UNIVERSITE DE TECHNOLOGIE DE COMPIÈGNE
Département de Génie Informatique

MOSS 7 - Syntax Summary

Jean-Paul Barthès

BP 349 COMPIÈGNE
Tel +33 3 44 23 44 66
Fax +33 3 44 23 44 77

Email: barthes@utc.fr

N220
July 2008

Warning

This document is intended to help the user with the MOSS 6 syntax.

The current research version of MOSS 7 runs in a MacIntosh Common Lisp environment (MCL 5.2 for OSX). It has been ported to Allegro Common Lisp (ACL 6.1 and 8.1 running under Windows XP).

Keywords

Object representation, object-oriented programming environment.

Revisions

Version	**Date**	**Author**	**Remarks**
0.1	Jul 06	Barthès	Draft
0.2	May 07	Barthès	Upgrade
0.3	May 08	Barthès	Upgrade
1.0	Jul 08	Barthès	upgrade to v7

MOSS documents

Related documents

- UTC/GI/DI/N196L - PDM4
- UTC/GI/DI/N206L - MOSS 7 : User Interface (Macintosh)
- UTC/GI/DI/N218L - MOSS 7 : User Interface (Windows)
- UTC/GI/DI/N219L - MOSS 7 : Primer
- UTC/GI/DI/N220L - MOSS 7 : Syntax
- UTC/GI/DI/N221L - MOSS 7 : Advanced Programming
- UTC/GI/DI/N222L - MOSS 7 : Query System
- UTC/GI/DI/N223L - MOSS 7 : Kernel Methods
- UTC/GI/DI/N224L - MOSS 7 : Low Level Functions
- UTC/GI/DI/N225L - MOSS 7 : Dialogs
- UTC/GI/DI/N228L - MOSS 7 : Paths to Queries

Readers unfamiliar with MOSS should read first N196 and N219 (Primer), then N220 (Syntax), N218 (User Interface). N223 (Kernel) gives a list of available methods of general use when programming. N222 (Query) presents the query system and gives its syntax. For advanced programming N224 (Low level Functions) describes some useful MOSS functions. N209 (Dialogs) describes the natural language dialog mechanism.

Contents

1 Introduction **6**
1.1 For Beginners . . . 6
1.2 Global Information . . . 6
1.3 Names, Multilingual Names, References, Internal Names, Identifiers . . . 6
1.3.1 Name . . . 6
1.3.2 Strings . . . 6
1.3.3 Multilingual name . . . 7
1.3.4 References . . . 7
1.3.5 Synonyms . . . 7
1.3.6 Internal Names . . . 7
1.3.7 Variables . . . 8
1.3.8 Identifiers . . . 8
1.3.9 Macros Options . . . 8
1.4 Namespaces . . . 8
1.5 Contexts or Versions . . . 9

2 Creating an Attribute **10**
2.1 Syntax . . . 10
2.2 Examples: . . . 11
2.2.1 Simple Example . . . 11
2.2.2 Variations . . . 12

3 Creating a Concept **14**
3.1 Syntax . . . 14
3.2 Examples . . . 14
3.2.1 Simple Example . . . 15
3.2.2 Variations . . . 15
3.3 Warning . . . 16

4 Creating a Relation **17**
4.1 Syntax . . . 17
4.2 Examples . . . 18
4.2.1 Simple Example . . . 18
4.2.2 Variations . . . 19

5 Creating an Individual **20**
5.1 Syntax . . . 20
5.2 Examples . . . 20
5.2.1 Simple Example . . . 20
5.2.2 Variations . . . 21

6 Creating an Orphan **22**
6.1 Syntax . . . 22
6.2 Example . . . 22

7 Creating an Instance Method **23**
7.1 Syntax . . . 23
7.2 Example . . . 23
7.3 Method Special Mechanisms . . . 23

8 Creating an Own Method **24**
8.1 Syntax 24
8.2 Example 24
8.3 Variations 25

9 Creating a Universal Method **26**
9.1 Syntax 26
9.2 Example 26

10 Multilingual Names **27**
10.1 Default Language when Inputting Data 27
10.2 Using a Multiple Languages when Inputting Data 27
10.3 Specifying a Language for Output 28

1 Introduction

1.1 For Beginners

This manual describes how to use the MOSS syntax, e.g. for designing ontologies. Beginners should also refer to
UTC/GI/DI/N219 - MOSS 7 : A Primer

1.2 Global Information

User space is `:common-lisp-user` in MCL (Macintosh) and `:common-graphics-user` in ACL (Windows). Creating objects is done by means of LISP macros. Errors during execution can be caught by wrapping the macro execution with a

```
(catch :error <macro execution>)
```

In case of error the result of the `catch` is a string that can be printed.

1.3 Names, Multilingual Names, References, Internal Names, Identifiers

MOSS allows different types of names, strings, references.

1.3.1 Name

Names are kept for backward compatibility. However, we strongly encourage to use string references, rather than symbols.

A name corresponds to a Lisp symbol, e.g. `PERSON, AGE`. When used in the defining macros it does not need to be quoted, e.g.

```
(defconcept PERSON (:att AGE))
```

Case is not important, thus the following line has the same effect as the previous one:

```
(defCONCEPT Person (att:  aGE))
```

After execution, the name of the concept will be transformed into the following multilingual name: `(:EN "PERSON")`, that of the attribute into `(:EN "AGE")`[1]. See Paragraph 1.3.3 for more details on multilingual names.
A name will print in capital letters in the various outputs.

1.3.2 Strings

If the concept or property contains several words, e.g. "Government Agency" then we cannot use a symbol. In that case we'll use a string, e.g.

```
(defconcept "Government Agency" (:att Acronym))
```

After execution the name of the class will be transformed into the following multilingual name: (:EN "Government Agency"), that of the attribute into (:EN "ACRONYM"). Note that since we used a string for the concept name, the case will be kept in the various printouts. This was not the case for the attribute.

We could have written:

[1]:EN denoting English is the default language, It is the value of the global `moss::*language*` variable

```
(defconcept "Government Agency" (:att "Acronym"))
```

which keeps the case of Acronym. Thus, any name expressed as a symbol can also be expressed as a string, which is the preferred way of MOSS.

1.3.3 Multilingual name

A multilingual name is used for defining concepts using different languages, e.g.

```
? (defconcept (:name :en "Person" :fr "Personne")
    (:att (:name :en "Age" :fr "Âge")))
```

Here we used English and French as specified by the tags (`:en` and `:fr`). Note that `:en` is used as a default by the system. Note also that French special letters may require using a special encoding like UNICODE UTF-8 (recommended).

Note finally that the `:name` tag in the list is not required:

```
? (defconcept (:en "Person" :fr "Personne")
    (:att (:en "Age" :fr "Âge")))
```

1.3.4 References

A reference stands for a name, string or multilingual name. References can be used in many places. When a reference is required, one may use any form for it, e.g.

```
? (defconcept (:en "Person" :fr "Personne")
    (:att "Age")))
```

1.3.5 Synonyms

It is possible to declare any number of synonyms in a string by using a semi column, e.g.

```
?  (defattribute "name ; given name" (:class "student")))
; Warning: reusing previously defined generic attribute HAS-NAME, ignoring eventual options.
; While executing: MOSS::%MAKE-GENERIC-TP
$T-STUDENT-NAME

? has-name
((MOSS::$TYPE (0 MOSS::$EP)) (MOSS::$ID (0 HAS-NAME))
 (MOSS::$PNAM.OF (0 $T-NAME $T-STUDENT-NAME)) (MOSS::$EPLS.OF (0 MOSS::$SYS.1)))
? has-given-name
((MOSS::$TYPE (0 MOSS::$EP)) (MOSS::$ID (0 HAS-GIVEN-NAME))
 (MOSS::$PNAM.OF (0 $T-STUDENT-NAME)) (MOSS::$EPLS.OF (0 MOSS::$SYS.1)))
```

Note that we have created 2 internal names `HAS-NAME` and `HAS-GIVEN-NAME`.

1.3.6 Internal Names

MOSS builds internal names for the different objects, creating new symbols.

Concept names repeat the external name as a symbol. Thus, `PERSON` will be the internal name of the PERSON concept, however it was defined. If a string is given containing several words the symbol is created by using hyphenation, e.g. `"Government Agency"` produces `GOVERNMENT-AGENCY`.

Property names are built by prefixing the symbol name with `"HAS-"`. E.g., `"Age"` will produce `HAS-AGE`. Multiple word names are hyphenated, e.g. `HAS-FIRST-NAME`. Inverse property internal names are built using prefix and suffix, e.g. `IS-BROTHER-OF`. Note that prefixes and suffixes are defined using English, which can lead to strange names when using a different language, e.g. `HAS-NOM` or `IS-NOM-OF`! However, internal names are not supposed to be seen by the user and their role is to provide unique identifiers. Thus, mixing languages at this level is not a problem.

1.3.7 Variables

Variables names can be used in the `:var` option for storing identifiers temporarily. They should start with an underscore, e.g.

```
? (defindividual
 Person ("name" "Dupond" "Smith")(:var _ds))
$E-PERSON.2
```

1.3.8 Identifiers

Identifiers are given by the MOSS system automatically and may change depending on the order in which definition macros are executed. Thus, one should not rely on them in application programs. Examples of identifiers:

```
$E-PERSON, $E-STUDENT, $T-PERSON-NAME, $S-STUDENT-BROTHER, $E-PERSON.2,
$S-BROTHER.OF, etc.
```

1.3.9 Macros Options

Macro options are specified as lists starting with a keyword, e.g.

```
(:max 3), (:entry), (:one-of "young" "old" "ancient"), etc.
```

1.4 Namespaces

Namespaces use the concept of Lisp package. All MOSS system objects are defined in the "MOSS" package. By default applications are run in the "COMMON-LISP-USER" package on Macs (MCL) and "COMMON-GRAPHICS-USER" package on PCs running XP (ACL). However, applications can use their own package(name space). The OMAS multi-agent platform uses a specific package for each agent. Thus each agent can develop its own ontology without interfering with the other agents. For simplified programming one can inherit symbols from the "MOSS" package by inserting a command:

```
(use-package :moss)
```

at the beginning of the application code.

In a programming environment the current namespace is given by the global variable `*package*`.

Note that in a Lisp environment keywords (e.g., `:ref`) and strings do not depend on packages, but symbols do. When defining a new ontology the desired package should be specified by inserting a command:

```
(in-package :my-package)
```

at the beginning of the file, or better a `defontology` macro command. See the documentation about SOL for a detailed treatment of ontologies.

1.5 Contexts or Versions

`All MOSS objects` can be versioned. The mechanism is implemented through a configuration graph that records the links between different versions called contexts in the MOSS jargon.

Versioning can be sequential or parallel. Sequential versioning allows accounting for changes in time easily, parallel versioning can be used for trying various hypothesis after an initial situation.

Versioning introduces a new level of complexity in the programming of functions. In a given programming environment the current version is recorded by a number associated to the global variable `*context*`

The initial context, root of the configuration tree is context 0. If no versioning is required then all objects will belong to context 0.

The following sections give examples of how to use the defining macros to create MOSS objects. Examples are taken from the MCL 5.2 environment on the Mac OSX 10.4 system in the "COMMON-LISP-USER" package (also known as `:cl-user`).

The syntax and corresponding examples will be given using English as a default language. More complex multilingual examples will be given in Section **??**.

2 Creating an Attribute

The MOSS formalism is based on properties. Properties are objects and exist by themselves as concepts, independently from any concept or model they can qualify. Thus, it is possible to create properties directly.

There are two kinds of properties: those that qualify an object with a value (e.g. `AGE` of a person), those that link objects to other objects (e.g. `BROTHER` of a person). The former are called *attributes*, and the latter *relations*. OWL calls them respectively *datatype properties* and *object properties*.

2.1 Syntax

defattribute *reference &rest option-list* **macro**

reference: reference of the property (symbol, string, multilingual name)

option-list:

`(:class <class-name>)` to attach the attribute to a class (incompatible with `:class-id` option)

`(:default <value>)` default value

`(:doc <doc-string>)` documentation

`(:entry <function-descriptor>)`
`(:index <function-descriptor>)`

specifies entry-point, meaning that values attached to the attribute will produce an index to the object in which the attribute appears. If no argument, uses `make-entry-symbols` to produce the index, otherwise uses specified function where
<function-descriptor> ::= <arg-list><doc><body>
e.g.
(:entry (value-list) "Entry for Company name" (list (intern (make-name value-list))))
The entry function must return a list of symbols.

`(:min <number>)` minimal cardinality

`(:max <number>)` maximal cardinality

`(:unique)` minimal and maximal cardinality are each 1

`(:one-of <value>+)`

restrictions on the values, that should be taken from a list. Selection is done according to the `:exist` and `:forall` options. Default is `:exists` Only applies when the class is specified (`:class` or `:class-id` options).

`(:forall)`

all values should be taken from the one-of list or be of the specified value-type Only applies when the class is specified (`:class` or `:class-id` option) and `:one-of` option is present.

`(:exists)`

there should be at least one value taken from the one-of list or of the specified value-type Only applies when the class is specified (`:class` or `:class-id` options) and `:one-of` option is present.

The following options are advanced and require knowledge or the internals of MOSS.

`(:id <id>)` prop id for generic property (if there, id is not synthesized). Reserved for system use.

`(:system)` indicates a system property. Reserved for system use.

`(:class-id <class-id>)` to attach the attribute to a class (incompatible with `:class` option)

`(:language <tag>)` to specify the language. Must be in the `moss::*language-tags*` list.

2.2 Examples:

Defining attributes directly is not a usual approach. In general attributes are defined as a side effect of the definition of a concept. Nevertheless, it is possible to define attributes independently, either as a preliminary step or a posteriori for adding the attribute to an already defined concept. The useful syntax presents many variations that are described in the following subsections.

2.2.1 Simple Example

The simplest possible syntax is the following:

```
Defining the property age (no options)
? (defattribute AGE)
$T-AGE
```

This creates a number of objects:

* a generic property:

  ```
  ? $T-AGE
  ((MOSS::$TYPE (0 MOSS::$EPT)) (MOSS::$ID (0 $T-AGE)) (MOSS::$PNAM (0 (:EN "Age")))
   (MOSS::$ETLS.OF (0 MOSS::$SYS.1)) (MOSS::$INV (0 $T-AGE.OF)))
  ```

  Note that in the internal format the name of the property has been set to `(:EN "Age")` meaning that English has been adopted as the default language.

* an internal name for the property:

  ```
  ? HAS-AGE
  ((MOSS::$TYPE (0 MOSS::$EP)) (MOSS::$ID (0 HAS-AGE)) (MOSS::$PNAM.OF (0 $T-AGE))
   (MOSS::$EPLS.OF (0 MOSS::$SYS.1)))
  ```

* a temporary variable containing the identifier of the property:

  ```
  ? _has-age
  $T-AGE
  ```

* an inverse property:

  ```
  ? $T-AGE.OF
  ((MOSS::$TYPE (0 MOSS::$EIL)) (MOSS::$INAM (0 (:EN "IS-AGE-OF")))
   (MOSS::$INV.OF (0 $T-AGE)) (MOSS::$EILS.OF (0 MOSS::$SYS.1)))
  ```

* a name for the inverse property:

```
? IS-AGE-OF
((MOSS::$TYPE (0 MOSS::$EP)) (MOSS::$ID (0 IS-AGE-OF))
 (MOSS::$INAM.OF (0 $T-AGE.OF)) (MOSS::$EPLS.OF (0 MOSS::$SYS.1)))
```

* an accessor function (defsettable), useful when writing methods (note that the name of the accessor function is the same as the name of the property):

```
? HAS-AGE
used as: (HAS-AGE _jd) ; where _jd is a person
     or: (setf (HAS-AGE _jd) 35)
```

Note however that no variable is created for the inverse property, nor any accessor function.

2.2.2 Variations

* The following definitions are equivalent:

```
? (defattribute "Name")          ; using a string
? (defattribute NAME)            ; using a symbol
? (defattribute (:en "name"))    ; using a multilingual name
```

  Although the definitions are equivalent, it is better to use strings since they keep the case.

* Defining the property name, with cardinality constraints, e.g. a name can have at least one value and a maximum of three values:

```
? (defattribute "Name" (:min 1)(:max 3))
$T-NAME
```

* Using a default on a generic attribute:

```
? (defattribute "Name" (:default "Dupond" "Durand"))
$T-NAME

? $T-NAME
((MOSS::$TYPE (0 MOSS::$EPT)) (MOSS::$ID (0 $T-NAME3))
 (MOSS::$PNAM (0 (:EN "Name"))) (MOSS::$ETLS.OF (0 MOSS::$SYS.1))
 (MOSS::$INV (0 $T-NAME.OF)) (MOSS::$DEFT (0 "Dupond" "Durand")))
```

  The default values (two in this case, are attached directly to the generic property.

* Defining the property name, with default indexing capabilities:

```
? (defattribute "Name" (:entry))
$T-NAME

? $T-NAME
((MOSS::$TYPE (0 MOSS::$EPT)) (MOSS::$ID (0 $T-NAME))
 (MOSS::$PNAM (0 (:EN "Name"))) (MOSS::$ETLS.OF (0 MOSS::$SYS.1))
 (MOSS::$INV (0 $T-NAME.OF)) (MOSS::$OMS (0 $FN.193)))
```

  A specific own method is created ($FN.193) and attached to the attribute. Whenever a name will be created, the method will be invoked to create the corresponding indexes.

* Defining the property name, with a documentation string:

```
? (defattribute "Name" (:doc "family name"))
$T-NAME

? $T-NAME
((MOSS::$TYPE (0 MOSS::$EPT)) (MOSS::$ID (0 $T-NAME))
 (MOSS::$PNAM (0 (:EN "Name"))) (MOSS::$ETLS.OF (0 MOSS::$SYS.1))
 (MOSS::$INV (0 $T-NAME.OF)) (MOSS::$DOCT (0 (:EN "family name"))))
```

Note that the string has been transformed into a string of the default language.

* Defining AGE and assigning the attribute to the class PERSON (must be defined)

```
? (defattribute "Age" (:class "Person"))
$T-PERSON-AGE

? $T-PERSON-AGE
((MOSS::$TYPE (0 MOSS::$EPT)) (MOSS::$ID (0 $T-PERSON-AGE))
 (MOSS::$PNAM (0 (:EN "age"))) (MOSS::$ETLS.OF (0 MOSS::$SYS.1))
 (MOSS::$IS-A (0 $T-AGE)) (MOSS::$INV (0 $T-PERSON-AGE.OF))
 (MOSS::$PT.OF (0 $E-PERSON)))
```

The newly created property is defined as a local attribute of the class PERSON and gets a different identifier ($T-PERSON-AGE). The attribute is related to the generic $T-AGE attribute by an $IS-A property.

* Defining that one of the names of a student should be taken from a list.

```
? (defattribute "name" (:class "student")
                       (:one-of (:en "judith") (:en "jim") (:en " george")))
$T-STUDENT-NAME

? $T-STUDENT-NAME
((MOSS::$TYPE (0 MOSS::$EPT)) (MOSS::$ID (0 $T-STUDENT-NAME))
 (MOSS::$PNAM (0 (:EN "name"))) (MOSS::$ETLS.OF (0 MOSS::$SYS.1))
 (MOSS::$IS-A (0 $T-NAME)) (MOSS::$INV (0 $T-STUDENT-NAME.OF))
 (MOSS::$PT.OF (0 $E-STUDENT))
 (MOSS::$ONEOF (0 (:EN "judith") (:EN "jim") (:EN " george")))
 (MOSS::$SEL (0 :EXISTS)))
```

3 Creating a Concept

A concept is an object describing the structure of an individual object that can be considered an instance of this concept. Thus, defining a concept amounts to defining the structure of its instances. One could think of the approach as similar to a class/instance relation in object-oriented languages. However, MOSS does not require that instances of a class have all the properties defined in the concept, or even lets such individuals have properties not present in the concept. Thus, a concept is the description of a typical individual, allowing exceptions to be present in actual individuals.

When specifying a property (attribute or relation) in a concept, MOSS uses the corresponding generic property, creating it if it does not exist. It then creates a local property associated with the new concept.

3.1 Syntax

defconcept *reference &rest option-list* **macro**

reference: reference of the concept (symbol, string, multilingual name)

option-list:

`(:is-a <class-id>*)` for defining inheritance (multiple inheritance is allowed)

`(:att <attribute-description>)`

defining an attribute for the current class. All attribute options are allowed except for `:class` and `:class-id`.

`(:rel <relation-description>)`

defining an attribute for the current class. All attribute options are allowed except for `:class` and `:class-id`.

`(:one-of <object-ids>+)` specifying the class as a set of objects. Note that the objects must already exist.

`(:doc <doc-string>)` documentation

The following options are advanced and require knowledge or the internals of MOSS.

`(:id <id>)` id for the class object (if not there, id is synthesized) Reserved for system use.

`(:no-counter)` Reserved for system use.

`(:system)` indicates a system class. Reserved for system use.

`(:export <truth value>)` to export the name of the class from the defining package.

`(:language <tag>)` to specify the language. Must be in the `moss::*language-tags*` list.

3.2 Examples

Defining concepts is the normal way of proceeding when defining an ontology. Properties are defined from within the definition of the concept and created if they do not exist.

3.2.1 Simple Example

The simplest example is:

```
(defconcept person)
or
(defconcept "person")
or
(defconcept (:en "person"))
or
(defconcept (:name :en "person"))
```

This defines the concept of person. As such, it does not have any specific properties. However, a number of objects are created:

* the concept object itself:

  ```
  ? $E-PERSON
  ((MOSS::$TYPE (0 MOSS::$ENT)) (MOSS::$ID (0 $E-PERSON))
   (MOSS::$ENAM (0 (:EN "Person"))) (MOSS::$RDX (0 $E-PERSON))
   (MOSS::$ENLS.OF (0 MOSS::$SYS.1)) (MOSS::$CTRS (0 $E-PERSON.CTR)))
  ```

  The concept is an instance of the entity metaconcept (MOSS::$ENT). It has a counter with initial value 1 and a radical ($E-PERSON) that will be used to create identifiers of individual persons.

* an index:

  ```
  ? PERSON
  ((MOSS::$TYPE (0 MOSS::$EP)) (MOSS::$ID (0 PERSON)) (MOSS::$ENAM.OF (0 $E-PERSON))
   (MOSS::$EPLS.OF (0 MOSS::$SYS.1)))
  ```

* a temporary variable whose value is the concept identifier:

  ```
  ? _person
  $E-PERSON
  ```

3.2.2 Variations

* Adding documentation:

  ```
  ? (defconcept "Person"
        (:doc "A PERSON is a human being."))
  $E-PERSON
  ```

* Concept with properties
  A concept has normally some properties (attributes or relations):

  ```
  ? (defconcept "Person"
     (:att "Name" (:entry))
     (:att "First-Name" (:max 3))
     (:rel "mother" "person" (:max 1))
  $E-PERSON
  ```

```
? $E-PERSON
((MOSS::$TYPE (0 MOSS::$ENT)) (MOSS::$ID (0 $E-PERSON))
 (MOSS::$ENAM (0 (:EN "Person" :FR "Personne; Individu")))
 (MOSS::$RDX (0 $E-PERSON)) (MOSS::$ENLS.OF (0 MOSS::$SYS.1))
 (MOSS::$CTRS (0 $E-PERSON.CTR))
 (MOSS::$PT (0 $T-PERSON-NAME $T-PERSON2-FIRST-NAME))
 (MOSS::$PS (0 $S-PERSON-MOTHER)) (MOSS::$SUC.OF (0 $S-PERSON-MOTHER)))
```

This creates a concept with a name (indexed), a first name (max cardinality constraint of 3), and a relation linking a person to the person's mother, another person (max cardinality of 1).

* An example with a subclass and documentation:

```
? (defconcept student2 (:is-a person)
   (:doc " A student is a person engaged in a studying program."))
$E-STUDENT
```

Meaning that a student is a person.

3.3 Warning

Concepts once defined cannot be redefined. Indeed, it is not a good idea to redefine a concept, because if individuals of this concept exist, it could lead to inconsistencies between the newly defined concept and the existing individuals.

4 Creating a Relation

Relations are used for linking objects, in particular the individuals of two concepts. In general relations are created as a consequence of creating concepts. However, they can be created independently. In that case, the concepts must be referenced and must have been defined beforehand for the property to be created. A different design choice could have been to create the missing classes automatically, but the option was dismissed. The adopted approach allows catching spelling mistakes.

4.1 Syntax

defrelation *reference,class-reference,successor-reference &rest option-list* **macro**

reference: reference of the property (symbol, string, multilingual name)

class-reference: reference of the class in the domain

successor-reference: reference of the class in the range

option-list:

`(:default <object-id>+)`

list of object identifiers of objects that can be used as default values for the relation. The objects should be instances of the successor class (range).

`(:doc <doc-string>)` documentation

`(:min <number>)` minimal cardinality

`(:max <number>)` maximal cardinality

`(:unique)` minimal and maximal cardinality are each 1

`(:one-of <value>+)`

restrictions on the values, that should be taken from a list. Selection is done according to the :exist and :forall options. Default is :exist Only applies when the class is specified (`:class` or `:class-id` options).

`(:forall)`

all values should be taken from the one-of list or be of the specified value-type Only applies when the class is specified (`:class` or `:class-id` option) and `:one-of` option is present.

`(:exists)`

there should be at least one value taken from the one-of list or of the specified value-type Only applies when the class is specified (`:class` or `:class-id` options) and `:one-of` option is present.

The following options are advanced and require knowledge or the internals of MOSS.

`(:id <id>)` prop id for generic property (if there, id is not synthesized). Reserved for system use.

`(:system)` indicates a system property. Reserved for system use.

`(:class-id <class-id>)` to attach the attribute to a class (incompatible with `:class` option)

`(:export <truth value>)` to export the name of the property from the defining package.

`(:language <tag>)` to specify the language. Must be in the `moss::*language-tags*` list.

Note that *class-reference* and *successor-reference* can be both replaced by the keyword `:any` when one wants to define a generic property that will apply to any object. However it is an error to have only one be assigned the keyword `:any`.

The options `:is-a, :min, :max, :unique, :var`, and `:doc` apply to generic properties. The options `:one-of, :forall`, and `:exist` do not.

4.2 Examples

4.2.1 Simple Example

The simplest syntax for creating a relation is the following (assuming that PERSON is a valid concept):

```
? (defrelation "brother" "person" "person")
$S-PERSON-BROTHER
```

A number of objects are created:

* An index to the property:

```
HAS-BROTHER
((MOSS::$TYPE (0 MOSS::$EP)) (MOSS::$ID (0 HAS-BROTHER))
 (MOSS::$PNAM.OF (0 $S-BROTHER $S-PERSON-BROTHER)) (MOSS::$EPLS.OF (0 MOSS::$SYS.1)))
```

* The property itself, first generic:

```
$S-BROTHER
((MOSS::$TYPE (0 MOSS::$EPS)) (MOSS::$ID (0 $S-BROTHER))
 (MOSS::$PNAM (0 (:EN "brother"))) (MOSS::$PS.OF (0 MOSS::*ANY*))
 (MOSS::$SUC (0 MOSS::*ANY*)) (MOSS::$ESLS.OF (0 MOSS::$SYS.1))
 (MOSS::$INV (0 $S-BROTHER.OF)) (MOSS::$IS-A.OF (0 $S-PERSON-BROTHER)))
```

* Then, local:

```
$S-PERSON-BROTHER
((MOSS::$TYPE (0 MOSS::$EPS)) (MOSS::$ID (0 $S-PERSON-BROTHER))
 (MOSS::$PNAM (0 (:EN "brother"))) (MOSS::$ESLS.OF (0 MOSS::$SYS.1))
 (MOSS::$IS-A (0 $S-BROTHER)) (MOSS::$INV (0 $S-PERSON-BROTHER.OF))
 (MOSS::$PS.OF (0 $E-PERSON)) (MOSS::$SUC (0 $E-PERSON)))
```

* An inverse property:

```
$S-BROTHER.OF
((MOSS::$TYPE (0 MOSS::$EIL)) (MOSS::$INAM (0 "IS-BROTHER-OF"))
 (MOSS::$INV.OF (0 $S-BROTHER))
 (MOSS::$EILS.OF (0 MOSS::$MOSSSYS)))
```

* An index to the inverse property, generic:

```
$S-BROTHER.OF
((MOSS::$TYPE (0 MOSS::$EIL)) (MOSS::$INAM (0 "IS-BROTHER-OF"))
 (MOSS::$INV.OF (0 $S-BROTHER)) (MOSS::$EILS.OF (0 MOSS::$SYS.1)))
```

* ... and local:

```
$S-PERSON-BROTHER.OF
((MOSS::$TYPE (0 MOSS::$EIL)) (MOSS::$INAM (0 "IS-BROTHER-OF"))
 (MOSS::$INV.OF (0 $S-PERSON-BROTHER)) (MOSS::$EILS.OF (0 MOSS::$SYS.1)))
```

* Internal (temporary variables), containing the property identifiers:

```
_has-BROTHER
$S-BROTHER
_HAS-PERSON-BROTHER
$S-PERSON-BROTHER
```

* Accessor functions (defsettable), useful when writing methods:

```
HAS-BROTHER
used as: (HAS-BROTHER _jd) ; where jd is a person
     or: (setf (HAS-BROTHER _jd) 35)
```

4.2.2 Variations

* The syntax of the defrelation macro is quite flexible.
  It can be positional using references:

```
? (defrelation BROTHER PERSON PERSON)
? (defrelation (:en "brother") (:en "person") (:en "person"))
? (defrelation "brother" "person" "person")
or any mixture of that
? (defrelation BROTHER "person" (:en "person"))
```

  Or it can use keywords (:from specifying the domain, :to the range):

```
? (defrelation BROTHER (:from PERSON) (:to PERSON))
? (defrelation (:en "brother") (:to "person")(:from PERSON))
```

  Note that when using keywords, only the relation name is required to be in the first position.

* We can impose restrictions on the property (meaning that one can have the French nationality and maybe other ones but no more than 3):

```
(defrelation FRENCH-NATIONALITY PERSON COUNTRY (:one-of _france) (:exists) (:max 3))
```

* Generic property (applying to all objects)

```
(defrelation (:en "neighborhood") :any :any)
```

* Relation one-to-all:

```
(defrelation (:en "zzz") person universal-class)
```

5 Creating an Individual

An individual can be created from a set of properties and values (see Orphans). However, if its type is known, one can use the corresponding concept for that.

5.1 Syntax

definidividual *concept-reference &rest option-list* **macro**

concept-reference: reference to a class (must have been defined)

option-list:

`(<attribute-reference> <value>*)`

attribute and values. If the attribute is referenced by a symbol (name), then it must be the internal name of the symbol, e.g. HAS-AGE (do not forget the prefix HAS-).

`(<relation-reference> <id>*)`
`(<relation-reference> (:new <class-ref> <individual description>)+)`

drelations and list of object identifiers or variables containing object identifiers. If the relation is referenced by a symbol (name), then it must be the internal name of the symbol, e.g. HAS-BROTHER (do not forget the prefix HAS-).
The second format allows creating local objects while avoiding to use variables.

`(:doc <doc-string>)`

documentation: string or multilingual name. If a string, then will be transformed into a multilingual name using the current language (default is English).

`(:var <var-name>)` temporary variable name, e.g. _dbb.

5.2 Examples

5.2.1 Simple Example

Defining the person DUPOND-DURAND:

```
(definidividual person (HAS-NAME "Dupond" "Durand")("AGE" 23)(:var _dd)
(:doc "Dupond-Durand is the mayor"))
```

Executing the previous command yields:

* The object:

```
? $E-PERSON.5
((MOSS::$TYPE (0 $E-PERSON)) (MOSS::$ID (0 $E-PERSON.5))
 ($T-NAME (0 "Dupond" "Durand")) ($T-AGE (0 23))
 (MOSS::$DOCT (0 (:EN "Dupond-Durand is the mayor"))))
```

* A temporary variable:

```
? _dd
$E-PERSON.55
```

* Entry-points:

```
? Dupond
((MOSS::$TYPE (0 MOSS::$EP)) (MOSS::$ID (0 DUPOND))
 ($T-PERSON-NAME.OF (0 $E-PERSON.5)) (MOSS::$EPLS.OF (0 MOSS::$SYS.1)))

? Durand
((MOSS::$TYPE (0 MOSS::$EP)) (MOSS::$ID (0 DURAND))
 ($T-PERSON-NAME.OF (0 $E-PERSON.5)) (MOSS::$EPLS.OF (0 MOSS::$SYS.1)))
```

5.2.2 Variations

* An example creating an anonymous address object:

```
? (defindividual person
     (HAS-NAME "Dupond" "Durand")
     ("AGE" 23)
     ("Address"
        (:new "Address" ("street" "10, rue de la Paix")
                        ("city" "Paris")))
     (:var _dd)
     (:doc "Dupond-Durand is the mayor))
```

The specific address object is created as an anonymous object and linked to the individual. It could be considered as a structured value in other formalisms. Note however that the address object could receive a `:var` option, and is a first class object that can be queried.

6 Creating an Orphan

An orphan is a classless individual that is created by assembling properties values and links.

6.1 Syntax

The syntax is essentially the same as for creating individuals without the concept reference.

defobject *&rest option-list* **macro**

option-list:

`(<attribute-reference> <value>*)`

attribute and values. If the attribute is referenced by a symbol (name), then it must be the internal name of the symbol, e.g. HAS-AGE (do not forget the prefix HAS-).

`(<relation-reference> <id>*)`
`(<relation-reference> (:new <class-ref> <individual description>)+)`

drelations and list of object identifiers or variables containing object identifiers. If the relation is referenced by a symbol (name), then it must be the internal name of the symbol, e.g. HAS-BROTHER (do not forget the prefix HAS-).
The second format allows creating local objects while avoiding to use variables.

`(:doc <doc-string>)`

documentation: string or multilingual name. If a string, then will be transformed into a multilingual name using the current language (default is English).

`(:var <var-name>)` temporary variable name, e.g. _dbb.

6.2 Example

Defining a red object:

```
? (defobject ("color" "red")(:doc "a red object")(:var _my-object))
```

Executing the previous command yields:

* The object:

```
? $0-3
((MOSS::$TYPE (0 MOSS::*NONE*)) (MOSS::$ID (0 $0-3)) ($T-COLOR (0 "red"))
 (MOSS::$DOCT (0 (:EN "a red object"))))
```

* A temporary variable:

```
? _my-object
$0-3
```

7 Creating an Instance Method

An instance method is analogous to a regular method in an object-oriented approach. The method is attached to the concept (class) and applies to the individuals (instances) belonging to this concept. Creating methods is similar to creating Lisp functions with some restrictions.

7.1 Syntax

The syntax is essentially the same as for creating individuals without the concept reference.

definstmethod *name selector arg-list &rest body* **macro**

name: name of the method (by convention should start with an = sign)

selector: name of the concept

arg-list: arguments of the method

body: (Lisp) code for the method

7.2 Example

The following code defines an instance of person and a method to print:

```
? (definstmethod =print PERSON (title)
    (format t "~&   ~A. ~A ~A"
            title (car (HAS-FIRST-NAME)) (car (HAS-NAME))))
```

Defining an individual and applying the method yields:

```
? (defindividual person ("name" "Dupont") ("first name" "John" "Jim")(:var _jjd))
$E-PERSON.8

? (send _jjd '=print "Mr")
   Mr. John Dupont
NIL
```

7.3 Method Special Mechanisms

While writing methods one can use the following features:

The special variable `*self*` contains the id of the object that will receive the message.

The special variable `*answer*` contains the answer returned by the last message.

Accessors like `HAS-XXX` can be used as follows:

```
(HAS-XXX obj-id) is equivalent to (send obj-id '=get 'HAS-XXX)
(HAS-XXX) is equivalent to (send *self* '=get 'HAS-XXX)
```

8 Creating an Own Method

An own method is attached to a particular object. Creating own method is similar to creating instance methods with the difference that we must specify the object onto which we attach the method.

8.1 Syntax

The syntax is essentially the same as for creating individuals without the concept reference.

defownmethod *name selector arg-list &rest body* **macro**

name: name of the method (by convention should start with an = sign)

selector: expression that is used to locate the object to which we want to attach the method

arg-list: arguments of the method

body: (Lisp) code for the method

The selector has a more complex syntax, e.g.

We want attach method to a person named ALBERT, and in case of ambiguity we apply the user-defined function agemax to the resulting list of object ids.

```
(ALBERT HAS-NAME PERSON :select agemax))
```

We want to attach the method to the NAME attribute of the concept PERSON (remember that AGE is the name of a generic property as well as the name of local properties attached to different classes. Thus, :class-ref is used to point to the wanted class).

```
(NAME HAS-PROPERTY-NAME ATTRIBUTE :class-ref PERSON)
```

Warning The selector syntax differs from the previous versions of MOSS: the property name must not include the HAS- prefix like previously:

```
(HAS-NAME HAS-PROPERTY-NAME ATTRIBUTE :class-ref PERSON)
```

8.2 Example

The following code defines a method attached to the age concept of a person, e.g. for making sure that input values to the age of a person is a number. It implements a semi-predicate.

```
? (defownmethod =xi ("age" HAS-PROPERTY-NAME ATTRIBUTE :class-ref PERSON) (data)
    (and (numberp data) data)))
$FN.194
```

* An object has been created for the method (yes, methods are first class objects):

```
? $FN.194
((MOSS::$TYPE (0 MOSS::$FN)) (MOSS::$ID (0 $FN.194)) (MOSS::$MNAM (0 =XI))
 (MOSS::$ARG (0 (DATA))) (MOSS::$CODT (0 ((PRINT DATA) (AND (NUMBERP DATA) DATA))))
 (MOSS::$FNLS.OF (0 MOSS::$SYS.1)) (MOSS::$OMS.OF (0 $T-PERSON-AGE))
 (MOSS::$FNAM (0 $T-PERSON-AGE=S=0=XI)))
```

... and attached to the local property:

```
? $T-PERSON-AGE
((MOSS::$TYPE (0 MOSS::$EPT)) (MOSS::$ID (0 $T-PERSON-AGE))
 (MOSS::$PNAM (0 (:EN "age"))) (MOSS::$ETLS.OF (0 MOSS::$SYS.1))
 (MOSS::$IS-A (0 $T-AGE)) (MOSS::$INV (0 $T-PERSON-AGE.OF))
 (MOSS::$PT.OF (0 $E-PERSON)) (MOSS::$OMS (0 $FN.194)))
```

8.3 Variations

* the selector can take many different forms:

```
("age" "property name" "attribute" :class-ref "person")
(AGE "property name" "attribute" :class-ref "person")
```

9 Creating a Universal Method

Creating universal method is easy since such methods are not attached to any object

9.1 Syntax

The syntax is essentially the same as for creating individuals without the concept reference.

defuniversal *name arg-list &rest body* **macro**

name: name of the method (by convention should start with an = sign)

arg-list: arguments of the method

body: (Lisp) code for the method

9.2 Example

The following example is taken from the MOSS Kernel Library. Any object can receive the =print-error message. Note that body starts with a documentation string.

```
? (defuniversalmethod =print-error (stream msg &rest arg-list)
  "Default method for printing errors - uses LISP format primitive directly.
Arguments:
   stream: printing stream
   msg: format string
   arg-list (rest): args to format string"
  (let ((*moss-output* stream))
    (if (stringp msg)
      (mformat "~%;***Error: ~?" msg arg-list)
      (mformat "~%;***Error:~{ ~A~}" msg))
    ))
```

An object has been created for the method:

The method is created and becomes available for any MOSS object until some own or instance method is created that will shadow it.

10 Multilingual Names

Using multilingual names can be quite tricky and must be done carefully. The language specification is handled by the global variable `moss::*language*`.

10.1 Default Language when Inputting Data

By default the `moss::*language*` variable has value `:en`, meaning that the default language is English. Thus, if nothing is done as was the case in the previous sections, all structures are built using English.

It is possible to change the default language to something else, e.g. French by setting the `moss::*language*` variable to `:fr`.

One cannot set the `moss::*language*` variable to anything. The language code (e.g. `:en` or `:fr` must be one of the code present in the global list `moss::*language-tags*` . Of course, it is possible to add new languages to this list.

10.2 Using a Multiple Languages when Inputting Data

In some cases, it is important to build multilingual structures, by accepting several languages simultaneously. In that case the global `moss::*language*` variable must be set to `:all`, and the used languages must be specified in the `moss::*language-tags*` global variable that gives the list of allowable languages (e.g. `(:en :fr :it :pl)`). Of course it is possible to add to the list.

The next paragraphs give some examples.

* attribute definition in English and French:

```
? (defattribute (:en "name" :fr "nom"))
$T-NAME

? $T-NAME
((MOSS::$TYPE (0 MOSS::$EPT)) (MOSS::$ID (0 $T-NAME))
 (MOSS::$PNAM (0 (:EN "name" :FR "nom"))
 (MOSS::$ETLS.OF (0 MOSS::$SYS.1)) (MOSS::$INV (0 $T-NAME.OF)))
```

However, if the generic property was already defined, the new definition does not change it.

```
? (defattribute (:en "name" :pl "nazwa"))
; Warning: reusing previously defined generic attribute HAS-NAME1,
; in package #<Package "COMMON-LISP-USER"> and context 0 ignoring eventual options.
; While executing: MOSS::%MAKE-GENERIC-TP
$T-NAME
```

* concept definition in English and French:

```
? (defconcept (:en "Cook" :fr "Cuisinier; Marmiton"))
;***** === creating class with name: COOK in package #<Package "COMMON-LISP-USER">
$E-COOK
? $E-COOK
((MOSS::$TYPE (0 MOSS::$ENT)) (MOSS::$ID (0 $E-COOK))
 (MOSS::$ENAM (0 (:EN "Cook" :FR "Cuisinier; Marmiton"))) (MOSS::$RDX (0 $E-COOK))
```

```
 (MOSS::$ENLS.OF (0 MOSS::$SYS.1)) (MOSS::$CTRS (0 $E-COOK.CTR)))
? cook
((MOSS::$TYPE (0 MOSS::$EP)) (MOSS::$ID (0 COOK)) (MOSS::$ENAM.OF (0 $E-COOK))
 (MOSS::$EPLS.OF (0 MOSS::$SYS.1)))
? cuisinier
((MOSS::$TYPE (0 MOSS::$EP)) (MOSS::$ID (0 CUISINIER)) (MOSS::$ENAM.OF (0 $E-COOK))
 (MOSS::$EPLS.OF (0 MOSS::$SYS.1)))
? marmiton
((MOSS::$TYPE (0 MOSS::$EP)) (MOSS::$ID (0 MARMITON)) (MOSS::$ENAM.OF (0 $E-COOK))
 (MOSS::$EPLS.OF (0 MOSS::$SYS.1)))
```

If the default language is English and we try to define a concept with a French name, then we get an error:

```
? (setq *language* :en)
:EN
? (defconcept (:fr "essai"))
;***Error (%mln-extract-mln-from-ref) no info for language :EN in
 ref: (:FR "essai")
```

* relation definition in English and French:

```
(defrelation (:en "uncle ; shushu" :fr "oncle; tonton")
             (:en "student ") (:fr "personne"))
```

Note that the domain and range specifications use one of the references.

* method names (e.g., =test, or =summary) cannot use multilingual names.

10.3 Specifying a Language for Output

Output is slightly different.

If the `moss::*language*` variable is set to `:all`, then the system does not know which language to choose. It chooses English by default, and if not possible, a random language or nil.

If the language is specified, it selects the labels in the specified language. If there is no label in this language, it may choose English if present or a random language or nil.