UNIVERSITE DE TECHNOLOGIE DE COMPIÈGNE
Département de Génie Informatique

MOSS 7 - Low Level Functions

Jean-Paul Barthès

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

Email: barthes@utc.fr

N224
July 2008

Warning

This document is a description of the MOSS low level library functions used in system programming. The functions could be useful to the advanced user in lieu of the kernel methods. The latter are defined mostly in terms of such functions. The gain is runtime efficiency, the danger is some lack of consistency checking.

However, using low level functions requires knowing the internals of the MOSS system.

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, kernel methods.

Revisions

Version	Date	Author	Remarks
1.0	Jul 06	Barthès	Initial Issue
1.1	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

Warning: MOSS 7 is an object language but implemented for efficiency reasons in terms of functional programming[1]. Thus, a number of service functions were written to deal with the object structure, names, versions, printing, etc. Such functions are documented in this document. At the beginning of the document a section deals with the macros used by the system.

The content of the document was produced automatically from the function inline documentation using the User Manual code developed by Mark Kantrowitz at CMU (1991).

1 Convention

The user should be aware of a certain number of conventions used in the system. Some are related to names, others are attached to internal obj-ids.

1.0.1 Elementary function names

All primitive functions start with a % sign, thus following the LISP machine LISP convention.

Some elementary functions are very primitive and somehow dangerous to use, they start with a double %% sign.

Some functions contain the string tp or sp; meaning they apply to attributes or to relations. This comes from a time when attributes were called terminal-properties and relations structural properties. Similarly, class or entity is used to refer to concepts and instances to individuals.

1.0.2 Predicates

Each function in the following set can apply to any object in the system. Predicates end with a question mark following the Scheme convention.

Example: %terminal-property?

1.0.3 Context

Most functions must execute within a *context* (or *version*), i.e. one of the states of the knowledge base corresponding to a node of the version graph. When the context is not passed as an argument to the function, and if it is needed, then the default context `*context*` is used. A version graph recorded in the global variable `*version-graph*` is used to impose the adequate structure onto the data.

1.0.4 Method names

By convention method names start with the = sign. Although this is in no way compulsory it is a good practice to avoid confusion with other kind of variables.

1.0.5 property names

External property names acquire the HAS- prefix, so that they can be distinguished from the class names that don't.

examples: HAS-NAME, HAS-DOCUMENTATION

Inverse property names acquire the IS- prefix and -OF suffix unless a special name is specified for them.

Example: IS-ENTITY-NAME-OF

[1]Not entirely true since Lisp is mostly written using CLOS (Common Lisp Obbject System).

1.0.6 Object ids

Object-ids are assigned by the system and are intended to be used only by the system functions. As a general rule one should not rely on such names, since the system may reassign them to different entities if the system is rebooted. They are only used to distinguish among various objects.

In the MOSS kernel they have names like $ENT, $EPT, $SUC, $CTR, etc. and are interned into the :moss package.

Some instances like methods have more complicated names like: $FN.34

User internal object-ids have systematic naming: (i) classes: $E-XXX, e.g. $E-PERSON, from the canonical external name; (ii) attributes: $T-XXX; (iii) relations: $S-XXX; (iv) individuals: <class-id>.nn, e.g. $E-PERSON.32. The corresponding symbols are interned in the user?s application package (most often :common-lisp-user).

1.0.7 Property ids

A special mention for property ids.

Properties are defined not as a single object but as an object tree. Namely, when a property is defined, a generic property is created, e.g.

```
(defattribute AGE)
```

creates a generic property named $T-NAME

If the property is defined in the context of a class or later added to a class, then a local version is created in addition to the generic version, e.g.

```
(defconcept PERSON (:att age))
```

creates (if needed) a generic property named $T-NAME and a local property $T-PERSON-NAME as a sub-property of the generic property. Both properties share the same HAS-NAME external name.

The mechanism applies also to relations.

2 Macros and Global Variables

The macros documented here can be grouped into several sets. Some macros are related to the object implemented format, some are predicates, some are offered to simplify the declaration of kernel classes or properties or methods, others are quite obscure service functions. In this section they are grouped according to their use. Later on, they are presented in alphabetical order

2.1 Defining Macros

They are intended to facilitate the definition of MOSS objects.

```
;;; DEFATTRIBUTE (name &rest option-list)                                [MACRO]
;;;
;;; %DEFCLASS (multilingual-name &rest option-list)                      [MACRO]
;;;
;;; DEFCHAPTER (&rest option-list)                                       [MACRO]
;;;
;;; DEFCONCEPT (multilingual-name &rest option-list)                     [MACRO]
;;;
;;; %DEFCONCEPT (multilingual-name &rest option-list)                    [MACRO]
;;;
;;; %DEFGENERICTP (name &rest option-list)                               [MACRO]
;;;    see %make-generic-tp doc
;;;
;;; DEFINDIVIDUAL (name &rest option-list)                               [MACRO]
;;;
;;; DEFINSTANCE (name &rest option-list)                                 [MACRO]
;;;
;;; %DEFINSTANCE (name &rest option-list)                                [MACRO]
;;;
;;; DEFINSTMETHOD (name selector arg-list &rest body)                    [MACRO]
;;;
;;; DEFMOSSINSTMETHOD (name selector arg-list &rest body)                [MACRO]
;;;
;;; DEFMOSSOWNMETHOD (name selector arg-list &rest body)                 [MACRO]
;;;
;;; DEFMOSSUNIVERSALMETHOD (name arg-list &rest body)                    [MACRO]
;;;
;;; DEFOBJECT (&rest option-list)                                        [MACRO]
;;;
;;; %DEFOBJECT (&rest option-list)                                       [MACRO]
;;;
;;; DEFONTOLOGY (&rest option-list)                                      [MACRO]
;;;
;;; DEFOWNMETHOD (name selector arg-list &rest body)                     [MACRO]
;;;
;;; %DEFOWNMETHOD (name selector arg-list &rest body)                    [MACRO]
;;;
;;; %DEFMETHOD (name selector arg-list &rest body)                       [MACRO]
;;;
;;; DEFRELATION (name class suc &rest option-list)                       [MACRO]
;;;
```

```
;;; DEFSECTION (&rest option-list)                                    [MACRO]
;;;
;;; %DEFSP (name class suc &rest option-list)                         [MACRO]
;;;
;;; %DEFSYSVAR (name doc value)                                       [MACRO]
;;;
;;; %DEFSYSVARNAME (sysvar-name)                                      [MACRO]
;;;
;;; %DEFTP (name &rest option-list)                                   [MACRO]
;;;
;;; DEFVIRTUALCONCEPT (multilingual-name &rest option-list)           [MACRO]
;;;
;;; %DEFVIRTUALCONCEPT (multilingual-name &rest option-list)          [MACRO]
;;;
;;; %DEFUNIVERSAL (name arg-list &rest body)                          [MACRO]
;;;
;;; DEFUNIVERSALMETHOD (name arg-list &rest body)                     [MACRO]
;;;
```

2.2 Printing Debugging and Tracing Macros

```
;;; DFORMAT (cstring &rest args)                                      [MACRO]
;;;
;;; VFORMAT (cstring &rest args)                                      [MACRO]
;;;
;;; TRFORMAT (fstring &rest args)                                     [MACRO]
;;;    debugging macro used locally explicitely temporarily
;;;
;;; RFORMAT (limit &rest ll)                                          [MACRO]
;;;    takes a text and limits its length. Because the silly Pascal strings
;;;    must be ~ less than some length.
;;;    Arguments:
;;;     limit: max length
;;;     ll: regular arguments to a text.
;;;
;;; TFORMAT (format-string &rest arg-list)                            [MACRO]
;;;    used to simplify global trace
;;;
```

2.3 Macros Dealing with Names

```
;;; %MAKE-NAME-FOR-CLASS (name &key package)                          [MACRO]
;;;    build the class name, e.g. SECTION-TITLE.
;;;    Argument:
;;;    name: symbol or (multilingual) string
;;;    package (opt): package if different from current
;;;    Return:
;;;    interned symbol in the current package.
;;;
;;; %MAKE-NAME-FOR-CLASS-PROPERTY (prop-ref class-ref &key package)   [MACRO]
;;;    build the property name symbol, e.g. HAS-PERSON-FIRST-NAME.
;;;    Argument:
```

```
;;;    prop-ref: symbol or (multilingual) string
;;;    class-ref: specifies the class
;;;    package (opt): package if different from current
;;;    Return:
;;;    interned symbol in the current package.
;;;
;;; %MAKE-NAME-FOR-CONCEPT (name &key package)                          [MACRO]
;;;    build the concept name, e.g. SECTION-TITLE.
;;;    Argument:
;;;    name: symbol or (multilingual) string
;;;    package (opt): package if different from current
;;;    Return:
;;;    interned symbol in the current package.
;;;
;;; %MAKE-NAME-FOR-ENTRY (name &key package)                            [MACRO]
;;;    build an entry name, e.g. SECTION-TITLE.
;;;    Argument:
;;;    name: symbol or (multilingual) string
;;;    package (opt): package if different from current
;;;    Return:
;;;    interned symbol in the current package.
;;;
;;; %MAKE-NAME-FOR-INVERSE-PROPERTY (name &key package)                 [MACRO]
;;;    build the inverse property name, e.g. IS-FIRST-NAME-OF.
;;;    Argument:
;;;    name: symbol or (multilingual) string
;;;    package (opt): package if different from current
;;;    Return:
;;;    interned symbol in the current package.
;;;
;;; %MAKE-NAME-FOR-INSTANCE-METHOD-FUNCTION (name class-id              [MACRO]
;;;                                          &key (context *context*)
;;;                                          package)
;;;    build the instance method function name, e.g. ~
;;;         $E-PERSON=I=0=PRINT-SELF.
;;;    Argument:
;;;    name: symbol or (multilingual) string
;;;    class-id: class id that owns the method
;;;    package (opt): package if different from current
;;;    Return:
;;;    interned symbol in the current package.
;;;
;;; %MAKE-NAME-FOR-OWN-METHOD-FUNCTION (name obj-id                     [MACRO]
;;;                                     &key (context *context*)
;;;                                     package)
;;;    build the own method function name, e.g. $E-PERSON=S=0=PRINT-SELF.
;;;    Argument:
;;;    name: symbol or (multilingual) string
;;;    class-id: id of the object that owns the method
;;;    package (opt): package if different from current
;;;    Return:
```

```
;;;    interned symbol in the current package.
;;;
;;; %MAKE-NAME-FOR-PROPERTY (name &key package)                          [MACRO]
;;;    build the property name, e.g. HAS-FIRST-NAME.
;;;    Argument:
;;;    name: symbol or (multilingual) string
;;;    package (opt): package if different from current
;;;    Return:
;;;    interned symbol in the current package.
;;;
;;; %MAKE-NAME-FOR-UNIVERSAL-METHOD-FUNCTION (name                       [MACRO]
;;;                                           &key context package)
;;;    build the universal method function name, e.g. *0=PRINT-UNIVERSAL.
;;;    Argument:
;;;    name: symbol or (multilingual) string
;;;    context (key): default current
;;;    package (key): package if different from current
;;;    Return:
;;;    interned symbol in the specified package.
;;;
;;; %MAKE-NAME-FOR-VARIABLE (name &key package)                          [MACRO]
;;;    Builds an external variable name starting with underscore.
;;;    Argument:
;;;    name: variable  name
;;;    Returns:
;;;    the name starting with an underscore
;;;
;;; %MAKE-NAME-STRING-FOR-CONCEPT (name)                                 [MACRO]
;;;    Cooks up a name string from the name of a concept.
;;;    Argument:
;;;    name: a symbol or (multilingual) string naming the concept
;;;    Return:
;;;    a string naming the concpet
;;;
;;; %MAKE-NAME-STRING-FOR-CONCEPT-PROPERTY (name-string class-ref)       [MACRO]
;;;    Builds an external property name string, starting with HAS-
;;;    Arguments:
;;;    name: a multilingual name naming the property given by the user as
;;;    an ~ argument to m-deftp for example (m-deftp (:fr "titre de
;;;    l'ouvrage" :en "book title") ...) Return:
;;;    a string naming the property, e.g. "HAS-TITLE"
;;;
;;; %MAKE-NAME-STRING-FOR-INVERSE-PROPERTY (name)                        [MACRO]
;;;    Cooks up an inverse property string from the name of the property by
;;;    sticking ~ IS- and -OF an each side. Remember the inverse property
;;;    has no semantic ~ content. It simply provides the possibility of
;;;    following the arcs in the ~ reverse direction.
;;;    Argument:
;;;    name: a symbol or string naming the direct property
;;;    Return:
;;;    a string naming the inverse property
```

```
;;;
;;; %MAKE-NAME-STRING-FOR-METHOD-FUNCTION (class-id method-type          [MACRO]
;;;                                        context method-name)
;;;    Builds an internal name to store compiled methods, e.g.
;;;    $CTR=I=2=summary ~ or $E-PERSON=S=1=print-self where 1 is the context
;;;    value. Arguments:
;;;    class-id: identifier of the class
;;;    method-type: :instance (I) or own (S)
;;;    context: context
;;;    method-name: name of the method
;;;    Returns:
;;;    a symbol with package that of the method name.
;;;
;;; %MAKE-NAME-STRING-FOR-PROPERTY (name-string)                         [MACRO]
;;;    Builds an external property name string, starting with HAS-
;;;    Arguments:
;;;    name: a multilingual name naming the property given by the user as
;;;    an ~ argument to m-deftp for example (m-deftp (:fr "titre de
;;;    l'ouvrage" :en "book title") ...) Return:
;;;    a string naming the property, e.g. "HAS-TITLE"
;;;
;;; %MAKE-NAME-STRING-FOR-RADIX (name &optional (type ""))               [MACRO]
;;;    build a radix name string for a property or for a class. Tries to
;;;    build a unique name ~ using the first 3 letters of the name. Error if
;;;    the symbol already exists. Arguments:
;;;    name: e.g. "SIGLE"
;;;    type (opt): e.g. "T " (for terminal property)
;;;    Return:
;;;    e.g. $T-SIG or error if already exists
;;;
;;; %MAKE-NAME-STRING-FOR-UNIVERSAL-METHOD-FUNCTION (context name)       [MACRO]
;;;    Builds an internal name to store compiled methods.
;;;    Arguments:
;;;    context: context
;;;    name: name of the universal method
;;;    Returns:
;;;    a symbol with package that of the method name.
;;;
;;; %MAKE-NAME-STRING-FOR-VARIABLE (name)                                [MACRO]
;;;    Builds an internal name string for a variable.
;;;    Arguments:
;;;    name: string, symbol or multilingual name
;;;    Returns:
;;;    a string.
;;;
;;; %MKNAME (ref type &key package)                                      [MACRO]
;;;    if ref is a symbol return symbol, otherwise call~
;;;         %make-name-for-<type>.
;;;    Arguments:
;;;    ref: a symbol or (multilingual) string
;;;    type: type of name to recover (keyword)
```

```
;;;    package (key): default current
;;;    Expansion:
;;;    (%make-name-for-<type> ref . more-args)
;;;
```

2.4 Macros Dealing with Ids

```
;;; %MAKE-ID-FOR-CLASS (name &key package)                              [MACRO]
;;;    function to cook up default entity ids: $E-<name>
;;;    If name is a symbol then the symbol is interned into the symbol's
;;;    ~ package, otherwise it is interned into the specified package, or ~
;;;         if not there into the current package.
;;;    Argument:
;;;    name: symbol or (multilingual) to be used for radix
;;;    package (key): default current
;;;    Return:
;;;    a symbol specifying the class, e.g. $E-PERSON
;;;
;;; %MAKE-ID-FOR-VIRTUAL-CLASS (name &key package)                      [MACRO]
;;;    function to cook up default entity ids: $E-<name>
;;;    If name is a symbol then the symbol is interned into the symbol's
;;;    ~ package, otherwise it is interned into the specified package, or ~
;;;         if not there into the current package.
;;;    Argument:
;;;    name: symbol or (multilingual) to be used for radix
;;;    package (key): default current
;;;    Return:
;;;    a symbol specifying the class, e.g. $E-PERSON
;;;
;;; %MAKE-ID-FOR-CLASS-SP (name class-name &key package)                [MACRO]
;;;    function to cook up a default relation id.
;;;    If name is a symbol then the symbol is interned into the symbol's
;;;    ~ package, otherwise it is interned into the specified package, or ~
;;;         if not there into the current package.
;;;    Argument:
;;;    name: symbol or (multilingual) to be used for radix
;;;    package (key): default current
;;;    Return:
;;;    a symbol specifying the class, e.g. $S-<class-name>-<name>
;;;
;;; %MAKE-ID-FOR-CLASS-TP (name class-name &key package)                [MACRO]
;;;    function to cook up default property ids.
;;;    If name is a symbol then the symbol is interned into the symbol's
;;;    ~ package, otherwise it is interned into the specified package, or ~
;;;         if not there into the current package.
;;;    Argument:
;;;    name: symbol or (multilingual) to be used for radix
;;;    package (key): default current
;;;    Return:
;;;    a symbol specifying the class, e.g. $T-<class-name>-<name>
;;;
;;; %MAKE-ID-FOR-CONCEPT (name &key package)                            [MACRO]
```

```
;;;    see %make-id-for-class
;;;
;;; %MAKE-ID-FOR-VIRTUAL-CONCEPT (name &key package)                    [MACRO]
;;;    see %make-id-for-virtual-class
;;;
;;; %MAKE-ID-FOR-COUNTER (class-id)                                     [MACRO]
;;;    makes an ID for a counter, e.g. $E-PERSON.CTR in class-id package.
;;;    Arguments:
;;;    class-id: id of the class to which the counter belongs
;;;    Returns:
;;;    the counter id.
;;;
;;; %MAKE-ID-FOR-IDEAL (class-id)                                       [MACRO]
;;;    makes an ID for the class ideal, e.g. $E-PERSON.0 in class-id
;;;    package. Arguments:
;;;    class-id: class ID
;;;    Return:
;;;    a symbol that is the ID of the ideal.
;;;
;;; %MAKE-ID-FOR-INSTANCE (model-id value &key context package)         [MACRO]
;;;    Builds internal keys for instances using the radix of a class; e.g. ~
;;;         $E-PERSON.34 (34 is the value obtained from the class counter).
;;;    The package of the new symbol is that of the class.
;;;    Arguments:
;;;    model-id: identifier of the class
;;;    value: value of the counter
;;;    context (key): default current
;;;    package (key): ignored
;;;    Returns:
;;;    an identifier obtained by applying make-name.
;;;
;;; %MAKE-ID-FOR-INSTANCE-METHOD (context value &key package)           [MACRO]
;;;    Builds an internal id for an own-method: e.g. $FN.34
;;;    Arguments:
;;;    context: context
;;;    value: counter for methods
;;;    package (key): default current
;;;    Return:
;;;    method id.
;;;
;;; %MAKE-ID-FOR-INVERSE-PROPERTY (id)                                  [MACRO]
;;;    Builds an inverse property id from a direct one. The function is
;;;    useful  ~ only for MOSS kernel. Indeed, for applications, inverse
;;;    properties are ~ instances of the $EIL class, and have $EIL.nn as
;;;    internal name. Arguments:
;;;    id: identifier of the direct property
;;;    Return:
;;;    symbol for the inverse property id.
;;;
;;; %MAKE-ID-FOR-ORPHAN (context value &key package)                    [MACRO]
;;;    Builds an internal id for an orphan object: e.g. $3-5 where 3 is ~
```

```
;;;         the context value, 5 the system's counter value.
;;;    Arguments:
;;;    context: context
;;;    value: counter for orphans
;;;    Return:
;;;    orphan id.
;;;
;;; %MAKE-ID-FOR-OWN-METHOD (context value &key package)                [MACRO]
;;;    Builds an internal id for an own-method: e.g. $FN.34
;;;    Arguments:
;;;    context: context
;;;    value: counter for methods
;;;    package (key): default current
;;;    Return:
;;;    method id.
;;;
;;; %MAKE-ID-FOR-SP (name &key package)                                  [MACRO]
;;;    function to cook up default property ids:
;;;    If name is a symbol then the symbol is interned into the symbol's
;;;    ~ package, otherwise it is interned into the specified package, or ~
;;;         if not there into the current package.
;;;    Arguments:
;;;    name: symbol, string or mln
;;;    package (key): default current
;;;    Return:
;;;    a symbol anem $S-<name>
;;;
;;; %MAKE-ID-FOR-TP (name &key package)                                  [MACRO]
;;;    function to cook up default property ids: $T-<name>
;;;    If name is a symbol then the symbol is interned into the symbol's
;;;    ~ package, otherwise it is interned into the specified package, or ~
;;;         if not there into the current package.
;;;    Arguments:
;;;    name: symbol or (multilingual) string
;;;    package (key): default current
;;;    Return:
;;;    a symbol anem $T-<name>
;;;
;;; %MAKE-ID-FOR-UNIVERSAL-METHOD (context value &key package)           [MACRO]
;;;    Builds an internal id for an own-method: e.g. $UNI.34
;;;    Arguments:
;;;    context: context
;;;    value: counter for methods
;;;    package (key): default current
;;;    Return:
;;;    method id.
;;;
```

2.5 Program Control

```
;;; CATCH-ERROR (msg &rest body)                                         [MACRO]
;;;    intended to catch errors due to poor MOSS syntax. Prints in active
```

```
;;;    window ~ using mformat unless msg is empty
;;;    Note: body should not return a string (considered as an error
;;;    message) Arguments:
;;;    msg: something like "while executing xxx"
;;;    body (rest): exprs to execute
;;;    Return:
;;;    the value of executing body if no error, nil and prints a message
;;;    when error
;;;
;;; CATCH-SYSTEM-ERROR (msg &rest body)                              [MACRO]
;;;
;;; MTHROW (control-string &rest args)                               [MACRO]
;;;    print a message and throws the message string to :error
;;;
;;; TERROR (cstring &rest args)                                      [MACRO]
;;;    throws an error message back to an :error catch. Prints message if
;;;    *verbose*
;;;
;;; TWARN (cstring &rest args)                                       [MACRO]
;;;    inserts a message into the *error-message-list* making sure to return
;;;    nil
;;;
;;; VERBOSE-THROW (label &rest format-args)                          [MACRO]
;;;    when *verbose* is T, prints a warning before throwing the error
;;;    message ~ string to the label.
;;;    Arguments:
;;;    label: a label to which to throw (usually :error)
;;;    format-args: arguments to build a message string using format nil
;;;    Return:
;;;    throws to the label the message string.
;;;
;;; VERBOSE-WARN (&rest format-args)                                 [MACRO]
;;;
;;; WHILE (condition &rest body)                                     [MACRO]
;;;
```

2.6 Environment

```
;;; IN-VERSION (%_context &rest ll)                                  [MACRO]
;;;
;;; WITH-CONTEXT (%_context &rest ll)                                [MACRO]
;;;
;;; WITH-PACKAGE (%_package &rest ll)                                [MACRO]
;;;
```

2.7 Low Level or Compatibility Macros and Functions

Such macros or functions are used for facilitating programming or for upward compatibility reasons.

```
;;; ASSREMPROP (prop ll)                                             [MACRO]
;;;
;;; GETAV (key list)                                                 [MACRO]
```

```
;;;
;;; GETV (prop ll)                                                    [MACRO]
;;;
;;; INTER (&rest ll)                                                  [MACRO]
;;;
;;; NASSREMPROP (prop ll)                                             [MACRO]
;;;
;;; NCONS (arg)                                                       [MACRO]
;;;
;;; PUTPROP (xx val prop)                                             [MACRO]
;;;
;;; ALISTP (ll)                                                     [FUNCTION]
;;;
;;; ALL-ALIKE? (ll)                                                 [FUNCTION]
;;;    T if all elements of the list are the same
;;;
;;; ALL-DIFFERENT? (ll)                                             [FUNCTION]
;;;    T if all elements of the list are different
;;;
;;; INTERSECT-SETS (list)                                           [FUNCTION]
;;;
;;; LAMBDAP (symbol)                                                [FUNCTION]
;;;    check if symbol is a lambda list
;;;
;;; LOB-GET (obj-id prop-id)                                        [FUNCTION]
;;;
;;; LOB-TYPEP (xx)                                                  [FUNCTION]
;;;
;;; MEXP (form)                                                     [FUNCTION]
;;;
;;; MAKE-NAME (&rest arg-list)                                      [FUNCTION]
;;;
;;; MAKE-VALUE-STRING (value &optional (interchar '#\.))            [FUNCTION]
;;;    Takes an input string, removing all leading and trailing blanks,
;;;    replacing ~ all substrings of blanks and simple quote with a single
;;;    underscore, and ~ building a new string capitalizing all letters. The
;;;    function is used ~ to normalize values for comparing a value with an
;;;    external one while ~ querying the database.
;;;         French accentuated letter are replaced with unaccentuated
;;;    capitals.
;;;
;;; MFORMAT (control-string &rest args)                             [FUNCTION]
;;;    prints to the output specified by output and a trace into the
;;;    listener. Arguments:
;;;    output: stream (main be a window pane
;;;    control-string: format control string
;;;    args (rest): args if any
;;;    Returns:
;;;    output-string (to be used in throw eventually)
;;;
;;; MGF-CLOSE "()"                                                  [FUNCTION]
```

```
;;;    Closes the currently opened database
;;;
;;; MGF-END-TRANSACTION "()"                                          [FUNCTION]
;;;
;;; MGF-ERRORP "()"                                                   [FUNCTION]
;;;    Returns the error code set by MGF functions.
;;;
;;; MGF-LOAD (xx)                                                     [FUNCTION]
;;;    noop as for now
;;;
;;; MGF-OPEN (filename)                                               [FUNCTION]
;;;    Opens a specific database and puts its name into the *base* global
;;;    variable
;;;
;;; MGF-START-TRANSACTION "()"                                        [FUNCTION]
;;;    In a multiuser environment prevents other users to access the MGF
;;;    file.
;;;
;;; MGF-STORE (key value)                                             [FUNCTION]
;;;    stores a given key and value into the database
;;;
;;; MGF-UPDATE (key value)                                            [FUNCTION]
;;;    stores a given key and value into the database replacing old value if
;;;    any.
;;;
;;; NO-P (word)                                                       [FUNCTION]
;;;    Returns some non nil value if arg is noor non or some substring
;;;
;;; NALIST-REPLACE-PV (obj-id prop-id value &aux obj-l)               [FUNCTION]
;;;
;;; NPUTV (prop-id value ll)                                          [FUNCTION]
;;;    nputv is defined for compatibility with UTC-Lisp. It is a destructive
;;;    function it adds a value to the list of values corresponding to a
;;;    property without duplicating it
;;;
;;; NPUTV-LAST (prop val ll)                                          [FUNCTION]
;;;    nputv-last is defined for compatibility with UTC-Lisp. It adds a
;;;    value to the list of values associated with a property in an a-list
;;;    at the last position. Destructive function.
;;;
;;; REF (symbol &optional show-file)                                  [FUNCTION]
;;;    traces in what function a specific symbol appears by examining the
;;;    code of the ~ function in the corresponding file.
;;;
;;; APPEARS-IN? (symbol expr)                                         [FUNCTION]
;;;
;;; YES-P (word)                                                      [FUNCTION]
;;;    Returns some non nil value if arg is yes or oui or some substring
```

2.8 Object Structure

```
;;; MOSS-SYMBOL? (obj-id)                                             [FUNCTION]
```

```
;;;    check whether symbol is defined in the moss package
;;;
;;; %%RESOLVE (id context)                                          [MACRO]
;;;    resolves references, pseudo objects with a $REF property, replacing
;;;    the ~ specified id with the id pointed to in the specified context.
;;;    If unbound or actual object, then returns the input id.
;;;    Arguments:
;;;    id: symbol containing the object id
;;;    context (opt): context (default current)
;;;    Return:
;;;    id or id pointed to
;;;
```

2.9 Global Variables

```
;;; *APPLICATION-NAME* ("moss-test")                                [PARAMETER]
;;;    default disk file
;;;
;;; *CONTEXT* (0)                                                   [PARAMETER]
;;;    number of the current ontology version
;;;
;;; *LANGUAGE* (:en)                                                [PARAMETER]
;;;    default language is English
;;;
;;; *VERSION-GRAPH* ('((0)))                                        [PARAMETER]
;;;    configuration lattice
;;;
;;; *SENDER* (nil)                                                  [PARAMETER]
;;;    sender of MOSS message
;;;
;;; *ANSWER* ("answer returned by a moss method")                    [VARIABLE]
;;;
;;; *SELF* (nil)                                                    [PARAMETER]
;;;    id of object that received the MOSS message
;;;
;;; *METHOD-NAME* ("name of the method being executed")             [PARAMETER]
;;;
;;; *EXPORT-ENTRY-SYMBOLS* (t)                                      [PARAMETER]
;;;    MOSS system exports entry symbols
;;;
;;; *DEBUG* (nil)                                                   [PARAMETER]
;;;    debug flag triggering the dformat macro
;;;
;;; *DELIMITERS* ('(#\space #\, #\' #\.))                           [PARAMETER]
;;;    characters delimiting a word
;;;
;;; *TRACED-AGENT* (nil)                                            [PARAMETER]
;;;    omas agent being traced
;;;
;;; *SYSTEM-ENTITIES* (nil)                                          [VARIABLE]
;;;    record the list of MOSS entities (classes)
;;;
```

```
;;; *WARNING-FOR-OWN-METHODS* (nil)                                  [PARAMETER]
;;;    flag for warning messages
;;;
;;; *VERBOSE* (nil)                                                  [VARIABLE]
;;;    trace error messages in warnings
;;;
;;; *ALLOW-FORWARD-REFERENCES* (nil)                                 [PARAMETER]
;;;    for relatiosns to an undefined object
;;;
;;; *ALLOW-FORWARD-INSTANCE-REFERENCES* (nil)                        [PARAMETER]
;;;    for relatiosns to an undefined instance
;;;
;;; *DEFERRED-ACTIONS* (nil)                                         [PARAMETER]
;;;    list of actions deferred until the end of the file
;;;
;;; *DEFERRED-INSTANCE-CREATIONS* (nil)                              [PARAMETER]
;;;    list of instance creation actions
;;;
;;; *DEFERRED-DEFAULTS* (nil)                                        [PARAMETER]
;;;    list of default creation actions
;;;
;;; *BASE* (nil)                                                     [VARIABLE]
;;;    Name of the currently opened base
;;;
;;; *DISK* (nil)                                                     [VARIABLE]
;;;    a-list simulating a disk file
;;;
;;; *LEFT-MARGIN* (0)                                                [VARIABLE]
;;;
;;; *LOB* (nil)                                                      [VARIABLE]
;;;    LOB object for handling update programs. Initialized in the LOB
;;;    module.
;;;
;;; *MGF-ERROR* (nil)                                                [VARIABLE]
;;;
;;; *MOSS-STAND-ALONE* (t)                                           [VARIABLE]
;;;    indicates whether MOSS is standalone or not
;;;
;;; *MOSS-TRACE-LEVEL* (0)                                           [VARIABLE]
;;;    Column number for starting the trace print out.
;;;
;;; *PRETTY-TRACE* (nil)                                             [VARIABLE]
;;;
;;; *STRING-HYPHEN* ("-")                                            [VARIABLE]
;;;
;;; *MOSS-SYSTEM* (nil)                                              [VARIABLE]
;;;
;;; *TRANSACTION-NUMBER* (nil)                                       [VARIABLE]
;;;    Used by LOB
;;;
;;; *CONVERSATION* (nil)                                             [PARAMETER]
```

```
;;;    current-conversation; not used for OMAS
;;;
;;; *MOSS-CONVERSATION* (nil)                                    [PARAMETER]
;;;    current-conversation; not used for OMAS
;;;
;;; *DIALOG-VERBOSE* (nil)                                       [PARAMETER]
;;;    trace of dialog flag
;;;
;;; *TRANSITION-VERBOSE* (nil)                                   [PARAMETER]
;;;    trace dialog transitions flag
;;;
;;; *MOSS-ENGINE-LOADED* (nil)                                    [VARIABLE]
;;;    indicates that the send mechanism is loaded
;;;
;;; *CACHE-METHODS* (nil)                                         [VARIABLE]
;;;    Flag that allows to cache methods onto p-list of ~
;;;                                                          classes and
;;;    objects. Turned off for debugging purposes
;;;
;;; *LEXICOGRAPHIC-INHERITANCE* (t)                               [VARIABLE]
;;;    Disable fancy inheritance scheme in favor ~
;;;
;;;    of a standard lexicographic scheme
;;;
;;; *USER* ('anonymous)                                           [VARIABLE]
;;;    The identity of the user of the system.
;;;
;;; *TRACE-FLAG* (nil)                                            [VARIABLE]
;;;    Flag for toggling trace of messages on and off.
;;;
;;; *TRACE-MESSAGE* (nil)                                         [VARIABLE]
;;;    Flag for tracing all messages to all objects.
;;;
;;; *MOSS-WINDOW* (nil)                                          [PARAMETER]
;;;    MOSS interface
;;;
;;; *MOSS-OUTPUT* (t)                                            [PARAMETER]
;;;    output channel: t or MOSS window
;;;
;;; *MOSS-INPUT* (t)                                             [PARAMETER]
;;;    input channel: t or MOSS window
;;;
;;; *ALLOWED-RESTRICTION-OPERATORS* ('(:type :not-type :unique    [VARIABLE]
;;;                                    :one-of :exists :forall :not
;;;                                    :value :between :outside
;;;                                    :filter :same :different :min
;;;                                    :max :unique))
;;;    allowed operators used to restrict the value of a property
;;;
;;; *LANGUAGE-TAGS* ('(:en :fr :it :pl))                         [PARAMETER]
;;;    allowed languages
```

```
;;;
;;; *TRANSLATION-TABLE* ('((#\à #\a) (#\é #\e) (#\è #\e) (#\ê #\e)   [VARIABLE]
;;;                       (#\ë #\e) (#\î #\i) (#\ï #\i) (#\ù #\u)
;;;                       (#\ü #\u) (#\û #\u) (#\' #\-)
;;;                       (#\? #\o #\e)))
;;;
```

3 Functions

In this section, functions have been grouped by features.

3.1 Resetting the environment

```
;;; %RESET (&key (context *context* there?))                          [FUNCTION]
;;;    Removes all classes and instances of an application, making the
;;;    corresponding ~ symbols unbound. Protects all data defined in the
;;;    kernel (moss  package). Arguments:
;;;    none
;;;    Return:
;;;    t
;;;
```

3.2 Predicates

```
;;; %ANCESTOR? (obj1 obj2 &key (prop-id '$is-a)                       [FUNCTION]
;;;             (context *context*))
;;;    Checks if obj2 is equal to obj1, or belongs to the transitive closure
;;;    of obj1 ~ using property prop-id.
;;;    Arguments:
;;;    obj1: first object
;;;    obj2: second object
;;;    prop-id: property to follow (default is $IS-A)
;;;    context (opt): context (default current)
;;;
;;; %CLASSLESS-OBJECT? (object-id &key (context *context*))           [FUNCTION]
;;;    Checks if the entity is a classless object, i.e. its type is *none*
;;;    Arguments:
;;;    object-id: identifier of object
;;;    context (opt): context (default current)
;;;    Return:
;;;    T if OK, NIL otherwise.
;;;
;;; %ENTRY? (object-id &key (context *context*)                       [FUNCTION]
;;;          (package *package* package-here?))
;;;    Checks if the id is that of an entry point.
;;;    Arguments:
;;;    object-id: identifier of object
;;;    context (opt): context (default current)
;;;    package (opt): if obj-id is a symbol, then that of obj-id default
;;;    current Returns:
;;;    nil or something not meaningful
;;;
;;; %IS-A? (obj1 obj2 &key (context *context*))                       [FUNCTION]
;;;    Checks if obj1 is equal to obj2, or obj-2 belongs to the transitive
;;;    closure of obj1~ using property $IS-A.
;;;    Arguments:
;;;    obj1: first object
;;;    obj2: second object
;;;    context (opt): context (default current)
```

```
;;;
;;; %IS-ATTRIBUTE? (prop-id &key (context *context*))                [FUNCTION]
;;;    Checks if the identifier points towards an attribute
;;;    Arguments:
;;;    prop-id: object identifier
;;;    context (opt): context (default current)
;;;    Return:
;;;    T or nil
;;;
;;; %IS-ATTRIBUTE-MODEL? (object-id &key (context *context*))        [FUNCTION]
;;;    Checks if the object is $EPT or some sub-type.
;;;    Arguments:
;;;    object-id: object identifier
;;;    context (opt): context (default current)
;;;    Return:
;;;    T or nil
;;;
;;; %IS-CLASS? (object-id &key (context *context*))                  [FUNCTION]
;;;    Checks if the object is a class, i.e. if its type is $ENT or a
;;;    subtype of $ENT. Arguments:
;;;    object-id: identifier of the object to check
;;;    context (opt): context default current
;;;    Return
;;;    nil or a list result of an intersection.
;;;
;;; %IS-CONCEPT? (object-id &key (context *context*))                [FUNCTION]
;;;    Checks if the object is a concept (same as class), i.e. if its type
;;;    is $ENT or a subtype of $ENT. Arguments:
;;;    object-id: identifier of the object to check
;;;    context (opt): context default current
;;;    Return
;;;    nil or a list result of an intersection.
;;;
;;; %IS-COUNTER-MODEL? (object-id &key (context *context*))          [FUNCTION]
;;;    Checks if the object is $CTR or some sub-type.
;;;    Arguments:
;;;    object-id: identifier of the object to check
;;;    context (opt): context default current
;;;    Return
;;;    nil or a list result of an intersection.
;;;
;;; %IS-ENTITY-MODEL? (object-id &key (context *context*))           [FUNCTION]
;;;    Checks if the object is $ENT or some sub-type.
;;;
;;; %IS-ENTRY-POINT-INSTANCE? (object-id &key (context *context*))   [FUNCTION]
;;;    Checks if the object is an instance of entry-point.
;;;
;;; %IS-GENERIC-PROPERTY-ID? (object-id &key (context *context*))    [FUNCTION]
;;;    Checks if the object is $EPR and is not a subproperty of some other
;;;    property.
;;;
```

```
;;; %IS-INSTANCE-OF? (object-id class-id)                              [FUNCTION]
;;;    Checks if the object is an instance of class id or of one of its
;;;    children. Arguments:
;;;    object-id: id of the object too test
;;;    class-id: id of the class
;;;    Return:
;;;    t if true nil otherwise.
;;;
;;; %IS-INVERSE-LINK-MODEL? (object-id &key (context *context*))      [FUNCTION]
;;;    Checks if the object is $EIL or some sub-type.
;;;
;;; %IS-INVERSE-PROPERTY? (xx &key (context *context*))               [FUNCTION]
;;;    Checks if the id points towards an inverse property, or to a property
;;;    containing $EIL in its transitive closure along $IS-A.
;;;
;;; %IS-INVERSE-PROPERTY-REF? (xx &key (context *context*))           [FUNCTION]
;;;    Checks if the string is the reference of an inverse property, i.e. a
;;;    string ~ starting with the > symbol whose name is that of a property.
;;;    Argument:
;;;    xx: string
;;;    Return:
;;;    the inverse property-id or nil.
;;;
;;; %IS-METHOD? (xx &key (context *context*))                          [FUNCTION]
;;;    Checks if the id points towards a method - Optional arg: context
;;;
;;; %IS-METHOD-MODEL? (object-id &key (context *context*))            [FUNCTION]
;;;    Checks if the object is $EFN or some sub-type.
;;;
;;; %IS-MODEL? (object-id &key (context *context*))                   [FUNCTION]
;;;    Checks whether an object is a model (class) or not - Models ~
;;;         are objects that have properties like $PT or $PS or $RDX or
;;;    $CTRS ~ or whose type is in the transitive closure of $ENT along the
;;;    $IS-A.OF link.
;;;
;;; %IS-ORPHAN? (object-id &key (context *context*))                  [FUNCTION]
;;;    Short synonym for %classless-object?
;;;
;;; %IS-RELATION? (prop-id &key (context *context*))                  [FUNCTION]
;;;    Checks if the object is $EPS or some sub-type.
;;;
;;; %IS-RELATIONSHIP? (prop-id &key (context *context*))              [FUNCTION]
;;;    Checks if the object is $EPS or some sub-type.
;;;
;;; %IS-RELATION-MODEL? (object-id &key (context *context*))          [FUNCTION]
;;;    checks if the object is $EPS or some sub-type
;;;
;;; %IS-STRUCTURAL-PROPERTY? (prop-id &key (context *context*))       [FUNCTION]
;;;    Checks if the identifier points towards a structural property -
;;;     Optional arg: context
;;;
```

```
;;; %IS-STRUCTURAL-PROPERTY-MODEL? (object-id                          [FUNCTION]
;;;                                 &key (context *context*))
;;;    Checks if the object is $EPS or some sub-type.
;;;
;;; %IS-SYSTEM? (object-id &key (context *context*))                   [FUNCTION]
;;;    Checks if the id points towards a system - Optional arg: context
;;;
;;; %IS-SYSTEM-MODEL? (object-id &key (context *context*))             [FUNCTION]
;;;    Checks if the entity is a model, and belongs to the MOSS package.~
;;;         Arguments:
;;;    object-id: identifier of object to test
;;;    context (opt): context default current
;;;    Return:
;;;    nil or t
;;;
;;; %IS-TERMINAL-PROPERTY? (prop-id &key (context *context*))          [FUNCTION]
;;;    Checks if the identifier points towards a terminal property -
;;;     Optional arg: context
;;;
;;; %IS-TERMINAL-PROPERTY-MODEL? (object-id                           [FUNCTION]
;;;                                &key (context *context*))
;;;    Checks if the object is $EPT or some sub-type.
;;;
;;; %IS-UNIVERSAL-METHOD? (xx &key (context *context*))                [FUNCTION]
;;;    Checks if the id points towards a universal method - Optional arg:
;;;    context
;;;
;;; %IS-UNIVERSAL-METHOD-MODEL? (object-id                            [FUNCTION]
;;;                               &key (context *context*))
;;;    Checks if the object is $UNI or some sub-type.
;;;
;;; %IS-VALUE-TYPE? (value value-type)                                [FUNCTION]
;;;    Checks that a value has the right type.
;;;    Arguments:
;;;    value: value to check
;;;    value-type: one of the XSD types (birkkkhh!)
;;;    Returns:
;;;    value if the right type, nil otherwise.
;;;
;;; %IS-VARIABLE-NAME? (ref)                                          [FUNCTION]
;;;    Checks if the refernce is a variable name, i.e. a symbol starting
;;;    with _ (underscore)
;;;
;;; %IS-VIRTUAL-CONCEPT? (object-id &key (context *context*))         [FUNCTION]
;;;    Checks if the object is a concept (same as class), i.e. if its type
;;;    is $ENT or a subtype of $ENT. Arguments:
;;;    object-id: identifier of the object to check
;;;    context (opt): context default current
;;;    Return
;;;    nil or a list result of an intersection.
;;;
```

```
;;; %PCLASS? (input class-ref)                                          [FUNCTION]
;;;    tests if input belongs to the transitive closure of class-ref. Only
;;;    works in ~ if the language is that if the input and class ref.
;;;    Arguments:
;;;    input: some expression that must be a string
;;;    class-ref: a string naming a class
;;;    Return:
;;;    nil if not the case, something not nil otherwise.
;;;
;;; %PDM? (xx)                                                          [FUNCTION]
;;;    Checks if an object had the right PDM format, i.e. it must be a
;;;    symbol, bound, ~ be an alist, and have a local $TYPE property.
;;;    References are PDM objects
;;;
;;; %SUBTYPE? (object-id model-id &key (context *context*))             [FUNCTION]
;;;    Checks if object-id is a sub-type of model-id including itself.
;;;    Arguments:
;;;    object-id: identifier of object to check for modelhood
;;;    model-id: reference-model
;;;    context (opt): context default current
;;;    Return:
;;;    T or nil.
;;;
;;; %TYPE? (object-id class-id &key (context *context*))                [FUNCTION]
;;;    Checks if one of the classes of object-id is class-id or one of its
;;;    ~ subclasses.
;;;       If class is *any* then the result is true.
;;;    Arguments:
;;;    object-id: the id of a pdm object
;;;    class-id: the id of a class
;;;    Returns
;;;    nil or T.
;;;
;;; %TYPE-OF (obj-id &key (context *context*))                          [FUNCTION]
;;;    Extracts the type from an object (name of its class(es)).
;;;    Careful: returns "CONS" if following a programming error ~
;;;         obj-id pass the value of the identifier...
;;;    Argument:
;;;    obj-id: identifier of a particular object
;;;    Return:
;;;    a list containing the identifiers of the class(es) of which the
;;;    object is an instance ~ or its lisp type if it is not a PDM object.
;;;
```

3.3 Functions Dealing with the Internal Object Structure

```
;;; %%ALIVE? (obj-id context)                                           [FUNCTION]
;;;    Checks if object is PDM, *context* is OK, and object is alive. If
;;;    not, throws ~ to error.
;;;    Arguments:
;;;    obj-id: id
;;;    context (opt): context (default current)
```

```
;;;    Return:
;;;    resoved id, if OK, throws to :error otherwise.
;;;
;;; %ALIVE? (obj-id &key (context *context*))                          [FUNCTION]
;;;    Checks if object exists in given context ~
;;;         each object has a tombstone which records if it has been killed
;;;    ~ by somebody in the given context. This is a source of inconsistency
;;;    ~ since if the tombstone exists then the properties must be removed
;;;    in ~ current context. I.e. one cannot have a link onto an object
;;;    which ~ has been killed.
;;;    If context is illegal throws to :error.
;;;    Arguments:
;;;    obj-id: identifier of the object to be checked
;;;    context (opt): context (default current context)
;;;    Return:
;;;    nil if object is dead, resolved id otherwise
;;;
;;; %%ALLOWED-CONTEXT? (context)                                       [FUNCTION]
;;;    Checks if the context is allowed. Does this by looking at the global
;;;    variable *version-graph*. If not allowed throws to :error.
;;;
;;; %SP-GAMMA (node arc-label &key (context *context*))                [FUNCTION]
;;;    Returns the transitive closure ~
;;;         of objects for a given version.
;;;       This is the famous gamma function found in graph theory.
;;;       No check that arc-label is a structural property.
;;;    Arguments:
;;;    node: starting node
;;;    arc-label: label to follow on the arcs (e.g. $IS-A)
;;;    context (opt): context, default value is *context* (current)
;;;
;;; %SP-GAMMA-L (node-list arc-label &key (context *context*))         [FUNCTION]
;;;    Computes a transitive closure
;;;    Arguments:
;;;    node-list: a list of nodes from which to navigate
;;;    arc-label: the label of the arc we want to use
;;;    context (opt): context default current
;;;    Return:
;;;    a list of nodes including those of the node-list.
;;;
;;; %SP-GAMMA1 (ll prop stack old-set version)                         [FUNCTION]
;;;    Called by %sp-gamma for computing a transitive closure.
;;;       gamma1 does the job.
;;;       computes the whole stuff depth first using a stack of yet
;;;    unprocessed ~ candidates.
;;;     ll contains the list of nodes being processed
;;;     stack the ones that are waiting
;;;     old-set contains partial results.
;;;    Arguments:
;;;    ll: candidate list of nodes to be examined
;;;    prop: property along which we make the closure
```

```
;;;    stack: stack of waiting candidates
;;;    old-set: list containing the result
;;;    version: context
;;;    Return:
;;;    list of nodes making the transitive closure.
;;;
;;; %VG-GAMMA (context)                                              [FUNCTION]
;;;    Computes the transitive closure upwards on the version graph.
;;;       Get the transitive closure of the context i.e. a merge of all the
;;;    paths ~ from context to root without duplicates, traversed in a
;;;    breadth first ~ fashion, since we assume that the value should not be
;;;    far from our context ~ This list should be computed once for all when
;;;    the context is set ~ e.g. in a %set-context function
;;;    Argument:
;;;    context: starting context
;;;    Return:
;;;    a list of contexts.
;;;
;;; %VG-GAMMA-L (lres lnext ll pred)                                 [FUNCTION]
;;;    Service function used be %vg-gamma
;;;    Arguments:
;;;    lres: resulting list
;;;    lnext: next candidates to examine later
;;;    ll: working list of candidates
;;;    pred: list of future nodes to process
;;;    Return:
;;;    lres
;;;
;;; %%ZAP-OBJECT (obj-id)                                            [FUNCTION]
;;;    Deletes an object by removing all values from all contexts and making
;;;    its ~ identifier unbound. This can destroy the database consistency.
;;;    Arguments:
;;;    obj-id: identifier of the object
;;;    Return:
;;;    not significant.
;;;
;;; %ZAP-PLIST (obj-id)                                              [FUNCTION]
;;;    Clears the plist of a given symbol.
;;;
```

3.4 Functions Dealing with Versions (Contest)

```
;;; %EXPLODE-OBJECT (obj-id &key into)                               [FUNCTION]
;;;    function that explodes an instance into a set of triples appending it
;;;    to the ~ list specified by the into variable.
;;;    Arguments:
;;;    obj-id: object identifier
;;;    into (key): list to which to append new triples
;;;    Return:
;;;    the list of triples.
;;;
;;; INSERT-BEFORE (val val-l &rest val-test)                         [FUNCTION]
```

```
;;;    Should be a primitive.
;;;         Inserts item into a list either at the end of the list, or in
;;;    front of the ~ specified test-item, possibly duplicating it.
;;;    Arguments:
;;;    val: value to insert
;;;    val-l: list into which to insert the value
;;;    val-test (&rest): value that will be checked for insertion (equal
;;;    test) Return:
;;;       modified list or list of the value when val-l was nil
;;;
;;; INSERT-NTH (val val-l nth)                                    [FUNCTION]
;;;    Should be a primitive.
;;;         Inserts an item into a list at the nth position. If the list is
;;;    too short ~ then the value is inserted at the end of it.
;;;    Arguments:
;;;    val: value to be inserted
;;;    val-l: list into which to insert the value
;;;    nth: position at which the value must be inserted (0 is in front)
;;;    Return:
;;;    modified list
;;;
;;; %%MERGE-OBJECTS (obj-id obj-list context)                     [FUNCTION]
;;;    obj-id is the id of an object already existing in memory when we are
;;;    loading a new object with same id from disk. The new format of the
;;;    loaded object is obj-list. We compare properties of the existing
;;;    (moss) object and the loaded object and update all properties that
;;;    are not system or that contain application references. For a tp, the
;;;    system object is the reference. The result is a merged object
;;;    including system and application data. Arguments:
;;;    obj-id: id of the object to be processed
;;;    obj-list: object saved by the application
;;;    Returns:
;;;    the list representing the system object merged with application
;;;    references.
;;;
;;; %PUTC (obj-id value prop-id context)                          [FUNCTION]
;;;    Stores a versioned value onto the p-list of the object used as a
;;;    cache.
;;;
;;; %PUTM (object-id function-name method-name context            [FUNCTION]
;;;        &rest own-or-instance)
;;;    Records the function name corresponding to the method onto the ~
;;;         p-list of the object. Own-or-instance can be either one of the
;;;    ~ keywords :own or :instance; If not present, then :general is used.
;;;
;;; %%REMPROP (obj-id prop-id context)                            [FUNCTION]
;;;    Removes the values associated with ~
;;;         the specified property in the specified context. The net result
;;;    is ~ that the object inherits the values from a higher level.
;;;       So the resulting value in the context will probably not be nil.
;;;    Hence ~ it is not equivalent to a set-value with a nil argument.
```

```
;;;
;;; %%REMVAL (obj-id val prop-id context)                              [FUNCTION]
;;;    Removes the value associated with ~
;;;         a property in a given context. Normally useful for terminal
;;;    properties ~ However does not check for entry-points. Also val must
;;;    be a normalized value n i.e. must be expressed using the internal
;;;    format (it cannot be an external value).
;;;
;;; %%REMNTHVAL (obj-id val prop-id context position)                  [FUNCTION]
;;;    Removes the nth value associated ~
;;;         with a property in a given context. Normally useful for
;;;    terminal properties ~ However does not check for entry-points. Does
;;;    not return any significant value.
;;;
;;; %RESOLVE (id &key (context *context*))                             [FUNCTION]
;;;    Replaces id with the actual id when it is a ref object. Otherwise
;;;    returns id. When id is unbound returns id.
;;;    Arguments:
;;;    id: object id or REF identifier
;;;    Return:
;;;    resolved id.
;;;
;;; %%SET-SYMBOL-PACKAGE (symbol package)                              [FUNCTION]
;;;    if package is nil set it to default *package*, and create symbol in
;;;    the ~ specified package. No check on input.
;;;    Arguments:
;;;    symbol: any symbol
;;;    package: a valid package or keyword
;;;    Return:
;;;    a symbol interned in the package.
;;;
;;; %%SET-VALUE (obj-id value prop-id context)                         [FUNCTION]
;;;    Resets the value associated ~
;;;         with a property in a given context. Replaces previous values.
;;;    Does not ~ try to inherit using the version-graph, but defines a new
;;;    context locally ~ Thus, it is not an %add-value, but an actual
;;;    %%set-value.
;;;
;;; %%SET-VALUE-LIST (obj-id value-list prop-id context)               [FUNCTION]
;;;    Resets the value associated ~
;;;         with a property in a given context. Replaces previous values.
;;;    Does not ~ try to inherit using the version-graph, but defines a new
;;;    context locally ~ Thus, it is not an %add-value, but an actual
;;;    %%set-value - ~ No entry point is created.
;;;
```

3.5 Functions Dealing with Properties

3.5.1 Attributes

```
;;; %%ADD-VALUE (obj-id prop-id value context)                         [FUNCTION]
;;;    Inserts a new value for prop-id, no checking done on arguments
```

```
;;;    Context must be ~ explicitly given.
;;;    Arguments:
;;;    obj-id: id of object
;;;    prop-id: id of property for which we add the value
;;;    value: value to add
;;;    context: context in which we add the value
;;;    Return:
;;;    the list representing the modified object.
;;;
;;; %ADD-VALUE (obj-id prop-id value &key (context *context*)        [FUNCTION]
;;;             before-value allow-duplicates)
;;;    Inserts a new value for a given property in the specified context. ~
;;;         Copies the old values from a previous context if necessary.
;;;    We assume that prop-id is the correct local property, and that ~
;;;         the value has been validated.
;;;    Arguments:
;;;    obj-id: id of the object to be modified
;;;    prop-id: id of the target property
;;;    value: value to be added
;;;    context: context in which the value is added
;;;    before-value (key): value - specifies that we want to insert in
;;;    front of value allow-duplicates (key): t - allows to insert duplicate
;;;    values. No cardinality checking done on args. If value is there does
;;;    not do ~ anything.
;;;    Throws to an :error tag if the context is not allowed
;;;    Return:
;;;    the internal format of the modified object
;;;
;;; %VALIDATE-TP (tp-id value-list &key (context *context*))         [FUNCTION]
;;;    Checks if the property values obey the restrictions attached to the
;;;    attribute. Arguments:
;;;    tp-id: id of the attribute
;;;    value-list: list of values to check
;;;    context (opt): context (default current)
;;;    Return:
;;;    2 values:
;;;          1st value value-list if OK, nil otherwise
;;;          2nd value: list of string error message.
;;;
;;; %VALIDATE-TP-VALUE (restriction data &key prop-id obj-id         [FUNCTION]
;;;                     (context *context*))
;;;    Takes some data and checks if they agree with a value-restriction
;;;    condition,~ associated with a terminal property. Used by the =xi
;;;    method whenever ~ we want to put data as the value of the property.
;;;    Processed conditions are those of *allowed-restriction-operators*
;;;    Arguments:
;;;    restriction: restriction condition associated with the property,
;;;    e.g. (:one-of (1 2 3)) data: a list of values to be checked
;;;    prop-id (key): identifier of the local property (unused)
;;;     obj-id (key): identifier of the object being processed (unued)
;;;    context (key): context default current (unused)
```

```
;;;    Return:
;;;    data when validated, nil otherwise.
;;;
```

3.5.2 Relations

```
;;; %LINK (obj1-id sp-id obj2-id &key (context *context*))          [FUNCTION]
;;;    Brute force link two objects using the structural property id - ~
;;;         No cardinality check is done - sp-id must be direct property.
;;;       (%link obj1-id sp-id obj2-id context) - links two objects ~
;;;         using sp-id - no checking done on args - crude link to be able
;;;    ~ to bootstrap without the properties really existing. Here we
;;;    manipulate ~ the names. We assume that sp-id is not an inverse
;;;    property ~ If the link already exists then does nothing. Otherwise,
;;;    introduces a ~ new link modifying or creating a value in the
;;;    specified context. ~ This might look like a bit complicated a piece
;;;    of code, however we try ~ to do simple tests first to avoid long
;;;    execution times. Arguments:
;;;    obj1-id: identifier for first object
;;;    sp-id: identifier of the local linking property
;;;    obj2-id: identifier for the second object
;;;    context: context
;;;    Return:
;;;    list representing the first object
;;;
;;; %LINK-ALL (obj1-id sp-id obj-list &key (context *context*))     [FUNCTION]
;;;    Brute force link an object to a list of objects using the structural
;;;    property ~ id - No cardinality check is done - sp-id must be direct
;;;    property. (%link obj1-id sp-id obj-list context) - links two objects
;;;    ~ using sp-id - no checking done on args - crude link to be able ~
;;;         to bootstrap without the properties really existing. Here we
;;;    manipulate ~ the names. We assume that sp-id is not an inverse
;;;    property ~ If the link already exists then does nothing. Otherwise,
;;;    introduces a ~ new link modifying or creating a value in the
;;;    specified context. ~ This might look like a bit complicated a piece
;;;    of code, however we try ~ to do simple tests first to avoid long
;;;    execution times. Arguments:
;;;    obj1-id: identifier for first object
;;;    sp-id: identifier of the local linking property
;;;    obj-list: identifier for the second object
;;;    context: context
;;;    Return:
;;;    list representing the first object
;;;
;;; %UNLINK (obj1-id sp-id obj2-id &key (context *context*))        [FUNCTION]
;;;    Disconnects to objects by removing the structural link.
;;;         in the given context. If the context does not exist then one
;;;    has to ~ recover the states of the links in the required context.
;;;    Then if the ~ link did not exist, then one exits doing nothing.
;;;    Otherwise, one removes ~ the link and stores the result.
;;;
;;; %VALIDATE-SP (sp-id suc-list &key (context *context*))          [FUNCTION]
```

```
;;;    Checks if the property successors obey the restrictions attached to
;;;    the relation. Arguments:
;;;    sp-id: id of the relation
;;;    suc-list: list of successors to check
;;;    context (opt): context (default current)
;;;    Return:
;;;    2 values:
;;;          1st value value-list if OK, nil otherwise
;;;          2nd value: list of string error message.
;;;
```

3.5.3 Inverse Relations

3.6 Constructors

```
;;; %BUILD-METHOD (function-name arg-list body)                    [FUNCTION]
;;;    build a MOSS method from its definition in the various defxxxmethof
;;;    macros. ~ Body may have a comment (string) and/or a declare
;;;    statement. The method ~ body is enclosed in a catch :return clause to
;;;    allow exits, since we cannot ~ use the return-from clause inside the
;;;    method. When the compiler is included in the environment, compiles
;;;    the code. Arguments:
;;;    function-name: the name of the function implementing the method
;;;    (e.g. $UNI=I=0=SUMMARY) arg-list: method arguments
;;;    body: the body of the function
;;;    Return:
;;;    the function name.
;;;
;;; %CREATE-BASIC-NEW-OBJECT (object-id &key (context *context*) id  [FUNCTION]
;;;                           ideal (package *package*))
;;;    Creates a skeleton of object containing only the $TYPE and $ID ~
;;;         properties in the current context.
;;;    Arguments:
;;;    object-id: presumably class of the object to be created
;;;    context (key): context default current
;;;    id (key): if there, id is specified (we do not create it)
;;;    ideal (key): if t, we want to create an ideal instance
;;;                          (with sequence number 0)
;;;    package (key): package (default is current)
;;;    Return:
;;;    id of the newly created object.
;;;
;;; %CREATE-BASIC-ORPHAN (&key (context *context*)                  [FUNCTION]
;;;                        (package *package*))
;;;    Creates a skeleton for an orphan object and the associated key.
;;;    Arguments:
;;;    context (key): context (default current)
;;;    Return:
;;;    the id of the new orphan.
;;;
;;; %CREATE-ENTRY-POINT-METHOD (object-id object-name               [FUNCTION]
;;;                             &optional args doc body)
```

```
;;;    Creates an entry point method, with default if no arguments. ~
;;;         the =make-entry symbol is defined in the same package as
;;;    object-id. If args is present it must contain an option &key (package
;;;    *package*). Arguments:
;;;    object-id: id to which the =make-entry method will be attached
;;;    object-name: name of the object (for error message)
;;;    args (opt): arguments of the method (default is (value-list))
;;;    doc (opt): documentation string
;;;    body (opt): body of the method
;;;    Return:
;;;    an own-method id.
;;;
;;; %INVERSE-PROPERTY-ID (prop-id &key (context *context*))          [FUNCTION]
;;;    Computes the inverse property id of a given structural or terminal
;;;    property Optional arg: version. When the property is still undefined,
;;;    calls %make-id-for-inverse-property.
;;;    For inverse properties of an inverse property return a list if
;;;    more than one, ~ e.g. when we have a property lattice.
;;;    Arguments:
;;;    prop-id: identifier of property
;;;    context (opt): context default current
;;;    Returns:
;;;    an inverse property identifier
;;;
;;; %MAKE-ENTITY-SUBTREE (ens &key (context *context*))              [FUNCTION]
;;;    takes an entity and builds the subtree of its sub-classes, e.g.
;;;    (A (B C (D E) F))
;;;    where (D E) are children of C and (B C F) are children of A.
;;;    Arguments:
;;;    ens: entity to process
;;;    Return:
;;;    (ens) if no subtree, or (ens <subtree>).
;;;
;;; %MAKE-ENTITY-TREE (&key (context *context*))                     [FUNCTION]
;;;    builds the forest corresponfing to the application classes.
;;;    Arguments:
;;;    context (opt): default is *context*
;;;    Return:
;;;    a tree like (A (B C (D E) F) G H)
;;;    where (D E) are children of C and (B C F) are children of A, and G
;;;    and H are parallel trees.
;;;
;;; %MAKE-ENTITY-TREE (&key (context *context*)                      [FUNCTION]
;;;                        (package *package*))
;;;    builds the forest keeping classes from the current package.
;;;    Arguments:
;;;    context (opt): default is *context*
;;;    Return:
;;;    a tree like (A (B C (D E) F) G H)
;;;    where (D E) are children of C and (B C F) are children of A, and G
;;;    and H are parallel trees.
```

```
;;;
;;; %MAKE-ENTRY-SYMBOLS (multilingual-name &key type prefix          [FUNCTION]
;;;                      (package *package*))
;;;    Takes a multilingual-name as input and builds entry symbols for each
;;;    of the ~ synonyms in each specified language.
;;;    Note that a list of symbols is not a valid input argument.
;;;    Arguments:
;;;    multilingual-name: string, symbol or multilingual-name specifying
;;;    the entry points tp-id: local attibute identifier
;;;    type (key): type of MOSS object (default is nil)
;;;    prefix (key): prefix for produced symbols
;;;    Return:
;;;    list of the entry point symbols
;;;
;;; %MAKE-EP (entry tp-id obj-id &key (context *context*) export)    [FUNCTION]
;;;    Adds a new object to an entry-point if it exists - Otherwise creates
;;;    the entry point(s) and record it/them in the current system
;;;    (*moss-system*). Export the entry symbol(s).
;;;    Arguments:
;;;    entry: symbol or list of symbols specifying the entry point(s)
;;;    tp-id: local attibute identifier
;;;    obj-id: identifier of object to index
;;;    context (opt): context (default current)
;;;    export (key): it t export entry points from the package
;;;    Return:
;;;    internal format of the entry point object or a list of them
;;;
;;; %MAKE-EP-FROM-MLN (mln tp-id obj-id &key type                    [FUNCTION]
;;;                    (context *context*) export)
;;;    Takes a multilingual-name as input and builds entries for each of the
;;;    ~ synonyms in each specified LEGAL language in package of obj-id.
;;;    Arguments:
;;;    mln: legal multilingual-name specifying the entry points
;;;    tp-id: local attibute identifier
;;;    obj-id: identifier of object to index
;;;    context (opt): context (default current)
;;;    type (key): type of object, e.g. $ENT, $EPR, $FN, ... (see
;;;    %make-name) export (key): if t then we should export entry points
;;;    from their package Return:
;;;    list of the entry point object (internal format)
;;;
;;; %MAKE-ID (class-id &key name prefix value context id ideal       [FUNCTION]
;;;           (package *package*))
;;;    Generic function to make object identifiers. If the symbol exists,
;;;    then ~ we throw to :error.
;;;    Arguments:
;;;    class-id: identifier of the class of the object (*none* is
;;;    authorized) name (key): useful name, e.g.class  name or property name
;;;    prefix (key): additional name (string), e.g. for defining
;;;    properties (the class name) value (key): value for an instance
;;;    context (key): context default current
```

```
;;;    id (key): id of class or property
;;;    ideal (key): if there indicates we want an ideal, :id option must
;;;    be there package (key): package into which new symbol must be
;;;    inserted (default *package*) Return:
;;;    unique new symbol.
;;;
;;; %MAKE-INVERSE-PROPERTY (id multilingual-name                    [FUNCTION]
;;;                         &key (context *context*) export)
;;;    create an inverse property for an attribute or a relation. Links it
;;;    to the ~ direct property, creates the name and entry point, add it to
;;;    MOSS. It is created in the same package as id and in the specified
;;;    context. Uses the default language.
;;;    Arguments:
;;;    id: identifier of the property to invert
;;;    multilingual-name: name of the property to invert
;;;    Return:
;;;    :done
;;;
;;; %MAKE-INVERSE-PROPERTY-MLN (mln)                                [FUNCTION]
;;;    build an inverse multilingual name for the inverse property.
;;;    Arguments:
;;;    mln: multilingual name
;;;    Result:
;;;    a list of strings and an inverse mln
;;;
;;; %MAKE-NAME (class-id &rest option-list &key name                [FUNCTION]
;;;             (package *package*) &allow-other-keys)
;;;    Makes a name with %make-name-string and interns is in the specified
;;;    package. Arguments:
;;;    class-id: identifier of the corresponding class
;;;    package (key): package for interning symbol (default current
;;;    execution package) name (key): string, symbol or multilingual string
;;;     more keys are allowed as specified in the lambda-list of the
;;;    %make-string function Return:
;;;    interned symbol.
;;;
;;; %MAKE-NAME-STRING (class-id &key name method-class-id           [FUNCTION]
;;;                    method-type state-type short-name prefix
;;;                    (type "") (context *context*)
;;;                    &allow-other-keys &aux pfx)
;;;    Builds name strings for various moss entities, e.g. method names,
;;;    inverse-property ~ names, internal method function names, etc.
;;;    Examples of resulting strings
;;;        XXX-Y-ZZZ from " Xxx y'zzz "
;;;        IS-XXX-OF inverse property
;;;        HAS-XXX  property
;;;        $E-PERSON=S=0=PRINT-SELF own method internal function name
;;;        $E-PERSON=I=0=SUMMARY instance method internal function name
;;;       *0=PRINT-OBJECT universal method internal function name
;;;        $-PER typeless radix
;;;        _HAS-BROTHER  internal variable
```

```
;;;    We mix English prefix and suffix with other languages, which is
;;;    not crucial ~ since such names are internal to MOSS.
;;;    Arguments:
;;;    class-id: identifier of the corresponding class
;;;    name (key): name, e.g. of method
;;;    context (key): context reference
;;;    prefix (key): symbol, string or mln specifying a class (for class
;;;    properties) method-type (key): instance or own or universal
;;;    method-class-id (key): class-id for a given method
;;;    state-type: for state objects {:entry-state, :success, :failure}
;;;    short-name: for state objects, prefix
;;;    package (key): package for interning symbol (default current
;;;    execution package) context (key): context used for building some
;;;    names (default current) language (key): language to use, e.g. :en,
;;;    :fr (default *language*) Return:
;;;    a single name string
;;;
;;; %MAKE-PHRASE (&rest words)                                    [FUNCTION]
;;;    Takes a list of words and returns a string with the words separated
;;;    by a space. Arguments:
;;;    words (rest): strings
;;;    Return:
;;;    a string.
;;;
;;; %MAKE-RADIX-FROM-NAME (name &key (type "") (package *package*))  [FUNCTION]
;;;    build a radix name for a property or for a class. Tries to build a
;;;    unique name ~ using the first 3 letters of the name. Error if the
;;;    symbol already exists. Arguments:
;;;    name: e.g. "SIGLE"
;;;    type (opt): e.g. "T " (for terminal property)
;;;    Return:
;;;    e.g. $T-SIG or error if already exists
;;;
;;; %MAKE-REF-OBJECT (class-ref obj-id &key (context *context*))     [FUNCTION]
;;;    Builds a reference object, checking the validity of the class. If
;;;    invalid ~ throws to :error tag.
;;;    Arguments:
;;;    class-ref: reference of the new class
;;;    obj-id: id of object being referenced
;;;    context (opt): context, default current
;;;    Return:
;;;    id of new ref object.
;;;
;;; %MAKE-STRING-FROM-PATTERN (control-string pattern                [FUNCTION]
;;;                            &rest properties)
;;;    makes a string from a pattern (XML like). The first element of
;;;    pattern is the ~ class of the  object. The function ~
;;;    is somewhat equivalent to the =summary method, but the method is
;;;    applied to ~ the structured object.
;;;    Ex: (fn "~{~A~^ ~}, ~{~A~^ ~}"
;;;              '("person" ("name" "Dupond") ("first-name" "Jean"))
```

```
;;;              "name" "first-name")
;;;    returns "Dupond, Jean"
;;;    Arguments:
;;;    pattern: the pattern
;;;    data: the data
;;;    Return:
;;;    string to be printed.
;;;
;;; %MAKE-WORD-COMBINATIONS (word-list)                              [FUNCTION]
;;;
;;; %MAKE-WORD-LIST (text &key (norm t))                             [FUNCTION]
;;;    Norms a text string by separating each word making it lower case with
;;;    a leading~ uppercased letter.
;;;    Arguments:
;;;    text: text string
;;;    norm (key): if true (default) capitalize the first letter of each
;;;    word Return:
;;;    a list of normed words.
;;;
```

3.7 Extraction Functions

```
;;; %ACCESS-ENTITIES-FROM-WORD-LIST (word-list                       [FUNCTION]
;;;                                  &rest empty-word-lists)
;;;    takes a list of words, removes the empty words and tries to find
;;;    entities ~ of the KB using entry points from the list of words and
;;;    filtering the results ~ to keep only best matches.
;;;    Arguments:
;;;    word-list: input list of words
;;;    empty-word-list (key): list of empty words to remove from word
;;;    list Return:
;;;    list of object ids or nil.
;;;
;;; %EXTRACT (entry tp class &key (context *context*)                [FUNCTION]
;;;           (filter nil filter-there?)
;;;           (class-ref nil class-there?)
;;;           (package *package* package-there?)
;;;           allow-multiple-values)
;;;    Extracts an object that is specified by its entry-point - attribute -
;;;    ~ concept - and context. When more than one, applies a function when
;;;    ~ specified in the option <:filter function>. If more than one value
;;;    left ~ then an error is declared, unless they are explicitely allowed
;;;    by setting the ~ keyword argument :allow-multiple-values to T.
;;;    Arguments:
;;;    entry: entry point (string, symbol or mln)
;;;    tp: attribute (string, symbol or mln)
;;;    class: class name (string, symbol or mln)
;;;    context (key): context default current
;;;    filter (key): filter function when more than one value is expected
;;;    filter-there?: variable set to T when a filter is specified
;;;    class-ref (key): reference of class to locate a property
;;;    class-there?: variable set to T when class-ref is specified
```

```
;;;    package (key): default current
;;;    allow-multiple-values (key): if T the function may return more
;;;    than one value Returns
;;;    nil if it cannot find anything
;;;    a symbol  for a single object
;;;    a list when it finds several objects and multiples are allowed
;;;
;;; %EXTRACT-FROM-ID (entry prop-id class-id                          [FUNCTION]
;;;                   &key (context *context*))
;;;    Returns the list of all objects  corresponding to entry point entry ~
;;;         with inverse attribute of prop-id and of class class-id.
;;;       E.g. (%extract-from-id 'ATTRIBUTE '$ENAM '$ENT 0) returns ($EPT)
;;;       obj-id are returned if they are in class class-id or in one of
;;;    the subclasses ~ (transitive closure along the $IS-A.OF link).
;;;       Also all meta-models $ENT, $EPS, $EPT, $FN, $CTR ($EIL?) are
;;;    defined ~ as instances of $ENT.
;;;       If context is illegal throws to an :error tag.
;;;       The function works with the *any* and *none* pseudo classes:
;;;     (%extract-from-id <entry> <prop-id> '*any*) returns everything
;;;     (%extract-from-id <entry> <prop-id> '*none*) returns only classless
;;;    objects. We also allow objects with multiple classes.
;;;    No checks are done to speed execution.
;;;    Arguments:
;;;    entry: entry point (symbol)
;;;    prop-id: id of property to be used to make the inverse
;;;                   should be the exact property associated with class-id
;;;    class-id: id of the potential class of the object
;;;    context (key): context (default current)
;;;    Returns:
;;;    a list of objects maybe one
;;;
;;; %GET-ALL-ATTRIBUTES "()"                                          [FUNCTION]
;;;    Gets the list of all attributes in all contexts. No arguments.
;;;
;;; %%GET-ALL-CLASS-ATTRIBUTES (class-id &key (context *context*))   [FUNCTION]
;;;    Collects all attributes starting with class-id through superclasses.
;;;    Attributes of the same property tree can be included in the
;;;    result. Arguments:
;;;    class-id: identifier of the class
;;;    context (opt): context (default current)
;;;    Return:
;;;    a list of class attributes.
;;;
;;; %%GET-ALL-CLASS-INVERSE-RELATIONS (class-id                       [FUNCTION]
;;;                                    &key (context *context*))
;;;    Collects all relations starting with class-id through superclasses.
;;;    Relations of the same property tree can be included in the result.
;;;    Arguments:
;;;    class-id: identifier of the class
;;;    context (opt): context (default current)
;;;    Return:
```

```
;;;    a list of class attributes.
;;;
;;; %%GET-ALL-CLASS-PROPERTIES (class-id &key (context *context*))   [FUNCTION]
;;;    Collects all properties starting with class-id through superclasses.
;;;    Relations of the same property tree can be included in the result.
;;;    Arguments:
;;;    class-id: identifier of the class
;;;    context (opt): context (default current)
;;;    Return:
;;;    a list of class attributes.
;;;
;;; %%GET-ALL-CLASS-RELATIONS (class-id &key (context *context*))    [FUNCTION]
;;;    Collects all relations starting with class-id through superclasses.
;;;    Relations of the same property tree can be included in the result.
;;;    Arguments:
;;;    class-id: identifier of the class
;;;    context (opt): context (default current)
;;;    Return:
;;;    a list of class attributes.
;;;
;;; %GET-ALL-CONCEPTS "()"                                            [FUNCTION]
;;;    Gets the list of all concepts in all contexts. No arguments.
;;;
;;; %GET-ALL-ENTRY-POINTS "()"                                        [FUNCTION]
;;;    Gets the list of all entry-points in all contexts. No arguments.
;;;
;;; %GET-ALL-INSTANCES (class-id &key package)                        [FUNCTION]
;;;    Gets the list of all instances of a class in all contexts. We assume
;;;    that ~ counter is in context 0.
;;;    The default behavior is that the package for the instances is the
;;;    same as ~ the package of the class-id symbol. Otherwise one should
;;;    know what ~ one does...
;;;    Arguments:
;;;    class-id: id of the class
;;;    return:
;;;    list of all bound identifiers of the instances of the class
;;;
;;; %GET-ALL-INVERSE-RELATIONS "()"                                   [FUNCTION]
;;;    Gets the list of all inverse relations in all contexts. No arguments.
;;;
;;; %GET-ALL-METHODS "()"                                             [FUNCTION]
;;;    Gets the list of all methods in all contexts. No arguments.
;;;
;;; %GET-ALL-MOSS-OBJECTS "()"                                        [FUNCTION]
;;;    Gets the list of all MOSS objects in all contexts. No arguments.
;;;
;;; %GET-ALL-ORPHANS "()"                                             [FUNCTION]
;;;    Gets the list of all orphans in all contexts. No arguments.
;;;
;;; %GET-ALL-RELATIONS "()"                                           [FUNCTION]
;;;    Gets the list of all relations in all contexts. No arguments.
```

```
;;;
;;; %%GET-ALL-SUBCLASS-INVERSE-RELATIONS (class-id                    [FUNCTION]
;;;                                       &key (context *context*))
;;;    Collects all properties starting with class-id through superclasses.
;;;    Relations of the same property tree can be included in the result.
;;;    Arguments:
;;;    class-id: identifier of the class
;;;    context (opt): context (default current)
;;;    Return:
;;;    a list of class attributes.
;;;
;;; %%GET-ALL-SUBCLASS-RELATIONS (class-id                            [FUNCTION]
;;;                                &key (context *context*))
;;;    Collects all relations starting with class-id through subclasses.
;;;    Relations of the same property tree can be included in the result.
;;;    Arguments:
;;;    class-id: identifier of the class
;;;    context (opt): context (default current)
;;;    Return:
;;;    a list of class attributes.
;;;
;;; %GET-ALL-SYMBOLS "()"                                             [FUNCTION]
;;;    Gets the list of all symbols in all contexts (counters, global
;;;    variables,... No arguments.
;;;
;;; %GET-ALL-VERSIONS (obj-id prop-id)                                [FUNCTION]
;;;    Returns all versions of values or of successors associated with ~
;;;         the given property prop-id. Removes ~
;;;         duplicates of the values corresponding to the various versions.
;;;    ~ The function is used when trying to get all objects of a kind ~
;;;         from the system, e.g. by %get-all-symbols.
;;;    Arguments:
;;;    obj-id: object identifier
;;;    prop-id: property id for which we want all values
;;;    Return:
;;;    a list of all values regardless of versions.
;;;
;;; %GET-AND-INCREMENT-COUNTER (obj-id                                [FUNCTION]
;;;                             &optional (counter-link-id '$ctrs))
;;;    Obtains the value of a counter associated with obj-id and increases
;;;    this value ~ in the counter itself. Throws to :error if obj-id is
;;;    invalid. Arguments
;;;    obj-id: id of the object that owns the counter (ususally a class)
;;;    counter-link-id (opt): name of the property linking to the counter
;;;    (default $CTRS) Return:
;;;    old value of the counter.
;;;
;;; %GET-APPLICATION-CLASSES (&key (context *context*))               [FUNCTION]
;;;    returns the list of classes of a given application by extracting them
;;;    from ~ the system object and removing the system classes.
;;;    If OMAS is active, then we consider objects from the current
```

```
;;;    package. Arguments:
;;;    context (opt): default is *context*
;;;    Returns:
;;;    a list of class ids.
;;;
;;; %GETC (obj-id prop-id context)                                    [FUNCTION]
;;;    Retrieves the value associated with prop-id ~
;;;         for the specified context from the p-list of obj-id used as a
;;;    cache. ~ If the context is illegal, then throws to an :error tag.
;;;    Arguments:
;;;    obj-id: identifier of object
;;;    prop-id: identifier of local property
;;;    context: context
;;;    Returns:
;;;    nil or the cached value, Throws to :error when context is illegal
;;;    or object dead.
;;;
;;; %GET-CLASS-CANONICAL-NAME-FROM-REF (ref)                          [FUNCTION]
;;;    uses ref as an entry point to the property. If it fails return nil.
;;;    Otherwise, ~ returns the canonical name in the current language.
;;;    Arguments:
;;;    ref: e.g. "country"
;;;    Return:
;;;    canonical name in the current language specified by *language*.
;;;
;;; %GET-CLASS-ID-FROM-REF (class-ref &key (context *context*)       [FUNCTION]
;;;                           (package *package*))
;;;    Retrieves the class id from its name. Should be unique. Error
;;;    otherwise. Argument:
;;;    class-ref: a symbol, string or multilingual name, e.g. PERSON,
;;;    "Person", (:en ...) context (opt): context default current
;;;    package (opt): default current
;;;    Return:
;;;    class-id, e.g. $E-PERSON
;;;
;;; GET-CURRENT-YEAR "()"                                             [FUNCTION]
;;;
;;; %GET-DEFAULT-FROM-CLASS (prop-id class-id)                        [FUNCTION]
;;;    gets the default value-list associated with a property in a given
;;;    class. ~ The way to do it is to ask the property itself; then the
;;;    ideal of the class. Defaults cannot be inherited.
;;;    Arguments:
;;;    prop-id: id of corresponding property
;;;    class-id: id of the class
;;;    Return:
;;;    defaul value or nil.
;;;
;;; %GET-OBJECT-CLASS-NAME (obj-id)                                   [FUNCTION]
;;;    get the class names of an object.
;;;    Arguments:
;;;    obj-id: object identifier
```

```
;;;    Return:
;;;    a list of strings representing the names of the classes to which
;;;    the object belongs.
;;;
;;; %GET-EMPTY-WORDS (language)                                    [FUNCTION]
;;;    recover a list of empty words in specified language. This assumes ~
;;;             that a class empty-words exists and has been initialized.
;;;    Arguments:
;;;    language: a string or keyword
;;;    Return:
;;;    a list of empty words or nil.
;;;
;;; %GET-GENERIC-PROPERTY (prop-list &key (context *context*))     [FUNCTION]
;;;    We start with a set of properties having the same name and try to
;;;    obtain the ~ one that is generic, i.e. that does not have an ancestor
;;;    (no $IS-A property). Argument:
;;;    prop-list: list of property identifiers
;;;    Return:
;;;    the identifier of the generic property or nil. Warns if there are
;;;    more than one left.
;;;
;;; %GET-GENERIC-PROPERTY-FROM-NAME (prop-name                     [FUNCTION]
;;;                                  &key (context *context*))
;;;    We start with a property name and try to obtain the id of the ~
;;;         one that is generic, i.e. that does not have an ancestor.
;;;    Argument:
;;;    prop-name: property name
;;;    Return:
;;;    the identifier of the generic property or nil. Warns if there are
;;;    more than one left.
;;;
;;; %GET-GENERIC-PROPERTY-FROM-REF (prop-ref)                      [FUNCTION]
;;;    We start with a set of properties having the same name and try to
;;;    obtain the ~ one that is generic, i.e. that does not have an ancestor
;;;    (no $IS-A property). Argument:
;;;    prop-ref: string or symbol e.g. "content" or CONTENT
;;;    Return:
;;;    the identifier of the generic property (e.g. HAS-CONTENT) or nil.
;;;
;;; %GET-INDEX-WEIGHTS (task-id)                                   [FUNCTION]
;;;    get a list of pairs index (a string) value from the list of ~
;;;        index patterns associated with the task.
;;;    Arguments:
;;;    task: a task object
;;;    Return:
;;;    a list of pairs.
;;;
;;; %GET-INSTANCE-COUNT (class-id &key (context *context*)         [FUNCTION]
;;;                      (allow-subclasses t))
;;;    Gets an upper-bound on the number of instances for a given class.
;;;    Arguments:
```

```
;;;    class-id: identifier of class
;;;    allow-subclasses (key): if T counts instances of subclasses
;;;    Returns:
;;;    a number
;;;
;;; %%GET-LIST-OF-VALUES-AND-CONTEXT (obj-id prop-id)                [FUNCTION]
;;;    Done to obtain raw value list of objects.
;;;    Arguments:
;;;    obj-id: identifier of object
;;;    prop-id: property local identifier
;;;    Returns:
;;;    a list of all values by context (internal format of the set of
;;;    values).
;;;
;;; %GETM (object-id method-name context &rest own-or-instance)      [FUNCTION]
;;;    When methods are compiled on p-list it is done as follows:
;;;     (=print-self (:own (nn <int-function name>)*) (:instance  <...>))
;;;       if the method type is not specified then the format is slightly ~
;;;         different from the more general format.
;;;       The version must be recorded locally otherwise we shall try ~
;;;         to inherit it. Remember this is a cache, not an object.
;;;       When using versions we can have the method cached onto the p-list
;;;    for a ~ previous version. If the method has not been modified since,
;;;    it is a waste ~ of energy to recover it again, to recompile it, and
;;;    to give it a new name. ~ Rather, we want to check that it was not
;;;    modified along the same branch indeed, ~ then we cache it under the
;;;    current version explicitely with the old internal ~ name. In order to
;;;    do that, we must record every time a method code or arguments ~ was
;;;    changed. Doing
;;;           (%get-value <method-id> <modif> *current-context*)
;;;       will return the last modification or nil (it could have been
;;;    modified, but ~ then it is not in the same branch.
;;;       So for recovering a cached method, we first get the current
;;;    branch of the ~ path from the current context to the root of the
;;;    version graph. We then loop on the successive values of the versions
;;;    in the branch starting ~ with the current one. At each step, either
;;;    we get a function, or this is the ~ step at which the version was
;;;    modified, in which case we loose and we must ~ recompute and
;;;    recompile the method. Otherwise, when we get a previously ~ recorded
;;;    method, we must cache it explicitely on the p-list for next time
;;;    around. Arguments: object-id: id of the object
;;;    method-name: name of the target method
;;;    context: context
;;;    own-or-instance (key): flag to specify type of method
;;;    Return:
;;;    method or nil
;;;
;;; %GET-NAME-FROM-REF (ref object-type)                             [FUNCTION]
;;;    Takes a reference and tries to obtain the official name for the
;;;    object, e.g. ~ "class" -> CONCEPT
;;;    If ref is a symbol, we assume it is the name we are looking for,
```

```
;;;       if ref is a multilingual name, we use the canonical name,
;;;       if ref is a string, we build a name from it, specific to the
;;;    object type, ~ and recover the canonical name from its recorded name.
;;;    if the object is a property yet undefined, we return the cooked up
;;;    name. Arguments:
;;;    ref: a symbol, string or multilingual name
;;;    object-type: a keyword giving the type of the object (:class,
;;;    :attribute, ...) package (opt): a package spec for the name
;;;    Return:
;;;    the canonical name of the object.
;;;
;;; %%GET-OBJECTS-FROM-ENTRY-POINT (entry &key (context *context*)   [FUNCTION]
;;;                                  keep-entry)
;;;    Sets up an a-list (inv-prop obj-id) of all objects for which entry is
;;;    an ~ entry point, eliminating the current-system (inv prop:$EPLS.OF).
;;;    Arguments:
;;;    entry: entry-point
;;;    context (key): context default current
;;;    keep-entry (key): if t keep the entry point
;;;    Return:
;;;    a list of pairs (<inv-prop-id> . <obj-id>)  or nil
;;;
;;; %GET-PROP-ID-FROM-NAME-AND-CLASS-REF (prop-name class-ref        [FUNCTION]
;;;                                       &key (context *context*)
;;;                                       (package *package*))
;;;    Determines the right property for the specific class. Throws to
;;;    :error when ~ something is wrong. Property may be inherited.
;;;    Arguments:
;;;    prop-name: name of a property
;;;    class-ref: class name or class identifier
;;;    context (opt): context default current
;;;    package (opt): defualt current
;;;    Return:
;;;    the property attached to the class.
;;;
;;; %GET-PROPERTIES (object-id &key (context *context*))             [FUNCTION]
;;;    Obtain all the possible properties for a given object. If an orphan,
;;;    then ~ we only get local properties attached to the orphan.
;;;    Otherwise, we get ~ all inherited properties corresponding to the
;;;    object class. Arguments:
;;;    object-id: identifier of object
;;;    context (opt): context (default current)
;;;    Return:
;;;    a list of properties associated with the object.
;;;
;;; %GET-PROPERTY-CANONICAL-NAME-FROM-REF (ref)                      [FUNCTION]
;;;    uses ref as an entry point to the property. If it fails return nil.
;;;    Otherwise, ~ returns the canonical name in the current language.
;;;    Arguments:
;;;    ref: e.g. "country"
;;;    Return:
```

```
;;;    canonical name in the current language specified by *language*.
;;;
;;; %GET-PROPERTY-ID-FROM-NAME (obj-id prop-ref                        [FUNCTION]
;;;                             &key (context *context*)
;;;                             (package *package*))
;;;    Function that has the difficult role of finding the right property
;;;    that ~ applies to the object, and corresponds to the prop-name. This
;;;    is ~ difficult in case we have tree-properties.
;;;    It is used by elementary universal methods like =get =put =delete
;;;    Result is cached onto the p-list of the property name (e.g.
;;;    HAS-NAME). Arguments:
;;;    obj-id: object for which we try to determine property
;;;    prop-ref: property ref, e.g. HAS-FIRST-NAME, "first name", or mln
;;;    context (opt): context default current
;;;    Return:
;;;    the identifier of the corresponding property or nil if the class
;;;    inherits property.
;;;
;;; %GET-PROPERTY-SYNONYMS (prop-ref &key (package *package*))       [FUNCTION]
;;;    takes a property reference and return the list of synonyms for this
;;;    property. Arguments:
;;;    prop-ref: a property reference, e.g. "street"
;;;    package (key): package in which property name has been defined
;;;    Return;
;;;    nil or the list of all property synonyms.
;;;
;;; %GET-RELEVANT-WEIGHTS (weight-list word-patterns                   [FUNCTION]
;;;                        &key used-patterns)
;;;    function to obtain weigths for a list of words.
;;;    Arguments:
;;;    weight-list: list of weights e.g. ((<string> .6))
;;;    word-patterns: the list of expressions (strings) to test
;;;    used-patterns (key): expressions that have already been tried
;;;    Return:
;;;    a list of pairs string weight.
;;;
;;; %GET-SP-ID-FROM-NAME-AND-CLASS-REF (prop-name class-ref            [FUNCTION]
;;;                                     &key (context *context*)
;;;                                     (package *package*))
;;;    Determines the right relation for the specific class. Throws to
;;;    :error when ~ something is wrong.
;;;    Arguments:
;;;    prop-name: name of a property
;;;    class-ref: class name or class identifier
;;;    context (opt): context (default current)
;;;    Return:
;;;    the property attached to the class.
;;;
;;; %GET-SP-VALUE-RESTRICTIONS (prop-id &key (context *context*))     [FUNCTION]
;;;    Obtains the value restrictions from the attribute representation. No
;;;    ~ inheritance is done.
```

```
;;;    Restrictions are
;;;       :exists     ; among the possible values one must be of that type
;;;       :forall     ; all values should be of that type
;;;       :type       ; values should be of that type
;;;       :not-type   ; no value should be of that type
;;;       :value      ; only this value is allowed
;;;       :one-of     ; the value should be a member of this list
;;;       :not        ; the value should not be one of this list
;;;    Arguments:
;;;    prop-id: attribute id
;;;    context (opt): context (default current)
;;;    Returns:
;;;    a list of restrictions: e.g. ((:one-of 1 2 3) (:type :integer)
;;;    (:one)).
;;;
;;; %GET-SUBCLASS-NAMES (class-ref)                                    [FUNCTION]
;;;    takes a class reference and return the list of synonyms for all
;;;    subclasses, ~ including the referenced class.
;;;    Arguments:
;;;    class-ref: a class reference, e.g. "address"
;;;    Return;
;;;    nil or the list of all subclass names.
;;;
;;; %GET-SUBCLASSES (obj-id)                                           [FUNCTION]
;;;    returns the list of subclasses of obj-id.
;;;    Argument:
;;;    obj-id: a class identifier
;;;    Return:
;;;    nil or the transitive closure along the inverse $IS-A property.
;;;
;;; %GET-TP-ID-FROM-NAME-AND-CLASS-REF (prop-name class-ref          [FUNCTION]
;;;                                     &key (context *context*))
;;;    Determines the right attribute for the specific class. Throws to
;;;    :error when ~ something is wrong. When class is *none* (orphans) all
;;;    attributes are possible. Arguments:
;;;    prop-name: name of a property
;;;    class-ref: class name or class identifier
;;;    context (opt): context default current
;;;    Return:
;;;    the attribute attached to the class or nil.
;;;
;;; %GET-TP-VALUE-RESTRICTIONS (prop-id &key (context *context*))    [FUNCTION]
;;;    Obtains the value restrictions from the attribute representation. No
;;;    ~ inheritance is done.
;;;    Restrictions are
;;;       :exists     ; among the possible values one must be of that type
;;;       :forall     ; all values should be of that type
;;;       :type       ; values should be of that type
;;;       :not-type   ; no value should be of that type
;;;       :value      ; only this value is allowed
;;;       :one-of     ; the value should be a member of this list
```

```
;;;       :not        ; the value should not be one of this list
;;;       :between    ; the numerical value should be in that range
;;;       :outside    ; the numerical value should be outside that range
;;;    Arguments:
;;;    prop-id: attribute id
;;;    context (opt): context (default current)
;;;    Returns:
;;;    a list of restrictions: e.g. ((:one-of 1 2 3) (:type :integer)
;;;    (:one)).
;;;
;;; %GET-VALUE (obj-id prop-id &key (context *context*))             [FUNCTION]
;;;    Gets the value corresponding to the specified context. To do so uses
;;;    ~ the *version-graph* to locate within a branch the latest valid
;;;    value. ~ The context graph is organized as an a-list of node, each
;;;    node points to ~ its predecessors: ((9 6)(8 3 2)(7 6)...(0)) it
;;;    contains all contexts, ~ context numbers are not necessarily
;;;    increasing along a branch, e.g. if we have ~ fused some contexts.
;;;       Algorithm: (i) gets the branch from the current context, if
;;;    legal, to the root. ~ (the branch is computed only once for a given
;;;    context and cached onto the ~ p-list of the global symbol
;;;    *version-graph*). ~ (ii) goes up in the branch, returning the first
;;;    non nil value associated with ~ a context in the object.
;;;    The exact property of the object must be used, not the generic
;;;    one. %get-value does not return defaults.
;;;       If context is illegal throws to an :error tag.
;;;    Arguments:
;;;    obj-id: identifier of object
;;;    prop-id: identifier of  property
;;;    context: context
;;;
;;; %%GET-VALUE (obj-id prop-id context)                             [FUNCTION]
;;;    Returns the value associated with a particular context using the
;;;    *version- ~ graph*. If context is illegal throws to an :error tag.
;;;       Like %get-value but does not require the object to have a
;;;    property $TYPE. object-id must not be a REF object, but a regular
;;;    object. Arguments:
;;;    obj-id: identifier of object
;;;    prop-id: identifier of local property
;;;    context: context
;;;    Return: the value associated with prop-id.
;;;
;;; %%GET-VALUE-AND-CONTEXT (obj-id prop-id)                         [FUNCTION]
;;;    Done to obtain value for objects which must not be versioned, like
;;;    counters ~ for entities.
;;;    Arguments:
;;;    obj-id: object identifier
;;;    prop-id: local property identifier.
;;;    Retrurn:
;;;    a list whose car is context and cadr is the value.
;;;
;;; %GET-VALUE-FROM-CLASS-REF (class-ref expr)                       [FUNCTION]
```

```
;;;    returns the value associated with class-ref from a MOSS pattern.
;;;    Arguments:
;;;    class-ref: e.g. "address"
;;;    Returns:
;;;    value or nil.
;;;
;;; %GET-VALUE-FROM-PROP-REF (prop-ref expr                          [FUNCTION]
;;;                           &key (package *package*))
;;;    returns the value associated with prop-ref from a MOSS pattern.
;;;    Arguments:
;;;    prop-ref: e.g. "town"
;;;    package (key): package in which property has been defined
;;;    Returns:
;;;    value or nil.
;;;
;;; %GET-WORDS-FROM-TEXT (text delimiters)                           [FUNCTION]
;;;    takes an input string and a set of delimiters and returns a list of
;;;    words ~ separated by delimiters.
;;;    Arguments:
;;;    text: a string
;;;    delimiters: a set of characters that delimit a word
;;;    Return:
;;;    a list of words.
;;;
;;; %HAS-ID-LIST (entry inv-id class-id &key (context *context*))    [FUNCTION]
;;;    Returns the list of all objects ~
;;;         corresponding to entry point entry with inverse terminal
;;;    property inv-id ~ and of class class-id.
;;;       E.g. (%has-id-list 'ATTRIBUTE '$PNAM.OF '$ENT) returns ($EPT) ~
;;;         class-id must be specified locally; i.e., will not return
;;;    objects ~ that are instances of subclasses of class-id.
;;;       If context is illegal throws to an :error tag
;;;    Arguments:
;;;    entry: entry symbol
;;;    inv-id: local inverse attribute
;;;    class-id: identifier of target class
;;;    context: context
;;;    Return:
;;;    a list of objects.
;;;
;;; %%HAS-INVERSE-PROPERTIES (obj-id context)                        [FUNCTION]
;;;    Returns the list of inverse properties local to an object and having
;;;    ~ associated values in the specified context
;;;    Arguments:
;;;    obj-id: object identifier
;;;    context: context
;;;    Return:
;;;    a list of properties or nil.
;;;
;;; %%HAS-PROPERTIES (obj-id context)                                [FUNCTION]
;;;    Returns the list of properties local to an object and having
```

```
;;;    associated ~ values in the specified context, looking directly at the
;;;    internal format ~ of the object. $TYPE is removed.
;;;    Arguments:
;;;    obj-id: object identifier
;;;    context: context
;;;    Return:
;;;    a list of properties or nil.
;;;
;;; %%HAS-PROPERTIES+ (obj-id context)                                [FUNCTION]
;;;    Returns the list of ALL properties LOCAL to an object even with no ~
;;;         value, in the specified context, looking directly at the
;;;    internal format ~ of the object.
;;;    Arguments:
;;;    obj-id: object identifier
;;;    context: context
;;;    Return:
;;;    a list of properties or nil.
;;;
;;; %%HAS-STRUCTURAL-PROPERTIES (obj-id context)                      [FUNCTION]
;;;    Returns the list of structural properties local to an object and
;;;    having  ~ associated values in the specified context. $TYPE and $ID
;;;    are removed. Arguments:
;;;    obj-id: object identifier
;;;    context: context
;;;    Return:
;;;    a list of properties or nil.
;;;
;;; %%HAS-TERMINAL-PROPERTIES (obj-id context)                        [FUNCTION]
;;;    Returns the list of terminal properties local to an object and having
;;;    ~ associated values in the specified context.
;;;    Arguments:
;;;    obj-id: object identifier
;;;    context: context
;;;    Return:
;;;    a list of properties or nil.
;;;
;;; %%HAS-VALUE (obj-id prop-id context &key no-resolve)              [FUNCTION]
;;;    Returns the value list associated to a particular context if recorded
;;;    locally. No checks are done, except for legal context.
;;;    Arguments:
;;;    obj-id: object or ref to object under consideration (can be dead)
;;;    prop-id: property (may be local or inherited)
;;;    context: context (default current)
;;;    no-resolve (key): if true, skip the call to resolve intended to
;;;    solve multiclass belonging
;;;    Return:
;;;    the local value if any.
;;;
```

3.8 Functions for Filtering

```
;;; %BUT-MOSS (item-list)                                             [FUNCTION]
```

```
;;;    Removes from a list of symbols any symbol that is not a PDM object or
;;;    that belongs ~ to the MOSS system. Thus we should only get PDM
;;;    application objects. Arguments:
;;;    item-list: a list of symbols
;;;    Return:
;;;    a list of application objects.
;;;
;;; %DETERMINE-PROPERTY-ID (obj-id prop-name                          [FUNCTION]
;;;                         &key (context *context*))
;;;    For the object with id obj-id, attempts to determine which of its ~
;;;         properties corresponds  to prop-name in the specified context.
;;;    If the property is present at the object level, then return it,
;;;    otherwise: - if the object has a prototype, try the prototype
;;;         - if the object has a class, look at the class level for
;;;    properties ~ defined at the class level.
;;;         - if the object belongs to several classes try each one in
;;;    turn. Problems are:
;;;    - tree properties: properties are in a tree (normally each node of
;;;    the ~ tree should have the same property name)
;;;    - lattices of prototypes, where a property could be derived from
;;;    another ~ one thru an IS-A relation (not implemented).
;;;    Arguments:
;;;    obj-id: identifier of the object
;;;    prop-name: name of property to check: e.g. HAS-NAME (must be
;;;    valid, no check) context: context
;;;    Returns:
;;;    a list of normally a single property-id.
;;;
;;; %DETERMINE-PROPERTY-ID-FOR-CLASS (prop-list class-id              [FUNCTION]
;;;                                   &key (context *context*)
;;;                                   inverse)
;;;    Determines the right prop-id for the specified class among a list of
;;;    properties. No checks on arguments. If property is inherited, then we
;;;    return the closest one. If class is *any* returns the list.
;;;    Arguments:
;;;    prop-list: list of property identifiers, e.g. $T-NAME,
;;;    $S-PERSON-HUSBAND.OF class-id: identifier of the class, e.g.
;;;    $E-PERSON (*any* is allowed) context (opt): context default current
;;;    inverse (key): specify that we look for inverse property
;;;    Returns:
;;;    nil or list of property-id
;;;
;;; %FILTER-ALIVE-OBJECTS (object-list &key (context *context*))      [FUNCTION]
;;;    Takes a list of objects and keeps the ones that are alived in the
;;;    specified ~ context.
;;;    Arguments:
;;;    object-list: list of objects
;;;    context (opt): context (default current)
;;;    Return:
;;;    a (possibly empty) list of objects alive in context.
;;;
```

```
;;; %REMOVE-REDUNDANT-PROPERTIES (prop-list                          [FUNCTION]
;;;                               &key (context *context*))
;;;    Removes properties that share the same generic property and keep the
;;;    leftmost ~ one.
;;;    Arguments:
;;;    prop-list: list of property ids
;;;    context (opt): context (default current)
;;;    Return:
;;;    a cleaned list of properties.
;;;
;;; %SELECT-BEST-ENTITIES (entity-score-list)                        [FUNCTION]
;;;    select entities with highest score and return the list.
;;;    Arguments:
;;;    entity-score-list: a list like (($E-PERSON.2 0.5)(..)...)
;;;    Returns:
;;;    the list of entities with the highest score.
;;;
```

3.9 Functions for Printing

```
;;; %PEP (obj-id &key (version *context*) (offset 30) (stream t))    [FUNCTION]
;;;    Prints versions of values from a given context up to the root.
;;;       (%pep obj-id version &rest offset)
;;;       For each property prints values in all previous context up to the
;;;    root ~ does not check for illegal context since it is a printing
;;;    function.
;;;
;;; %PEP-PV (value context-branch &optional (stream t) offset)       [FUNCTION]
;;;    Used by %pep, prints ~
;;;         a set of value associated with a property from the root to
;;;    current context. Offset is currently set to 30.
;;;
;;; %PFORMAT (fstring input &rest prop-list)                         [FUNCTION]
;;;    produces a string for printing from a list produced typically by the
;;;    ~ =make-print-string method.
;;;    Arguments:
;;;    input: a-list, e.g. (("name" "Dupond")("first name" "Jean")...)
;;;    fstring: string control format
;;;    prop-list: a list of properties appearing in the a-list. last
;;;    entry can be the specification of a particular package
;;;    Return:
;;;    a string: e.g. "Dupond, Jean"
;;;
```

3.10 Functions Dealing with Multilingual Names

```
;;; %MLN? (expr)                                                     [FUNCTION]
;;;    Checks whether a list is a multilingual name Uses global
;;;    *language-tags* variable. ~ nil is not a valid MLN.
;;;    Argument:
;;;    expr: something like (:en "London" :fr "Londres") or (:name :en
;;;    "London") Result:
```

```
;;;    T if OK, nil otherwise
;;;
;;; %MLN-1? (expr)                                                    [FUNCTION]
;;;    Checks whether a list is a multilingual string. Uses global
;;;    *language-tags* variable. Argument:
;;;    expr: something like (:en "London" :fr "Londres")
;;;    Result:
;;;    T if OK, nil otherwise
;;;
;;; %MLN-ADD-VALUE (mln value language-tag)                           [FUNCTION]
;;;    Adds a value corresponding to a specific language at the end of the
;;;    list. ~ Does not check if value is already there.
;;;    Arguments:
;;;    mln: multilingual name
;;;    value: coerced to a string
;;;    language-tag: must be legal, i.e. part of *language-tags*
;;;    Return:
;;;    the modified mln.
;;;
;;; %MLN-EQUAL (mln1 mln2 &key language)                              [FUNCTION]
;;;    Checks whether two multilingual-names are equal. They may also be
;;;    strings. ~ When the language key argument is not there uses global
;;;    *language* variable.~ When *language* is unbound, tries all possible
;;;    languages. The presence of ~ language is only necessary if one of the
;;;    arguments is a string. The function uses %string-norm before
;;;    comparing strings. Argument:
;;;    mln1: a multilingual name or a string
;;;    mln2: id.
;;;    language (key): a specific language tag
;;;    Result:
;;;    T if OK, nil otherwise
;;;
;;; %MLN-EXTRACT-ALL-SYNONYMS (mln)                                   [FUNCTION]
;;;    Extracts all synonyms as a list of strings regardless of the
;;;    language. Arguments:
;;;    mln: multilingual name
;;;    Return
;;;    a list of strings.
;;;
;;; %MLN-EXTRACT-MLN-FROM-REF (ref language                           [FUNCTION]
;;;                            &key no-throw-on-error (default :en)
;;;                            &aux $$$)
;;;    takes a ref input symbol string or mln and builds a specific mln
;;;    restricted to ~ the specified language. If language is :all and ref
;;;    is a string or symbol, then ~ language defaults to default. If ref is
;;;    an MLN, then language defaults to default ~ if there, first found
;;;    language otherwise (canonical behavior). Default beavior ~ is enabled
;;;    only if no-throw-on-error option is set to T. Removes also the
;;;    leading :name marker if present in mln. Arguments:
;;;    ref: a string symbol or mln
;;;    language a language tag or :all
```

```
;;;    no-throw (key): if true function does not throw on error but
;;;    returns nil default (key): default language tag (default: :EN)
;;;    Result:
;;;    a valid mln or throws to :error
;;;
;;; %MLN-FILTER-LANGUAGE (mln language-tag &key always)              [FUNCTION]
;;;    Extracts from the MLN the synonym string corresponding to specified
;;;    language. ~ If mln is a string or language is :all, then returns mln
;;;    untouched, unless always ~ is true in which case returns the English
;;;    terms if there, otherwise random terms. If language is not legal,
;;;    throws to :error. Arguments:
;;;    mln: a multilingual name (can also be a simple string)
;;;    language-tag: a legal language tag (part of *language-tags*)
;;;    always (key): if t then returns either the English terms or a
;;;    random one Return:
;;;    a string or nil if none.
;;;
;;; %MLN-GET-CANONICAL-NAME (mln &aux names)                          [FUNCTION]
;;;    Extracts from a multilingual name the canonical name that will be
;;;    used to build ~ an object ID. By default it is the first name
;;;    corresponding to the value of *language*, ~ or else the first name of
;;;    the English entry, or else the name of the list. ~ An error occurs
;;;    when the argument is not multilingual name. Arguments:
;;;    mln: a multilingual name
;;;    Return:
;;;    2 values: a simple string (presumably for building an ID) and the
;;;    language tag. Error:
;;;    error if not a multilingual name.
;;;
;;; %MLN-GET-FIRST-NAME (name-string)                                 [FUNCTION]
;;;    Extracts the first name from the name string. Names are separated by
;;;    semi-columns. E.g. "identity card ; papers" returns "identity card "
;;;    Argument:
;;;    name-string
;;;    Return:
;;;    string with the first name.
;;;
;;; %MLN-MEMBER (input-string tag mln)                               [FUNCTION]
;;;    checks whether the input string is one of the synonyms of the mln in
;;;    the ~ language specified by tag. If tag is :all then we check against
;;;    any synonym ~ in any language.
;;;    Arguments:
;;;    input-string: input string (to be normed with %norm-string)
;;;    tag; a legal language tag or :all
;;;    mln: a mltilingual name
;;;    Return:
;;;    non nil value if true.
;;;
;;; %MLN-MERGE (&rest mln)                                            [FUNCTION]
;;;    Takes a list of MLN and produces a single MLN as a result, merging
;;;    the ~ synonyms (using or norm-string comparison) and keeping the
```

```
;;;    order of the ~ inputs.
;;;    Arguments:
;;;    mln: a multilingual name
;;;    more-mln (rest): more multilingual names
;;;    Return:
;;;    a single multilingual name
;;;    Error:
;;;    in case one of the mln has a bad format.
;;;
;;; %MLN-NORM (expr)                                                [FUNCTION]
;;;    Norms an mln expr by removing the :name keyword if there.
;;;    Argument:
;;;    expr: expression to test
;;;    Return:
;;;    normed expr, nil is an error)
;;;
;;; %MLN-PRINT-STRING (mln &optional (language-tag :all))           [FUNCTION]
;;;    Prints a multilingual name nicely.
;;;    Arguments:
;;;    mln: multilingual name
;;;    language-tag: legal language-tag (default :all)
;;;    Result:
;;;    a string ready to be printed (can be empty).
;;;
;;; %MLN-REMOVE-LANGUAGE (mln language-tag)                         [FUNCTION]
;;;    Removes the entry corresponding to a language from a multilingual
;;;    name. Arguments:
;;;    mln: multilanguage name
;;;    language-tag: language to remove (must be legal
;;;    Return:
;;;    modified mln.
;;;
;;; %MLN-REMOVE-VALUE (mln value language-tag)                      [FUNCTION]
;;;    Removes a value in the list of synonyms of a particular language.
;;;    Arguments:
;;;    mln: multilingual name
;;;    value: coerced to a string (loose package context)
;;;    language-tag: must be legal
;;;    Return:
;;;    the modified mln.
;;;
;;; %MLN-SET-VALUE (mln language-tag syn-string)                    [FUNCTION]
;;;    Sets the language part to the specified value, erasing the old value.
;;;    Arguments:
;;;    mln: a multilingual name
;;;    language-tag: language tag
;;;    syn-string: coerced to a string (loose package context)
;;;    Return:
;;;    the modified normed MLN.
;;;
;;; %MULTILINGUAL-NAME? (expr)                                      [FUNCTION]
```

```
;;;    Checks whether a list is a multilingual string. Uses global
;;;    *language-tags* variable. Argument:
;;;    expr: something like (:en "London" :fr "Londres")
;;;    Result:
;;;    T if OK, nil otherwise.
;;;
```

3.11 Functions Dealing with Strings and Synonyms

```
;;; STR-EQUAL (aa bb)                                              [FUNCTION]
;;;
;;; %STRING-NORM (input &optional (interchar #\-))                 [FUNCTION]
;;;    Same as make-value-string but with more explicit name.
;;;    If input is a multilingual string, uses the canonical name.
;;;    If input is NIL throws to :error
;;;    Arguments:
;;;    input: may be a string, a symbol or a multilingual string
;;;    interchar: character to hyphenate the final string (defaul is #-
;;;    Return:
;;;    a normed string, e.g. "AU-JOUR-D-AUJOURD-HUI ".
;;;
;;; %SYNONYM-ADD (syn-string value)                                [FUNCTION]
;;;    Adds a value to a string at the end.
;;;    Sends a warning if language tag does not exist and does not add
;;;    value. Arguments:
;;;    syn-string: a synonym string or nil
;;;    value: value to be included (coerced to string)
;;;    Return:
;;;    modified string when no error, original string otherwise.
;;;
;;; %SYNONYM-EXPLODE (text)                                        [FUNCTION]
;;;    Takes a string, consider it as a synonym string and extract items
;;;    separated ~ by a semi-column. Returns the list of string items.
;;;    Arguments:
;;;    text: a string
;;;    Return
;;;    a list of strings.
;;;
;;; %SYNONYM-MAKE (&rest item-list)                                [FUNCTION]
;;;    Builds a synonym string with a list of items . Eadh item is coerced
;;;    to a string. Arguments:
;;;    item-list: a list of items
;;;    Return:
;;;    a synonym string.
;;;
;;; %SYNONYM-MEMBER (value text)                                   [FUNCTION]
;;;    Checks if a string is part of the synonym list. Uses the %string-norm
;;;    ~ function to normalize the strings before comparing.
;;;    Arguments:
;;;    value: a value coerced to a string
;;;    text: a synonym string
;;;    Return:
```

```
;;;    a list of unnormed remaining synonyms if succcess, NIL if nothing
;;;    left.
;;;
;;; %SYNONYM-MERGE-STRINGS (&rest names)                          [FUNCTION]
;;;    Merges several synonym string, removing duplicates (using a
;;;    norm-string ~ comparison) and preserving the order.
;;;    Arguments:
;;;    name: a possible complex string
;;;    more-names (rest): more of that
;;;    Return:
;;;    a single complex string
;;;    Error:
;;;    when some of the aguments are not strings.
;;;
;;; %SYNONYM-REMOVE (value text)                                  [FUNCTION]
;;;    Removes a synonym from the list of synonyms. If nothing is left
;;;    return ~ empty string.
;;;    Arguments:
;;;    value: a value coerced to a string
;;;    text: a synonym string
;;;    Return:
;;;    the string with the value removed NIL if nothing left.
;;;
```

3.12 Miscellaneous

```
;;; %CLEAN-WORD-LIST (word-list &rest empty-word-lists)           [FUNCTION]
;;;    remove empty words from the list using empty-word lists.
;;;    Arguments:
;;;    word-list: list of words to be cleaned
;;;    empty-word-lists (rest): lists of empty words
;;;    Returns:
;;;    the cleaned list.
;;;
;;; %INTERSECT-SYMBOL-LISTS (l11 l12)                             [FUNCTION]
;;;    Intersect two lists of symbols by using their p-list. Done in linear
;;;    time.
;;;
;;; %RANK-ENTITIES-WRT-WORD-LIST (entity-list word-list)          [FUNCTION]
;;;    takes a list of entity ids and a list of words, and returns a score
;;;    for ~ each entity that is function of the number of apparitions of
;;;    the words in ~ attributes of the entity.
;;;    Arguments:
;;;    entity-list: a list of object ids (ei)
;;;    word-list: a list of words (wordj)
;;;    Returns:
;;;    a list of pairs (ei wi), e.g. (($e-person.2 0.75)($E-PERSON.3
;;;    0.5)).
;;;
```

4 Service Functions by Alphabetical Order

```
;;;
;;; %ACCESS-ENTITIES-FROM-WORD-LIST (word-list                          [FUNCTION]
;;;                                  &rest empty-word-lists)
;;;    takes a list of words, removes the empty words and tries to find
;;;    entities ~ of the KB using entry points from the list of words and
;;;    filtering the results ~ to keep only best matches.
;;;    Arguments:
;;;    word-list: input list of words
;;;    empty-word-list (key): list of empty words to remove from word
;;;    list Return:
;;;    list of object ids or nil.
;;;
;;; %%ADD-VALUE (obj-id prop-id value context)                          [FUNCTION]
;;;    Inserts a new value for prop-id, no checking done on arguments
;;;    Context must be ~ explicitly given.
;;;    Arguments:
;;;    obj-id: id of object
;;;    prop-id: id of property for which we add the value
;;;    value: value to add
;;;    context: context in which we add the value
;;;    Return:
;;;    the list representing the modified object.
;;;
;;; %ADD-VALUE (obj-id prop-id value &key (context *context*)          [FUNCTION]
;;;             before-value allow-duplicates)
;;;    Inserts a new value for a given property in the specified context. ~
;;;         Copies the old values from a previous context if necessary.
;;;    We assume that prop-id is the correct local property, and that ~
;;;         the value has been validated.
;;;    Arguments:
;;;    obj-id: id of the object to be modified
;;;    prop-id: id of the target property
;;;    value: value to be added
;;;    context: context in which the value is added
;;;    before-value (key): value - specifies that we want to insert in
;;;    front of value allow-duplicates (key): t - allows to insert duplicate
;;;    values. No cardinality checking done on args. If value is there does
;;;    not do ~ anything.
;;;    Throws to an :error tag if the context is not allowed
;;;    Return:
;;;    the internal format of the modified object
;;;
;;; %%ALIVE? (obj-id context)                                           [FUNCTION]
;;;    Checks if object is PDM, *context* is OK, and object is alive. If
;;;    not, throws ~ to error.
;;;    Arguments:
;;;    obj-id: id
;;;    context (opt): context (default current)
;;;    Return:
```

```
;;;    resoved id, if OK, throws to :error otherwise.
;;;
;;; %ALIVE? (obj-id &key (context *context*))                    [FUNCTION]
;;;    Checks if object exists in given context ~
;;;         each object has a tombstone which records if it has been killed
;;;    ~ by somebody in the given context. This is a source of inconsistency
;;;    ~ since if the tombstone exists then the properties must be removed
;;;    in ~ current context. I.e. one cannot have a link onto an object
;;;    which ~ has been killed.
;;;    If context is illegal throws to :error.
;;;    Arguments:
;;;    obj-id: identifier of the object to be checked
;;;    context (opt): context (default current context)
;;;    Return:
;;;    nil if object is dead, resolved id otherwise
;;;
;;; %%ALLOWED-CONTEXT? (context)                                 [FUNCTION]
;;;    Checks if the context is allowed. Does this by looking at the global
;;;    variable *version-graph*. If not allowed throws to :error.
;;;
;;; %ANCESTOR? (obj1 obj2 &key (prop-id '$is-a)                  [FUNCTION]
;;;             (context *context*))
;;;    Checks if obj2 is equal to obj1, or belongs to the transitive closure
;;;    of obj1 ~ using property prop-id.
;;;    Arguments:
;;;    obj1: first object
;;;    obj2: second object
;;;    prop-id: property to follow (default is $IS-A)
;;;    context (opt): context (default current)
;;;
;;; %BUILD-METHOD (function-name arg-list body)                  [FUNCTION]
;;;    build a MOSS method from its definition in the various defxxxmethof
;;;    macros. ~ Body may have a comment (string) and/or a declare
;;;    statement. The method ~ body is enclosed in a catch :return clause to
;;;    allow exits, since we cannot ~ use the return-from clause inside the
;;;    method. When the compiler is included in the environment, compiles
;;;    the code. Arguments:
;;;    function-name: the name of the function implementing the method
;;;    (e.g. $UNI=I=0=SUMMARY) arg-list: method arguments
;;;    body: the body of the function
;;;    Return:
;;;    the function name.
;;;
;;; %BUT-MOSS (item-list)                                        [FUNCTION]
;;;    Removes from a list of symbols any symbol that is not a PDM object or
;;;    that belongs ~ to the MOSS system. Thus we should only get PDM
;;;    application objects. Arguments:
;;;    item-list: a list of symbols
;;;    Return:
;;;    a list of application objects.
;;;
```

```
;;; %CLASSLESS-OBJECT? (object-id &key (context *context*))          [FUNCTION]
;;;    Checks if the entity is a classless object, i.e. its type is *none*
;;;    Arguments:
;;;    object-id: identifier of object
;;;    context (opt): context (default current)
;;;    Return:
;;;    T if OK, NIL otherwise.
;;;
;;; %CLEAN-WORD-LIST (word-list &rest empty-word-lists)              [FUNCTION]
;;;    remove empty words from the list using empty-word lists.
;;;    Arguments:
;;;    word-list: list of words to be cleaned
;;;    empty-word-lists (rest): lists of empty words
;;;    Returns:
;;;    the cleaned list.
;;;
;;; %CREATE-BASIC-NEW-OBJECT (object-id &key (context *context*) id  [FUNCTION]
;;;                           ideal (package *package*))
;;;    Creates a skeleton of object containing only the $TYPE and $ID ~
;;;         properties in the current context.
;;;    Arguments:
;;;    object-id: presumably class of the object to be created
;;;    context (key): context default current
;;;    id (key): if there, id is specified (we do not create it)
;;;    ideal (key): if t, we want to create an ideal instance
;;;                         (with sequence number 0)
;;;    package (key): package (default is current)
;;;    Return:
;;;    id of the newly created object.
;;;
;;; %CREATE-BASIC-ORPHAN (&key (context *context*)                   [FUNCTION]
;;;                        (package *package*))
;;;    Creates a skeleton for an orphan object and the associated key.
;;;    Arguments:
;;;    context (key): context (default current)
;;;    Return:
;;;    the id of the new orphan.
;;;
;;; %CREATE-ENTRY-POINT-METHOD (object-id object-name                [FUNCTION]
;;;                             &optional args doc body)
;;;    Creates an entry point method, with default if no arguments. ~
;;;         the =make-entry symbol is defined in the same package as
;;;    object-id. If args is present it must contain an option &key (package
;;;    *package*). Arguments:
;;;    object-id: id to which the =make-entry method will be attached
;;;    object-name: name of the object (for error message)
;;;    args (opt): arguments of the method (default is (value-list))
;;;    doc (opt): documentation string
;;;    body (opt): body of the method
;;;    Return:
;;;    an own-method id.
```

```
;;;
;;; %DETERMINE-PROPERTY-ID (obj-id prop-name                          [FUNCTION]
;;;                         &key (context *context*))
;;;    For the object with id obj-id, attempts to determine which of its ~
;;;         properties corresponds  to prop-name in the specified context.
;;;    If the property is present at the object level, then return it,
;;;    otherwise: - if the object has a prototype, try the prototype
;;;         - if the object has a class, look at the class level for
;;;    properties ~ defined at the class level.
;;;         - if the object belongs to several classes try each one in
;;;    turn. Problems are:
;;;    - tree properties: properties are in a tree (normally each node of
;;;    the ~ tree should have the same property name)
;;;    - lattices of prototypes, where a property could be derived from
;;;    another ~ one thru an IS-A relation (not implemented).
;;;    Arguments:
;;;    obj-id: identifier of the object
;;;    prop-name: name of property to check: e.g. HAS-NAME (must be
;;;    valid, no check) context: context
;;;    Returns:
;;;    a list of normally a single property-id.
;;;
;;; %DETERMINE-PROPERTY-ID-FOR-CLASS (prop-list class-id              [FUNCTION]
;;;                                   &key (context *context*)
;;;                                   inverse)
;;;    Determines the right prop-id for the specified class among a list of
;;;    properties. No checks on arguments. If property is inherited, then we
;;;    return the closest one. If class is *any* returns the list.
;;;    Arguments:
;;;    prop-list: list of property identifiers, e.g. $T-NAME,
;;;    $S-PERSON-HUSBAND.OF class-id: identifier of the class, e.g.
;;;    $E-PERSON (*any* is allowed) context (opt): context default current
;;;    inverse (key): specify that we look for inverse property
;;;    Returns:
;;;    nil or list of property-id
;;;
;;; %ENTRY? (object-id &key (context *context*)                       [FUNCTION]
;;;          (package *package* package-here?))
;;;    Checks if the id is that of an entry point.
;;;    Arguments:
;;;    object-id: identifier of object
;;;    context (opt): context (default current)
;;;    package (opt): if obj-id is a symbol, then that of obj-id default
;;;    current Returns:
;;;    nil or something not meaningful
;;;
;;; %EXPLODE-OBJECT (obj-id &key into)                                [FUNCTION]
;;;    function that explodes an instance into a set of triples appending it
;;;    to the ~ list specified by the into variable.
;;;    Arguments:
;;;    obj-id: object identifier
```

```
;;;    into (key): list to which to append new triples
;;;    Return:
;;;    the list of triples.
;;;
;;; %EXTRACT (entry tp class &key (context *context*)                [FUNCTION]
;;;           (filter nil filter-there?)
;;;           (class-ref nil class-there?)
;;;           (package *package* package-there?)
;;;           allow-multiple-values)
;;;    Extracts an object that is specified by its entry-point - attribute -
;;;    ~ concept - and context. When more than one, applies a function when
;;;    ~ specified in the option <:filter function>. If more than one value
;;;    left ~ then an error is declared, unless they are explicitely allowed
;;;    by setting the ~ keyword argument :allow-multiple-values to T.
;;;    Arguments:
;;;    entry: entry point (string, symbol or mln)
;;;    tp: attribute (string, symbol or mln)
;;;    class: class name (string, symbol or mln)
;;;    context (key): context default current
;;;    filter (key): filter function when more than one value is expected
;;;    filter-there?: variable set to T when a filter is specified
;;;    class-ref (key): reference of class to locate a property
;;;    class-there?: variable set to T when class-ref is specified
;;;    package (key): default current
;;;    allow-multiple-values (key): if T the function may return more
;;;    than one value Returns
;;;    nil if it cannot find anything
;;;    a symbol  for a single object
;;;    a list when it finds several objects and multiples are allowed
;;;
;;; %EXTRACT-FROM-ID (entry prop-id class-id                         [FUNCTION]
;;;                   &key (context *context*))
;;;    Returns the list of all objects  corresponding to entry point entry ~
;;;         with inverse attribute of prop-id and of class class-id.
;;;       E.g. (%extract-from-id 'ATTRIBUTE '$ENAM '$ENT 0) returns ($EPT)
;;;       obj-id are returned if they are in class class-id or in one of
;;;    the subclasses ~ (transitive closure along the $IS-A.OF link).
;;;       Also all meta-models $ENT, $EPS, $EPT, $FN, $CTR ($EIL?) are
;;;    defined ~ as instances of $ENT.
;;;       If context is illegal throws to an :error tag.
;;;       The function works with the *any* and *none* pseudo classes:
;;;     (%extract-from-id <entry> <prop-id> '*any*) returns everything
;;;     (%extract-from-id <entry> <prop-id> '*none*) returns only classless
;;;    objects. We also allow objects with multiple classes.
;;;    No checks are done to speed execution.
;;;    Arguments:
;;;    entry: entry point (symbol)
;;;    prop-id: id of property to be used to make the inverse
;;;                   should be the exact property associated with class-id
;;;    class-id: id of the potential class of the object
;;;    context (key): context (default current)
```

```
;;;    Returns:
;;;    a list of objects maybe one
;;;
;;; %FILTER-ALIVE-OBJECTS (object-list &key (context *context*))     [FUNCTION]
;;;    Takes a list of objects and keeps the ones that are alived in the
;;;    specified ~ context.
;;;    Arguments:
;;;    object-list: list of objects
;;;    context (opt): context (default current)
;;;    Return:
;;;    a (possibly empty) list of objects alive in context.
;;;
;;; %GET-ALL-ATTRIBUTES "()"                                          [FUNCTION]
;;;    Gets the list of all attributes in all contexts. No arguments.
;;;
;;; %%GET-ALL-CLASS-ATTRIBUTES (class-id &key (context *context*))   [FUNCTION]
;;;    Collects all attributes starting with class-id through superclasses.
;;;    Attributes of the same property tree can be included in the
;;;    result. Arguments:
;;;    class-id: identifier of the class
;;;    context (opt): context (default current)
;;;    Return:
;;;    a list of class attributes.
;;;
;;; %%GET-ALL-CLASS-INVERSE-RELATIONS (class-id                       [FUNCTION]
;;;                                    &key (context *context*))
;;;    Collects all relations starting with class-id through superclasses.
;;;    Relations of the same property tree can be included in the result.
;;;    Arguments:
;;;    class-id: identifier of the class
;;;    context (opt): context (default current)
;;;    Return:
;;;    a list of class attributes.
;;;
;;; %%GET-ALL-CLASS-PROPERTIES (class-id &key (context *context*))   [FUNCTION]
;;;    Collects all properties starting with class-id through superclasses.
;;;    Relations of the same property tree can be included in the result.
;;;    Arguments:
;;;    class-id: identifier of the class
;;;    context (opt): context (default current)
;;;    Return:
;;;    a list of class attributes.
;;;
;;; %%GET-ALL-CLASS-RELATIONS (class-id &key (context *context*))    [FUNCTION]
;;;    Collects all relations starting with class-id through superclasses.
;;;    Relations of the same property tree can be included in the result.
;;;    Arguments:
;;;    class-id: identifier of the class
;;;    context (opt): context (default current)
;;;    Return:
;;;    a list of class attributes.
```

```
;;;
;;; %GET-ALL-CONCEPTS "()"                                          [FUNCTION]
;;;    Gets the list of all concepts in all contexts. No arguments.
;;;
;;; %GET-ALL-ENTRY-POINTS "()"                                      [FUNCTION]
;;;    Gets the list of all entry-points in all contexts. No arguments.
;;;
;;; %GET-ALL-INSTANCES (class-id &key package)                      [FUNCTION]
;;;    Gets the list of all instances of a class in all contexts. We assume
;;;    that ~ counter is in context 0.
;;;    The default behavior is that the package for the instances is the
;;;    same as ~ the package of the class-id symbol. Otherwise one should
;;;    know what ~ one does...
;;;    Arguments:
;;;    class-id: id of the class
;;;    return:
;;;    list of all bound identifiers of the instances of the class
;;;
;;; %GET-ALL-INVERSE-RELATIONS "()"                                 [FUNCTION]
;;;    Gets the list of all inverse relations in all contexts. No arguments.
;;;
;;; %GET-ALL-METHODS "()"                                           [FUNCTION]
;;;    Gets the list of all methods in all contexts. No arguments.
;;;
;;; %GET-ALL-MOSS-OBJECTS "()"                                      [FUNCTION]
;;;    Gets the list of all MOSS objects in all contexts. No arguments.
;;;
;;; %GET-ALL-ORPHANS "()"                                           [FUNCTION]
;;;    Gets the list of all orphans in all contexts. No arguments.
;;;
;;; %GET-ALL-RELATIONS "()"                                         [FUNCTION]
;;;    Gets the list of all relations in all contexts. No arguments.
;;;
;;; %%GET-ALL-SUBCLASS-INVERSE-RELATIONS (class-id                  [FUNCTION]
;;;                                       &key (context *context*))
;;;    Collects all properties starting with class-id through superclasses.
;;;    Relations of the same property tree can be included in the result.
;;;    Arguments:
;;;    class-id: identifier of the class
;;;    context (opt): context (default current)
;;;    Return:
;;;    a list of class attributes.
;;;
;;; %%GET-ALL-SUBCLASS-RELATIONS (class-id                          [FUNCTION]
;;;                               &key (context *context*))
;;;    Collects all relations starting with class-id through subclasses.
;;;    Relations of the same property tree can be included in the result.
;;;    Arguments:
;;;    class-id: identifier of the class
;;;    context (opt): context (default current)
;;;    Return:
```

```
;;;    a list of class attributes.
;;;
;;; %GET-ALL-SYMBOLS "()"                                          [FUNCTION]
;;;    Gets the list of all symbols in all contexts (counters, global
;;;    variables,... No arguments.
;;;
;;; %GET-ALL-VERSIONS (obj-id prop-id)                             [FUNCTION]
;;;    Returns all versions of values or of successors associated with ~
;;;         the given property prop-id. Removes ~
;;;         duplicates of the values corresponding to the various versions.
;;;    ~ The function is used when trying to get all objects of a kind ~
;;;         from the system, e.g. by %get-all-symbols.
;;;    Arguments:
;;;    obj-id: object identifier
;;;    prop-id: property id for which we want all values
;;;    Return:
;;;    a list of all values regardless of versions.
;;;
;;; %GET-AND-INCREMENT-COUNTER (obj-id                             [FUNCTION]
;;;                             &optional (counter-link-id '$ctrs))
;;;    Obtains the value of a counter associated with obj-id and increases
;;;    this value ~ in the counter itself. Throws to :error if obj-id is
;;;    invalid. Arguments
;;;    obj-id: id of the object that owns the counter (ususally a class)
;;;    counter-link-id (opt): name of the property linking to the counter
;;;    (default $CTRS) Return:
;;;    old value of the counter.
;;;
;;; %GET-APPLICATION-CLASSES (&key (context *context*))            [FUNCTION]
;;;    returns the list of classes of a given application by extracting them
;;;    from ~ the system object and removing the system classes.
;;;    If OMAS is active, then we consider objects from the current
;;;    package. Arguments:
;;;    context (opt): default is *context*
;;;    Returns:
;;;    a list of class ids.
;;;
;;; %GETC (obj-id prop-id context)                                 [FUNCTION]
;;;    Retrieves the value associated with prop-id ~
;;;         for the specified context from the p-list of obj-id used as a
;;;    cache. ~ If the context is illegal, then throws to an :error tag.
;;;    Arguments:
;;;    obj-id: identifier of object
;;;    prop-id: identifier of local property
;;;    context: context
;;;    Returns:
;;;    nil or the cached value, Throws to :error when context is illegal
;;;    or object dead.
;;;
;;; %GET-CLASS-CANONICAL-NAME-FROM-REF (ref)                       [FUNCTION]
;;;    uses ref as an entry point to the property. If it fails return nil.
```

```
;;;    Otherwise, ~ returns the canonical name in the current language.
;;;    Arguments:
;;;    ref: e.g. "country"
;;;    Return:
;;;    canonical name in the current language specified by *language*.
;;;
;;; %GET-CLASS-ID-FROM-REF (class-ref &key (context *context*)       [FUNCTION]
;;;                         (package *package*))
;;;    Retrieves the class id from its name. Should be unique. Error
;;;    otherwise. Argument:
;;;    class-ref: a symbol, string or multilingual name, e.g. PERSON,
;;;    "Person", (:en ...) context (opt): context default current
;;;    package (opt): default current
;;;    Return:
;;;    class-id, e.g. $E-PERSON
;;;
;;; GET-CURRENT-YEAR "()"                                            [FUNCTION]
;;;
;;; %GET-DEFAULT-FROM-CLASS (prop-id class-id)                       [FUNCTION]
;;;    gets the default value-list associated with a property in a given
;;;    class. ~ The way to do it is to ask the property itself; then the
;;;    ideal of the class. Defaults cannot be inherited.
;;;    Arguments:
;;;    prop-id: id of corresponding property
;;;    class-id: id of the class
;;;    Return:
;;;    defaul value or nil.
;;;
;;; %GET-OBJECT-CLASS-NAME (obj-id)                                  [FUNCTION]
;;;    get the class names of an object.
;;;    Arguments:
;;;    obj-id: object identifier
;;;    Return:
;;;    a list of strings representing the names of the classes to which
;;;    the object belongs.
;;;
;;; %GET-EMPTY-WORDS (language)                                      [FUNCTION]
;;;    recover a list of empty words in specified language. This assumes ~
;;;             that a class empty-words exists and has been initialized.
;;;    Arguments:
;;;    language: a string or keyword
;;;    Return:
;;;    a list of empty words or nil.
;;;
;;; %GET-GENERIC-PROPERTY (prop-list &key (context *context*))       [FUNCTION]
;;;    We start with a set of properties having the same name and try to
;;;    obtain the ~ one that is generic, i.e. that does not have an ancestor
;;;    (no $IS-A property). Argument:
;;;    prop-list: list of property identifiers
;;;    Return:
;;;    the identifier of the generic property or nil. Warns if there are
```

```
;;;    more than one left.
;;;
;;; %GET-GENERIC-PROPERTY-FROM-NAME (prop-name                          [FUNCTION]
;;;                                  &key (context *context*))
;;;    We start with a property name and try to obtain the id of the ~
;;;         one that is generic, i.e. that does not have an ancestor.
;;;    Argument:
;;;    prop-name: property name
;;;    Return:
;;;    the identifier of the generic property or nil. Warns if there are
;;;    more than one left.
;;;
;;; %GET-GENERIC-PROPERTY-FROM-REF (prop-ref)                           [FUNCTION]
;;;    We start with a set of properties having the same name and try to
;;;    obtain the ~ one that is generic, i.e. that does not have an ancestor
;;;    (no $IS-A property). Argument:
;;;    prop-ref: string or symbol e.g. "content" or CONTENT
;;;    Return:
;;;    the identifier of the generic property (e.g. HAS-CONTENT) or nil.
;;;
;;; %GET-INDEX-WEIGHTS (task-id)                                        [FUNCTION]
;;;    get a list of pairs index (a string) value from the list of ~
;;;        index patterns associated with the task.
;;;    Arguments:
;;;    task: a task object
;;;    Return:
;;;    a list of pairs.
;;;
;;; %GET-INSTANCE-COUNT (class-id &key (context *context*)              [FUNCTION]
;;;                      (allow-subclasses t))
;;;    Gets an upper-bound on the number of instances for a given class.
;;;    Arguments:
;;;    class-id: identifier of class
;;;    allow-subclasses (key): if T counts instances of subclasses
;;;    Returns:
;;;    a number
;;;
;;; %%GET-LIST-OF-VALUES-AND-CONTEXT (obj-id prop-id)                   [FUNCTION]
;;;    Done to obtain raw value list of objects.
;;;    Arguments:
;;;    obj-id: identifier of object
;;;    prop-id: property local identifier
;;;    Returns:
;;;    a list of all values by context (internal format of the set of
;;;    values).
;;;
;;; %GETM (object-id method-name context &rest own-or-instance)         [FUNCTION]
;;;    When methods are compiled on p-list it is done as follows:
;;;     (=print-self (:own (nn <int-function name>)*) (:instance  <...>))
;;;       if the method type is not specified then the format is slightly ~
;;;         different from the more general format.
```

```
;;;       The version must be recorded locally otherwise we shall try ~
;;;         to inherit it. Remember this is a cache, not an object.
;;;       When using versions we can have the method cached onto the p-list
;;;    for a ~ previous version. If the method has not been modified since,
;;;    it is a waste ~ of energy to recover it again, to recompile it, and
;;;    to give it a new name. ~ Rather, we want to check that it was not
;;;    modified along the same branch indeed, ~ then we cache it under the
;;;    current version explicitely with the old internal ~ name. In order to
;;;    do that, we must record every time a method code or arguments ~ was
;;;    changed. Doing
;;;           (%get-value <method-id> <modif> *current-context*)
;;;       will return the last modification or nil (it could have been
;;;    modified, but ~ then it is not in the same branch.
;;;       So for recovering a cached method, we first get the current
;;;    branch of the ~ path from the current context to the root of the
;;;    version graph. We then loop on the successive values of the versions
;;;    in the branch starting ~ with the current one. At each step, either
;;;    we get a function, or this is the ~ step at which the version was
;;;    modified, in which case we loose and we must ~ recompute and
;;;    recompile the method. Otherwise, when we get a previously ~ recorded
;;;    method, we must cache it explicitely on the p-list for next time
;;;    around. Arguments: object-id: id of the object
;;;    method-name: name of the target method
;;;    context: context
;;;    own-or-instance (key): flag to specify type of method
;;;    Return:
;;;    method or nil
;;;
;;; %GET-NAME-FROM-REF (ref object-type)                          [FUNCTION]
;;;    Takes a reference and tries to obtain the official name for the
;;;    object, e.g. ~ "class" -> CONCEPT
;;;    If ref is a symbol, we assume it is the name we are looking for,
;;;       if ref is a multilingual name, we use the canonical name,
;;;       if ref is a string, we build a name from it, specific to the
;;;    object type, ~ and recover the canonical name from its recorded name.
;;;    if the object is a property yet undefined, we return the cooked up
;;;    name. Arguments:
;;;    ref: a symbol, string or multilingual name
;;;    object-type: a keyword giving the type of the object (:class,
;;;    :attribute, ...) package (opt): a package spec for the name
;;;    Return:
;;;    the canonical name of the object.
;;;
;;; %%GET-OBJECTS-FROM-ENTRY-POINT (entry &key (context *context*)   [FUNCTION]
;;;                                 keep-entry)
;;;    Sets up an a-list (inv-prop obj-id) of all objects for which entry is
;;;    an ~ entry point, eliminating the current-system (inv prop:$EPLS.OF).
;;;    Arguments:
;;;    entry: entry-point
;;;    context (key): context default current
;;;    keep-entry (key): if t keep the entry point
```

```
;;;    Return:
;;;    a list of pairs (<inv-prop-id> . <obj-id>)  or nil
;;;
;;; %GET-PROP-ID-FROM-NAME-AND-CLASS-REF (prop-name class-ref        [FUNCTION]
;;;                                       &key (context *context*)
;;;                                       (package *package*))
;;;    Determines the right property for the specific class. Throws to
;;;    :error when ~ something is wrong. Property may be inherited.
;;;    Arguments:
;;;    prop-name: name of a property
;;;    class-ref: class name or class identifier
;;;    context (opt): context default current
;;;    package (opt): defualt current
;;;    Return:
;;;    the property attached to the class.
;;;
;;; %GET-PROPERTIES (object-id &key (context *context*))             [FUNCTION]
;;;    Obtain all the possible properties for a given object. If an orphan,
;;;    then ~ we only get local properties attached to the orphan.
;;;    Otherwise, we get ~ all inherited properties corresponding to the
;;;    object class. Arguments:
;;;    object-id: identifier of object
;;;    context (opt): context (default current)
;;;    Return:
;;;    a list of properties associated with the object.
;;;
;;; %GET-PROPERTY-CANONICAL-NAME-FROM-REF (ref)                      [FUNCTION]
;;;    uses ref as an entry point to the property. If it fails return nil.
;;;    Otherwise, ~ returns the canonical name in the current language.
;;;    Arguments:
;;;    ref: e.g. "country"
;;;    Return:
;;;    canonical name in the current language specified by *language*.
;;;
;;; %GET-PROPERTY-ID-FROM-NAME (obj-id prop-ref                      [FUNCTION]
;;;                             &key (context *context*)
;;;                             (package *package*))
;;;    Function that has the difficult role of finding the right property
;;;    that ~ applies to the object, and corresponds to the prop-name. This
;;;    is ~ difficult in case we have tree-properties.
;;;    It is used by elementary universal methods like =get =put =delete
;;;    Result is cached onto the p-list of the property name (e.g.
;;;    HAS-NAME). Arguments:
;;;    obj-id: object for which we try to determine property
;;;    prop-ref: property ref, e.g. HAS-FIRST-NAME, "first name", or mln
;;;    context (opt): context default current
;;;    Return:
;;;    the identifier of the corresponding property or nil if the class
;;;    inherits property.
;;;
;;; %GET-PROPERTY-SYNONYMS (prop-ref &key (package *package*))       [FUNCTION]
```

```
;;;    takes a property reference and return the list of synonyms for this
;;;    property. Arguments:
;;;    prop-ref: a property reference, e.g. "street"
;;;    package (key): package in which property name has been defined
;;;    Return;
;;;    nil or the list of all property synonyms.
;;;
;;; %GET-RELEVANT-WEIGHTS (weight-list word-patterns                  [FUNCTION]
;;;                        &key used-patterns)
;;;    function to obtain weigths for a list of words.
;;;    Arguments:
;;;    weight-list: list of weights e.g. ((<string> .6))
;;;    word-patterns: the list of expressions (strings) to test
;;;    used-patterns (key): expressions that have already been tried
;;;    Return:
;;;    a list of pairs string weight.
;;;
;;; %GET-SP-ID-FROM-NAME-AND-CLASS-REF (prop-name class-ref           [FUNCTION]
;;;                                     &key (context *context*)
;;;                                     (package *package*))
;;;    Determines the right relation for the specific class. Throws to
;;;    :error when ~ something is wrong.
;;;    Arguments:
;;;    prop-name: name of a property
;;;    class-ref: class name or class identifier
;;;    context (opt): context (default current)
;;;    Return:
;;;    the property attached to the class.
;;;
;;; %GET-SP-VALUE-RESTRICTIONS (prop-id &key (context *context*))    [FUNCTION]
;;;    Obtains the value restrictions from the attribute representation. No
;;;    ~ inheritance is done.
;;;    Restrictions are
;;;       :exists     ; among the possible values one must be of that type
;;;       :forall     ; all values should be of that type
;;;       :type       ; values should be of that type
;;;       :not-type   ; no value should be of that type
;;;       :value      ; only this value is allowed
;;;       :one-of     ; the value should be a member of this list
;;;       :not        ; the value should not be one of this list
;;;    Arguments:
;;;    prop-id: attribute id
;;;    context (opt): context (default current)
;;;    Returns:
;;;    a list of restrictions: e.g. ((:one-of 1 2 3) (:type :integer)
;;;    (:one)).
;;;
;;; %GET-SUBCLASS-NAMES (class-ref)                                  [FUNCTION]
;;;    takes a class reference and return the list of synonyms for all
;;;    subclasses, ~ including the referenced class.
;;;    Arguments:
```

```
;;;    class-ref: a class reference, e.g. "address"
;;;    Return;
;;;    nil or the list of all subclass names.
;;;
;;; %GET-SUBCLASSES (obj-id)                                        [FUNCTION]
;;;    returns the list of subclasses of obj-id.
;;;    Argument:
;;;    obj-id: a class identifier
;;;    Return:
;;;    nil or the transitive closure along the inverse $IS-A property.
;;;
;;; %GET-TP-ID-FROM-NAME-AND-CLASS-REF (prop-name class-ref          [FUNCTION]
;;;                                     &key (context *context*))
;;;    Determines the right attribute for the specific class. Throws to
;;;    :error when ~ something is wrong. When class is *none* (orphans) all
;;;    attributes are possible. Arguments:
;;;    prop-name: name of a property
;;;    class-ref: class name or class identifier
;;;    context (opt): context default current
;;;    Return:
;;;    the attribute attached to the class or nil.
;;;
;;; %GET-TP-VALUE-RESTRICTIONS (prop-id &key (context *context*))    [FUNCTION]
;;;    Obtains the value restrictions from the attribute representation. No
;;;    ~ inheritance is done.
;;;    Restrictions are
;;;       :exists     ; among the possible values one must be of that type
;;;       :forall     ; all values should be of that type
;;;       :type       ; values should be of that type
;;;       :not-type   ; no value should be of that type
;;;       :value      ; only this value is allowed
;;;       :one-of     ; the value should be a member of this list
;;;       :not        ; the value should not be one of this list
;;;       :between    ; the numerical value should be in that range
;;;       :outside    ; the numerical value should be outside that range
;;;    Arguments:
;;;    prop-id: attribute id
;;;    context (opt): context (default current)
;;;    Returns:
;;;    a list of restrictions: e.g. ((:one-of 1 2 3) (:type :integer)
;;;    (:one)).
;;;
;;; %GET-VALUE (obj-id prop-id &key (context *context*))             [FUNCTION]
;;;    Gets the value corresponding to the specified context. To do so uses
;;;    ~ the *version-graph* to locate within a branch the latest valid
;;;    value. ~ The context graph is organized as an a-list of node, each
;;;    node points to ~ its predecessors: ((9 6)(8 3 2)(7 6)...(0)) it
;;;    contains all contexts, ~ context numbers are not necessarily
;;;    increasing along a branch, e.g. if we have ~ fused some contexts.
;;;       Algorithm: (i) gets the branch from the current context, if
;;;    legal, to the root. ~ (the branch is computed only once for a given
```

```
;;;    context and cached onto the ~ p-list of the global symbol
;;;    *version-graph*). ~ (ii) goes up in the branch, returning the first
;;;    non nil value associated with ~ a context in the object.
;;;    The exact property of the object must be used, not the generic
;;;    one. %get-value does not return defaults.
;;;       If context is illegal throws to an :error tag.
;;;    Arguments:
;;;    obj-id: identifier of object
;;;    prop-id: identifier of  property
;;;    context: context
;;;
;;; %%GET-VALUE (obj-id prop-id context)                          [FUNCTION]
;;;    Returns the value associated with a particular context using the
;;;    *version- ~ graph*. If context is illegal throws to an :error tag.
;;;       Like %get-value but does not require the object to have a
;;;    property $TYPE. object-id must not be a REF object, but a regular
;;;    object. Arguments:
;;;    obj-id: identifier of object
;;;    prop-id: identifier of local property
;;;    context: context
;;;    Return: the value associated with prop-id.
;;;
;;; %%GET-VALUE-AND-CONTEXT (obj-id prop-id)                      [FUNCTION]
;;;    Done to obtain value for objects which must not be versioned, like
;;;    counters ~ for entities.
;;;    Arguments:
;;;    obj-id: object identifier
;;;    prop-id: local property identifier.
;;;    Retrurn:
;;;    a list whose car is context and cadr is the value.
;;;
;;; %GET-VALUE-FROM-CLASS-REF (class-ref expr)                    [FUNCTION]
;;;    returns the value associated with class-ref from a MOSS pattern.
;;;    Arguments:
;;;    class-ref: e.g. "address"
;;;    Returns:
;;;    value or nil.
;;;
;;; %GET-VALUE-FROM-PROP-REF (prop-ref expr                       [FUNCTION]
;;;                           &key (package *package*))
;;;    returns the value associated with prop-ref from a MOSS pattern.
;;;    Arguments:
;;;    prop-ref: e.g. "town"
;;;    package (key): package in which property has been defined
;;;    Returns:
;;;    value or nil.
;;;
;;; %GET-WORDS-FROM-TEXT (text delimiters)                        [FUNCTION]
;;;    takes an input string and a set of delimiters and returns a list of
;;;    words ~ separated by delimiters.
;;;    Arguments:
```

```
;;;    text: a string
;;;    delimiters: a set of characters that delimit a word
;;;    Return:
;;;    a list of words.
;;;
;;; %HAS-ID-LIST (entry inv-id class-id &key (context *context*))    [FUNCTION]
;;;    Returns the list of all objects ~
;;;         corresponding to entry point entry with inverse terminal
;;;    property inv-id ~ and of class class-id.
;;;       E.g. (%has-id-list 'ATTRIBUTE '$PNAM.OF '$ENT) returns ($EPT) ~
;;;         class-id must be specified locally; i.e., will not return
;;;    objects ~ that are instances of subclasses of class-id.
;;;       If context is illegal throws to an :error tag
;;;    Arguments:
;;;    entry: entry symbol
;;;    inv-id: local inverse attribute
;;;    class-id: identifier of target class
;;;    context: context
;;;    Return:
;;;    a list of objects.
;;;
;;; %%HAS-INVERSE-PROPERTIES (obj-id context)                        [FUNCTION]
;;;    Returns the list of inverse properties local to an object and having
;;;    ~ associated values in the specified context
;;;    Arguments:
;;;    obj-id: object identifier
;;;    context: context
;;;    Return:
;;;    a list of properties or nil.
;;;
;;; %%HAS-PROPERTIES (obj-id context)                                [FUNCTION]
;;;    Returns the list of properties local to an object and having
;;;    associated ~ values in the specified context, looking directly at the
;;;    internal format ~ of the object. $TYPE is removed.
;;;    Arguments:
;;;    obj-id: object identifier
;;;    context: context
;;;    Return:
;;;    a list of properties or nil.
;;;
;;; %%HAS-PROPERTIES+ (obj-id context)                               [FUNCTION]
;;;    Returns the list of ALL properties LOCAL to an object even with no ~
;;;         value, in the specified context, looking directly at the
;;;    internal format ~ of the object.
;;;    Arguments:
;;;    obj-id: object identifier
;;;    context: context
;;;    Return:
;;;    a list of properties or nil.
;;;
;;; %%HAS-STRUCTURAL-PROPERTIES (obj-id context)                     [FUNCTION]
```

```
;;;    Returns the list of structural properties local to an object and
;;;    having  ~ associated values in the specified context. $TYPE and $ID
;;;    are removed. Arguments:
;;;    obj-id: object identifier
;;;    context: context
;;;    Return:
;;;    a list of properties or nil.
;;;
;;; %%HAS-TERMINAL-PROPERTIES (obj-id context)                        [FUNCTION]
;;;    Returns the list of terminal properties local to an object and having
;;;    ~ associated values in the specified context.
;;;    Arguments:
;;;    obj-id: object identifier
;;;    context: context
;;;    Return:
;;;    a list of properties or nil.
;;;
;;; %%HAS-VALUE (obj-id prop-id context &key no-resolve)              [FUNCTION]
;;;    Returns the value list associated to a particular context if recorded
;;;    locally. No checks are done, except for legal context.
;;;    Arguments:
;;;    obj-id: object or ref to object under consideration (can be dead)
;;;    prop-id: property (may be local or inherited)
;;;    context: context (default current)
;;;    no-resolve (key): if true, skip the call to resolve intended to
;;;    solve multiclass belonging
;;;    Return:
;;;    the local value if any.
;;;
;;; INSERT-BEFORE (val val-l &rest val-test)                          [FUNCTION]
;;;    Should be a primitive.
;;;         Inserts item into a list either at the end of the list, or in
;;;    front of the ~ specified test-item, possibly duplicating it.
;;;    Arguments:
;;;    val: value to insert
;;;    val-l: list into which to insert the value
;;;    val-test (&rest): value that will be checked for insertion (equal
;;;    test) Return:
;;;       modified list or list of the value when val-l was nil
;;;
;;; INSERT-NTH (val val-l nth)                                        [FUNCTION]
;;;    Should be a primitive.
;;;         Inserts an item into a list at the nth position. If the list is
;;;    too short ~ then the value is inserted at the end of it.
;;;    Arguments:
;;;    val: value to be inserted
;;;    val-l: list into which to insert the value
;;;    nth: position at which the value must be inserted (0 is in front)
;;;    Return:
;;;    modified list
;;;
```

```
;;; %INTERSECT-SYMBOL-LISTS (l11 l12)                                    [FUNCTION]
;;;    Intersect two lists of symbols by using their p-list. Done in linear
;;;    time.
;;;
;;; %INVERSE-PROPERTY-ID (prop-id &key (context *context*))              [FUNCTION]
;;;    Computes the inverse property id of a given structural or terminal
;;;    property Optional arg: version. When the property is still undefined,
;;;    calls %make-id-for-inverse-property.
;;;    For inverse properties of an inverse property return a list if
;;;    more than one, ~ e.g. when we have a property lattice.
;;;    Arguments:
;;;    prop-id: identifier of property
;;;    context (opt): context default current
;;;    Returns:
;;;    an inverse property identifier
;;;
;;; %IS-A? (obj1 obj2 &key (context *context*))                          [FUNCTION]
;;;    Checks if obj1 is equal to obj2, or obj-2 belongs to the transitive
;;;    closure of obj1~ using property $IS-A.
;;;    Arguments:
;;;    obj1: first object
;;;    obj2: second object
;;;    context (opt): context (default current)
;;;
;;; %IS-ATTRIBUTE? (prop-id &key (context *context*))                    [FUNCTION]
;;;    Checks if the identifier points towards an attribute
;;;    Arguments:
;;;    prop-id: object identifier
;;;    context (opt): context (default current)
;;;    Return:
;;;    T or nil
;;;
;;; %IS-ATTRIBUTE-MODEL? (object-id &key (context *context*))            [FUNCTION]
;;;    Checks if the object is $EPT or some sub-type.
;;;    Arguments:
;;;    object-id: object identifier
;;;    context (opt): context (default current)
;;;    Return:
;;;    T or nil
;;;
;;; %IS-CLASS? (object-id &key (context *context*))                      [FUNCTION]
;;;    Checks if the object is a class, i.e. if its type is $ENT or a
;;;    subtype of $ENT. Arguments:
;;;    object-id: identifier of the object to check
;;;    context (opt): context default current
;;;    Return
;;;    nil or a list result of an intersection.
;;;
;;; %IS-CONCEPT? (object-id &key (context *context*))                    [FUNCTION]
;;;    Checks if the object is a concept (same as class), i.e. if its type
;;;    is $ENT or a subtype of $ENT. Arguments:
```

```
;;;    object-id: identifier of the object to check
;;;    context (opt): context default current
;;;    Return
;;;    nil or a list result of an intersection.
;;;
;;; %IS-COUNTER-MODEL? (object-id &key (context *context*))          [FUNCTION]
;;;    Checks if the object is $CTR or some sub-type.
;;;    Arguments:
;;;    object-id: identifier of the object to check
;;;    context (opt): context default current
;;;    Return
;;;    nil or a list result of an intersection.
;;;
;;; %IS-ENTITY-MODEL? (object-id &key (context *context*))           [FUNCTION]
;;;    Checks if the object is $ENT or some sub-type.
;;;
;;; %IS-ENTRY-POINT-INSTANCE? (object-id &key (context *context*))   [FUNCTION]
;;;    Checks if the object is an instance of entry-point.
;;;
;;; %IS-GENERIC-PROPERTY-ID? (object-id &key (context *context*))    [FUNCTION]
;;;    Checks if the object is $EPR and is not a subproperty of some other
;;;    property.
;;;
;;; %IS-INSTANCE-OF? (object-id class-id)                            [FUNCTION]
;;;    Checks if the object is an instance of class id or of one of its
;;;    children. Arguments:
;;;    object-id: id of the object too test
;;;    class-id: id of the class
;;;    Return:
;;;    t if true nil otherwise.
;;;
;;; %IS-INVERSE-LINK-MODEL? (object-id &key (context *context*))     [FUNCTION]
;;;    Checks if the object is $EIL or some sub-type.
;;;
;;; %IS-INVERSE-PROPERTY? (xx &key (context *context*))              [FUNCTION]
;;;    Checks if the id points towards an inverse property, or to a property
;;;    containing $EIL in its transitive closure along $IS-A.
;;;
;;; %IS-INVERSE-PROPERTY-REF? (xx &key (context *context*))          [FUNCTION]
;;;    Checks if the string is the reference of an inverse property, i.e. a
;;;    string ~ starting with the > symbol whose name is that of a property.
;;;    Argument:
;;;    xx: string
;;;    Return:
;;;    the inverse property-id or nil.
;;;
;;; %IS-METHOD? (xx &key (context *context*))                        [FUNCTION]
;;;    Checks if the id points towards a method - Optional arg: context
;;;
;;; %IS-METHOD-MODEL? (object-id &key (context *context*))           [FUNCTION]
;;;    Checks if the object is $EFN or some sub-type.
```

```
;;;
;;; %IS-MODEL? (object-id &key (context *context*))                    [FUNCTION]
;;;    Checks whether an object is a model (class) or not - Models ~
;;;         are objects that have properties like $PT or $PS or $RDX or
;;;    $CTRS ~ or whose type is in the transitive closure of $ENT along the
;;;    $IS-A.OF link.
;;;
;;; %IS-ORPHAN? (object-id &key (context *context*))                   [FUNCTION]
;;;    Short synonym for %classless-object?
;;;
;;; %IS-RELATION? (prop-id &key (context *context*))                   [FUNCTION]
;;;    Checks if the object is $EPS or some sub-type.
;;;
;;; %IS-RELATIONSHIP? (prop-id &key (context *context*))               [FUNCTION]
;;;    Checks if the object is $EPS or some sub-type.
;;;
;;; %IS-RELATION-MODEL? (object-id &key (context *context*))           [FUNCTION]
;;;    checks if the object is $EPS or some sub-type
;;;
;;; %IS-STRUCTURAL-PROPERTY? (prop-id &key (context *context*))        [FUNCTION]
;;;    Checks if the identifier points towards a structural property -
;;;     Optional arg: context
;;;
;;; %IS-STRUCTURAL-PROPERTY-MODEL? (object-id                          [FUNCTION]
;;;                                 &key (context *context*))
;;;    Checks if the object is $EPS or some sub-type.
;;;
;;; %IS-SYSTEM? (object-id &key (context *context*))                   [FUNCTION]
;;;    Checks if the id points towards a system - Optional arg: context
;;;
;;; %IS-SYSTEM-MODEL? (object-id &key (context *context*))             [FUNCTION]
;;;    Checks if the entity is a model, and belongs to the MOSS package.~
;;;         Arguments:
;;;    object-id: identifier of object to test
;;;    context (opt): context default current
;;;    Return:
;;;    nil or t
;;;
;;; %IS-TERMINAL-PROPERTY? (prop-id &key (context *context*))          [FUNCTION]
;;;    Checks if the identifier points towards a terminal property -
;;;     Optional arg: context
;;;
;;; %IS-TERMINAL-PROPERTY-MODEL? (object-id                            [FUNCTION]
;;;                               &key (context *context*))
;;;    Checks if the object is $EPT or some sub-type.
;;;
;;; %IS-UNIVERSAL-METHOD? (xx &key (context *context*))                [FUNCTION]
;;;    Checks if the id points towards a universal method - Optional arg:
;;;    context
;;;
;;; %IS-UNIVERSAL-METHOD-MODEL? (object-id                             [FUNCTION]
```

```
;;;                              &key (context *context*))
;;;    Checks if the object is $UNI or some sub-type.
;;;
;;; %IS-VALUE-TYPE? (value value-type)                                [FUNCTION]
;;;    Checks that a value has the right type.
;;;    Arguments:
;;;    value: value to check
;;;    value-type: one of the XSD types (birkkkhh!)
;;;    Returns:
;;;    value if the right type, nil otherwise.
;;;
;;; %IS-VARIABLE-NAME? (ref)                                          [FUNCTION]
;;;    Checks if the refernce is a variable name, i.e. a symbol starting
;;;    with _ (underscore)
;;;
;;; %IS-VIRTUAL-CONCEPT? (object-id &key (context *context*))         [FUNCTION]
;;;    Checks if the object is a concept (same as class), i.e. if its type
;;;    is $ENT or a subtype of $ENT. Arguments:
;;;    object-id: identifier of the object to check
;;;    context (opt): context default current
;;;    Return
;;;    nil or a list result of an intersection.
;;;
;;; %LINK (obj1-id sp-id obj2-id &key (context *context*))            [FUNCTION]
;;;    Brute force link two objects using the structural property id - ~
;;;         No cardinality check is done - sp-id must be direct property.
;;;       (%link obj1-id sp-id obj2-id context) - links two objects ~
;;;         using sp-id - no checking done on args - crude link to be able
;;;    ~ to bootstrap without the properties really existing. Here we
;;;    manipulate ~ the names. We assume that sp-id is not an inverse
;;;    property ~ If the link already exists then does nothing. Otherwise,
;;;    introduces a ~ new link modifying or creating a value in the
;;;    specified context. ~ This might look like a bit complicated a piece
;;;    of code, however we try ~ to do simple tests first to avoid long
;;;    execution times. Arguments:
;;;    obj1-id: identifier for first object
;;;    sp-id: identifier of the local linking property
;;;    obj2-id: identifier for the second object
;;;    context: context
;;;    Return:
;;;    list representing the first object
;;;
;;; %LINK-ALL (obj1-id sp-id obj-list &key (context *context*))       [FUNCTION]
;;;    Brute force link an object to a list of objects using the structural
;;;    property ~ id - No cardinality check is done - sp-id must be direct
;;;    property. (%link obj1-id sp-id obj-list context) - links two objects
;;;    ~ using sp-id - no checking done on args - crude link to be able ~
;;;         to bootstrap without the properties really existing. Here we
;;;    manipulate ~ the names. We assume that sp-id is not an inverse
;;;    property ~ If the link already exists then does nothing. Otherwise,
;;;    introduces a ~ new link modifying or creating a value in the
```

```
;;;    specified context. ~ This might look like a bit complicated a piece
;;;    of code, however we try ~ to do simple tests first to avoid long
;;;    execution times. Arguments:
;;;    obj1-id: identifier for first object
;;;    sp-id: identifier of the local linking property
;;;    obj-list: identifier for the second object
;;;    context: context
;;;    Return:
;;;    list representing the first object
;;;
;;; %MAKE-ENTITY-SUBTREE (ens &key (context *context*))              [FUNCTION]
;;;    takes an entity and builds the subtree of its sub-classes, e.g.
;;;    (A (B C (D E) F))
;;;    where (D E) are children of C and (B C F) are children of A.
;;;    Arguments:
;;;    ens: entity to process
;;;    Return:
;;;    (ens) if no subtree, or (ens <subtree>).
;;;
;;; %MAKE-ENTITY-TREE (&key (context *context*))                     [FUNCTION]
;;;    builds the forest corresponfing to the application classes.
;;;    Arguments:
;;;    context (opt): default is *context*
;;;    Return:
;;;    a tree like (A (B C (D E) F) G H)
;;;    where (D E) are children of C and (B C F) are children of A, and G
;;;    and H are parallel trees.
;;;
;;; %MAKE-ENTITY-TREE (&key (context *context*)                      [FUNCTION]
;;;                     (package *package*))
;;;    builds the forest keeping classes from the current package.
;;;    Arguments:
;;;    context (opt): default is *context*
;;;    Return:
;;;    a tree like (A (B C (D E) F) G H)
;;;    where (D E) are children of C and (B C F) are children of A, and G
;;;    and H are parallel trees.
;;;
;;; %MAKE-ENTRY-SYMBOLS (multilingual-name &key type prefix          [FUNCTION]
;;;                       (package *package*))
;;;    Takes a multilingual-name as input and builds entry symbols for each
;;;    of the ~ synonyms in each specified language.
;;;    Note that a list of symbols is not a valid input argument.
;;;    Arguments:
;;;    multilingual-name: string, symbol or multilingual-name specifying
;;;    the entry points tp-id: local attibute identifier
;;;    type (key): type of MOSS object (default is nil)
;;;    prefix (key): prefix for produced symbols
;;;    Return:
;;;    list of the entry point symbols
;;;
```

```
;;; %MAKE-EP (entry tp-id obj-id &key (context *context*) export)    [FUNCTION]
;;;    Adds a new object to an entry-point if it exists - Otherwise creates
;;;    the entry point(s) and record it/them in the current system
;;;    (*moss-system*). Export the entry symbol(s).
;;;    Arguments:
;;;    entry: symbol or list of symbols specifying the entry point(s)
;;;    tp-id: local attibute identifier
;;;    obj-id: identifier of object to index
;;;    context (opt): context (default current)
;;;    export (key): it t export entry points from the package
;;;    Return:
;;;    internal format of the entry point object or a list of them
;;;
;;; %MAKE-EP-FROM-MLN (mln tp-id obj-id &key type                     [FUNCTION]
;;;                    (context *context*) export)
;;;    Takes a multilingual-name as input and builds entries for each of the
;;;    ~ synonyms in each specified LEGAL language in package of obj-id.
;;;    Arguments:
;;;    mln: legal multilingual-name specifying the entry points
;;;    tp-id: local attibute identifier
;;;    obj-id: identifier of object to index
;;;    context (opt): context (default current)
;;;    type (key): type of object, e.g. $ENT, $EPR, $FN, ... (see
;;;    %make-name) export (key): if t then we should export entry points
;;;    from their package Return:
;;;    list of the entry point object (internal format)
;;;
;;; %MAKE-ID (class-id &key name prefix value context id ideal       [FUNCTION]
;;;           (package *package*))
;;;    Generic function to make object identifiers. If the symbol exists,
;;;    then ~ we throw to :error.
;;;    Arguments:
;;;    class-id: identifier of the class of the object (*none* is
;;;    authorized) name (key): useful name, e.g.class  name or property name
;;;    prefix (key): additional name (string), e.g. for defining
;;;    properties (the class name) value (key): value for an instance
;;;    context (key): context default current
;;;    id (key): id of class or property
;;;    ideal (key): if there indicates we want an ideal, :id option must
;;;    be there package (key): package into which new symbol must be
;;;    inserted (default *package*) Return:
;;;    unique new symbol.
;;;
;;; %MAKE-INVERSE-PROPERTY (id multilingual-name                      [FUNCTION]
;;;                         &key (context *context*) export)
;;;    create an inverse property for an attribute or a relation. Links it
;;;    to the ~ direct property, creates the name and entry point, add it to
;;;    MOSS. It is created in the same package as id and in the specified
;;;    context. Uses the default language.
;;;    Arguments:
;;;    id: identifier of the property to invert
```

```
;;;    multilingual-name: name of the property to invert
;;;    Return:
;;;    :done
;;;
;;; %MAKE-INVERSE-PROPERTY-MLN (mln)                                    [FUNCTION]
;;;    build an inverse multilingual name for the inverse property.
;;;    Arguments:
;;;    mln: multilingual name
;;;    Result:
;;;    a list of strings and an inverse mln
;;;
;;; %MAKE-NAME (class-id &rest option-list &key name                    [FUNCTION]
;;;             (package *package*) &allow-other-keys)
;;;    Makes a name with %make-name-string and interns is in the specified
;;;    package. Arguments:
;;;    class-id: identifier of the corresponding class
;;;    package (key): package for interning symbol (default current
;;;    execution package) name (key): string, symbol or multilingual string
;;;     more keys are allowed as specified in the lambda-list of the
;;;    %make-string function Return:
;;;    interned symbol.
;;;
;;; %MAKE-NAME-STRING (class-id &key name method-class-id               [FUNCTION]
;;;                    method-type state-type short-name prefix
;;;                    (type "") (context *context*)
;;;                    &allow-other-keys &aux pfx)
;;;    Builds name strings for various moss entities, e.g. method names,
;;;    inverse-property ~ names, internal method function names, etc.
;;;    Examples of resulting strings
;;;        XXX-Y-ZZZ from " Xxx y'zzz "
;;;        IS-XXX-OF inverse property
;;;        HAS-XXX  property
;;;        $E-PERSON=S=0=PRINT-SELF own method internal function name
;;;        $E-PERSON=I=0=SUMMARY instance method internal function name
;;;       *0=PRINT-OBJECT universal method internal function name
;;;        $-PER typeless radix
;;;        _HAS-BROTHER  internal variable
;;;    We mix English prefix and suffix with other languages, which is
;;;    not crucial ~ since such names are internal to MOSS.
;;;    Arguments:
;;;    class-id: identifier of the corresponding class
;;;    name (key): name, e.g. of method
;;;    context (key): context reference
;;;    prefix (key): symbol, string or mln specifying a class (for class
;;;    properties) method-type (key): instance or own or universal
;;;    method-class-id (key): class-id for a given method
;;;    state-type: for state objects {:entry-state, :success, :failure}
;;;    short-name: for state objects, prefix
;;;    package (key): package for interning symbol (default current
;;;    execution package) context (key): context used for building some
;;;    names (default current) language (key): language to use, e.g. :en,
```

```
;;;    :fr (default *language*) Return:
;;;    a single name string
;;;
;;; %MAKE-PHRASE (&rest words)                                          [FUNCTION]
;;;    Takes a list of words and returns a string with the words separated
;;;    by a space. Arguments:
;;;    words (rest): strings
;;;    Return:
;;;    a string.
;;;
;;; %MAKE-RADIX-FROM-NAME (name &key (type "") (package *package*))  [FUNCTION]
;;;    build a radix name for a property or for a class. Tries to build a
;;;    unique name ~ using the first 3 letters of the name. Error if the
;;;    symbol already exists. Arguments:
;;;    name: e.g. "SIGLE"
;;;    type (opt): e.g. "T " (for terminal property)
;;;    Return:
;;;    e.g. $T-SIG or error if already exists
;;;
;;; %MAKE-REF-OBJECT (class-ref obj-id &key (context *context*))     [FUNCTION]
;;;    Builds a reference object, checking the validity of the class. If
;;;    invalid ~ throws to :error tag.
;;;    Arguments:
;;;    class-ref: reference of the new class
;;;    obj-id: id of object being referenced
;;;    context (opt): context, default current
;;;    Return:
;;;    id of new ref object.
;;;
;;; %MAKE-STRING-FROM-PATTERN (control-string pattern                [FUNCTION]
;;;                            &rest properties)
;;;    makes a string from a pattern (XML like). The first element of
;;;    pattern is the ~ class of the  object. The function ~
;;;    is somewhat equivalent to the =summary method, but the method is
;;;    applied to ~ the structured object.
;;;    Ex: (fn "~{~A~^ ~}, ~{~A~^ ~}"
;;;              '("person" ("name" "Dupond") ("first-name" "Jean"))
;;;              "name" "first-name")
;;;    returns "Dupond, Jean"
;;;    Arguments:
;;;    pattern: the pattern
;;;    data: the data
;;;    Return:
;;;    string to be printed.
;;;
;;; %MAKE-WORD-COMBINATIONS (word-list)                                 [FUNCTION]
;;;
;;; %MAKE-WORD-LIST (text &key (norm t))                                [FUNCTION]
;;;    Norms a text string by separating each word making it lower case with
;;;    a leading~ uppercased letter.
;;;    Arguments:
```

```
;;;    text: text string
;;;    norm (key): if true (default) capitalize the first letter of each
;;;    word Return:
;;;    a list of normed words.
;;;
;;; %%MERGE-OBJECTS (obj-id obj-list context)                    [FUNCTION]
;;;    obj-id is the id of an object already existing in memory when we are
;;;    loading a new object with same id from disk. The new format of the
;;;    loaded object is obj-list. We compare properties of the existing
;;;    (moss) object and the loaded object and update all properties that
;;;    are not system or that contain application references. For a tp, the
;;;    system object is the reference. The result is a merged object
;;;    including system and application data. Arguments:
;;;    obj-id: id of the object to be processed
;;;    obj-list: object saved by the application
;;;    Returns:
;;;    the list representing the system object merged with application
;;;    references.
;;;
;;; %MLN? (expr)                                                 [FUNCTION]
;;;    Checks whether a list is a multilingual name Uses global
;;;    *language-tags* variable. ~ nil is not a valid MLN.
;;;    Argument:
;;;    expr: something like (:en "London" :fr "Londres") or (:name :en
;;;    "London") Result:
;;;    T if OK, nil otherwise
;;;
;;; %MLN-1? (expr)                                               [FUNCTION]
;;;    Checks whether a list is a multilingual string. Uses global
;;;    *language-tags* variable. Argument:
;;;    expr: something like (:en "London" :fr "Londres")
;;;    Result:
;;;    T if OK, nil otherwise
;;;
;;; %MLN-ADD-VALUE (mln value language-tag)                      [FUNCTION]
;;;    Adds a value corresponding to a specific language at the end of the
;;;    list. ~ Does not check if value is already there.
;;;    Arguments:
;;;    mln: multilingual name
;;;    value: coerced to a string
;;;    language-tag: must be legal, i.e. part of *language-tags*
;;;    Return:
;;;    the modified mln.
;;;
;;; %MLN-EQUAL (mln1 mln2 &key language)                         [FUNCTION]
;;;    Checks whether two multilingual-names are equal. They may also be
;;;    strings. ~ When the language key argument is not there uses global
;;;    *language* variable.~ When *language* is unbound, tries all possible
;;;    languages. The presence of ~ language is only necessary if one of the
;;;    arguments is a string. The function uses %string-norm before
;;;    comparing strings. Argument:
```

```
;;;    mln1: a multilingual name or a string
;;;    mln2: id.
;;;    language (key): a specific language tag
;;;    Result:
;;;    T if OK, nil otherwise
;;;
;;; %MLN-EXTRACT-ALL-SYNONYMS (mln)                                    [FUNCTION]
;;;    Extracts all synonyms as a list of strings regardless of the
;;;    language. Arguments:
;;;    mln: multilingual name
;;;    Return
;;;    a list of strings.
;;;
;;; %MLN-EXTRACT-MLN-FROM-REF (ref language                            [FUNCTION]
;;;                            &key no-throw-on-error (default :en)
;;;                            &aux $$$)
;;;    takes a ref input symbol string or mln and builds a specific mln
;;;    restricted to ~ the specified language. If language is :all and ref
;;;    is a string or symbol, then ~ language defaults to default. If ref is
;;;    an MLN, then language defaults to default ~ if there, first found
;;;    language otherwise (canonical behavior). Default beavior ~ is enabled
;;;    only if no-throw-on-error option is set to T. Removes also the
;;;    leading :name marker if present in mln. Arguments:
;;;    ref: a string symbol or mln
;;;    language a language tag or :all
;;;    no-throw (key): if true function does not throw on error but
;;;    returns nil default (key): default language tag (default: :EN)
;;;    Result:
;;;    a valid mln or throws to :error
;;;
;;; %MLN-FILTER-LANGUAGE (mln language-tag &key always)                [FUNCTION]
;;;    Extracts from the MLN the synonym string corresponding to specified
;;;    language. ~ If mln is a string or language is :all, then returns mln
;;;    untouched, unless always ~ is true in which case returns the English
;;;    terms if there, otherwise random terms. If language is not legal,
;;;    throws to :error. Arguments:
;;;    mln: a multilingual name (can also be a simple string)
;;;    language-tag: a legal language tag (part of *language-tags*)
;;;    always (key): if t then returns either the English terms or a
;;;    random one Return:
;;;    a string or nil if none.
;;;
;;; %MLN-GET-CANONICAL-NAME (mln &aux names)                           [FUNCTION]
;;;    Extracts from a multilingual name the canonical name that will be
;;;    used to build ~ an object ID. By default it is the first name
;;;    corresponding to the value of *language*, ~ or else the first name of
;;;    the English entry, or else the name of the list. ~ An error occurs
;;;    when the argument is not multilingual name. Arguments:
;;;    mln: a multilingual name
;;;    Return:
;;;    2 values: a simple string (presumably for building an ID) and the
```

```
;;;    language tag. Error:
;;;    error if not a multilingual name.
;;;
;;; %MLN-GET-FIRST-NAME (name-string)                                [FUNCTION]
;;;    Extracts the first name from the name string. Names are separated by
;;;    semi-columns. E.g. "identity card ; papers" returns "identity card "
;;;    Argument:
;;;    name-string
;;;    Return:
;;;    string with the first name.
;;;
;;; %MLN-MEMBER (input-string tag mln)                               [FUNCTION]
;;;    checks whether the input string is one of the synonyms of the mln in
;;;    the ~ language specified by tag. If tag is :all then we check against
;;;    any synonym ~ in any language.
;;;    Arguments:
;;;    input-string: input string (to be normed with %norm-string)
;;;    tag; a legal language tag or :all
;;;    mln: a mltilingual name
;;;    Return:
;;;    non nil value if true.
;;;
;;; %MLN-MERGE (&rest mln)                                           [FUNCTION]
;;;    Takes a list of MLN and produces a single MLN as a result, merging
;;;    the ~ synonyms (using or norm-string comparison) and keeping the
;;;    order of the ~ inputs.
;;;    Arguments:
;;;    mln: a multilingual name
;;;    more-mln (rest): more multilingual names
;;;    Return:
;;;    a single multilingual name
;;;    Error:
;;;    in case one of the mln has a bad format.
;;;
;;; %MLN-NORM (expr)                                                 [FUNCTION]
;;;    Norms an mln expr by removing the :name keyword if there.
;;;    Argument:
;;;    expr: expression to test
;;;    Return:
;;;    normed expr, nil is an error)
;;;
;;; %MLN-PRINT-STRING (mln &optional (language-tag :all))            [FUNCTION]
;;;    Prints a multilingual name nicely.
;;;    Arguments:
;;;    mln: multilingual name
;;;    language-tag: legal language-tag (default :all)
;;;    Result:
;;;    a string ready to be printed (can be empty).
;;;
;;; %MLN-REMOVE-LANGUAGE (mln language-tag)                          [FUNCTION]
;;;    Removes the entry corresponding to a language from a multilingual
```

```
;;;    name. Arguments:
;;;    mln: multilanguage name
;;;    language-tag: language to remove (must be legal
;;;    Return:
;;;    modified mln.
;;;
;;; %MLN-REMOVE-VALUE (mln value language-tag)                    [FUNCTION]
;;;    Removes a value in the list of synonyms of a particular language.
;;;    Arguments:
;;;    mln: multilingual name
;;;    value: coerced to a string (loose package context)
;;;    language-tag: must be legal
;;;    Return:
;;;    the modified mln.
;;;
;;; %MLN-SET-VALUE (mln language-tag syn-string)                   [FUNCTION]
;;;    Sets the language part to the specified value, erasing the old value.
;;;    Arguments:
;;;    mln: a multilingual name
;;;    language-tag: language tag
;;;    syn-string: coerced to a string (loose package context)
;;;    Return:
;;;    the modified normed MLN.
;;;
;;; %MULTILINGUAL-NAME? (expr)                                    [FUNCTION]
;;;    Checks whether a list is a multilingual string. Uses global
;;;    *language-tags* variable. Argument:
;;;    expr: something like (:en "London" :fr "Londres")
;;;    Result:
;;;    T if OK, nil otherwise.
;;;
;;; %PCLASS? (input class-ref)                                    [FUNCTION]
;;;    tests if input belongs to the transitive closure of class-ref. Only
;;;    works in ~ if the language is that if the input and class ref.
;;;    Arguments:
;;;    input: some expression that must be a string
;;;    class-ref: a string naming a class
;;;    Return:
;;;    nil if not the case, something not nil otherwise.
;;;
;;; %PDM? (xx)                                                    [FUNCTION]
;;;    Checks if an object had the right PDM format, i.e. it must be a
;;;    symbol, bound, ~ be an alist, and have a local $TYPE property.
;;;    References are PDM objects
;;;
;;; %PEP (obj-id &key (version *context*) (offset 30) (stream t))    [FUNCTION]
;;;    Prints versions of values from a given context up to the root.
;;;       (%pep obj-id version &rest offset)
;;;       For each property prints values in all previous context up to the
;;;    root ~ does not check for illegal context since it is a printing
;;;    function.
```

```
;;;
;;; %PEP-PV (value context-branch &optional (stream t) offset)       [FUNCTION]
;;;    Used by %pep, prints ~
;;;         a set of value associated with a property from the root to
;;;    current context. Offset is currently set to 30.
;;;
;;; %PFORMAT (fstring input &rest prop-list)                         [FUNCTION]
;;;    produces a string for printing from a list produced typically by the
;;;    ~ =make-print-string method.
;;;    Arguments:
;;;    input: a-list, e.g. (("name" "Dupond")("first name" "Jean")...)
;;;    fstring: string control format
;;;    prop-list: a list of properties appearing in the a-list. last
;;;    entry can be the specification of a particular package
;;;    Return:
;;;    a string: e.g. "Dupond, Jean"
;;;
;;; %PUTC (obj-id value prop-id context)                             [FUNCTION]
;;;    Stores a versioned value onto the p-list of the object used as a
;;;    cache.
;;;
;;; %PUTM (object-id function-name method-name context               [FUNCTION]
;;;        &rest own-or-instance)
;;;    Records the function name corresponding to the method onto the ~
;;;         p-list of the object. Own-or-instance can be either one of the
;;;    ~ keywords :own or :instance; If not present, then :general is used.
;;;
;;; %RANK-ENTITIES-WRT-WORD-LIST (entity-list word-list)             [FUNCTION]
;;;    takes a list of entity ids and a list of words, and returns a score
;;;    for ~ each entity that is function of the number of apparitions of
;;;    the words in ~ attributes of the entity.
;;;    Arguments:
;;;    entity-list: a list of object ids (ei)
;;;    word-list: a list of words (wordj)
;;;    Returns:
;;;    a list of pairs (ei wi), e.g. (($e-person.2 0.75)($E-PERSON.3
;;;    0.5)).
;;;
;;; %REMOVE-REDUNDANT-PROPERTIES (prop-list                          [FUNCTION]
;;;                               &key (context *context*))
;;;    Removes properties that share the same generic property and keep the
;;;    leftmost ~ one.
;;;    Arguments:
;;;    prop-list: list of property ids
;;;    context (opt): context (default current)
;;;    Return:
;;;    a cleaned list of properties.
;;;
;;; %%REMPROP (obj-id prop-id context)                               [FUNCTION]
;;;    Removes the values associated with ~
;;;         the specified property in the specified context. The net result
```

```
;;;    is ~ that the object inherits the values from a higher level.
;;;       So the resulting value in the context will probably not be nil.
;;;    Hence ~ it is not equivalent to a set-value with a nil argument.
;;;
;;; %%REMVAL (obj-id val prop-id context)                          [FUNCTION]
;;;    Removes the value associated with ~
;;;         a property in a given context. Normally useful for terminal
;;;    properties ~ However does not check for entry-points. Also val must
;;;    be a normalized value n i.e. must be expressed using the internal
;;;    format (it cannot be an external value).
;;;
;;; %%REMNTHVAL (obj-id val prop-id context position)              [FUNCTION]
;;;    Removes the nth value associated ~
;;;         with a property in a given context. Normally useful for
;;;    terminal properties ~ However does not check for entry-points. Does
;;;    not return any significant value.
;;;
;;; %RESET (&key (context *context* there?))                       [FUNCTION]
;;;    Removes all classes and instances of an application, making the
;;;    corresponding ~ symbols unbound. Protects all data defined in the
;;;    kernel (moss  package). Arguments:
;;;    none
;;;    Return:
;;;    t
;;;
;;; %RESOLVE (id &key (context *context*))                         [FUNCTION]
;;;    Replaces id with the actual id when it is a ref object. Otherwise
;;;    returns id. When id is unbound returns id.
;;;    Arguments:
;;;    id: object id or REF identifier
;;;    Return:
;;;    resolved id.
;;;
;;; %SELECT-BEST-ENTITIES (entity-score-list)                      [FUNCTION]
;;;    select entities with highest score and return the list.
;;;    Arguments:
;;;    entity-score-list: a list like (($E-PERSON.2 0.5)(..)...)
;;;    Returns:
;;;    the list of entities with the highest score.
;;;
;;; %%SET-SYMBOL-PACKAGE (symbol package)                          [FUNCTION]
;;;    if package is nil set it to default *package*, and create symbol in
;;;    the ~ specified package. No check on input.
;;;    Arguments:
;;;    symbol: any symbol
;;;    package: a valid package or keyword
;;;    Return:
;;;    a symbol interned in the package.
;;;
;;; %%SET-VALUE (obj-id value prop-id context)                     [FUNCTION]
;;;    Resets the value associated ~
```

```
;;;         with a property in a given context. Replaces previous values.
;;;    Does not ~ try to inherit using the version-graph, but defines a new
;;;    context locally ~ Thus, it is not an %add-value, but an actual
;;;    %%set-value.
;;;
;;; %%SET-VALUE-LIST (obj-id value-list prop-id context)            [FUNCTION]
;;;    Resets the value associated ~
;;;         with a property in a given context. Replaces previous values.
;;;    Does not ~ try to inherit using the version-graph, but defines a new
;;;    context locally ~ Thus, it is not an %add-value, but an actual
;;;    %%set-value - ~ No entry point is created.
;;;
;;; %SP-GAMMA (node arc-label &key (context *context*))             [FUNCTION]
;;;    Returns the transitive closure ~
;;;         of objects for a given version.
;;;       This is the famous gamma function found in graph theory.
;;;       No check that arc-label is a structural property.
;;;    Arguments:
;;;    node: starting node
;;;    arc-label: label to follow on the arcs (e.g. $IS-A)
;;;    context (opt): context, default value is *context* (current)
;;;
;;; %SP-GAMMA-L (node-list arc-label &key (context *context*))      [FUNCTION]
;;;    Computes a transitive closure
;;;    Arguments:
;;;    node-list: a list of nodes from which to navigate
;;;    arc-label: the label of the arc we want to use
;;;    context (opt): context default current
;;;    Return:
;;;    a list of nodes including those of the node-list.
;;;
;;; %SP-GAMMA1 (ll prop stack old-set version)                      [FUNCTION]
;;;    Called by %sp-gamma for computing a transitive closure.
;;;       gamma1 does the job.
;;;       computes the whole stuff depth first using a stack of yet
;;;    unprocessed ~ candidates.
;;;     ll contains the list of nodes being processed
;;;     stack the ones that are waiting
;;;     old-set contains partial results.
;;;    Arguments:
;;;    ll: candidate list of nodes to be examined
;;;    prop: property along which we make the closure
;;;    stack: stack of waiting candidates
;;;    old-set: list containing the result
;;;    version: context
;;;    Return:
;;;    list of nodes making the transitive closure.
;;;
;;; STR-EQUAL (aa bb)                                               [FUNCTION]
;;;
;;; %STRING-NORM (input &optional (interchar #\-))                  [FUNCTION]
```

```
;;;    Same as make-value-string but with more explicit name.
;;;    If input is a multilingual string, uses the canonical name.
;;;    If input is NIL throws to :error
;;;    Arguments:
;;;    input: may be a string, a symbol or a multilingual string
;;;    interchar: character to hyphenate the final string (defaul is #-
;;;    Return:
;;;    a normed string, e.g. "AU-JOUR-D-AUJOURD-HUI ".
;;;
;;; %SYNONYM-ADD (syn-string value)                                [FUNCTION]
;;;    Adds a value to a string at the end.
;;;    Sends a warning if language tag does not exist and does not add
;;;    value. Arguments:
;;;    syn-string: a synonym string or nil
;;;    value: value to be included (coerced to string)
;;;    Return:
;;;    modified string when no error, original string otherwise.
;;;
;;; %SYNONYM-EXPLODE (text)                                        [FUNCTION]
;;;    Takes a string, consider it as a synonym string and extract items
;;;    separated ~ by a semi-column. Returns the list of string items.
;;;    Arguments:
;;;    text: a string
;;;    Return
;;;    a list of strings.
;;;
;;; %SYNONYM-MAKE (&rest item-list)                                [FUNCTION]
;;;    Builds a synonym string with a list of items . Eadh item is coerced
;;;    to a string. Arguments:
;;;    item-list: a list of items
;;;    Return:
;;;    a synonym string.
;;;
;;; %SYNONYM-MEMBER (value text)                                   [FUNCTION]
;;;    Checks if a string is part of the synonym list. Uses the %string-norm
;;;    ~ function to normalize the strings before comparing.
;;;    Arguments:
;;;    value: a value coerced to a string
;;;    text: a synonym string
;;;    Return:
;;;    a list of unnormed remaining synonyms if succcess, NIL if nothing
;;;    left.
;;;
;;; %SYNONYM-MERGE-STRINGS (&rest names)                           [FUNCTION]
;;;    Merges several synonym string, removing duplicates (using a
;;;    norm-string ~ comparison) and preserving the order.
;;;    Arguments:
;;;    name: a possible complex string
;;;    more-names (rest): more of that
;;;    Return:
;;;    a single complex string
```

```
;;;    Error:
;;;    when some of the aguments are not strings.
;;;
;;; %SYNONYM-REMOVE (value text)                                    [FUNCTION]
;;;    Removes a synonym from the list of synonyms. If nothing is left
;;;    return ~ empty string.
;;;    Arguments:
;;;    value: a value coerced to a string
;;;    text: a synonym string
;;;    Return:
;;;    the string with the value removed NIL if nothing left.
;;;
;;; %SUBTYPE? (object-id model-id &key (context *context*))          [FUNCTION]
;;;    Checks if object-id is a sub-type of model-id including itself.
;;;    Arguments:
;;;    object-id: identifier of object to check for modelhood
;;;    model-id: reference-model
;;;    context (opt): context default current
;;;    Return:
;;;    T or nil.
;;;
;;; %TYPE? (object-id class-id &key (context *context*))             [FUNCTION]
;;;    Checks if one of the classes of object-id is class-id or one of its
;;;    ~ subclasses.
;;;       If class is *any* then the result is true.
;;;    Arguments:
;;;    object-id: the id of a pdm object
;;;    class-id: the id of a class
;;;    Returns
;;;    nil or T.
;;;
;;; %TYPE-OF (obj-id &key (context *context*))                       [FUNCTION]
;;;    Extracts the type from an object (name of its class(es)).
;;;    Careful: returns "CONS" if following a programming error ~
;;;         obj-id pass the value of the identifier...
;;;    Argument:
;;;    obj-id: identifier of a particular object
;;;    Return:
;;;    a list containing the identifiers of the class(es) of which the
;;;    object is an instance ~ or its lisp type if it is not a PDM object.
;;;
;;; %UNLINK (obj1-id sp-id obj2-id &key (context *context*))         [FUNCTION]
;;;    Disconnects to objects by removing the structural link.
;;;         in the given context. If the context does not exist then one
;;;    has to ~ recover the states of the links in the required context.
;;;    Then if the ~ link did not exist, then one exits doing nothing.
;;;    Otherwise, one removes ~ the link and stores the result.
;;;
;;; %VALIDATE-SP (sp-id suc-list &key (context *context*))           [FUNCTION]
;;;    Checks if the property successors obey the restrictions attached to
;;;    the relation. Arguments:
```

```
;;;    sp-id: id of the relation
;;;    suc-list: list of successors to check
;;;    context (opt): context (default current)
;;;    Return:
;;;    2 values:
;;;          1st value value-list if OK, nil otherwise
;;;          2nd value: list of string error message.
;;;
;;; %VALIDATE-TP (tp-id value-list &key (context *context*))         [FUNCTION]
;;;    Checks if the property values obey the restrictions attached to the
;;;    attribute. Arguments:
;;;    tp-id: id of the attribute
;;;    value-list: list of values to check
;;;    context (opt): context (default current)
;;;    Return:
;;;    2 values:
;;;          1st value value-list if OK, nil otherwise
;;;          2nd value: list of string error message.
;;;
;;; %VALIDATE-TP-VALUE (restriction data &key prop-id obj-id         [FUNCTION]
;;;                      (context *context*))
;;;    Takes some data and checks if they agree with a value-restriction
;;;    condition,~ associated with a terminal property. Used by the =xi
;;;    method whenever ~ we want to put data as the value of the property.
;;;    Processed conditions are those of *allowed-restriction-operators*
;;;    Arguments:
;;;    restriction: restriction condition associated with the property,
;;;    e.g. (:one-of (1 2 3)) data: a list of values to be checked
;;;    prop-id (key): identifier of the local property (unused)
;;;     obj-id (key): identifier of the object being processed (unued)
;;;    context (key): context default current (unused)
;;;    Return:
;;;    data when validated, nil otherwise.
;;;
;;; %VG-GAMMA (context)                                              [FUNCTION]
;;;    Computes the transitive closure upwards on the version graph.
;;;       Get the transitive closure of the context i.e. a merge of all the
;;;    paths ~ from context to root without duplicates, traversed in a
;;;    breadth first ~ fashion, since we assume that the value should not be
;;;    far from our context ~ This list should be computed once for all when
;;;    the context is set ~ e.g. in a %set-context function
;;;    Argument:
;;;    context: starting context
;;;    Return:
;;;    a list of contexts.
;;;
;;; %VG-GAMMA-L (lres lnext ll pred)                                 [FUNCTION]
;;;    Service function used be %vg-gamma
;;;    Arguments:
;;;    lres: resulting list
;;;    lnext: next candidates to examine later
```

```
;;;    ll: working list of candidates
;;;    pred: list of future nodes to process
;;;    Return:
;;;    lres
;;;
;;; %%ZAP-OBJECT (obj-id)                                          [FUNCTION]
;;;    Deletes an object by removing all values from all contexts and making
;;;    its ~ identifier unbound. This can destroy the database consistency.
;;;    Arguments:
;;;    obj-id: identifier of the object
;;;    Return:
;;;    not significant.
;;;
;;; %ZAP-PLIST (obj-id)                                            [FUNCTION]
;;;    Clears the plist of a given symbol.
;;;
```