applies:
    "an invoice has been sent to {CUSTOMER}"
```

❖ *Specifies that invoices are to be sent to exactly those customers who placed an order in the last month.*

1.3 Contents of this document

Note about navigating this document: The following Table of Contents links to each of the chapters and top-level sections. Each section and chapter starts with a ⇧ symbol, which is also an internal link. In the top-level sections, these links point to the start of the chapter; and in the chapters, they link back to this Table of Contents.
In addition, the standard Google Docs document outline, to the left of the page, can also be used to navigate the document.

2 Organisation of SMART Notation specifications

2.1 Requirement definitions and specifications

🛈 In SMART Notation there are three types of requirements.

1. **Process result definitions**: describe the results of a process.
2. **Sentence definitions**: describe partial results (with result expressions) and calculations of values (with value expressions).
3. **Entity definitions**: describe entities and their attributes.

We use the term **reusable definitions** as a common name for sentence definitions and entity definitions.

When writing down the requirements of an information system, we speak of **specifications**. A specification is basically a group of definitions: we distinguish three types of these groups.

1. **Process specification:** A group of definitions that contains a process result definition.
2. **Reusable specification:** A group of definitions that contains only reusable definitions.
3. **Views specification:** A group of view definitions, where a view definition specifies how an entity can be displayed on the front-end.

Specifications in SMART Notation are usually stored in files, which use the extension `.sn`. Note that the requirements of an information system will generally consist of multiple process specifications, along with any number of reusable sentence definitions and entity definitions. The totality of all these specifications describing an information system is called a **project**.

2.1.1 Syntax

```
requirementsSpecification =
    processSpecification | reusableSpecification | viewsSpecification ;
```

2.2 Process specifications

🛈 A process specification is a group of requirement definitions that contains a process result definition, and may additionally include any number of reusable definitions. A process specification always contains exactly one process result definition, and always starts with this process result definition.

2.2.1 Syntax

```
processSpecification =
     processResultDefinition { definitionSeparator reusableDefinition } ;

processResultDefinition = processHeader definitionSeparator processBody ;
```

```
    processHeader =
        "process" processNameLiteral
        { "with key" processKey
        | "startable by" actorName { "," actorName }
        | "with actor" actorMappingClause { "," actorMappingClause }
        | "with subject" tupleExpression
        | "description" descriptionLiteral
        } ;

    actorMappingClause =
        actorName "with role" roleNameLiteral ;

    processBody =
        "results in:"
        bodySeparator
        resultExpression
        [ definitionSeparator withClause ] ;

#   processNameLiteral = // string of length > 0 enclosed in single quotes;
                         // no placeholders, newline/tab characters not allowed
#   roleNameLiteral = [a-z][a-z0-9_]+  // enclosed in single quotes; no placeholders
#   processKey = name { "." name }  // lower case only
#   actorName = name  // upper case only
#   descriptionLiteral = // an HTML clob enclosed in single quotes; no placeholders
#   definitionSeparator = // one or more newlines (no indent)
#   bodySeparator = // one or more newlines with increase of indent level
```

2.2.2 Semantics and usage notes

The processHeader contains information that is used to control and administer the generated process. The process header always starts with the keyword `Process` followed by a process name. A process name is generally chosen to be a unique name, but this is not mandatory: only the process key *must* be unique (see below). In addition, the process header may contain the following clauses. Each of the following clauses *must* appear 0 or 1 times (i.e., no clause may appear more than once).

- The processKey in the **`with-key`** clause contains a unique identification of the process. Syntactically this is made up of a sequence of names, separated by periods (e.g., `a.b.c`). The names form a hierarchy: in the example, the name `c` would refer to the process (in the context of `a.b`), while `a.b` and `a` are clusters of processes to which `c` belongs.
  It is important to note that the process keys in the complete set of process specifications of a project *implicitly* define a hierarchical structuring of system functionality: in other words, there is no separate (*explicit*) specification of this hierarchy.
  The complete key (the entire string of names and periods) *must* be unique. Also, a string of names that identifies a cluster in one process key (e.g., `a.b` in the above example) *cannot* also

be used as the key of another process.
If the key is omitted, the generator will derive a key from the process name (see § 2.12). Derived keys are always simple names (not sequences of names); consequently, if all processes in a given project do not have a process key, the above-mentioned hierarchy will be flat.

- The actorName (or Names) in the **`startable-by`** clause indicate(s) the category/ies of users who may initiate a new instance of the process. Normally, if this clause is not present, a process can be started by the actor (or actors) associated with the first step(s) in the generated process diagram, if any. The startable by clause supersedes this arrangement, and allows only the named actor(s) to start the process. This can also be used to make a process startable by a specific actor if it has no (human) input roles at all (autonomous process): in the absence of a startable-by clause, an autonomous process can only be started by users with admin privileges.

- The **`with-actor`** clause allows to specify process-specific actor names and map them directly to system roles. This can be used when it is necessary to distinguish between different *process roles* for several actors that have the same *system role*. See § 2.8 for details about user, actors and roles.
  A process-specific actor name *must* be distinct from any existing system role.

- The **`with-subject`** clause contains a tuple expression, which consists of one or more key-value pairs that can be used to hold identifying information about a running instance of a process. This information is intended to allow a user to distinguish between different instances of the same process.

- The **`description`** keyword introduces a free-format text explaining what the process does: this serves as additional documentation. The description may contain HTML markup.
  The description is shown in the generated documentation, but does not affect the generated processes.

The processBody consists of a result expression (cf. Ch. 3) which specifies the result the process needs to achieve, optionally followed by a with clause (cf. § 2.5).

The processResultDefinition may be followed by any number of reusable definitions; see § 2.4.2 and § 6.2.2 for scope considerations.

2.2.3 Examples

```
process 'Register leave of absence'
   with key hr.employees.register_leave
   startable by EMPLOYEE, SECRETARY
   with subject { 'Employee'   : EMPLOYEE.name
                , 'Start date' : LEAVE_REQUEST.start_date }

description 'With this process an employee can request a leave of absence which is
reviewed by a manager <b>if the employee is old enough</b>. <p> The leave request can
also be submitted by a secretary for the employee.'
```

```
results in:
   "employee has requested leave of absence"
   and
   if "{EMPLOYEE} is currently old enough"
   then
       "manager has reviewed the request"
```

- *The above example specifies a process for submitting and handling a request for a leave of absence.*
- *This process is part of the cluster 'hr', and within 'hr' it is part of the cluster 'employees'. Within the cluster Employees the process is known as 'register_leave'; another process in the same cluster could be 'register_sickness'. Another cluster under 'hr' could be e.g. 'contractors'.*
- *The generated process will (presumably, depending on the details) specify that it is normally started by an EMPLOYEE; in this example, explicitly, both a SECRETARY and an EMPLOYEE are allowed to start it. Note that the EMPLOYEE has to be explicitly named in the startable by clause, otherwise he wouldn't be able to start the process.*
- *In a user's task list, tasks that are available to the user are displayed using the name of the process, the subject, and the name of the task: for instance, for the above example, this could be something like* "Register leave of absence; Employee: John Doe; Start date: 1 July 2018; Review request"*. If there are dozens of active instances of the same process, the subject (consisting here of employee name and start date) allows the user to distinguish between them and pick up the task she wants to pick up.*

2.3 Reusable specifications

ⓘ A reusable specification is a group of requirement definitions that contains only sentence definitions and/or entity definitions.

2.3.1 Syntax

```
    reusableSpecification =
        reusableSpecificationHeader definitionSeparator reusableDefinitions ;

    reusableSpecificationHeader =
        "reusable definitions" ;

    reusableDefinitions =
        reusableDefinition { definitionSeparator reusableDefinition } ;

    reusableDefinition = sentenceDefinition | entityDefinition ;

#   definitionSeparator = // one or more newlines (no indent)
```

2.3.2 Semantics and usage notes

Reusable definitions in a reusable specification have global scope, i.e., they can be referenced in all processes that are part of the system.

As a matter of style, it is generally recommended to keep sentence definitions and entity definitions separate (i.e., in separate reusable specifications, i.o.w. in separate files).

2.3.3 Examples

```
reusable definitions

"{PERSON} is currently old enough" =
   "{PERSON} is old enough on {currentDate()}"
```

❖ *The above example contains a single definition in a reusable specification.*

2.4 Sentence definitions

ℹ Sentence definitions are used to structure and reuse functionality.

In order to be able to reuse a requirement, it is given a name, which is called a sentence name. What this sentence name “means” is made explicit in a sentence definition. When this requirement is needed in another requirement definition, its sentence name can be used in the other definition: this usage of a sentence is called a sentence reference (see § 2.6).

2.4.1 Syntax

```
      sentenceDefinition =
         sentenceName
         [ "description" descriptionLiteral ]
         "="
         expression
         [ definitionSeparator withClause ]
         ;

      sentenceName = '"' { sentenceNamePart | formalParameter } '"' ;

      formalParameter = '{' parameterName '}' ;

#     sentenceNamePart = // a string of Unicode characters,
                         // excluding double quotes ("), newlines, braces ({}), and tabs
#     parameterName = name  // either upper-case only, or lower-case only
#     descriptionLiteral = // an HTML clob enclosed in single quotes; no placeholders
#     definitionSeparator = // one or more newlines (no indent)
```

2.4.2 Semantics and usage notes

A sentenceName is any sequence of characters between double quotes, optionally containing one or more parameters. Parameters in a sentence name are formal parameters: they are represented by a simple name. A sentence name must contain at least one character: “” is not a valid sentence name, nor is “{ORDER_LINE}”.

The **description** is a free-format text explaining what the process does: this serves as additional documentation. The description may contain HTML markup.

The scope of a sentence is determined by the specification in which the definition is placed. Sentences that are defined in a process specification (cf. § 2.2) can be (re)used by any requirement definition used in that process. Sentences that are defined in a reusable specification (cf. § 2.3) can be (re)used from any process or requirement definition in any process in the entire project.

A sentence name must be unique within its scope i.e., the same sentence name cannot be used twice in a single process specification, and the same sentence name cannot be used twice in the collection of all reusable specifications for a project. It is therefore allowed to use a sentence name in a process specification even if it also exists in a reusable specification, and it is also allowed to use the same sentence name in two different process specifications. See § 2.6 for details and precedence rules, also for a definition of when two sentence names are considered the same.

Sentence definitions may not contain recursive references; i.e., a sentence definition may not contain a direct reference to itself, but also it may not contain an indirect reference, as in two sentence definitions referring to each other.

Formal parameters in the sentence name can only be used within the sentence definition itself. If a formal parameter is the same as a name in the global scope, then it is interpreted as the formal parameter within the sentence definition.

The definition of the sentence consists of an expression, which may be either a result expression or a value expression.

A with-clause may optionally be given at the bottom of the sentence definition, which defines any local sentences (called “inner sentences”) used within the expression defining the sentence. With-clauses are described in § 2.5.

2.4.3 Examples

```
"employee has requested leave of absence"
description 'This requirement describes the result of an employee requesting a leave of
absence. This results in the creation of a new element LEAVE_REQUEST.'
=
   one LEAVE_REQUEST in REQUESTS is created with:
       start    =           input from EMPLOYEE
       end      = optional  input from EMPLOYEE
```

```
      reason   =           input from EMPLOYEE

"manager has approved the request" =
   LEAVE_REQUEST.approved =  input from MANAGER
                             based on {LEAVE_REQUEST.start,
                                       LEAVE_REQUEST.end,
                                       LEAVE_REQUEST.reason}
```

❖ *The above are examples of sentence definitions defining result expressions.*

2.5 With-clauses

A with-clause can be used at the bottom of either a sentence definition or a process result definition. It contains local definitions of sentences that are used as parameters or as values in other expressions in the enclosing sentence definition or process result definition. Such sentences are called inner sentences. These local definitions mostly serve to improve readability.

2.5.1 Syntax

```
     withClause = "with:" bodySeparator innerSentenceDefinitions ;

     innerSentenceDefinitions = innerSentenceDefinition
                              { sequenceSeparator innerSentenceDefinition } ;

     innerSentenceDefinition = sentenceName "=" valueExpression ;

#    sentenceName = // any sequence of characters between double quotes,
                    // optionally containing one or more parameters (cf. § 2.4)
#    bodySeparator = // one or more newlines with increase of indent level
#    sequenceSeparator = // one or more newlines with same indent level
```

2.5.2 Semantics and usage notes

A with-clause is used to present local definitions of value expressions. The inner sentence definitions contained in a with-clause are syntactically similar to sentence definitions, but with important restrictions.

- Inner sentence definitions have an indent level of 1 (i.e., they do not start at the beginning of a new line, but are indented).
- Inner sentence definitions cannot have a description clause.
- Inner sentence definitions cannot have a with-clause.
- Also, the body of an inner sentence definition cannot be a result expression, it must be a value expression.

An inner sentence definition has a very limited scope: the associated sentence name can only be used directly in the definition where its with-clause is located. See also the examples below for details.

2.5.3 Examples

```
"leave of absence for {EMPLOYEE} is not blocked" =
  not "{EMPLOYEE} is blocked by {"department head HR or Legal for employee unit"}"
  and
  not "{EMPLOYEE} is blocked between {LEAVE_REQUEST.start} and {LEAVE_REQUEST.end}"
with
  "department head HR or Legal for employee unit" =
      EMPLOYEE.MANAGER.MANAGER[dept in ['002', '007'] and funct == '01']]

"{EMPLOYEE} is blocked by {MANAGER}" =
   // body of another sentence definition
```

- *In this example, the sentence "department head [etc]" in the with-clause (the inner sentence definition) is used as a parameter of the sentence reference "{EMPLOYEE} is blocked by etc.". This is done because this is considered more readable than the precise definition given in the with clause. The meaning however is exactly the same as it would be if the precise definition was directly used.*
- *Note also that the inner sentence "department head [etc]" can only be used directly in the sentence definition "leave of absence for {EMPLOYEE} is not blocked", the parent of the with-clause. It cannot be used in the sentence definition of the sentence "{EMPLOYEE} is blocked by {MANAGER}" (the place marked "body of another sentence definition"), or any other requirement definition.*

2.6 Sentence references

A sentence reference is used to refer to an expression as defined by either a sentence definition, or by the inner sentence definition in a with clause. The expression referred to can be either a value expression or a result expression.

2.6.1 Syntax

```
    sentenceReference =
       '"' { sentenceNamePart | actualParameter } '"' ;

    actualParameter = '{' valueExpression '}' ;

#   sentenceNamePart = // a string of Unicode characters,
                       // excluding double quotes ("), newlines, braces ({}), and tabs
```

2.6.2 Semantics and usage notes

A sentence reference is any sequence of one or more characters between double quotes, optionally containing one or more parameters. The parameter in a sentence reference is an *actual parameter*, which is represented by a value expression between curly brackets.

When a sentence reference is used, there *must* be a corresponding sentence definition or inner sentence definition within the scope of the reference. If there is more than 1 corresponding definition, the following precedence rules apply.

- If there is one corresponding sentence definition in the process specification and one in a reusable specification, then the definition in the process specification takes precedence.
- If both a regular sentence and an inner sentence definition exist within this scope, the inner sentence definition takes precedence.

In order to determine whether the sentence name in a sentence reference matches the sentence name in a sentence definition, the following rules apply. Note that the same rules also apply to determine whether the sentence names of two sentence definitions are the same.

- For each sentence, a sentence identifier is derived by replacing each parameter in the sentence name with a question mark. So for instance, the sentence identifier of `"{PERSON} is old enough on {date}"` is `"{?} is old enough on {?}"`.
- If the two sentence identifiers are identical, then the sentence reference matches the sentence definition (or, when comparing two sentence definitions, the two sentences names are the same).

As noted above (in § 2.4), sentence references may not be recursive, whether directly or indirectly.

2.6.3 Examples

```
//Example 1
   "employee has requested leave of absence"
   and
   if "{REQUESTED_LEAVE.EMPLOYEE} is old enough on {REQUESTED_LEAVE.start_date}"
   then
       "the manager of {REQUESTED_LEAVE.EMPLOYEE} has reviewed the request"
```

❖ *The above example shows three sentence references. The second one has two parameters, one referring to an instance of an entity, and one referring to a date value.*

```
//Example 2
   "employee has requested leave of absence"
   and
   if "{"the requesting employee"} is old enough on {"the start date of the leave"}"
   then
       "the manager of {"the requesting employee"} has reviewed the request"
with:
   "the requesting employee" = REQUESTED_LEAVE.EMPLOYEE
   "the start date of the leave" = REQUESTED_LEAVE.start_date
```

❖ *In the above example, the parameters "the requesting employee" and "the start date of the leave" are placeholders, which are defined in the "with" clause at the bottom of the context.*

2.7 View specifications

A view specification is a group of view definitions, where a view definition specifies how an entity can be displayed in the front-end.

2.7.1 Syntax

```
    viewsSpecification =
        viewsSpecificationHeader { definitionSeparator viewDefinition } ;

    viewsSpecificationHeader =
        "view definitions" ;

    viewDefinition
        = recordDefinition
        | listDefinition
        | detailDefinition
        ;

#   definitionSeparator = // one or more newlines (no indent)
```

2.7.2 Semantics and usage notes

All view specifications are part of the same scope, i.e., they can cross-reference each other freely.

2.7.3 Examples

```
view definitions

list ALL_ORDERS =
  view ORDERS
  with fields:
    quantity labeled 'Amount'
    PRODUCT.name
```

- *The above example contains a single list definition in a view specification.*

```
view definitions

// this is a global trait definition
// every record, list, and detail in this file,
// will now only be visible to those with the MANAGER role
visible to MANAGER
```

```
// we define a view record to display an order
record STANDARD_FIELDS =
  // for every view definition, we need to specify what entity it views
  view ORDER
  // view records require at least one field!
  with fields:
    // fields can access the entity's attributes
    // the label is inferred, in this case "Nr" will be chosen as label
    nr
    // but you may specify a label manually
    date_ordered labeled 'Order date'
    // more complex expressions are also available
    BUYER.username labeled 'Buyer'
    // further fields:
    delivery_date
    comments
    approved

// record views by themselves are not available directly in the frontend,
// but they are referenced by *lists* and *details*

// here we define a reusable list view
// this list will not be exported to the frontend, but can be inherited from
reusable list ORDER_LIST =
  // for list views, we specify what *collection* of entities it views
  // hence the plural ORDERS instead of the singular ORDER as before
  // we may also filter and sort the entity collection
  view ORDERS sorted by date_ordered ascending
  // lists must have at least one field, which they may inherit from a record
  with STANDARD_FIELDS
  // lists may link to a detail view for extra information about a specific entity
  // this specific detail view will be defined later
  detailed by ORDER_DETAIL
  // lists may have a search bar, here we specify which attributes will be searched
  with search fields:
    nr
    comments
    // as with regular fields, search fields may also be complex expressions
    BUYER.username

// now we define a few child views from ORDER_LIST
```

```
list ORDERS_ALL =
  // we specify what list this inherits from
  ORDER_LIST
  // and we may also give it a manual label
  labeled 'All Orders'
list ORDERS_APPROVED =
  // lists may have filters using the conventional filter syntax
  ORDER_LIST[approved]
  labeled 'Approved Orders'
list ORDERS_MINE =
  // these filters may be complex expressions
  ORDER_LIST[CREATOR == current_user()]
  labeled 'My Orders'
  // here we *override* the global roles trait
  // we specify that any CUSTOMER may view this list
  visible to CUSTOMER
  // and we can also link to the 'Order products' process
  with link to process product.order
  // we can also give values to the process form
  with link to
    process product.order
    filled with {
      orderLines: [{ product: the first PRODUCTS, quantity: 1 }]
    }
    labeled 'Order whatever the first product is'

// we now define the detail view that was referenced before
detail ORDER_DETAIL =
  view ORDER
  // detail views are a collection of *sections*
  // these sections are either record views or list views
  // there must be at least 1 section defined
  with sections:
    // when defining a section, we use the same syntax as before
    record MAIN =
      // MAIN inherits from STANDARD_FIELDS
      STANDARD_FIELDS
      // and gets its own label
      labeled 'Basic info'
      // and we give it an extra field!
      // this field will show up after all inherited fields
```

```
    with fields:
      concat(MANUAL_APPROVAL_REASONS.text, '\n') labeled 'Manual approval reasons'
  // we can also define a section without inheritance
  record CONDITIONAL_SECTION =
    // without inheritance, we do still need to specify what we view,
    // even if it is nested inside a detail view like this
    view ORDER
    with fields:
      // just to drive home the point,
      // section fields may be ANY arbitrary expression!
      // you do not have to look at the entity at all:
      current_user().given_name labeled 'Your name'
    // sections may have a condition to specify when they show up on the frontend
    // this example is a bit contrived: the section shows up 50% of the time
    if random_number(0, 1) == 0
  // sections may also be lists
  list LINES =
    // here we view the LINES attribute of an ORDER
    view LINES
    // and we can also define list fields directly
    with fields:
      nr
      PRODUCT.name labeled 'Product'
      quantity
    // the parser is smart enough to know this label belongs to the list,
    // and not to the quantity field above
    labeled 'Order lines'
// we can also link to further detail views from here
// here we link to the PRODUCT attribute using the detail view PRODUCT_DETAIL
// PRODUCT_DETAIL is defined in another file, but can still be referenced here!
with link to
  PRODUCT
  detailed by PRODUCT_DETAIL
  // we can also put conditions on links
  if current_user().ROLES.name contains 'manager'
// and we can override the visibility as well
// we can specify multiple roles that may view this
visible to USER, EMPLOYEE, MANAGER
```

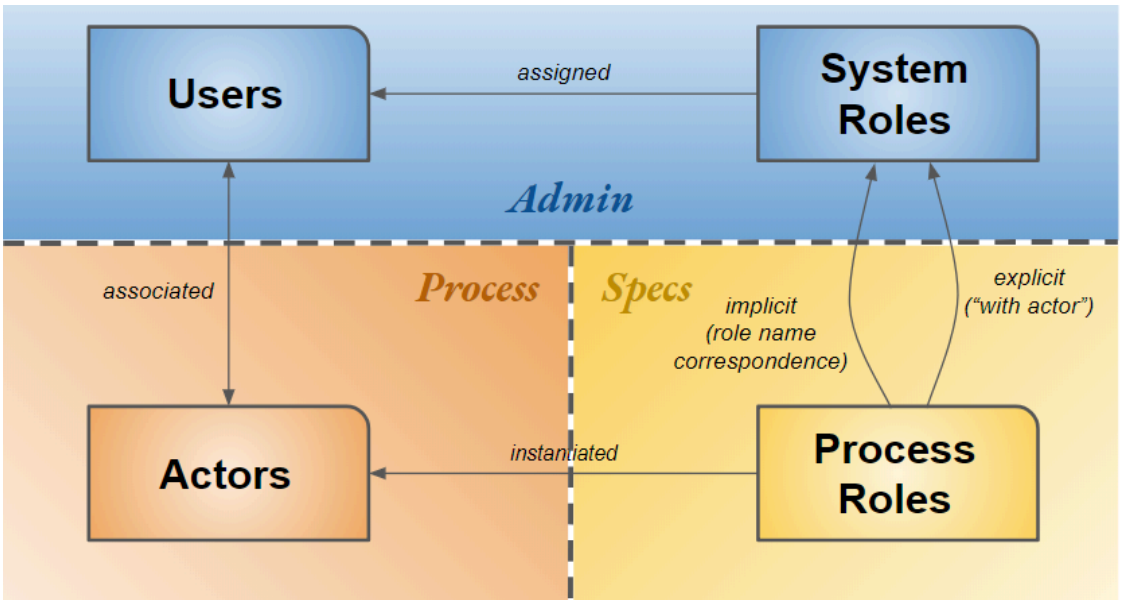
❖ *A bigger example containing almost all supported features, with explanatory comments*

2.8 Actors, Users and Roles

This section details the distinction between the static concepts of Users and System Roles on the one hand, and the dynamic concepts of Actors and Process Roles on the other.

2.8.1 Semantics and usage notes

The following image summarises the differences and relations between actors, users, and roles, as the terms are used in SN3.



Creating and managing **Users** and **System Roles** should be seen as a manual administrative task that can be carried out separately from any generated process. Both Users and System Roles are standard entities, which are stored as records in the database (see § 6.8). A **User** can be thought of as a log-in identity, and usually corresponds to a person. A **System Role** is a collective name that is used to divide Users into groups depending on their organisational tasks and authorizations (e.g., manager, secretary, salesperson, hr_worker). In order for Users to be able to use an application, they must have a System Role: assigning these roles is also a manual administrative task. Note that external systems that carry out tasks in a process (non-human actors) also need to be known as Users and assigned a System Role.

Actors and **Process Roles** are not statically stored in the database, but come into existence when SMART Requirements processes are instantiated and carried out. **Process Role** is the formal term for the names used in SMART Requirements specifications in input expressions, startable-by clauses and with-actor clauses. In the syntax rules these are referred to as **actorNames**: a Process Role and an Actor Name are in fact the same thing. In a process instance (i.e., when a process is actually executed), the Process Role named in a task is instantiated as an **Actor**, and when a User claims and executes that task in that process instance, then that user is associated with that Actor. In other words, if User johndoe gives input for a task specified as `input from EMPLOYEE`, then any further reference of `EMPLOYEE` in that process instance will be identified with User johndoe.

In order to determine whether a User is authorised for a task, the process role of the task needs to be mapped to the System Role of the user. The default mechanism for this is name correspondence: if the User has a System Role which is identical to the Process Role for the task, then the User is authorized to claim and execute the task (Note however that by convention, System Roles are written in lowercase, and Process Roles in uppercase.)

It is also possible to explicitly map a Process Role (with a different name) to a System Role: this is done using the with-actor clause in the process specification (see § 2.2). When, through this mechanism, more than one actorName is linked to the same System Role, then these Actors cannot be associated with the same User.

2.8.2 Examples

```
process 'Add new employee'
   with actor HR_CHECKER with role HR_EMPLOYEE

results in:
   one NEW_EMPLOYEE in EMPLOYEES is created with:
       name          = input from HR_EMPLOYEE
       age           = input from HR_EMPLOYEE
       doublechecked = input from HR_CHECKER
                       based on {NEW_EMPLOYEE.name, NEW_EMPLOYEE.age}
```

- *In this process, a new employee is entered into the database by an HR employee, where another HR employee has to double-check the first HR employee's work.*
- *The input expressions for name and age are linked to the process role HR_EMPLOYEE, and can be claimed and executed by any User with System Role hr_employee.*
- *The input expression for* `doublechecked` *is linked to the process role HR_CHECKER. This role is declared in the with-actor clause as a process-specific actor name, and therefore does not exist as a System Role. As per the with-actor clause, any User with system role hr_employee can claim and execute this task, with the restriction that it cannot be the same User who is already associated with the actor HR_EMPLOYEE in this process (the User who gave the input for name and age).*

2.9 Formatting: indentation & newlines

In SN 3.0 indentation and newlines are used as a syntactic mechanism to represent language features such as grouping and precedence of constructs. A specification that is not properly formatted is not syntactically correct and cannot be parsed.

2.9.1 Semantics and usage notes

Only spaces may be used to indent SN3 constructs. Tabs are not allowed. A single space is enough to indicate a higher indent level, though in practice (depending on the editor) it is more common to use indents consisting of 4 space characters.

The formatting rules for SN3 are as follows.

1. Specification headers always start on a line at position 0 (no indentation).
2. Syntactic constructs called separators are used in the EBNF rules to *explicitly* indicate indentation in the following cases.
    a. A definitionSeparator indicates that the subsequent construct *must* start on a new line without indentation. A definitionSeparator can consist of any number of newline characters (at least one).
    b. A bodySeparator can indicate one of two things.
        i. The subsequent construct starts on a new line, and is indented one level higher than the preceding construct.
        ii. The subsequent construct stays on the same line; in this case the start position of this construct defines a new indent level for other constructs depending on it or belonging with it.
    c. A sequenceSeparator indicates that the subsequent construct *must* start on a new line with the same indent level as the preceding construct.
3. As a general rule, when dividing a construct over several lines, all parts of the construct must have at least the same indent level as the start of the construct.
4. The specific rule for dividing binary and ternary expressions over several lines is as follows: if any constituent of a binary or ternary expression (whether an operand or an operator) is placed on a new line, it must have the same horizontal position (i.e., the same column) as the previous constituent of the expression.
   I.e., for an expression like `<left> <operator> <right>`, if <operator> is placed on a new line it must have the same column as <left>, and if <right> is placed on a new line it must have the same column as <operator>.
5. Any other rule that may apply is a guidelines, i.e., not enforced by the parser and not in scope of this document.

Newlines have the same function as parentheses, in that they override the standard precedence rules.

2.9.2 Examples

```
//Example 1
reusable definitions

"{PERSON} is allowed to drive independently"
description 'Lists all the conditions under which a person is allowed to drive without supervision.'
=
  "{PERSON} is old enough to drive on {currentDate()}"
  and
  "{PERSON} passed the driving test" and "license of {PERSON} has not been revoked"
  and
  "{PERSON} has no more than allowed alcohol level in blood"
```

- ❖ *Above example shows various formatting rules. Note that this is rather messy and is not recommended indenting style.*
    - ➢ *ReusableDefinitionsHeader starts on new line.*
    - ➢ *ReusableDefinition (sentenceDefinition) starts on new line.*

- ➢ *Description clause and = sign have no explicit separator; here positioned on new line at same indent level.*
- ➢ *First sentence "person is old enough to drive" must start on a new line with increased indent level (bodySeparator).*
- ➢ *Keyword* and *and second sentence start on new line with same indent level (sequenceSeparator).*
- ➢ *Second and and third sentence start on same line; indent level for new line (as seen for the third and) remains what it was.*

```
//Example 2
"{PERSON} is allowed to drive independently"
= "{PERSON} is old enough to drive on {currentDate()}"
   and
   "{PERSON} passed the driving test"
```

- ❖ *This example has incorrect indentation. The sentence "{PERSON} is old enough to drive on [date}" is preceded by a bodySeparator according to the syntax rules and stays on the same line, which thereby defines a new indent level. However the subsequent* `and` *is indented one space deeper than the preceding sentence. It would also be an error if the* `and` *was aligned with the =.*

```
//Example 3
   x + y * z        //#1
   (x + y) * z      //#2
   x + y            //#3
   *
   z
```

- ❖ *According to the normal precedence rules, in #1, the multiplication is carried out first. Both the parentheses in #2 and the formatting in #3 override the normal precedence rules, so that #2 and #3 are equivalent.*

2.10 Comments

Comments can be used at any position in the specifications. There are two types of comments: single-line comments and multi-line comments.

Single-line comments start with two forward slashes (// comments). Anything between the slashes and the end of the line (including of course the slashes themselves) is a comment and is not part of any SN3 expression preceding and/or following it.

Multi-line comments start with a forward slash and an asterisk, and end with an asterisk and a forward slash (/* comments */). Anything between these opening and closing markers, including of course the markers themselves, is a comment and is not part of any SN3 expression preceding and/or following it.

2.11 Keywords

SN3 uses reserved words and phrases in various syntactic constructs, as seen in the preceding paragraphs as well as the remainder of this document. Examples of these that have been mentioned in

this chapter are `process`, `with subject`, `reusable definitions`, and, as we will see below, `if`, `then`, `else`, and various others. SN3 has close to 100 keywords.

These reserved words and phrases are called keywords, and may not be used as names (of e.g. entities, attributes, roles or actors), nor as process keys. Keywords can however be used as (parts of) sentence names. Note that in case of multi-word keywords (such as `with subject`), only the entire keyword is reserved, not its constituent parts ("with" is a keyword because it's used in with-clauses, but "subject" isn't and can freely be used).

In the case of multi-word keywords (such as `with subject`), the whitespace separating the two constituent words is part of the keyword. If for instance there are two whitespace characters between with and subject, or if they are on separate lines (with a newline character between them), this is *not* recognised as a keyword.

A complete list of SN3 keywords is given in Appendix A.

2.12 Derivation of process keys

When process keys are not explicitly specified, they are derived from the process name literal. Since process keys may contain only lowercase letters, numerals, and underscores, any other character that may appear in a process name literal needs to be converted, which is done according to the following rules.

- Uppercase characters in the process name literal are converted to lowercase.
- Any characters in the process name literal that are not allowed in a process key are converted to underscores.
- Any sequence of underscores (i.e., 2 or more consecutive underscores) is converted to a single underscore.

The process keys that are derived based on these rules *must* be unique.

Since derived process keys tend to get rather unwieldy, it is generally recommended to specify explicit process keys.

2.13 About translation and localisation

The requirements specification language SN3 is based on English. Its keywords are English words, or phrases made up of English words. Of SN3 itself, there are no other-language versions.

However the entity names, attribute names and sentences of an SN3 requirements specification can be created in any language (and character set, for sentences) desired by the owner of the processes. This would still leave the keywords in English in the original SN3 specification; but documentation can be generated from the SN3 specification in other languages than English, including translated keywords to match the language of the sentences. While the generation of documentation from SN3 specifications is not in scope of this language reference, it is relevant to note that this feature makes it possible to use SN3 to create process documentation in other languages than English.

As for the generated information system, SN3 allows to internationalise the specifications, i.e., to specify multiple different language versions of a process. From such an internationalised specification, localised versions of the resulting information system (with translated messages, labels and such) can be generated by adapting the language and/or locale settings in the property files. The language features for internationalisation are as follows.
(TBD - BLF029)

3 Result expressions

3.1 Overview

Result expressions are one of the two types of expressions that exist in SN3, alongside value expressions. There are three basic kinds of result expressions: create expressions, update expressions and delete expressions. In addition, there are ways to combine result expressions into a more complex result expression, using conjunction, if-expressions, and for-expressions. Finally, sentence references can be result expressions, depending on their definitions, and result expressions can be placed between parentheses (which doesn't affect their meaning).

The formal syntax of all expressions is given in Ch. 4; the syntax section below contains an informal enumeration of the constructs discussed in this chapter.

3.1.1 Syntax

```
#    resultExpression
#       = createdResult
#       | updatedResult
#       | deletedResult
#       | resultAndExpression
#       | resultIfExpression
#       | resultForEachExpression
#       | resultForTheExpression
#       | sentenceReference
#       | "(" resultExpression ")"
#       ;
```

3.2 Created result expressions

A created result expression (or create expression) is used to specify one single new element or multiple new elements that is/are created in a given collection (i.e., for a given entity). The created element is named, so that it can be referred to in other parts of the process specification.

Create expressions are special in SN3 in that they not only define a result (the creation of one or more elements), but also have a value which consists of the created element or elements. A create expression is therefore both a result expression, and a value expression. The context determines which of the two is syntactically prominent.

3.2.1 Syntax

```
    createdResult = singleCreatedResult | multipleCreatedResult ;
```

```
    singleCreatedResult =
        "one" elementName "in" collectionName "is created"
        [ createdWithClause ] ;

    multipleCreatedResult =
        "multiple" elementName "in" collectionName "are created"
        [ createdWithClause ] ;

    createdWithClause =
        "with:" bodySeparator
        attributeValueAssignment { sequenceSeparator attributeValueAssignment } ;

    attributeValueAssignment = attributeName "=" valueExpression ;

#   elementName = name  // upper case only
#   collectionName = name  // upper case only
#   attributeName = name  // either upper case only, or lower case only

#   bodySeparator = // one or more newlines with increase of indent level
#   sequenceSeparator = // one or more newlines with same indent level
```

3.2.2 Semantics and usage notes

3.2.2.1 Element, collection, and attribute names

In a create expression, the elementName can be freely chosen and can be used elsewhere in the process specification to refer to the created element or elements. The elementName is not strictly required to be unique: it is possible to have two (or more) create expressions in a process with the same elementNames, or a createExpression with an elementName that is identical to the elementName of a for-expression. However, if this leads to ambiguity (references to this name where it cannot be determined which one is intended), then this will be flagged as an error.

The collectionName *must* be an existing collection name from the domain model (see Ch. 5 “Entity definitions”). It names the entity for which new elements are created.

In both single- and multiple-create result expressions, both the elementName and collectionName *must* be all-caps. It is customary (and recommended) to use a singular noun for the elementName (even in the case of a multiple-create expression), and a plural noun for the collectionName.

The name in each attribute value assignment of a create expression *must* be the name of an attribute of the entity of which new instances are created. The same attribute name *may not* be used twice in the same create expression.

3.2.2.2 Element references

An elementName in a single-create expression that is used just once in a process specification can be used freely anywhere in the process specification. The reference is always to the single created element.

An elementName in a multiple-create expression can *only* be used within the attribute value assignments of the create expression (in the value expression on the right-hand side) to refer to a single instance of the entity that is created. Outside the create expression, it would be unclear which of the multiple created instances the name is meant to refer to, and the usage of the elementName is considered an error.

If a single-create expression is used in a for-each expression, it is de facto used to create multiple elements. Within the for-each body the reference is still to the single created element, but outside the for-each body the elementName cannot be used as such, for it would be unclear which created element it refers to.

3.2.2.3 Create expression as value expression

A create expression (both single and multiple) is the only type of expression in SN3 that functions both as a result expression and as a value expression. When used in a context where a value expression is expected, the create expression engenders the required result (creation of one or more new elements), but also returns the created element or elements as values that can be used in an encapsulating value expression (e.g. an update expression).

3.2.2.4 Number of created elements in multiple-create

A multiple-create expression creates multiple elements, but the expression doesn't specify how many. The number of created elements is decided at run-time; for this to happen, at least one of the attributes of the multiple-created element needs to be provided by an actor using an input expression, so that the actor decides how many elements are created. It is recommended, but not mandatory, that the input expression is part of the multiple-create expression (i.e., one of the attribute-value assignments). However, when the multiple-created element is finalized, at least one of its attributes *must* have been provided through an input expression.

3.2.2.5 Finalization, and mandatory attributes in create expressions

The creation of a new instance of an entity as part of a process can be seen as a complex operation consisting of three stages: first the creation of the entity itself; then the assignment of values to the attributes of the entity, insofar as specified in the process; and the third step is called the finalization of the entity. Finalization is an artificial step which is automatically executed for a newly created entity after all assignments to its attributes that are prescribed in the process specification have been executed.

Finalization has various functional and technical implications, but in terms of SN3, the most important aspect of finalization is that this is the moment when any necessary checks on the entity are performed. When an entity is finalized, all mandatory attributes of the entity *must* have been assigned a value; also, all entity constraints specified for the entity and any of its attributes *must* be met.

3.2.3 Examples

```
//Example 1
```

```
"customer has ordered products" =
  one NEW_ORDER in ORDERS is created with:
      date  = currentDate()
      BUYER = CUSTOMER
      LINES = "order lines"
```

- *Single create result expression which assigns a value to three attributes of the entity ORDERS. The name ORDER can be used anywhere in the requirements specifications to refer to the specific instance that is created here.*

```
//Example 2
"order lines" =
  multiple LINE in ORDER_LINES are created with:
      PRODUCT  = input from CUSTOMER
      number   = input from CUSTOMER
      delivery = if "{LINE.number} of {LINE.PRODUCT} are in stock"
                 then 'immediate'
                 else 'deferred'
```

- *Multiple create result expression which creates multiple instances of the entity ORDER_LINES (as many as the CUSTOMER chooses to provide), and assigns values to three attributes of each instance. For each instance, the value of the attribute delivery is determined based on the attributes PRODUCT and number of that instance.*

3.3 Update expressions

An update expression is used when the result of a requirement is that an attribute of an entity receives a new value.

3.3.1 Syntax

```
      updatedResult = attributePath "=" valueExpression ;
      attributePath = pathNamepart  { "." pathNamepart } "." attributeName ;

#     pathNamepart = name  // upper case only
#     attributeName = name  // either upper case only, or lower case only
```

3.3.2 Semantics and usage notes

An update expression assigns the result of a value expression (i.e., a value) to an attribute of a given entity.

The string of pathNameParts before the attributeName *must* evaluate to a single element or attribute. This requires that the first namepart in the path denotes a single element (and not a collection), and that all subsequent nameparts stand for a 1-1 or n-1 relationship.

The attributePath and valueExpression *must* be of the same or compatible types. The rules for type conversion (cf. § 5.20) apply here, so that e.g. a numerical value can be assigned to an attribute of type text (but not the other way around). Note that if the value expression denotes a single value (i.e., not a collection), then the attributePath must also denote a single element or attribute.This requires that the attribute name stands for either a simple value or a 1-1 or n-1 relationship.

3.3.3 Examples

```
//Example 1
"the leave of absence request has been evaluated automatically" =
   REQUEST.evaluation = "automatic evaluation"

"automatic evaluation" =
   if REQUEST.reason = 'sick'  then 'automatic approval' else
   if EMPLOYEE.days_left <= then 'no days left'       else
   if EMPLOYEE.days_left <= 10 then 'needs approval'
   else 'plenty days left'
```

- *In this example, the attribute evaluation of the entity REQUEST is assigned the value resulting from the expression represented by the sentence "automatic evaluation".*

```
//Example 2
   for the RESIGNING_EMPLOYEE in "employee requesting termination" applies:
       RESIGNING_EMPLOYEE.status           = 'inactive'
       RESIGNING_EMPLOYEE.termination_date = currentDate()
```

- *The above construct contains two update expressions, and shows how multiple attributes of a single entity can be updated.*

3.4 Delete expressions

A delete expression is used when the result of a requirement is that one or more elements of a given entity are deleted. We distinguish between outright or physical deletion on the one hand (when the record is removed from the database), and logical deletion on the other (when the record becomes invisible, but is marked as "archived" and remains in the database for auditing purposes).

3.4.1 Syntax

```
      deletedResult = valueExpression ( "are" | "is" ) ( "deleted" | "archived" )
                      [ "from" namePath ] ;

      namePath = collectionName { "." attributeName } ;

#     collectionName = name  // upper case only
#     attributeName = name  // either upper case only, or lower case only
```

3.4.2 Semantics and usage notes

The value expression *must* refer to either a single element of, or a collection of elements of an entity defined in the domain model.

When the keyword `is deleted` is used, the value expression *must* refer to a single element. This may be a name that is defined as a single element, or a the-expression (see § 5.14).

When the keyword `are deleted` is used, the value expression *must* refer to a collection. The size of the collection may be 0, 1, or greater than 1. Note that if the collection is empty nothing is actually deleted, but this is not an error.

The optional from-clause indicates from which collection the element(s) is/are deleted. The type of the namePath *must* match the element or collection being deleted. When the type is a domain model collection, this information is implicit in the value expression and doesn't need to be specified. The from-clause may also indicate a relational attribute: if this is the case, then the deleted element(s) itself isn't (aren't) deleted, but they are removed from the relation. For instance, "MARY is deleted from JOHN.CHILDREN" doesn't remove either MARY or JOHN from the domain, but it severs the father-child relationship between them.

When the keyword `deleted` is used, the deleted element (or elements) is physically deleted from the DB, which means it no longer exists and cannot be accessed in any way. When the keyword `archived` is used, the element is only logically deleted. It remains in the DB, but is flagged as archived and is automatically ignored in all searches and filters. Note that once an element has been logically deleted, it cannot be physically deleted as well (i.e., not through a SMART Requirements process). If an archived element needs to be physically removed from the DB, this should be done directly on the DB. For more information about archived elements, see § 6.7 for details.

When an element is deleted (physically or logically), the referential integrity of the data must be maintained. When an element X is removed, all references to this element, whether directly or via links that existed to it from other elements, become invalid. Note that constraints on the data may make a delete expression invalid: for instance, if teacher X is the sponsor of student Y and every student must have exactly one sponsor in the DB, then deleting X is not possible.

In order to prevent generation of applications where an element is updated or used after it is deleted, some special dependency rules apply to delete statements. A delete step itself has dependencies on publish steps of deleted elements, updates of deleted elements, and finally update steps of elements contained in deleted steps.

The dependencies for update steps also behave differently than usual so that they sometimes aren't dependent or are "reverse dependent" on delete steps - i.e. the delete is dependent on the update because the dependency arrow is reversed. The left hand side of updates is never dependent on a delete. For instance "EMPLOYEE.name = PERSON.name" should never be dependent on a "PERSON is deleted" statement. The right hand side of an update step is reverse dependent on direct deletes of elements it uses. A direct delete is a delete statement that deletes an element, but not by deleting from a containing collection, for instance "PERSON is deleted" is a direct delete of PERSON, but "PERSONS

are deleted" is not. Note that deletion through an alias is also considered a direct delete, i.e. if ADAM.SPOUSE is EVE, then "ADAM.SPOUSE is deleted" is a direct delete of EVE.

3.4.3 Examples

```
//Example 1
"employee records are deleted when required by law" =
   EMPLOYEES[creation_date < "legal expiry date"] are deleted

"legal expiry date" =
   currentDate() - 5 years
```

- *In this example the value expression that represents the elements that are to be deleted is* `EMPLOYEES[creation_date < "legal expiry date"]`*. This expression always results in a collection, so the keyword "are" is warranted.*

```
//Example 2
"the {review_year} review record for {EMPLOYEE} is deleted" =
   the only REVIEW_RECORDS[employee_number == EMPLOYEE.number
                           and year == review_year] is archived
```

- *In this example the value expression that represents the element that is to be deleted is a the-expression, which always results in a single element; so the keyword "is" is warranted.*

```
//Example 3
"{THIS_PRODUCT} is removed from the lineup of {THIS_AD_CAMPAIGN}" =
   THIS_PRODUCT is deleted from THIS_AD_CAMPAIGN.FEATURED_PRODUCTS
```

- *In this example the link that existed between the ad campaign THIS_AD_CAMPAIGN and the product THIS_PRODUCT is removed. The product THIS_PRODUCT itself is not deleted or archived: it remains in the DB.*

3.5 Simple conjunction (and expressions, implicit and)

Result expressions can be conjoined using the keyword and, which means that both results are achieved. Also, the keyword and may be omitted if this improves readability (implicit and).

3.5.1 Syntax

```
#    resultAndExpression = resultExpression "and" resultExpression
#                        | resultExpression sequenceSeparator resultExpression ;

#    sequenceSeparator = // one or more newlines with same indent level
```

Note: The above syntax rules are not part of the formal syntax of SN3. Formally, result and-expressions are handled by the same syntax rules as binary expressions: cf. § 5.6.

3.5.2 Semantics and usage notes

When two result expressions are conjoined, this means that both results are achieved, in unspecified order.

When the and operator is left out, there has to be a newline between the expressions, and both expressions must have the same indent level. Note that even when the and is explicit, it is recommended, especially for result expressions, to place the two expressions on different lines, for the sake of readability.

3.5.3 Examples

```
"{PERSON} is old enough to drive on {currentDate()}"
"{PERSON} passed the driving test"
and
"license of {PERSON} has not been revoked"
"{PERSON} has no more than allowed alcohol level in blood"
```

- *This example contains four conjoined requirements, where the conjunction between the 2nd and 3rd requirements is explicit, while the other two conjunctions are implicit.*

3.6 If-expressions (results)

⍐ A (result) if-expression is used when the requirement is for a result to be conditionally achieved. Either the result in the `then` clause is achieved, or the result in the `else` clause (if present) is achieved; or, if there is no `else` clause, no result may be achieved.

Note that an if-expression can be either a value expression or a result expression, depending on whether the *then* (and *else*) clause(s) contain value expressions or result expressions. Value if-expressions are described in § 5.17.

3.6.1 Syntax

```
#    resultIfExpression = "if" condition
#                         "then" resultExpression
#                         [ "else" resultExpression ] ;

#    condition = valueExpression ;
```

Note: The above syntax rules are not part of the formal syntax of SN3. Formally, result if-expressions are handled by the same syntax rules as value if-expressions: cf. § 5.17.

3.6.2 Semantics and usage notes

If the if-expression is a result expression, both the then-clause and the else-clause (if present) *must* be result expressions.

The value expression in the condition *must* be of Boolean type. Also, it *must* have a value of either `true` or `false` (that is, a null or empty value, as in a Boolean attribute that has not been assigned a value, is not allowed).

In nested if-expressions, if the attachment of an else-clause cannot be derived from the formatting (indentation), then the else-clause is presumed to belong with the nearest preceding if-clause (that doesn't have an else yet).

Under specific circumstances, a result if-expression in SN3 may be interpreted as a kind of while-loop. This has to do with dependencies: if the condition of the if-expression *depends on* the results in the then-branch, then the if-expression is interpreted as requiring the production of the then-branch result *as long as* the condition is true; then, when the condition becomes false, the else branch (if present) is evaluated. (A similar reasoning applies if the condition is dependent on the else branch; the condition may not be dependent on both branches.)

3.6.3 Examples

```
//Example 1
   if "the requesting user already has a profile"
   then
       "the profile of the requesting user is updated"
   else
       "a new profile is created for the requesting user"
```

- *This example shows an if expression that, based on the definition of the sentences in the* `then` *and* `else` *clauses, is a result expression. This means that depending on the condition, either the one or the other result is achieved.*

```
//Example 2
   if A then if B then C else D
```

- *In this example, without indentation or parentheses, it is ambiguous whether the* `else` *is intended to belong with the first or the second* `if`*. In that case, the else is presumed to apply to the nearest (i.e. the second)* `if`*.*
  Note that this kind of ambiguity in a SMART Requirements specification is bad practice, and should be avoided by using indentation or parentheses.

3.7 For-each expressions (results)

A for-each (result) expression is used when the requirement is for a result to be achieved for each of a collection of elements.

A for-each expression can be either a value expression or a result expression, depending on the enclosed expression. For-each (value) expressions are described in § 5.18.

3.7.1 Syntax

```
#    resultForEachExpression =
#        "for each" elementName "in" [ "numerous" ] valueExpression "applies:"
#        bodySeparator
#        resultExpression ;

#    elementName = name  // upper case only
```

```
#     bodySeparator = // one or more newlines with increase of indent level
```

Note: The above syntax rules are not part of the formal syntax of SN3. Formally, result for-each expressions are handled by the same syntax rules as value for-each expressions: cf. § 5.18.

3.7.2 Semantics and usage notes

The valueExpression *must* result in a collection, so that the result is achieved for each element in the collection, represented by the name elementName. If the size of the collection is 0, then no results are achieved.

The keyword `numerous`, when present, has an effect on how the resulting code is generated, and is intended to be used when the collection defined by the valueExpression is large. It has no effect on the meaning of the for-each expression.

Normally, the elementName is intended to be used in the enclosed expression, but this is not enforced. If the elementName is not used, a warning will result.

3.7.3 Examples

```
for each CUSTOMER in "customers who have placed an order in the past month" applies:
    "{CUSTOMER} has received an invoice"
```

- *Leads to as many results (invoices) as there are customers who placed an order in the last month.*

3.8 For expressions (results)

A for (result) expression is used when the requirement is for a result to be achieved for one or more named elements.

Note that a for expression can be either a value expression or a result expression, depending on the enclosed expression and the presence of the 'applies' keyword. For (value) expressions are described in § 5.19.

3.8.1 Syntax

```
#     forExpression =
#         "for" name "=" valueExpression { "," name "=" valueExpression }
#         "applies:"
#         bodySeparator
#         resultExpression ;

#     bodySeparator = // one or more newlines with increase of indent level
```

Note: The above syntax rules are not part of the formal syntax of SN3. Formally, result for expressions are handled by the same syntax rules as value for expressions: cf. § 5.19.

3.8.2 Semantics and usage notes

In a for result expression, the keyword `applies` is mandatory.

Normally, the elementName is intended to be used in the enclosed expression, but this is not enforced. If the elementName is not used, a warning will result.

3.8.3 Examples

```
for APPROVING_MANAGER = "approving manager of employee" applies:
    "{APPROVING_MANAGER} receives printed confirmation of request"
    and
    APPROVING_MANAGER.work_items = APPROVING_MANAGER.work_items + 1
```

- *This expression specifies one result that is achieved for this manager.*

4. View definitions

[↑]

4.1 Record definitions and traits

A record view displays a single entity using a set of fields. These cannot be viewed by themselves, but can be included within lists and details.

4.1.1 Syntax

```
recordDefinition = [ "reusable" ] "record" viewName "="
                       viewBase
                       { [ sequenceSeparator ] recordTrait } ;

viewBase
    = "view" entityType
    | viewName
    ;

recordTrait
    = "with" viewName
    | "with fields:"
          bodySeparator fieldDefinition
          { sequenceSeparator fieldDefinition }
    | viewTrait
    ;

viewTrait
    = "labeled" simpleStringLiteral
    | "with key" translationKey
    | "with link to" viewLink // if this is top-level
    | "if" condition          // if this is a section
    | "visible to" roleNameLiteral { "," roleNameLiteral }
    ;

#   viewName = name  // upper case only
#   entityType = name  // upper case only
#   sequenceSeparator = // one or more newlines with same indent level
#   bodySeparator = // one or more newlines with increase of indent level
#   simpleStringLiteral = // a string literal, but without placeholders,
                          // and without newlines, tabs and such special characters
```

```
#     condition = valueExpression ;
#     roleNameLiteral = [a-z][a-z0-9_]+  // enclosed in single quotes; no placeholders
#     translationKey = name { "." name } ;
```

4.1.2 Semantics and usage notes

Records view a specific entity type, which is specified after the “view” keyword if the record does not inherit from an existing parent record. For the rest of the view definition, the viewed entity’s attributes are in scope in expressions. The entity itself is available using the name ‘this’.

Records may inherit from a parent record by specifying the parent name in the `viewBase`. Any traits defined on the parent will be inherited by the child. Most traits will *override* any previously defined trait, but any trait starting with the word “with” - like “with link to” or “with fields” - will *append* new links and fields, respectively.

A non-reusable record *must* have at least one field. This is checked for every record definition, so inheriting from a reusable record with 0 fields without specifying any extra fields will give an error.

4.1.3 Examples

(TBD)

4.2 Field definitions

A single field inside a record or list.

4.2.1 Syntax

```
    fieldDefinition = valueExpression { fieldTrait } ;

    fieldTrait
        = "labeled" simpleStringLiteral
        | "with key" translationKey
        | "formatted" simpleStringLiteral
        ;

#     simpleStringLiteral = // a string literal, but without placeholders,
                            // and without newlines, tabs and such special characters
#     translationKey = name { "." name } ;
```

4.2.2 Semantics and usage notes

Fields can be any arbitrary expression: it does not have to strictly refer to a view attribute. The generator will try to come up with a sensible translation key and label for the field, but may fall back to using the name of the type of the field. Labels can be manually set using the “labeled” trait. Specifying the translation key is not available currently.

Fields may also have a “formatted” trait: this is equivalent to calling `to_text` with the value and given format. Using the trait instead of the function call is preferred as the generator can then better infer a translation key and label for the value.

4.2.3 Examples

(TBD)

4.3 List definitions and traits

A list view, viewing a collection of entities.

4.3.1 Syntax

```
    listDefinition = [ "reusable" ] "list" viewName "="
                         listBase [ listFilter ] [ listSort ]
                         { [ sequenceSeparator ] listTrait } ;

    listBase
        = "view" entityCollection // if this is top-level
        | "view" valueExpression  // if this is a section
        | viewName
        ;

    listFilter = "[" filterCondition "]" ;
    listSort = "sorted by" sortClause { "," sortClause } ;

    listTrait
        = "with search fields:"
              bodySeparator valueExpression
              { sequenceSeparator valueExpression }
        | "detailed by" viewName
        | recordTrait
        | viewTrait
        ;

#   viewName = name  // upper case only
#   entityCollection = name  // upper case only
```

4.3.2 Semantics and usage notes

Unlike records, lists must specify a specific *collection* instead of an entity type. Top-level lists may only refer to a global entity collection, but list *sections* (see 6.4) may refer to any arbitrary collection of entities. In the future, even top-level lists may refer to an arbitrary entity collection.

Lists may have filters and sort clauses using the usual syntax. Even child lists may filter or sort their parent list.

A non-reusable list *must* have at least one field. Lists may capture all the fields from a record using the "with" trait, or specify them manually with the "with fields" traits. A list may even capture from a *reusable* record. A list may link to a detail (see 6.4) using the "detailed by" trait. This detail must be *non-reusable* to make sure all checks are met.

All non-reusable lists are exported to the frontend.

4.3.3 Examples

(TBD)

4.4 Detail and section definitions

A detail view, containing multiple sections displaying details about a single entity.

4.4.1 Syntax

```
detailDefinition = [ "reusable" ] "detail" viewName "="
                        viewBase
                        { [ sequenceSeparator ] detailTrait } ;

detailTrait
    = "with sections:"
          bodySeparator sectionDefinition
          { sequenceSeparator sectionDefinition }
    | viewTrait
    ;

sectionDefinition = recordDefinition | listDefinition

#    viewName = name  // upper case only
#    entityType = name  // upper case only
#    sequenceSeparator = // one or more newlines with same indent level
#    bodySeparator = // one or more newlines with increase of indent level
```

4.4.2 Semantics and usage notes

Sections are simply records (see 6.2) or lists (see 6.3), and reuse the syntax for these. Record sections *must* view the same entity as the enclosing detail. List sections may refer to any arbitrary collection, including collection attributes of the viewed entity.

Sections may be conditional by adding the "if" trait to them.

A non-reusable detail *must* have at least one section. This section cannot have the “if” trait or be “visible to” a smaller subsection of roles than the detail itself is visible to. This way, a detail will always display at least one section to the user.

All non-reusable details are exported to the frontend.

4.4.3 Examples

(TBD)

4.5 View links

Top-level views may link to a detail view or a process. These links will appear on the side.

4.5.1 Syntax

```
    viewLink
        = "process" processReference
          { [ sequenceSeparator ] processLinkTrait }
        | valueExpression
          [ sequenceSeparator ] "detailed by" viewName
          { [ sequenceSeparator ] linkTrait }
        ;

    processReference = processNameLiteral | processKey;

    processLinkTrait
        = "filled with" tupleExpression
        | "submit with" tupleExpression
        | "submit"
        | linkTrait
        ;

    linkTrait
        = "labeled" simpleStringLiteral
        | "with key" translationKey
        | "with icon" simpleStringLiteral
        | "if" condition
        | "visible to" roleNameLiteral { "," roleNameLiteral }
        ;

#   viewName = name  // upper case only
#   formItemName = name  // should be lowercase or camelCase
```

```
#     processNameLiteral = [a-zA-Z0-9 -]+ // enclosed in single quotes
#     processKey = name { "." name } ;
#     translationKey = name { "." name } ;
#     simpleStringLiteral = // a string literal, but without placeholders,
                            // and without newlines, tabs and such special characters
#     condition = valueExpression ;
#     roleNameLiteral = [a-z][a-z0-9_]+  // enclosed in single quotes; no placeholders
#     sequenceSeparator = // one or more newlines with same indent level
#     tupleExpression = "{" keyValuePair { "," keyValuePair } "}" ;
```

4.5.2 Semantics and usage notes

A process link may contain fill values for the initial form using the “fill with” trait. These are specified using a tuple expression. The tuple keys for these are in *camelCase*, which deviates from the usual *snake_case* or *CAPITAL_SNAKE_CASE* that can be seen in smart specifications. These keys are taken directly from the html identifiers of the form.

Any lists in the form may be filled using a list expression of tuple expressions.

A process link may also *submit* the initial form automatically with the values using the “submit with” trait. It will be checked if all form values are specified or have default values. In case it is possible to submit without *any* values, the “submit” trait may be used.

4.5.3 Examples

(TBD)

5 Value expressions

[1]

5.1 Expressions

[1] Strictly speaking, as mentioned in chapter 1, the syntax of SN3 is overly permissive. The most prominent example of this is in the definition of expressions: in the syntax section below, we see that valueExpressions and resultExpressions are formally reduced to just expressions. So even though in other syntax sections we have stated that specifically valueExpressions, or alternatively resultExpressions, are expected in certain contexts, in the end the syntax allows any kind of expression anywhere. Consider for instance:

```
#     resultForEachExpression =
#         "for each" elementName "in" valueExpression "applies:"
#         resultExpression ;
```

This rule suggests that a valueExpression in the body of the result for-each expression (after “applies:”) is syntactically wrong, which conceptually is indeed the case, but formally, according to the EBNF rules, it is not wrong. Instead, in the definition of SN3, other rules - mostly semantic - are specified to make sure this is flagged as wrong. We will not pursue this matter here, other than to say this is a design choice informed by parsing considerations.

It is important to keep this in mind when reading the syntax rules: wherever the syntax rules say a valueExpression or a resultExpression is expected, this is intended *informatively*. The formal syntax allows both kinds, and *semantic* rules make the actual distinction.

5.1.1 Syntax

```
resultExpression = expression ;
valueExpression = expression ;

expression =
// value expressions only
   literal
   | name
   | unaryExpression
   | binaryExpression
   | betweenExpression
   | listExpression
   | tupleExpression
   | inputExpression
   | attributeExpression
   | filterExpression
```

```
    | sortExpression
    | theExpression
    | functionExpression
// both value and result expressions
    | implicitAndExpression
    | ifExpression
    | forEachExpression
    | forTheExpression
// result expressions only
    | createdResult
    | updatedResult
    | deletedResult
// general
    | sentenceReference
    | "(" expression ")"
    ;
```

Of the above list, the three marked “result expressions only” are discussed in Ch. 3, as are the result expression variants of the four marked “both value and result expressions”. From the “general” subsection, sentence references are discussed in § 2.6, and the use of parentheses in expressions is discussed in the following section on operator precedence. All the other expression types are described in sections 4.3 and lower.

5.2 Parentheses and operator precedence

Operator precedence in SN3 follows the general rules that are common in mathematics and computer programming, also known as PEMDAS. The binary operators (§ 5.6) are presented in this document in order of precedence, i.e., the operators that are shown first have higher precedence and are therefore evaluated first. The unary operators (§ 5.5) have specific precedence rules, and as a group have higher precedence than the binary operators.

Since the precedence rules affect more than just the unary and binary operators, the following list shows the order of precedence (high to low) of all expressions in SN3 for which this is relevant.

1. Attribute expressions (§ 5.11), filter expressions (§ 5.12), sort expressions (§ 5.13)
2. Unary operators `old` and `current` (§ 5.5)
3. The-expressions (§ 5.14)
4. Deleted expressions (§ 3.4)
5. Unary postfix operators `exists` and `does not exist` (§ 5.5)
6. Other unary prefix operators (except `old` and `current`) (§ 5.5)
7. Binary expressions (in the order of the paragraph) (§ 5.6)
   Note that the precedence of between-expressions (§ 5.7) is directly above that of the binary relational expressions.

As formalised in the EBNF rules in § 4.1.1 above, any expression between parentheses is itself an expression with the same value or meaning as the original expression. It follows that, as is common in mathematics and computer programming, the parentheses override the general rules for operator precedence. As mentioned in § 2.9, formatting with newlines and indentation also overrides the precedence rules.

5.3 Literals

A literal is a value of any simple type, written out; so for instance, an integer or decimal number, a Boolean value (`true` or `false`), or a string value.

5.3.1 Syntax

```
literal = integerLiteral | decimalLiteral | booleanLiteral | stringLiteral
        | dateLiteral | timeLiteral | datetimeLiteral | periodLiteral
        | "undefined" | "empty" ;

integerLiteral = ['-'][0-9]+ ;
decimalLiteral = ['-'][0-9]*.[0-9]+ ;
booleanLiteral = 'true' | 'false' ;

stringLiteral = "'" { stringPart | placeholder } "'" ;
stringPart = [^']+ ;
placeholder = "{" valueExpression "}" ;

dateLiteral = yearLiteral "-" monthLiteral "-" dayLiteral ;
  // (* ^\d{4}\-(0?[1-9]|1[012])\-(0?[1-9]|[12][0-9]|3[01])$ *)
yearLiteral = // any four-digit number
monthLiteral = // any integer from 1 up to and including 12,
               // optionally left-padded with a 0 to two positions
dayLiteral = // any integer from 1 up to and including 31,
             // optionally left-padded with a 0 to two positions
timeLiteral = hoursLiteral ":" minutesLiteral
              [ ":" secondsLiteral [ ":" millisecondsLiteral ] ] ;
  // (* ^(?:(?:([01]?\d|2[0-3]):)?([0-5]?\d):)?([0-5]?\d)$ ; *)
hoursLiteral = // any integer from 0 up to and including 23,
               // left-padded with a 0 to two positions
minutesLiteral = // any integer from 0 up to and including 59,
                 // left-padded with a 0 to two positions
secondsLiteral = // same as minutesLiteral
millisecondsLiteral = // any integer from 0 up to and including 999,
                      // left-padded with 0's to three positions
datetimeLiteral = dateLiteral [ timeLiteral ] ;
```

```
periodLiteral = [ integerLiteral " year" ["s"] ] [" "]
                [ integerLiteral " month" ["s"] ] [" "]
                [ integerLiteral " week" ["s"] ] [" "]
                [ integerLiteral " day" ["s"] ] [" "]
                [ integerLiteral " hour" ["s"] ] [" "]
                [ integerLiteral " minute" ["s"] ] [" "]
                [ integerLiteral " second" ["s"] ] [" "]
                [ integerLiteral " millisecond" ["s"] ] ;
```

5.3.2 Semantics and usage notes

In a decimal number literal, the integer part (digits to the left of the period) may be omitted. When omitted the integer part is taken to be `0`, so `.25` is equivalent to `0.25`.

The decimal separator for number literals in SN3 is always a period. Thousands separators are not used in number literals.

A string literal can contain placeholders. These are values that are calculated when the string literal is evaluated. The placeholder may be any type of expression, and will automatically be converted to a string for inclusion in the string literal. The characters `'`, `\`, `{`, and `}` are special characters and need to be escaped (using the escape character `\`) if they must be part of the string. Some character sequences have special meaning: for instance, `\n` is a newline character and `\t` is a tab.

There are two special literals, `undefined` and `empty`. The literal `undefined` corresponds to the concept of NULL as defined in the SQL ANSI standard: it is not a value, but the absence of a value, for any data type. Note that an equality check "`x == undefined`" would always return `false` for any x, even if x is in fact undefined: for this reason, it should not be used. To check for attributes that have no value, the exists operator should be used ("`x does not exist`" is `true` if x is undefined). It is possible to set a value to undefined through an Update expression ("`x = undefined`").

Where the literal value `undefined` can be used for any data type, the value `empty` is reserved for referring to empty collections, empty lists and empty strings. To understand the difference with `undefined`, consider the following example: `PERSON.middle_name = empty` means that PERSON has no middle name; `PERSON.middle_name = undefined` means that it is unknown whether PERSON has a middle name or not. Note that an empty list can also be represented by `[]`, and an empty string by `''`. It is not possible to say that a number, boolean or date value is `empty`.

Date literals are defined by the regular expression as YYYY-MM-DD, where a leading zero of the month and/or day can be omitted. Time literals are defined as HH:mm:ss:SSS; leading zeroes may not be omitted in any of the constituents, but the seconds and milliseconds (and the associated operators) are optional. If omitted, they are assumed to be 00:00. A datetime literal is a date optionally followed by a time, where the time is assumed to be 00:00:00:00 if omitted.

A period literal contains at least one of the constituents listed for it in the syntax. Between two constituents there must always be at least one space character. Note that SN3 is not interested in linguistic correctness of singular/plural distinctions in periods: both `1 years` and `9 year` are accepted as correct literals.

5.3.3 Examples

```
1
3.14
.3333
true
false
empty
'A text without placeholders.'
'Another text with a placeholder {1+3} which is converted to 4.'
```

- *In the last example, the placeholder {1+3} will be replaced by its value, 4, which is automatically converted to a string in the process.*

5.4 Names

Names are used to directly refer to an element or an attribute.

5.4.1 Syntax

```
#     name = [A-Za-z][A-Za-z0-9_]* ;
```

5.4.2 Semantics and usage notes

Names are used to refer to either an element (such as an instance of an entity), or to an attribute.

When referring to an element, it is possible to define a new name for an element, and to refer to that element by this newly defined name. When referring to an attribute, it is only possible to refer to an existing attribute.

An attribute can only be referred to by just its name when the entity is clear from the context, which in practice is only in create expressions and filter expressions; otherwise an attribute expression must be used (see § 5.11).

As a convention, names are written in upper case when referring to an element or entity, and in lower case when referring to an attribute. This is however not always enforced in the language.

5.4.3 Examples

```
results in:
    "employee has requested leave of absence"
    if   "{EMPLOYEE} is currently old enough"
    then "manager has approved the request"

"employee has requested leave of absence" =
    one REQUEST in REQUESTS is created with:
        start  =          input from EMPLOYEE
```

```
        end    = optional input from EMPLOYEE
        reason =          input from EMPLOYEE

"{PERSON} is currently old enough" =
    "{PERSON} is old enough on {currentDate()}"

"manager has approved the request" =
    REQUEST.approved = input from MANAGER
                       based on {REQUEST.start, REQUEST.end, REQUEST.reason,
                                 EMPLOYEE.name}
```

- *In the above example, the orange highlights show where a name is used to assign an identifying name to an element, so that it can be reused later. There are examples in create expressions, input expressions, and parameters in sentence definitions. The green highlights show where these names are then (re)used. Note also that REQUESTS in the create expression is a reuse, this is the name of an entity that is defined in the domain model. The blue highlights show where attribute names are used; these are all defined in the domain model.*
- *Note that the three uses of EMPLOYEE with orange highlights in the input expression all refer to the same employee, so strictly speaking they can't all be defining a new name. Actor names in input expressions are a special case where identification of actors is concerned; see § 5.10 for details.*

5.5 Unary expressions

Unary expressions are expressions where a value is combined with a unary operator to produce a new value.

5.5.1 Syntax

```
unaryExpression = ( unaryPrefixOperator valueExpression )
                | ( valueExpression unaryPostfixOperator ) ;

unaryPrefixOperator = "-" | "not" | "old" | "current" | "published" ;
unaryPostfixOperator = "exists" | "does not exist" ;
```

5.5.2 Semantics and usage notes

All unary operators require a separator (at least one space character) between the operator and operand(s).

The operators are described below. In this section, examples are given immediately after the descriptions (if needed).

For operator precedence, see § 5.2.

5.5.2.1 Unary prefix operators

Operator	Input → output	Description
`- x`	num → num	Results in the negated value of x. If x is `undefined`, the result is also `undefined`. Note that negative numeric literals (e.g. -3) are written without separator (no space between - and 3): this is not the same thing as the unary operator `-`.
`not x`	bool → bool	Results in the inverse of x. If x is `undefined`, the result is also `undefined`.
`old x`	expr → any	Applied to a value expression x, this operator ensures that the expression is evaluated as early as possible in the process; see also below.
`current x`	expr → any	Applied to a value expression x, this operator states that the evaluation of that expression should be done independently of other expressions; see also below.

`published ENTITY_SYMBOL`	expr → any	Applied to a value expression `ENTITY_SYMBOL`, this operator states that the evaluation of that expression should be done after the publish step of `ENTITY_SYMBOL`.
		E.g.
		`if (published ORDER).decision == 'Abort' then current ORDER.LINES are deleted`
		or
		`ORDER.order_pdf_file = to_file( published ORDER_PDF_DOCUMENT)`
		to make sure the fully completed ORDER_PDF_DOCUMENT is used to make a pdf file of.

5.5.2.2 Unary postfix operators

Operator	Input → output	Description
`x exists`	any → bool	If the value is a collection, this results in `true` if the collection contains at least one element, `false` otherwise. If the value is not a collection, this results in `true` if the value is not undefined; `false` otherwise.
`x does not exist`	any → bool	Inverse of `x exists`; equivalent to `not ( x exists )`.

5.5.2.3 Notes about current and old

The unary operators **`current`** and **`old`** are unique to SN3, and have an effect on the dependencies of the expressions to which the operators are applied, as mentioned above. The difference between the two can be understood as follows.

```
x1 = y + z
x2 = old y + z
x3 = current y + z
```

- When calculating the value of x1, any dependencies resulting from the evaluations of y and z apply. As a result, if the value of y is changed at some point in the process, then the changed value of y is the one used in the calculation of x1.
- When calculating the value of x2, the value that y had before any changes were made to it is used. If the value of y has been changed at some point in the process, this guarantees that the calculation of x2 will not use the changed value but rather the original (old) one.
- When calculating the value of x3, any dependencies that would normally apply to the evaluation of y are ignored. If the value of y has been changed at some point in the process, then the calculation of x3 might use either the old or the new value of y, depending on other dependencies (e.g. the dependencies resulting from the expression z).

The operators old and current can apply to the following types of values.

- Basic attributes: when applied to a basic attribute, the operator impacts the effect of updates on this attribute.
- Relational attribute: when applied to a relational attribute, the operator impacts the effect of updates to this relational attribute, i.e. of adding or removing related elements.
- Entity collections: when applied to an entity collection, the operator impacts the effect of creation into or deletion from the entity collection.

The operators `old` and `current` can be applied to the following expressions.

- Attribute expressions: applied to an attribute expression `A.B.C.x`, the operator refers to the value of the last constituent of the expression, i.e. x (‘*the old (or current) x of A.B.C*’). Any dependencies of `A.B.C` do apply: therefore it does not signify ‘*the old x of the old C of A.B*’ or anything like that. This also applies when the attribute expression is followed by a filter.
- Symbol expressions that are normalised to attribute expressions: same as above. An example of this kind of expression would be `ORDERS[old attr1 == 0]`.
- Symbol expressions that refer to entity collections: applied to an expression such as ORDERS, as in `for the first ORDER in old ORDERS applies`, the operator is used to refer to this collection before any elements are added to it or deleted from it.
- Note that this also applies to filter expressions: in other words, in `old ORDERS[size > 100]`, the operator `old` applies to the entity collection `ORDERS`, not the attribute `size`.

The operators `old` and `current` *cannot* be applied to any other types of expression. Note also that when the keyword `old` is applied to an attribute expression of which the left-hand side is a created symbol (e.g. `old NEW_ORDER.value`), then the keyword makes no sense and is *not allowed*. Keyword

`current` however *can* be used in this capacity. The operator `published` can only be applied to entities, as only these have a publish step.

5.6 Binary expressions

Binary expressions are expressions where two values are combined with an operator to produce a new value. The list of operators below is ordered from high to low precedence: see § 5.2 for details.

5.6.1 Syntax

```
binaryExpression = expression binaryOperator expression ;
binaryOperator = orOperator | andOperator | membershipOperator
               | relationalOperator | equalityOperator | additiveOperator
               | multiplicativeOperator | powerOperator ;
powerOperator = "^" ;
multiplicativeOperator = "*" | "/" | "mod" | "div" ;
additiveOperator = "+" | "-" ;
relationalOperator = "<" | "<=" | ">" | ">=" ;
membershipOperator = "in" | "contains" | "intersects" ;
equalityOperator = "==" | "<>" | "like" ;
andOperator = "and" ;
orOperator = "or" ;
```

5.6.2 Semantics and usage notes

All operators require a separator (at least one space character) between the operator and operand(s).

The operators are described below. In this section, examples are given immediately after the descriptions (if needed).

Note that although the formal syntax for binary expressions specifies that any type of expression is allowed, in practice most operators work only with value expressions, and not with result expressions. In all descriptions below, only value expressions are allowed unless explicitly stated otherwise.

For operator precedence, see § 5.2.

5.6.2.1 Power operator

Operator	Input → output	Description
x ^ y	number, number → number	Result in x to the power y. Decimals and negative numbers are allowed for both x and y. When both x and y are integers and y is positive, the result is an integer; in all other cases the result is a decimal.

5.6.2.2 Multiplicative operators

Multiplicative operators can only be used with numerical values; the rules for number reconciliation apply (see § 5.20), but other implicit type conversion rules do not apply. Note that mod and div allow only integer operands.

Operator	Input → output	Description
`x * y`	number, number → number	Regular mathematical multiplication.
	period, integer -> period	Multiplies all components of the period by the integer number.
`x / y`	number, number → decimal	Regular mathematical division. The second operator y *must* be non-zero. Note that the result is always a decimal number.
`x div y`	integer, integer → integer	Whole-number division: equivalent to round-down of `x / y`. For example: `10 div 3 == 3` The second operator y *must* be non-zero.

`x mod y`	integer, integer → integer	Remainder of whole-number division. Equivalent to:
		`x - ((x div y) * y)`
		For example:
		`10 mod 3 == 1`
		The second operator y *must* be non-zero.

5.6.2.3 Additive operators

Additive operators can only be used with specific combinations of operand types. The allowed combinations are listed in the table below; the implicit type conversion rules of § 5.20 also apply.

Operator	Input → output	Description
`x + y`	number, number → number	Normal mathematical addition of x and y.
	text, text → text	Concatenation of x and y.
	date, time → datetime	Combination of a date and a time into a datetime. Operands must be in this order.
	date, period → date	Adding a period to a date. E.g. `to_date('2022-09-21') + to_period('2 days') == to_date('2022-09-23')`
	datetime, period → datetime	Same as date, period → date but for datetime values.

	collection, element → collection	Addition of an element to a collection. The element *must* be of compatible type for the collection.
	number, text → text	When a number is added to a text, irrespective of the order of the operands, the number is appended to the text, as if it were a text itself. See also § 5.20.1.4.
`x - y`	number, number → number	Normal mathematical subtraction.
	date, period → date	Subtracting a period from a date. E.g. `to_date('2022-09-23') - to_period('2 days') == to_date('2022-09-21')`
	datetime, period → datetime	Same as date, period → date but for datetime values.
	collection, element → collection	Removal of an element from a collection.

5.6.2.4 Equality operators

The general rules for implicit type conversion do not apply for equality operators. Both operands *must* be of the same type, except that integer and decimal numbers can be compared to each other.

Operator	Input → output	Description
`x == y`	any, any → bool	Result is `true` if the value of x is equal to the value of y; `false` otherwise.
	text, text → bool	Result is `true` if both strings are exactly the same; `false` otherwise. This comparison is case-sensitive.
	num, num → bool	Result is `true` if the value of x is equal to the value of y; `false` otherwise. When integers and decimals are compared, the integer is taken to be equivalent to a decimal with only zeros in its decimal part.
	Bool, Bool → bool	Result is `true` if x and y are both `true`, or both `false`; `false` otherwise.
	(date/time/ datetime), (date/ time/datetime) → bool	Both operands have to be of the same type: times can only be compared to times, dates to dates, and datetimes to datetimes.
	period, period → bool	Number of years and months must be the same, if present; for lower constituents, the normalised periods are compared, so that e.g. `24 hours == 1 day` → `true`; see § 5.21.3 for details.

	tuple, tuple → bool	Two tuples are equal if they have the same key-value pairs, with the same keys and the same values, in the same order.
	enum, enum → bool	Result is `true` if the value of x is equal to the value of y; `false` otherwise.
	Collection, Collection → bool	Two collections are equal if they contain the same elements, irrespective of ordering.
	-	In all cases, if x and/or y are `undefined`, the result is `undefined`, i.e., neither `true` nor `false` .
`x <> y`	any, any → bool	Result is `true` if the value x is not equal to the value of y; `false` otherwise. Equivalent to `not(x == y)`. All type-specific comments given for `x == y` apply.

`x like y`	text, text → bool	Applies only to text values, where the second operator may contain wildcards. The wildcard characters are _ for a single character, and % for a string of any number of characters (i.e. 0 or more). Both wildcard characters can be escaped with a \.
		Result is `true` if x matches the wildcard string y; `false` otherwise. If y doesn't contain wildcards, this is equivalent to `x == y`.
		For example:
		`'John' like 'Jo%'`
		`'John' like 'Jo__'`
		`'Jonathan' like 'Jo%an'`

5.6.2.5 Relational operators

Relational operators can be used with the following types: text, number, date, time, datetime. Periods can be compared as long as the month and year constituents are 0; otherwise they *cannot* be compared. All other types *cannot* be compared.

The general rules for implicit type conversion do not apply for relational operators. Both operands *must* be of the same type, except that integer and decimal numbers can be compared to each other.

Operator	Input → output	Description
`x < y`	text, text → bool	Result is `true` if x comes before y if they are sorted (see § 5.21.1 on sorting order for strings); `false` otherwise.
	num, num → bool	Result is `true` if x is smaller than y; `false`otherwise. When integers and decimals are compared, the integer is taken to be equivalent to a decimal with only zeros in its decimal part.

	(date/time/ datetime), (date/ time/datetime) → bool	Result is `true` if x is an earlier moment in time than y; `false` otherwise. Both operands have to be of the same type: times can only be compared to times, dates to dates, and datetimes to datetimes.
	period, period → bool	Can only be compared if the month and year constituents of both operands are 0. The expression compares normalised versions of the two operands, so that e.g. `25 hours < 1 day` → `false`; see § 5.21.3 for details.
	-	In all cases, if x and/or y are `undefined`, the result is `undefined`, i.e., neither `true` nor `false`.
`x <= y`	(type), (type) → bool	Result is `true` if x comes before or is smaller/earlier, or equal compared to y; `false` otherwise. All type-specific comments given for `x < y` apply.
`x > y`	(type), (type) → bool	Result is `true` if x comes after or is greater/later compared to y; `false` otherwise. All type-specific comments given for `x < y` apply.
`x >= y`	(type), (type) → bool	Result is `true` if x comes after or is greater/later, or equal compared to y; `false` otherwise. All type-specific comments given for `x < y` apply.

See also the between-expressions (§ 5.7), which make use of the relational operators in a ternary expression.

5.6.2.6 Membership operators

Operator	Input → output	Description
`x in y`	any, coll → bool	Result is `true` if the value x occurs in collection y, `false` otherwise. The collection y *must* be a collection of elements of the type of x. If x and/or y are `undefined`, the result is `undefined`.
`x contains y`	coll, any → bool	Result is `true` if the value y occurs in collection x, `false` otherwise. Equivalent to `y in x`. The collection x *must* be a collection of elements of the type of y. If x and/or y are `undefined`, the result is `undefined`.
`x intersects y`	coll, coll → bool	Result is `true` if there is at least one element from either collection that also occurs in the other collection; `false` otherwise. If x and/or y are `undefined`, the result is `undefined`.

5.6.2.7 Or and And operators

Operator	Input → output	Description

`x or y`	bool, bool → bool	Result is `true` if one or both operands are true; `false` if both operands are false; `undefined` otherwise.
`x and y`	bool, bool → bool	Result is `true` if both operands are true; `false` if one or both of the operands are false; `undefined` otherwise. Note that "and" is also used as a conjunction between two result expressions, see § 3.5. The and operator may be omitted: this construct, the implicit-and expression, is discussed in § 4.16.

5.7 Between-expressions

🛈 A between-expression is a Boolean-valued ternary expression allowing to check whether one value is between two others, but written with two explicit operators. A between-expression like `a < x < b` is shorthand for `a < x and x < b`.

5.7.1 Syntax

```
    betweenExpression = valueExpression relationalOperator valueExpression
                        relationalOperator valueExpression ;

#   relationalOperator = "<" | "<=" | ">" | ">=" ; // (cf. § 5.6.1)
```

5.7.2 Semantics and usage notes

In a between expression, both relational operators *must* "point in the same direction": i.e. < and <= can be used together in one expression, but < and >, or > and <= cannot.

Between expressions can be used with the same data types as relational operators; the same remarks apply as in §5.6.2.5.

In terms of precedence, between-expressions are below the binary equality operators and above the binary relational operators.

5.7.3 Examples

```
if "minimum for {year}" < "salary of {PERSON} in {year}" <= "maximum for {year}"
then
    "regular bonus calculation is applied to salary of {PERSON} for {year}"
```

❖ *Example of usage of between expression.*

5.8 List expressions

🛈 List expressions are expressions that consist of zero or more values of the same type in a certain order.

5.8.1 Syntax

```
listExpression = "[" [ valueExpression { "," valueExpression } ] "]" ;
```

5.8.2 Semantics and usage notes

Lists can have any length. The elements of a list can have any value, as long as all values in the list are of the same type. Lists are ordered as given.

One of the things lists can be used for is to specify an ordered collection of explicit values for e.g. a drop-down in an input field.

5.8.3 Examples

```
[3, 1, 10, 3]
['1st item', '2nd item', '3rd item']
[ FATHER.age, MOTHER.age, SPOUSE.age, round("average age of {CHILDREN}") ]
[]
```

❖ *Some examples of list expressions.*

5.9 Tuple expressions

Tuple expressions are used to represent freely configurable typed data structures. These are called tuples in SN3, and are defined as ordered collections of one or more key-value pairs. SN3 tuples are very similar to JSON data structures.

5.9.1 Syntax

```
tupleExpression = "{" keyValuePair { "," keyValuePair } "}" ;
keyValuePair = [ tupleKey ":" ] tupleValue ;
tupleKey     = attributeName ;
tupleValue   = valueExpression ;

#    attributeName = name | simpleStringLiteral
```

5.9.2 Semantics and usage notes

A tuple expression consists of at least one key-value pair.

The keys in a tuple expression *must* be unique.

A key-value pair consists of a key (a name) and a value. The key can be omitted if it can be derived from the expression representing the value (e.g., the attribute name in an attribute expression; see below for details). In principle the key should be thought of as a string, but the enclosing quotes can be omitted if the name string contains no spaces.

While tuple itself is a datatype, we also speak of the type of a tuple expression, which we call the tuple-type. The tuple-type of a tuple expression is defined by the collection of its keys and the collection of the types of its values. The ordering of the pairs is relevant: if two tuples have the same keys but defined in a different order, then they are of different types, even if the values have the same types. This definition is not recursive: if one of the values of a tuple is itself a tuple, then the keys of the embedded tuple are not part of the tuple-type of the enclosing tuple.

The tuple-type is relevant in collections: two tuples can be combined in a collection or list if and only if they are of the same tuple-type.

Note that values in a tuple can be accessed through attribute expressions (see § 5.11).

5.9.3 Derivation of tuple keys from value expressions

When the key in a key-value pair is omitted, it is derived from the expression that specifies the value based on the following rules.

- If the value expression is a name, then the name itself is used as the key.
- If the value expression is an attribute expression, then the attribute name in the attribute expression is used as the key.
- If the value expression is a sentence reference, then a normalised form of the sentence is used as the key. This normalised form is produced as follows.
    - All actual parameters are replaced by the corresponding formal parameters from the sentence definition.
    - The brackets enclosing the parameters are removed.
    - The enclosing double quotes are removed and the resulting character string is transformed to a string literal.
- If the value expression is an input expression and it has a labeled clause, then the label is used as the key.
- If the value expression is a the-expression, then the key is derived from the value expression within the the-expression using these same rules.

In all other cases, the key cannot be derived, and therefore may not be omitted. For instance, if the value is a literal, the key cannot be omitted (because it cannot be derived using the above rules).

5.9.4 Examples

```
//Example 1
    { name          : 'John Smith'
    , birth_date    : 1973-03-27
    , CHILDREN      : [ {name: 'Eric Smith',  birth_date: 2004-11-21},
                        {name: 'Alexa Smith', birth_date: 2006-06-16}
                      ]
    }
```

❖ *Tuple consisting of three key-value pairs. Each key-value pair has a different type; the value of CHILDREN is itself a list of tuples. In this way, tuples can be used to build complex information structures.*

```
//Example 2
    { SUBJECT.name, SUBJECT.birth_date, SUBJECT.CHILDREN }
```

- *Tuple consisting only of values (keys omitted). In this case, the key can be derived from the attribute expression that represents the value. Note that this tuple is of the same type as the one in the previous example.*

5.10 Input expressions

Input expressions are used when the process requires information to be provided by an external actor (human or system).

5.10.1 Syntax

```
    inputExpression =
        [ "required" | "optional" ]
        "input from" actorName
        { inputOption } ;

    inputOption
        = "labeled" simpleStringLiteral
        | "chosen from" valueExpression
        | "based on" valueExpression
        | "default" valueExpression
        | "of type" attributeType
        ;

#   actorName = name  // upper case only
#   simpleStringLiteral = // a string literal, but without placeholders,
                          // and without newlines, tabs and such special characters
```

5.10.2 Semantics and usage notes

Input expressions may be explicitly specified as either required or optional. When nothing is specified, the input is ‘required’ by default.

The actor name in an input expression is shorthand for “specific actor with role actorName”. For more details on roles and actors, see § 2.8.

The input options in an input expression can occur in any order. Any input option can be used only once within an input expression.

The option `labeled` specifies a label that is shown to the actor as a prompt. If this option is not present, the attribute to which the input expression is assigned is used as the basis for deriving a label text (see also § 5.21.2).

The option `chosen from` *must* evaluate to a collection of values (elements or basic value) that *must* be of the right type for the attribute that the input expression is assigned to. This collection is presented to the actor in the indicated order, if any, as candidate values for the input expression: the actor must choose one of these values, and cannot input another value.

The option `based on` indicates information that is presented to the actor in order to help him or her to determine the correct input. The based on information is presented to the actor as labeled values, where the labels are derived from the values given in the value expression. If the value expression evaluates to a tuple, then the keys of the tuple are used as labels for the values. If the value expression evaluates to an element or a collection of elements, the values shown are those attributes, if any, that are marked as displayed values in the domain model (see § 6.4); otherwise a default is applied.

The option `default` *must* evaluate to a single value (element or basic value) of the correct type for the attribute that the input expression is assigned to. This value is prefilled in the input form that is presented to the actor, and may be overridden by the actor.

The option `of type` specifies the type of input that is allowed from the actor (see also § 6.3). This is intended to be used when the target of the input cannot (easily) be derived from the target of the expression. In most cases, the type can be derived from the target (in the expression `x = input from` USER, if x is of type text, then the input type is text). If the target is known, the explicit type *must* correspond to the type of the target.

5.10.3 Examples

```
"manager has approved the request" =
   REQUEST.approved = input from MANAGER
                      based on {REQUEST.start, REQUEST.end, REQUEST.reason}
```

❖ *Input expression, implicitly required, with based on information.*

5.11 Attribute Expressions

An attribute expression is used to refer to an attribute of an entity, or to a specific key-value pair of a tuple. It may represent a single value, or a collection of values.

5.11.1 Syntax

```
      attributeExpression = valueExpression "." attributeName ;

#     attributeName = name | simpleStringLiteral
```

5.11.2 Semantics and usage notes

The valueExpression in an attributeExpression *must* represent either a single entity or tuple, or a collection of elements of composite type (i.e., a collection of entities or tuples). Note that as a general rule, if the valueExpression represents a collection (in both cases), all elements in the collection *must* be of the same type.

If the valueExpression refers to an entity or a collection of entities, the attributeName *must* correspond to one of the attributes of the entity type. If the valueExpression corresponds to a tuple or a collection of tuples, the attributeName *must* correspond to one of the keys of the tuple type.

An attributeExpression may result in a single value or in a collection. If the valueExpression represents a single value, then the attributeExpression as a whole has the same type as determined by the attributeName. If the attributeName represents a single value (e.g., a string or a single entity), then the attributeExpression also represents a single value. If the attributeName represents a collection, then the attributeExpression also represents a collection.

If the valueExpression represents a collection, then the attributeExpression also represents a collection irrespective of the attributeName. Note that in this case, if the attributeName represents a collection, the attributeExpression does not result in a collection of collections, but is flattened to a single collection.

If the valueExpression refers to a collection, then the collection resulting from the attributeExpression has the same sorting order as the valueExpression. If the attributeName refers to a collection, the sorting order of this collection is also maintained in the outcome.

If an attributeExpression results in a collection, then this collection is not deduplicated (i.e., may contain duplicates).

5.11.3 Examples

```
//Example 1
THIS_PERSON.family_name
```

- *Single value: the family name of THIS_PERSON*

```
//Example 2
PERSONS.given_name
```

- *The collection of all given names of PERSONS. This may contain multiple instances of the name 'John', for instance. If the collection of PERSONS was ordered by family name, the instances of 'John' will be spread out over the collection.*

```
//Example 3
THIS_PERSON.CHILDREN
```

- *The collection of all children of THIS_PERSON. This collection would be ordered in the default way for relation*

```
//Example 4
PERSONS.CHILDREN
```

- *The collection of all children of all PERSONS. Since most people are children of two parents, most children would appear in this list twice. If there are two children, Jack and Joe, who are half-brothers of the same mother, Jill, and fathers John and James respectively; and the PERSONS are ordered by first name; then the resulting collection would be:*
  [Joe, Jack, Joe, Jack]
  (child of James; children of Jill; child of John)

```
//Example 5
"persons export list".PARENTS
```

- *Assuming the sentence returns a collection of tuples, this results in a collection of the values of the key-value pairs with key PARENTS: in this example this could be a collection of entities.*

5.12 Filter Expressions

A filter expression consists of an expression (which represents a collection of elements of composite type) to which a filter condition is applied, so that the result of the filter expression is a subset of the original collection, consisting only of members of the original collection that satisfy the filter condition.

5.12.1 Syntax

```
filterExpression = valueExpression "[" filterCondition "]" ;
filterCondition = valueExpression | elementName ":" valueExpression ;

elementName = name ;  // upper case only
```

5.12.2 Semantics and usage notes

The valueExpression to which the filterCondition is applied *must* evaluate to a collection of elements of composite type, i.e., a collection of entities or a collection of tuples. All elements in the collection (in both cases) *must* be of the same type.

There are two kinds of filterConditions. For the first kind, represented by valueExpression in the formal syntax, the filterCondition *must* be a Boolean expression, possibly a coordinated expression consisting of multiple atomic Boolean statements. Normally each of the constituents of the filterCondition would make reference to at least one attribute of the entities, or key of the tuples in the original collection.

The second kind of filterCondition starts with a name expression, which introduces a new name that represents the elements of the collection that match the filter condition. The valueExpression in this type of filterCondition specifies the conditions these elements must meet, and therefore *must* make reference to the name.

Only elements for which the filterCondition evaluates to `true` are included in the result collection.

A filterExpression always results in a collection: this can be an empty collection, a collection of 1, or a larger collection. The resulting collection has the same sorting order as the original collection.

5.12.3 Examples

```
//Example 1
PERSONS.CHILDREN[given_name == 'John']
```

- *All elements of PERSONS.CHILDREN whose name is 'John'.*

```
//Example 2
```

```
PERSONS[CHILDREN[given_name == 'John'] exists]
```

- *All elements of PERSONS who have at least one child named 'John'.*

```
//Example 3
PERSONS.CHILDREN[given_name == 'John' and age == 18]
```

- *All elements of PERSONS.CHILDREN whose name is 'John' and who are 18 years old.*

```
//Example 4
PERSONS.CHILDREN[given_name == 'John'][age == 18]
```

- *This is equivalent to:* `PERSONS.CHILDREN[given_name == 'John' and age == 18]`.

```
//Example 5
"children of person{THIS_PERSON}"[residence_status == 'with parents']
```

- *Assuming the sentence results in a collection of PERSONS, returns a subset of this collection.*

```
//Example 6
"persons export list"[domicile == 'Amsterdam']
```

- *Assuming the sentence returns a list of tuples, this results in a list consisting only of those tuples that have a key-value pair where the key is domicile and the value is Amsterdam.*

```
//Example 7
PERSONS.CHILDREN[1 == 1]
```

- *Results in the entire collection PERSONS.CHILDREN.*

```
//Example 8
PERSONS.CHILDREN[P: P <> PERSON1]
```

- *Results in the collection of all elements of PERSONS.CHILDREN that are not PERSON1.*

```
//Example 9
PERSONS.CHILDREN[1 == 2]
```

- *Results in an empty collection.*

```
//Example 10
PERSONS.family_name[age == 18]
```

- *Results in an error:* `PERSONS.family_name` *is a collection of strings, which is not a composite type.* `PERSONS[age == 18].family_name` *on the other hand is a valid expression.*

5.13 Sort expressions

Sort expressions result in a collection being sorted in the order specified by the sort clause. These can be used for collections of composite types, as well as for lists of simple values.

5.13.1 Syntax

```
      sortExpression = valueExpression "sorted by" sortClause { "," sortClause } ;
      sortClause = attributeName { '.' attributeName } [ 'ascending' | 'descending' ] ;

 #    attributeName = name | simpleStringLiteral // either upper case only, or
lower-case only
```

5.13.2 Semantics and usage notes

The value expression in a sort expression *must* be a collection.

If the value expression is a collection (list) of simple values, then the sort expression *must* have a single sort clause. The sort clause *must* have a single attribute name, and in place of the attribute name the keyword ‘value’ *must* be used.

If the value expression is a collection of elements of composite type, then each of the attribute names in each of the sort clauses *must* correspond to an attribute (or a key) of the composite type in the collection.

Collections can only be sorted on attributes of the following types: text, number, date, time, and datetime. A sort expression referring to any other type of attribute results in an error.

For dates and times, ascending sort order is defined as old to new; descending sort order is new to old.

Note that an expression that is not explicitly sorted is conceptually "unsorted", which is to say the elements exist in some technically defined order which cannot functionally be taken for granted.

5.13.3 Examples

```
//Example 1
PERSONS sorted by last_name, first_name ascending, date_of_birth descending
```

- *Results in a list of PERSONS sorted alphabetically by last name; PERSONS with the same last name are subsorted alphabetically by first name; PERSONS with the same first and last name are sub-subsorted from young to old. If two PERSONS have the same first name, last name, and birthday, then their sort order is undefined.*

```
//Example 2
PERSONS.last_name sorted by value ascending
```

- *Results in a list of last names, sorted alphabetically.*

5.14 The-expressions

A the-expression results in the selection of a single element from a collection of elements. The construct “the only” transforms a collection of 1 to a single element; the constructs “the first” and “the last” select the first and the last element of a collection, respectively.

5.14.1 Syntax

```
theExpression = "the" ( "only" | "first" | "last" ) collectionExpression
                [ "otherwise" alternativeOutcomeExpression ] ;

collectionExpression = valueExpression ;
alternativeOutcomeExpression = valueExpression ;
```

5.14.2 Semantics and usage notes

The collectionExpression in a the-expression *must* be a collection. This can be either a collection of elements of composite type, or a collection (list) of simple values.

Note that the keywords `the first` and `the last` are only meaningful for ordered collections; if the collection is unordered (no ordering is specified), then conceptually, a random element of the collection will be selected (whichever happens to be first or last in the (unspecified/technical) ordering).

If the collectionExpression refers to an empty collection, and the the-expression has no otherwise-clause, then the the-expression itself results in the value `undefined`. If however the otherwise-clause is present, then the the-expression results in the value of alternativeOutcomeExpression.

If the keyword `the only` is used and the collectionExpression refers to a collection of more than one element, then the the-expression is incorrect. In a generated application, this would result in a runtime error. Note that this is the only kind of error that cannot be caught by the otherwise clause!

The value expression of the alternative outcome expression *must* have a value of the same type as the elements in the collectionExpression of the the-expression.

5.14.3 Examples

```
//Example 1
the only PERSONS[last_name == 'Jones' and date_of_birth < 1950-01-01]
```

- *Results in the single entity of type PERSON who meets the criteria. Results in an empty list if there is no PERSON who meets the criteria. If there is more than one person who meets these criteria, then this expression results in an error.*

```
//Example 2
the last PERSONS sorted by last_name, first_name ascending, date_of_birth descending
```

- *Results in the single entity of type PERSON who is last in the sorted list.*

```
//Example 3
the first "measured temperatures"
```

- *Given that the sentence results in a list of temperature measurements, which are decimal numbers, this expression returns the first of that list.*

5.15 Function expressions

Function expressions are used to apply predefined calculations to produce values of various data types. Functions can be used in two ways: in function notation and in postfix notation.

Note that this language reference does not include a list of available functions; that can be found in the separate document SN3 Functions.

5.15.1 Syntax

```
      functionExpression
         = functionName "(" [ functionParameterList ] ")"
         | valueExpression "." functionName "(" [ functionParameterList ] ")" ;

      functionParameterList = valueExpression { "," valueExpression } ;
#     functionName = name ;
```

5.15.2 Semantics and usage notes

Functions are a flexible way of adding functionality to SN3. An exhaustive list of available functions and their exact definitions is outside of the scope of this document: that information can be found in the document SN3 Functions.

Function notation consists of the function name, followed by the complete parameter list between parentheses. In postfix notation, the function expression is appended to a value expression using dot notation. In this case, the value expression that is written before the function name is the first parameter, and the parameter list between parentheses contains the remaining parameters, if any.

A function can have zero or more parameters. Most functions have at least 1 parameter; common examples of functions with zero parameters are `currentDate()` and `pi()`. Note that if a function is used in postfix notation, it *must* have at least one parameter: i.e., `pi()` cannot be used in postfix notation.

The parameters in the function parameter list have a fixed ordering, which is defined as part of the function definition.

Parameters in the parameter list may be optional. If a parameter is optional, the parameter list may either omit that parameter, or it may be given the value `empty`. The definition of the function specifies how optional parameters are to be represented.

Note that in postfix notation, the first parameter is always supplied. If the first parameter is optional and needs to be omitted in a particular situation, postfix notation cannot be used in that situation.

Functions require their parameters to be typed correctly, as per the descriptions in the documentation for SN3 functions; however the rules for implicit type conversion apply (see § 5.20).

5.15.3 Examples

```
//Example 1
    day(2018-04-17)                                 // => 17
    month(CONTRACT.start_date)                      // month nr of start_date, e.g. 4
    count(['1st item', '2nd item', '3rd item'])     // => 3
    round("average age of subjects")                // rounds average age, e.g. 21
    round(3.14, 1)                                  //=> 3.1
```

- *Some examples of functions in function notation. The results are given as comments. Note that in the examples, the function round() has an optional parameter, which is the number of decimals the first parameter must be rounded to. The default is 0.*

```
//Example 2
    currentDate().day()                             //== day(currentDate())
    'text,with,commas'.split(',')                   //== split('text,with,commas', ',')
                                                    //=> ['text', 'with', 'commas']
```

- *This example shows a few functions in postfix notation.*

5.16 Implicit and-expressions

The binary operator `and` may be left out: an expression with a "missing" `and` operator is called an implicit and-expression. This construct can be used if this is considered to improve the readability of the specification.

5.16.1 Syntax

```
    implicitAndExpression = expression sequenceSeparator expression ;

#   sequenceSeparator = one or more newlines with same indent level
```

5.16.2 Semantics and usage notes

When the `and` operator is left out, there has to be a newline between the expressions, and both expressions must have the same indent level.

Since and is a Boolean operator, this is allowed only if both expressions are Boolean-valued. Note that the implicit and construct is also allowed for result expressions: this usage is discussed in § 3.5.

5.16.3 Examples

```
"{PERSON} is allowed to drive independently"
description 'Lists all the conditions under which a person is allowed to drive without supervision.'
=
   "{PERSON} is old enough to drive on {currentDate()}"
```

```
"{PERSON} passed the driving test" and "{PERSON} has a valid license"
"{PERSON} has no more than allowed alcohol level in blood"
```

❖ *Collocated list of Boolean-valued expressions with implicit and operators. Note that there is also one explicit and operator.*

5.17 If-expressions (values)

🛈 An if (value) expression results in one or the other value, depending on a condition; or potentially in no value at all.

5.17.1 Syntax

```
ifExpression = "if" condition
               "then" expression
               "else" expression ;

condition = valueExpression
```

5.17.2 Semantics and usage notes

If the if-expression is a value expression, both the then-clause and the else-clause *must* be value expressions. Also, both clauses must evaluate to the same type of value. Note that in a value if-expression, the else-clause is mandatory: the expression has to represent an explicit value, whether the condition is `true` or `false`.

The value expression in the condition *must* be of Boolean type. Also, it *must* have a value of either `true` or `false` (in other words an `undefined` value, as in a Boolean attribute that has not been assigned a value, is not allowed).

5.17.3 Examples

```
//Example 1
   ORDER.approved = if ORDER.price < 1000 then true else input from CONTROLLER
```

❖ *Example of if-expression (value).*

5.18 For-each expression (values)

🛈 A for-each expression represents a list of values, where each value is calculated based on an element of a given collection.

5.18.1 Syntax

```
forEachExpression =
       "for each" elementName
       "in" ["numerous"] collectionExpression ["applies"] ":"
       bodySeparator
       expression ;
```

```
#     elementName = name ;  // upper case only
#     collectionExpression = valueExpression ;  // cf. § 5.14.1

#     bodySeparator = // one or more newlines with increase of indent level
```

5.18.2 Semantics and usage notes

The collectionExpression *must* result in a collection of elements, so that the value is computed for each element in that collection. Each of these elements is in turn represented by the name elementName. The collection may be a collection of composite values, or a collection of simple values. If the size of the collection is 0, then no values are computed.

In a for-each *value* expression, the expression in the body *must* be a value expression. Also, the keyword `applies` *must* be omitted (this keyword indicates the expression is a result expression), and the keyword `numerous` *must* be omitted (only allowed for result expressions).

In the body of the for-each expression, the element name is taken to refer to the element of the for-each expression. Any other possible interpretation of the name is overriden.

Normally, the elementName is intended to be used in the enclosed expression, but this is not enforced. If the elementName is not used, a warning will result.

5.18.3 Examples

```
for each APPROVED_EXPENSE in EXPENSES[approved]:
   APPROVED_EXPENSE.total_value * 0.15
```

- *Results in a list of numbers, where each value is 15% of the total value of an approved expense. If the collection of EXPENSES had been sorted, then the list of values would be in the same sorting order.*

5.19 For expression (values)

A for expression is like a for-each expression, except that the value is calculated for only one element and not for a collection of several elements. This also means it represents a single value, rather than a list of values.

5.19.1 Syntax

```
     forExpression = "for" name "=" valueExpression { "," name "=" valueExpression }
                     ["applies"] ":"
                     bodySeparator
                     expression ;

#    bodySeparator = // one or more newlines with increase of indent level
```

5.19.2 Semantics and usage notes

In the scope of the for expression, all of the defined names are taken to refer to the element of the for expression. Any other possible interpretation of the name is overridden.

Normally, the names are intended to be used in the enclosed expression, but this is not enforced. If the names are not used, a warning will result.

In a for value expression, the expression in the body *must* be a value expression. Also, the keyword applies *must* be omitted (this keyword indicates the expression is a result expression).

5.19.3 Examples

```
for ORDER = the only ORDERS[CUSTOMER == CUSTOMER1 and date_placed == 2017-02-01]:
    ORDER.total_value / count(ORDER.ORDERLINES)
```

- *Assuming that there is only one ORDER that meets the criteria, this expression results in the average value of its ORDER_LINES. If there is more than one ORDER that meets these criteria, then this expression results in an error.*

5.20 Calculations, precision and types

When performing calculations, or really operations of any kind, errors with operand types and precision mismatches are among the most common and frustrating computing problems. In SN3, this may affect unary, binary and ternary expressions, functions, lists, and assignments. In this section we discuss how SN3 helps to handle such issues, and what specific pitfalls exist involving types in SMART specs.

5.20.1 Data type considerations

5.20.1.1 Calculating with numbers

We use the term *number* in SN3 to mean any kind of numerical value (numbers are formally defined in § 6.3). In addition, we may distinguish between integer numbers or *integers* (i.e., numbers with no decimal part), and decimal numbers (or *decimals*). Decimal numbers consist of two parts, which we call the *integer part* and the *decimal part*, separated by the decimal separator (which is always a period in SN3). Floating point numbers (the general computer representation of numbers with variable precision) are not used in SN3: every number has a specific precision (number of digits in the decimal part) which must be made explicit in its type.

Precision is a persistent problem when doing calculations with numbers. For instance, if you multiply two numbers like the following: `9.9 * 9.9`, then the result is `98.01`: both original numbers had a precision of 1, but the result has a precision of 2. If you try to assign this result to an attribute which is defined as `number(2,1)`, then this is a type error, as we will discuss below. Limited precision can also cause rounding errors: if you work with numbers of type number(4,2), then 10.00 divided by 3 is 3.33; if you store this number, then retrieve it and multiply it by 3, the outcome would be 9.99.

The basic rules when performing calculations with numbers in SN3 are as follows. This applies for the regular arithmetic operators, i.e. power `^`, multiplication `*`, division `/`, addition `+`, subtraction `-`.

- Any operand of a numerical operation can be any kind of number, whether integer or decimal, with any number of digits in the decimal part. A formula like `4 * 8.76414` means exactly what you'd expect it to.
- The result of such a calculation will have the precision it needs to represent the result as well as possible (see technical note below). Internally, the result of a formula like `1.1 / 3.03` can never be represented exactly, but semantically it will always have at least the precision it needs to get correct outcomes, if possible.
- Issues may arise when trying to *use* the results of a numerical calculation, for instance when assigning it to an attribute of an entity. These are discussed below.

Note that the above is not true for the arithmetic operators `div` and `mod`, which both allow only integers as operands and always yield integers: so these are not part of this discussion.

Technical note: Internally, the results of (specifically) divisions are represented in the decimal 128 number format, a floating point standard that allows for (roughly) 34 digits. What this means in practice is the following. Take the formula `x = y * (a / b)`, where `a = 1` and `b = 3`. If y equals one septillion (a very large number consisting of a 1 followed by 21 zeros), then the result x (rounded down) will be a number consisting of 21 times the digit 3. If y equals 1 quindecillion (a 1 followed by 48 zeros), then the result x *should* be a number consisting of 48 times the digit 3; but in reality this will not be the case, since the intermediate representation of `1/3` does not have sufficient precision for that.

5.20.1.2 Comparisons with numbers

In general, comparison operators (equality operators, relational operators, and membership operators) can only be applied if both operands are of the same type. While it is possible to argue that the expression `3 == 'hello world'` is meaningful (and, obviously, false), in SN3 it is regarded as nonsensical and therefore an error.

When comparing numbers however, it is possible to compare decimals with integers, or differently-dimensioned decimals with each other. In these cases, the implicit assumption is that the less precise operand has a longer decimal part with as many additional zeros as necessary. For instance, `33 < 33.001` is interpreted as `33.000 < 33.001`; `33.1 < 33.001` is the same as `33.100 < 33.001`; and `33 == 33.000` evaluates to `true`.

5.20.1.3 Operations with dates and times

When using temporal types in expressions (i.e., dates, times, and datetimes), the rule is that the type must be correct. Dates and times can be converted to datetimes, and vice-versa, explicitly through the use of the applicable functions: this is never done implicitly.

5.20.1.4 Operations with text

The + operator can be used with one operand of type text, and one of type number: in this case the number operand will be assumed to have been intended as a text, implicitly converted to text, and appended to the text operand. For example, in the expression `'the result is ' + 3.14`, it will be assumed that the number 3.14 is intended as a text, so it will be appended as a text after the string,

resulting in a value of type text (`'the result is 3.14'`). This feature improves readability, by reducing the number of functions in a specification in cases where the conversion is pretty obvious.

Note that this implicit type conversion is subordinate to the regular operator precedence rules explained in § 5.2. This means that expressions are evaluated in the order that is prescribed by the precedence rules, and that implicit type conversion only starts to apply at the moment the first operation involving a text value is processed. The following examples make this clear.

```
//Example 1
   'The result is ' + 3 + 4 ==> 'The result is 34'
```

- *The first + is evaluated first, so 3 is converted to a text value and appended to the sentence; then 4 is also converted to a text value and also appended.*

```
 //Example 2
  'The result is ' + (3 + 4) ==> 'The result is 7'
```

- *The parentheses force the last + to be evaluated first, leading to the result 7, which is then converted to text and appended to the sentence.*

```
//Example 3
   3 + 4 + ' is the result' ==> '7 is the result'
```

- *The first + is evaluated first, as a normal addition, and only after that, when the second + operator is processed, is the result converted to text and prepended to the sentence.*

```
//Example 4
   'The result is ' + 3 * 4 ==> 'The result is 12'
```

- *According to the precedence rules, the multiplication is evaluated first; then its result is converted to text and appended to the sentence.*

As these examples also make clear, it is generally advisable to use parentheses to force the correct evaluation order when implicit type conversion is in play.

Note also that when a decimal number is implicitly converted to text, the precision of that number is whatever is needed to represent its value as well as possible. Therefore if the value is the result of the calculation `9.30 / 3`, the resulting string after implicit conversion will be `'3.1'` - it is not correct to assume the result is `'3.10'`. Also if the value is the result of the calculation `1.00 / 3`, then the resulting value should have a large number of 3's in the decimal part - it is not correct to assume the resulting string will be `'0.33'`.

Other types are not automatically converted to text. Although booleans, dates, times, datetimes and periods are fairly easy to convert, the requirements for how these should be formatted are usually very specific, and should be made explicit.

5.20.2 Considerations for specific expressions

5.20.2.1 Assignments with numbers

As mentioned above, assignments involving numbers with the wrong dimensions can lead to errors. The general rule is that assigning to a higher precision is allowed, while assigning to a lower precision isn't. I.e., if you have an attribute `price` of an entity type PRODUCT which is defined as `number(4,2)` (which means the maximum value is 99.99), then you can assign the value 12 to it: this will be interpreted as 12.00. However you cannot assign the value 12.345 to it: this number would need to be rounded to two decimal positions, and the specifications must make explicit just how the number is rounded.

When assigning numbers, problems with dimensions can occur in two ways: in the integer part (the assigned number is too big), or in the decimal part (too many decimal digits).

- Assigning a number with too many decimal digits is considered a language error. When SMART specifications are analyzed, this type of error is detected and reported as much as possible. Note though that this is not foolproof, so runtime errors cannot completely be excluded. As a guideline, it is recommended to take explicit measures to make sure that the decimal part of a number is dimensioned correctly for the use that is made of it (the attribute it is assigned to).
- As a specific case of the above, the result of a division can *never* be directly assigned to a variable of type number, no matter what the precision is. This is because the result of a division potentially has infinite precision: it therefore always needs to be explicitly rounded before it can be stored. An assignment of an unrounded result of a division is considered a type error.
- Assigning a number that is too large (overflow) is *not* considered a language error, and will not be reported as such when analyzing SMART specifications. If it happens operationally however it will result in a runtime error (i.e., if the actual result of an actual calculation turns out to be too big). It is recommended to be aware of this risk as much as possible, and to take measures to explicitly deal with numbers that exceed the allowed size if there is any chance of that happening.

5.20.2.2 Assignments with dates and datetimes

When assigning values to targets of type date or datetime, the rule is that the type of the value must match the type of the target. If the types do not match, they must be converted explicitly.

5.20.2.3 Assignments to text types

When assigning values to targets of type text, the type of the value must match the type of the target. Implicit type conversion does not apply in this case: assigning a number to an attribute of type text is an error.

5.20.2.4 Functions

Function expressions make use of functions, which are named chunks of functionality that are directly programmed into a programming language such as Java. Functions are defined in function definitions, which are descriptions (not specifically in SMART Notation) of what exactly the function achieves and what kinds of parameters it takes.

In the case of functions, the function definition has to be completely explicit in how it deals with different kinds of input (parameters of different types). Function parameters are never implicitly converted to another type.

5.20.2.5 Lists

In list expressions, there is no implicit type conversion. All elements of a list must have the same type, and in a list of numbers, all elements must also have the same dimensions. A list mixing integers with decimal numbers is considered a type error.

5.21 Additional rules

5.21.1 Sorting of text strings

Text strings are sorted based on "collation". This is basically the country and language setting, a.k.a. "locale". In western world countries the most used collation is "US English". This collation places punctuation characters (a space, `!`, `"`, `$`, …, `/`) first, followed by digits (0, 1, 2, …, 9) followed by alphabetical characters (a, b, …, z).

Lowercase alphabetical characters, capital case characters and diacritic characters are considered equal during sorting, however amongst each other lowercase characters come first, followed by capital case and lastly followed by diacritics (e.g. a, A, á).

See Appendix C Extended ASCII and default sorting for a detailed list of characters in their (US English) order.

5.21.2 Derivation of labels

(TBD - regels voor afleiding labels uitleggen)

5.21.3 Normalization of periods

(TBD)

6 Entity definitions

[1]

6.1 Logical Domain Model

Entity definitions are used to define the domain model (or logical information model), i.e., the set of entities that exist in the application domain, and their attributes. The technical data model is generated based on the entity definitions: what precise form this technical model takes depends on the implementation, but the most natural way to think about it is in terms of tables with columns and foreign keys in a relational database. It is important to keep in mind that the domain model, which is a logical model, is a more abstract representation than the underlying technical model.

6.2 Entities and attributes

[1] An entity is a collection of elements of the same type. An entity type is defined by an entity definition and determined by the name of the entity and the set of its attributes.

6.2.1 Syntax

```
    entityDefinition = entityType "in" entityCollection
                       { "description" descriptionLiteral }
                       "=" bodySeparator entityBody ;

    entityBody = attributeDefinitions
                 [ definitionSeparator entityConstraintsClause ] ;
    attributeDefinitions = attributeDefinition
                           { sequenceSeparator attributeDefinition } ;
    attributeDefinition = attributeName
                          ":" attributeType
                          { attributeTrait }
                          [ calculatedAttributeDefinition ] ;

    entityType = name  // upper case only
    entityCollection = name  // upper case only
#   attributeName = name  // either upper case only, or lower-case only

#   descriptionLiteral = // an HTML clob enclosed in single quotes
#   bodySeparator = // one or more newlines with increase of indent level
#   definitionSeparator = // one or more newlines
#   sequenceSeparator = // one or more newlines with same indent level
```

6.2.2 Semantics and usage notes

Both entity type names and entity collection names *must* be unique throughout all requirements of a project. To avoid confusion, it is strongly recommended to also ensure that there is no overlap between these two collections (i.e., that no entity type name is identical to any entity collection name).

The order in which entities are defined, and also the order in which their attributes are defined has only cosmetic relevance in a SMART Requirements specification: it may affect the order in which information is presented in some circumstances, but does not affect the resulting process in terms of outcomes.

Entity types and entity collections *must* be written in all-caps. As a guideline, moreover, entity types should be singular nouns (e.g. PERSON, COMPANY), and entity collections should be plural nouns (e.g. PERSONS, COMPANIES). It is strongly recommended to adhere to this guideline for the readability of the SMART Requirements specifications.

Attribute names of relational attributes and of attributes of type tuple (see § 6.3) *must* be written in all-caps; attribute names of all other basic attributes *must* be written in lowercase. Various guidelines apply to the naming of entities and attributes, which are comparable to general data modelling guidelines; these are not further discussed here; see the appropriate guidelines and training material.

Attribute names *must* be unique within an entity definition: no two attributes of an entity type can have the same name. This restriction is case-insensitive: e.g., birthplace and BIRTHPLACE are considered to be the same attribute name and therefore not both allowed on one entity.

When an attribute value is displayed in a user interface, it is usually labeled in order to make clear what it represents. If no label is explicitly specified (see attribute trait `labeled`), then a default label is derived from the attribute name by substituting spaces for underscores, making the first character uppercase and the rest lowercase (e.g. `birth_date` will become Birth date, `ORDER_LINES` becomes Order lines).

The scope of an entity definition is determined by the specification in which the definition is placed. Entities that are defined in a process specification (cf. § 2.2) can be referenced by any requirement definition used in that process. Entities that are defined in a reusable specification (cf. § 2.3) can be referenced from any process or requirement definition in any process in the entire project.

As mentioned above, entity types and collections must be unique for the entire project. So even though an entity type placed in a process specification can only be referenced from within that process, another entity type in another process, or another entity type in a reusable, *may not* have the same name.

6.2.3 Examples

```
PERSON in PERSONS =
   ss_number    : number unique labeled 'Social security number'
   first_name   : text(25)
   last_name    : text
   birthdate    : date
   EMPLOYERS    : multiple COMPANY optional opposite of EMPLOYEES
```

```
MOTHER      : PERSON
```

❖ *The first four attributes are basic attributes which can be assigned a value. The attribute EMPLOYERS is a link to potentially several entities of type COMPANY (which is why the attribute name is a plural noun). The attribute MOTHER is a relation to another PERSON.*

6.3 Attribute types

The type of an attribute determines what kinds of values the attribute can assume.

6.3.1 5.3.1 Syntax

```
attributeType
   = "multiple" attributeType
   | "text" [ "(" nonzeroInteger ")" ]
   | "number" [ "(" nonzeroInteger [ "," nonzeroInteger ] ")" ]
   | "boolean"
   | "date"
   | "time"
   | "datetime"
   | "period"
   | "tuple"
   | "binary"
   | enumerationType
   | entityType
   ;

enumerationType = "[" simpleTypeLiteral { "," simpleTypeLiteral } "]"
simpleTypeLiteral = integerLiteral | decimalLiteral | simpleStringLiteral
                  | dateLiteral | timeLiteral | datetimeLiteral | periodLiteral
                  | booleanLiteral ;

#  simpleStringLiteral = // a string literal, but without placeholders,
                         // and without newlines, tabs and such special characters
#  entityType = name  // upper case only, cf. § 5.2
```

6.3.2 Semantics and usage notes (basic types)

Concerning attributes and attribute types, we use the following categorizations.

- **Relational attributes** are attributes whose type is an entity type. A relational attribute signifies a relation from this entity type to another entity type. Relations from an entity with itself are allowed.
- We use the term **basic attributes** to refer to all other attributes, with any of the other attribute types (such as text, number, date, etc.). A basic attribute signifies a quality or characteristic of an entity. Complex values such as tuples and multiple-valued types are considered basic. So in essence, basic means “not relational”.

- **Complex attribute types** is a term that refers to relational attribute types, to the tuple type, and to multiple-valued attribute types..
- All other attribute types are called **simple types**; so simple, unsurprisingly, means “not complex”.

All attribute types can exist as multiple-valued types. A multiple-valued text attribute, for instance, has a value that is not a string, but a list of strings. The most common use of multiple-valued types is to indicate a higher-order cardinality for relational attributes: when a relational type is multiple-valued, this means the cardinality is not to-one, but to-n (see below).

6.3.2.1 Semantics and usage notes (basic types)

The table below contains a brief description of each of the types of basic attributes, and adds semantics and usage notes. Relational attributes are discussed in a separate section below.

Type	Description & notes
`text`	Denotes textual values (i.e., character strings allowing any Unicode character). String literals are written between single quotes (e.g. `'this is a string literal'`); for details see § 5.3. The maximum allowed length of the character string can optionally be specified by a nonzero integer between parentheses. If no maximum length is specified, the default maximum length is 255 characters. If the maximum length is explicitly specified, it may be larger than the default. When a value is assigned to an attribute, this value *must* conform to the length restrictions. Newlines are not allowed in a text-valued attribute unless it has the multiline trait (see § 6.4).
`number`	Denotes numerical values, both integers and decimals. All numbers can be positive (unsigned), negative or zero, unless otherwise constrained. A number type may optionally be followed by 1 or 2 quantifiers (nonzero integers) between parentheses. This has the following interpretation. • *No quantifiers*: the type denotes an integer, with no maximum length specified. In this case the default maximum length applies, which is 18 digits • *One quantifier between parentheses*: the type denotes an integer, the number indicates the maximum length. When explicitly specified, the maximum length may be larger than the default maximum length. • *Two quantifiers between parentheses*: the type denotes a decimal number, i.e., a number consisting of an integer part (the part before the decimal separator) and a decimal or fractional part (the part after the decimal separator). The *first* quantifier signifies the maximum number of digits of the entire number. The *second* quantifier signifies the maximum number of digits of the decimal part (after the decimal separator) - so *first* - *second* = maximum number of digits of the integer part. In a decimal type, the first quantifier *must* be larger than the second. When a value is assigned to an attribute, this value *must* conform to the length restrictions.

`boolean`	Denotes Boolean values, i.e., either `true` or `false`.
`date`	Denotes a date value, which consists of a year, a month and a day. Date literals are written in the YYYY-MM-dd format, with hyphens as separators; see § 5.3 for details.
`time`	Denotes a time value, which consists of an hour, minute, and optionally second and millisecond value. Time literals are written in the HH:mm:ss:SSS format, with colons as separators; see § 5.3 for details. Note that this is not a duration, but denotes a point in time (on an unspecified date). The seconds and milliseconds are optional. Times are specified using the 0-23 format, i.e., 24:00 hrs is not a legal time expression (this should be 00:00 hrs). If the seconds and milliseconds in a literal time value are omitted, they are assumed to be 00:000.
`datetime`	Denotes a datetime value, i.e., a combination of a date and a time. Datetime literals are written as a date followed by a time, separated by a space (or equivalent character). The time 00:00 of a day denotes the start of the day, not the end of it. Therefore 23:59:59:999 is the last millisecond of a day. The time part is optional. If the time part of a datetime value is not specified, it is assumed to be 00:00:00:000.
`period`	Period values denote a length of time, expressed in any combination of years, months, weeks, days, hours, minutes, seconds, and milliseconds. Note that the exact amount of time in a period (which we call the duration) is unknown if it contains years or months. A period literal is written as a combination of whole numbers and text labels, e.g. 1 year 20 days 19 minutes; see § 5.3 for details. Any of the constituents of a period literal may be omitted; the ones that are present *must* be in the normal order (i.e. big to small). Note that periods can be written in various ways: e.g., 73 hours 120 minutes is a valid (though counterintuitive) alternative way of writing 3 days 3 hours.
Duration	*Note about durations: The exact amount of time in a period, expressed most exactly in milliseconds, is called its duration. This is not a separate type in SN3: if a duration is needed in a specification, it should be modeled as an integer. Depending on the required accuracy, this could represent milliseconds, hours, or days, among others: the requirements should make clear what exactly is meant by a duration, and how it is computed or used in computations.* *The duration of a given period (if it contains months or years) depends on the start date that is chosen for it. E.g., depending on the start date, a period of 1 month can consist of 28, 29, 30, or 31 days.*
`tuple`	A tuple is a list of key-value pairs. The optional key is a character string that is used to identify the value, and the value can be of any type, including lists and collections of elements or other tuples. This tuple is therefore a structural type similar to JSON or XML.

	Tuples are written as comma-separated lists of key-value pairs enclosed in curly brackets; the key and value are separated by a colon. The keys in a tuple must be unique. While tuple itself is a datatype, we also speak of the type of a tuple expression, which we call the tuple-type; see § 5.9 for details. It is important to note that the attribute type `tuple` merely signifies that an attribute is some kind of tuple, not the exact tuple-type. In other words, if you have an attribute `attr` of type tuple belonging to an entity type ENT, and two entities ENT1 and ENT2 of type ENT, then `ENT1.attr1` may have a different tuple-type than `ENT2.attr1`.
`binary`	Used for binary or encoded values such as pictures, videos, various types of files or any other binary value. *(Note: maxlength currently undefined, TBD)*
Enumeration type	An enumeration type value is a text value that is restricted to a predetermined number of allowed values. The attribute type definition consists of a list of the allowed values. All values must have the same simple type.

6.3.2.2 Semantics and usage notes (relation types)

A relation type represents a relation of the source entity type with one or more instances of another entity type, the target entity type. Note however that a relation type only signifies one direction of the relation, from the source to the target entity. The other side of the relation needs to be separately specified on the target entity type using the `opposite of` trait (see § 6.4).

The degree of a relation may be single or multiple, i.e., the value of the relational attribute can be either a single element of the target entity type, or a collection of elements of the target entity type. The default is single: multiple-valued relation types are indicated with the keyword `multiple`.

In general data modelling terms, the cardinality of a relation is customarily expressed using shorthand like 1-1 (one to one), 1-n or 1-+ (one to many), 0..1-1 or ?-1 (zero or one to one). In SN3 entity definitions, these three examples are expressed as follows (see also explanation of `opposite of` trait below).

- 1-1: relation is single and required, opposite relation is also single and required.
- 1-+: relation is multiple and required, opposite relation is single and required.
- ?-1: relation is single and required, opposite relation is single and optional.

In this way, all kinds of cardinality can be expressed using the right combinations of opposite relations and the trait `optional` (if omitted the attribute is required).

Although it is recommended to always create opposites of relations, in specific cases there may be reasons to NOT specify an opposite relation. In that case, the opposite relation does not exist in the requirements, and cannot be used. If for instance there exists a 1-n relation STREETS between a city and its streets, but not the opposite 1-n relation from streets to the city they're in, then we can easily query for "the streets of city X". We can however not query directly for "the city of street Y": this would have to be done indirectly ("find the only city X such that given Y is one of the streets of X").

Note that in the technical model, in a relational database, the foreign key relation may actually be placed on the "other" side: in the example with streets and cities, the foreign key can *technically* only be placed on the side of the streets, pointing to its city. This does not change the fact that *logically* (i.e., in the SMART Requirements specification), the only relation you can use in this example is from cities to streets, unless you specify the opposite relation in the domain model.

6.3.3 Examples

(TBD)

6.4 Attribute traits

An attribute trait is a feature that applies to an attribute in addition to its type.

6.4.1 Syntax

```
      attributeTrait
         = "unique" [ simpleStringLiteral ]
         | "optional"
         | "autovalue" valueExpression
         | "displayed"
         | "multiline"
         | "action"
         | "default" valueExpression
         | "labeled" simpleStringLiteral
         | "group" simpleStringLiteral
         | "opposite of" attributeName
         | "description" descriptionLiteral
         ;

#     attributeName = name  // either upper case only, or lower-case only
#     simpleStringLiteral = // a string literal, but without placeholders,
                            // and without newlines, tabs and such special characters
#     descriptionLiteral = // an HTML clob enclosed in single quotes
```

6.4.2 Semantics and usage notes

Attribute traits can be specified on an attribute in any order. Any trait can be used no more than once per attribute.

The trait `unique` can be applied to any attribute type, and ensures that no two elements of the entity collection can have the same value for this attribute. The attribute value *must* be unique when an element of this collection is saved. The trait `unique` cannot be used with multiple-valued attribute types.

The `unique` trait can optionally be supplemented with an attribute cluster name. If this is the case, then the attribute by itself does not need to be unique, but the combination of all attribute values with the `unique` trait and the same cluster name *must* be unique when an element of this collection is saved.

The trait `optional` defines if an attribute is optional and therefore can have an undefined value. When this trait is omitted, the attribute will be required when an element in this collection is created by a process.

If an attribute has the trait `autovalue`, then it is automatically assigned a value when an element of this collection is created. The `valueExpression` defines what the value will be. In many cases the function `sequence('some_sequence_name')` will be used. This function will result in a guaranteed unique sequence number.

The trait `displayed` can only be used on basic attributes and on calculated attributes, and affects the way an element is displayed in the front-end. If an element is to be displayed in the front-end without explicit reference to an attribute, then the `displayed` attribute is used. A maximum of two attributes of an entity type can have this trait: if there are two such attributes, then one of them must be of type `tuple` and the other of another basic type. If there are two attributes with the `displayed` trait, then GEARS may choose which one to use depending on circumstances.

The `multiline` trait can only be used with attributes of type `text`. Newline characters may be used in a text attribute only if the `multiline` trait is present. This trait also influences how the attribute is displayed in the front-end (typically, a multi-line scrollable text area).

The `action` trait can only be used with attributes of type `boolean`. When an attribute with the action trait is part of a user task (i.e., a request to the user to provide input), then the user must provide the value `true` for this attribute, otherwise the task does not complete. This trait can for instance be used to mark actions that are required to be performed outside the system scope, and would force the user to mark this action as completed (by explicitly assigning the value true) before he can finish his task.

The trait `default` is followed by a value expression, which specifies the default value assigned to the attribute when a newly created element of this collection is created. The default is used only if no value was explicitly assigned in the process; otherwise the assigned value is used. The value expression *must* result in a value of the same type as the attribute.

The `labeled` trait is followed by a label text: this indicates that when the attribute is shown in a front-end, it has to be labelled with the label text rather than the default label for the attribute (see § 6.2.2 for default labels).

The `group` trait is followed by a group name, and impacts the way attributes are shown in a front-end. Attributes with the same group name are logically (e.g. visually) grouped together in the front-end.

The `opposite of` trait can only be used for relational attributes. Relations between entities are usually thought of as two-way streets: if person A is an inventor of patent B, it follows that patent B is invented by A. In SN3, both sides of that relation need to be made explicit on the two entity types PERSON and PATENT, and these two relational attributes are linked to each other using the `opposite of` trait. See example 1 below.

Two relational attributes of an entity cannot have the same opposite relation. Note though that “the same opposite relation” here means that both the opposite relation name, *and* the opposite entity type are the same. Two relational attributes can use the same opposite-of name, as long as they point to different entities. See example 2 below.

The `description` trait is followed by a description literal, which is a free-format text explaining what the attribute signifies: this serves as additional documentation. The description may contain HTML markup.

6.4.3 Examples

```
//Example 1
PERSON in PERSONS =
   first_name   : text
   last_name    : text
   ...
   INVENTIONS   : multiple PATENT optional opposite of INVENTED_BY

PERSON in PERSONS =
   number       : text
   ...
   INVENTED_BY : multiple PERSON required opposite of INVENTIONS
```

- *The attribute trait `opposite of` here makes it clear that the two relations INVENTIONS and INVENTED_BY are in fact the same relation, seen from different sides.*

```
//Example 2
TOWN in TOWNS =
   name         : text
   ...
   STREETS      : multiple PUBLIC_ROAD opposite of IN_TOWN

VILLAGE in VILLAGES =
   name         : text
   ...
   STREETS      : multiple PUBLIC_ROAD opposite of IN_VILLAGE

PUBLIC_ROAD in PUBLIC_ROADS =
   name         : text
   ...
   IN_TOWN      : multiple TOWN opposite of STREETS
   IN_VILLAGE   : multiple VILLAGE opposite of STREETS
```

- *In the entity type PUBLIC_ROAD there are two relational attributes that are opposites of relations with the same name, STREETS. In this case that is not a problem because the opposite relations are on different entity types, so there's no confusion.*

6.5 Calculated attributes

Calculated attributes are a special class of attributes whose values are calculated by the system on the fly when needed. These attributes exist only in the logical model: no corresponding columns are generated in the technical data model, and the values, once used, are not stored with the entity.

6.5.1 Syntax

```
calculatedAttributeDefinition = "=" valueExpression ;
```

6.5.2 Semantics and usage notes

Calculated attributes can be used with both basic and relational attribute types.

The value expression *must* result in a value of the same type as the attribute.

In the value expression, other attributes of the same element can be referred to by using only the attribute name.

It is possible to refer to other entities in the value expression of a calculated attribute. However, if a name is ambiguous between a relational attribute and a separate entity, then it will be interpreted as a relational attribute.

Calculated attributes cannot coexist with all attribute traits. The following lists show which traits are and aren't allowed for calculated attributes.

- Not allowed: `unique`, `optional`, `required`, `autovalue`, `action`, `default`, `opposite of`.
- Allowed: `displayed`, `multiline`, `labeled`, `group`, `description`.

6.5.3 Examples

(TBD)

6.6 Entity constraints

Entity constraints are used to place specific constraints on the entity or its attributes, which are enforced when the entity is saved.

6.6.1 Syntax

```
entityConstraintsClause = "with:" entityConstraints ;
entityConstraints = entityConstraint { sequenceSeparator entityConstraint } ;
entityConstraint = valueExpression "otherwise" exceptionMessage ;

#    sequenceSeparator = // one or more newlines with same indent level
```

```
#     exceptionMessage = stringLiteral ;
```

6.6.2 Semantics and usage notes

The value expression in a constraint should state a condition on one or more of the attributes of the entity, or on the entity as a whole. The exception message specifies what message to return if the constraint is violated.

The value expression *must* result in a Boolean value.

The constraint is evaluated whenever a newly created element is finalized, and whenever an existing element is saved (committed).

In the value expression, any attributes of the element can be referred to by using only the attribute name.

It is possible to refer to other entities in the value expression of an entity constraint. However, if a name is ambiguous between a relational attribute and a separate entity, then it will be interpreted as a relational attribute.

In the exception message placeholders can be used. In the value expressions in these placeholders, attributes of the element can be referred to by using only the attribute name. The exception message is shown to the user.

6.6.3 Examples

(TBD)

6.7 Technical data model and element commits

When generating an information system, a technical data model is created based on the entity definitions. Although this is strictly not within the purview of a language reference document, in this section we provide some information about how the technical data model is generated and how data are created, updated and deleted in the database. This is based on the assumption that the underlying data layer uses a relational database.

The technical data model is primarily based on the entity definitions. For each entity definition, a table is created in the database; the entity collection name (i.e., the plural form) is used as the name of the table. By convention, table names are always written in uppercase. Each element that is created becomes a new row in the table.

Also, for each attribute of the entity with a basic type, a column is created in the database table. The attribute name is used as the name of the column. By convention, column names are always written in uppercase.

The attribute traits `unique` and `required`/`optional` are directly translated to constraints on the database. Other traits are enforced through various mechanisms in the runtime.

Relational attributes are realised through foreign key relations. The following rules apply.

- If the relation is one to one, and one side of the relation is mandatory, then a column for the foreign key is created on the mandatory side of the relation. This column has the same name as the relational attribute it represents, appended with '_ID'. The other table (the non-mandatory side) will have no column corresponding to this relation.
  If both or neither of the sides of the relation are mandatory, then the column for the foreign key is placed on the table whose name is alphabetically lower (closer to A).
- If the relation is one to many, then a column for the foreign key is created on the "many" side of the relation. This column has the same name as the relational attribute it represents, appended with '_ID'. The other table (the "one" side) will have no column corresponding to this relation.
- If the relation is many to many, a special table is created (a so-called link table) which has attributes for both sides of the relation, both named for the attribute they are based on appended with '_ID'. Each record in this table represents one distinct relation between two elements. This table is named by combining the names of both entities, appended with '_LINK'. Neither of the entity tables will have a column corresponding to this relation.

In addition to the columns derived from attributes, each table gets a few additional columns for various forms of housekeeping (including but not limited to temporal housekeeping). These are the following columns.

- `primary_key`: unique ID for the row. This is immutable (set once and never changed) and unique.
- `ratified_at`: timestamp of the moment that the row was ratified, i.e., marked as final by the process in which it was created. This is set once, at some point during the creating process, and never updated afterwards.
- `created_at`: timestamp of the moment that the row was created. This is set once, at creation, and never updated afterwards.
- `created_by`: username of the user who created the data. This is set once, at creation, and never updated afterwards.
- `last_updated_at`: timestamp of the moment that the row was last updated. This is updated every time the row is updated (not on initial create).
- `last_updated_by`: username of the user who was responsible for the last update. This is updated every time the row is updated (not on initial create).
- `deleted_at`: timestamp of the moment that the row was deleted or archived. This is set once, when the row is deleted/archived, and never updated afterwards.
- `deleted_by`: username of the user who was responsible for deleting the row. This is set once, when the row is deleted, and never updated afterwards.

In SMART Requirements specifications, these attributes can be accessed (read). However, they cannot be updated. Also, the above names are reserved: it is not allowed to define attributes on an entity type using these same names.

In a SMART Requirements process, new elements can be created and existing elements can be updated, or deleted (logically or physically).

- When a new element is created in a process, attributes of the element may be supplied from various sources at various times in the process. This information may be committed to the DB in several steps. A determination is made which is the last step (when all specified information has been supplied): this is also the moment when all explicit and implicit entity constraints are checked (including the unique and required traits).
- When an existing element is updated in a process, then at every commit all explicit and implicit entity constraints must be met.
- When an existing element is deleted (logically or physically), the referential integrity of the DB is checked to make sure there are no dangling references.

6.8 Standard entities

TBD

While the domain model for an application always needs to be defined as part of a SMART Requirements specification, there are some standard entities that can be used “out of the box” without needing to be defined. These entities have specific roles in the SMART specs, as well as in the generated application.

There are roughly two kinds of standard entities: access entities (USERS, ROLES), and action entities (MAILS, MESSAGES, REPORTS, and DOCUMENTS). Access entities are used to control access to the GEARS runtime and the generated application. Action entities are linked to specific services in the runtime, which take action when they detect that a new entity has been created and finalized: for instance, when an entity of type MAIL is created in a SMART Requirements process, the runtime sends an e-mail.

6.8.1 USERS

Users are log-in identities, which are used to control access to both the default admin front-end and the generated front-end of information systems based on SN3 specifications. In combination with the ROLES entity, this also determines which actors have access to which tasks (see also § 2.8).

6.8.1.1 Entity definition

```
USER in USERS =
   username        : text unique required
   surname         : text
   given_name      : text
   email_address   : text
   mobile_number   : text
   ROLES           : multiple ROLE opposite of USERS
```

6.8.1.2 Semantics and usage notes

The username is used to log in to the front-end. Note that the password is not stored in the domain model, this is for the authentication & authorization service to handle.

The email_address can be used to implement forgot-my-password functionality; the mobile_number is used for two-factor authentication. Both of these functionalities are handled by the authentication & authorization service.

Managing (creating, updating, deleting) users is standard functionality in the front-end; users can also be created, updated and deleted through a SMART Requirements process. The system always has one default user with username sysadmin, and role sysadmin. This user cannot be deleted, and the role cannot be removed.

6.8.2 ROLES

This entity holds the system roles, which are used in processes to indicate when actors are allowed to take up tasks. Simply put, if a system user has the role `manager`, then she is authorized to take up a task labeled as **`input from`** MANAGER in a process. For a more precise description, see § 2.8.

6.8.2.1 Entity definition

```
ROLE in ROLES =
   name            : text
   USERS           : multiple USER opposite of ROLES
```

6.8.2.2 Semantics and usage notes

Managing (creating, updating, deleting) roles is standard functionality in the front-end; roles can also be created, updated and deleted through a SMART Requirements process. The system always has two default roles, sysadmin and useradmin. The sysadmin role has the authorization to perform system administration tasks; also, the sysadmin role has automatic authorization to do anything that any other role can, including tasks in processes that are assigned to a specific rolename. The useradmin role has authorization to manage users and roles. These two roles cannot be deleted.

6.8.3 MAILS

This entity is used to send mails. It is linked to the mail service: creating an entity of this type automatically results in an e-mail being sent based on the attributes of the entity.

6.8.3.1 Entity definition

```
MAIL in MAILS =
   from_address    : text required
   to_address      : text required
   subject         : text
   body            : text
   attachments     : binary
```

6.8.3.2 Semantics and usage notes

When a newly created entity of type MAIL is finalized, the mail service of the runtime sends out an e-mail based on the entity's attributes.

Note that updating an existing entity of type MAIL has no effect beyond updating the attributes. It is allowed to update an existing entity, but this will not lead to a new e-mail being sent.

6.8.4 MESSAGES

This entity is used to send messages to users of the runtime and/or the generated application. It is linked to the messaging service: creating an entity of this type automatically results in message being created based on the attributes of the entity, which will be shown to the target user when he logs in (or immediately, if he's already logged in).

6.8.4.1 Entity definition

```
MESSAGE in MESSAGES =
   SENDER          : USER
   RECIPIENT       : USER
   title           : text
   body            : text
```

6.8.4.2 Semantics and usage notes

When a newly created entity of type MESSAGE is finalized, the messaging service of the runtime creates a message based on the entity's attributes.

Note that updating an existing entity of type MESSAGE has no effect beyond updating the attributes. It is allowed to update an existing entity, but this will not lead to a new message being sent.

6.8.5 REPORTS

This entity is used to present structured information in the standard front-end of GEARS. Unlike DOCUMENTS REPORTS do not make use of templates: the information is placed in a tuple, and presented in a standard way.

6.8.5.1 Entity definition

```
REPORT in REPORTS =
   title           : text
   body            : tuple
   rewritable      : boolean
```

6.8.5.2 Semantics and usage notes

When a newly created entity of type REPORT is finalized, the reporting service of the runtime creates a report based on the entity's attributes.

Note that updating an existing entity of type REPORT has no effect beyond updating the attributes. It is allowed to update an existing entity, but this will not lead to a new report being created.

The boolean attribute `rewritable`, when set to `true`, will have the effect that any new report that is created with the same name as an existing report will overwrite the existing report. If `rewritable` is

`false`, then existing reports are not overwritten: new reports are placed alongside existing ones, with an added serial number to make the distinction.

6.8.6 DOCUMENTS

This entity is used to create documents of various types, for instance Word documents or PDF documents. The general mode of operation is that first a template of the document must be created, with named fields for variable information. In the SMART Requirements process, the content of the document is placed in a tuple; when an entity of type DOCUMENT is created, the runtime uses the contents of the tuple to fill the template and thus creates a new document, where the keys of the tuple are matched to the names of the fields in the template.

6.8.6.1 Entity definition

```
DOCUMENT in DOCUMENTS =
   name            : text
   type            : text
   template        : text
   parameters      : tuple
```

6.8.6.2 Semantics and usage notes

When a newly created entity of type DOCUMENT is finalized, the document service of the runtime creates a document based on the entity's attributes and using the template named in the document.

Note that updating an existing entity of type DOCUMENT has no effect beyond updating the attributes. It is allowed to update an existing entity, but this will not lead to a new document being created.

The `type` attribute of a DOCUMENT determines what type of document is created. Allowed document types at the time of writing are docx and pdf.

Note that the template type does not necessarily have to match the document type: for instance, a PDF document can be created from a .docx template.

7 Generic constructs

This chapter contains syntax rules and short descriptions of generic and mostly trivial constructs. For the most part, when they are used in other parts of the syntax, they are directly (and redundantly, and sometimes informally) included in those sections. Here they are all collected in one place, and formally defined.

7.1 Names and keys

Names are used to refer to various basic constructs in the syntax, such as elements, attributes, and actors.

7.1.1 Syntax

```
actorName = name ;          (* should be UPPERCASE *)
parameterName = name ;      (* UPPER or lowercase *)
elementName = name ;        (* should be UPPERCASE *)
collectionName = name ;     (* should be UPPERCASE *)
entityName = name ;         (* should be UPPERCASE *)
functionName = name ;       (* should be lowercase or camelCase *)
attributeName = name | simpleStringLiteral ; (* UPPER or lowercase *)

processKey = name { "." name } ;

name = [A-Za-z][A-Za-z0-9_]*;
```

7.1.2 Semantics and usage notes

There are various rules concerning naming of elements, actors, attributes and such; however these are not formalised in the syntax of SN3. One general rule however is that names may not contain spaces nor hyphens, and must start with a letter.

7.1.3 Examples

(TBD)

7.2 Special literals

Literals (generally speaking) are literal values of a given type, and are described as a type of expression (see § 5.3). One specific type of literal is the string literal, which is an expression that signifies a literal value of type text.

Some restricted kinds of string literals are used for specific purposes in SN3; these are described in this section.

7.2.1 Syntax

```
descriptionLiteral = '[.,\n,\r,\f,\u2028,\u2029,\u0085,\u2B7F]*'
                     (* Any sequence of zero or more Unicode characters,
                        including newlines and the like,
                        enclosed in single quotes. *)
simpleStringLiteral = '[.]+' ;
                      (* Any sequence of at least 1 Unicode character,
                         excepting newlines and the like,
                         enclosed in single quotes. *)
processNameLiteral = simpleStringLiteral;
roleNameLiteral = '[a-z][a-z0-9_]*'
```

7.2.2 Semantics and usage notes

The following table succinctly describes the differences between the various types of literals.

	String literals	Description lit's	Simple string lit's	Name literals
Character set	Unicode	Unicode	Unicode	Alphabet
Newlines	✅	✅	❌	❌
Spaces	✅	✅	✅	❌
Placeholders	✅	❌	❌	❌
Special chars	' { } \	' \	' \	❌

A red cross in the row Newlines means no line termination characters, such as newline, linefeed, carriage return, or anything like that is allowed in this type of string literal.

If placeholders are allowed, the literal may contain a parameter-like construction with a value expression between curly brackets (e.g. {name_of_client}), which is replaced with its value when the string is actually used.

Special characters in any type of literal *must* either be escaped, or are interpreted as having a specific meaning. Escaping of special characters is done using a backslash. This means that e.g. `'\'` (a literal consisting of a single backslash) and `'\a'` are not valid literals. Also, `'{'` is valid as a description literal, but not as a string literal.

The various types of literals are used, among others, for the following purposes.

- String literals: Any process-specific literal expressions of type text; body texts of e-mails; exception messages.
- Description literals: Descriptions of processes, definitions, entities and attributes. In these cases the literal will actually contain text with various formatting tags, spec. HTML.

- Simple string literals: Labels, attribute groups, keys in key-value pairs (in tuples); process names; also used in enumeration types.
- Name literals: Only used for role names in actor mapping clauses (see § 2.2).

7.2.3 Examples

(TBD)

7.3 Characters and strings

(TBD - not sure we use these?)

7.3.1 Syntax

```
      digit = [0-9] ;
      lower_case_letter = [a-z] ;
      upper_case_letter = [A-Z] ;
      letter = lower_case_letter | upper_case_letter ;
      whitespace = " " | newline ;
(* These ones not used? Just directly regexes *)

      nonzeroInteger = ( [1-9] ) { digit } ;
      sentenceNamePart = [A-Za-z0-9 -_/]+ ; (* characters allowed in a sentence name. *)
      sentenceNamePart = '[.,\n,\r,\f,\u2028,\u2029,\u0085,\u2B7F]+'

      exceptionMessage = stringLiteral ;
(*
Note: an exceptionMessage must be localisable.
*)
```

7.3.2 Semantics and usage notes

(TBD)

7.3.3 Examples

(TBD)

7.4 Separators

Separators are syntactic constructs used to indicate scoping and attachment; see § 2.9 for details.

7.4.1 Syntax

```
      definitionSeparator = newlines ;
      bodySeparator = newlines existingIndent additionalIndent | additionalIndent ;
      sequenceSeparator = newlines existingIndent ;
```

```
newlines = \n+ ; (* One or more newlines *)
existingIndent = (\t*) ; (* Same indent level (same number of spaces)
                            as preceding line *)
additionalIndent = (\t*) ; (* Additional indent (at least one space) *)
```

7.4.2 Semantics and usage notes

(TBD)

7.4.3 Examples

(TBD)

Appendices

Appendix A Keywords

TBD

Appendix B Functions

See SN3 functions.

Appendix C Extended ASCII and default sorting

The next list shows the sorting order based on the default collation, which on most systems matches the US English sorting order. Note that if your system uses a different default collation, it will result in a different sorting order.

(space), !, ", #, %, &, ', (,), *, +, ,, -, ., /, :, ;, <, =, >, ?, @, [, \,], ^, _, `, {, |, }, ~, ¡, ¦, ©, «, ¬, ®, », ¿, ×, ―, ¦, ┌, ┐, └, ┘, ├, ┤, ┬, ┴, ┼, ═, ╟, ╔, ╗, ╚, ╝, ╠, ╣, ╦, ╩, ╬, ▀, ▄, ▌, ▒, ▒, ▓, ¤, ¢, $, £, ¥, 0, 1, ½, ¼, 2, 3, 4, 5, 6, 7, 8, 9, a, A, ª, á, Á, à, À, â, Â, å, Å, ä, Ä, ã, Ã, æ, Æ, b, B, c, C, ç, Ç, d, D, ð, Ð, e, E, E, é, é, É, è, È, ê, Ê, ë, Ë, E, é, f, F, ƒ, g, G, h, H, i, I, í, Í, ì, Ì, î, Î, ï, Ï, ı, j, J, k, K, l, L, m, M, n, N, ñ, Ñ, o, O, º, ó, Ó, ò, Ò, ô, Ô, ö, Ö, õ, Õ, ø, Ø, p, P, q, Q, r, R, s, S, ß, t, T, u, U, ú, Ú, ù, Ù, û, Û, ü, Ü, v, V, w, W, x, X, y, Y, ý, Ý, ÿ, z, Z, þ, Þ, µ