RacerPro Reference Manual
Version 1.9.2

Racer Systems GmbH & Co. KG

October 18, 2007

Contents

Chapter 1

Knowledge Base Management Functions

A knowledge base is just a tuple consisting of a TBox and an associated ABox. Note that a TBox and its associated ABox may have the same name. This section documents the functions for managing TBoxes and ABoxes and for specifying queries.

Racer provides a default knowledge base with a TBox called `default` and an associated ABox with the same name.

in-knowledge-base *macro*

Description: This form is an abbreviation for the sequence:
`(in-tbox` *TBN*`)`
`(in-abox` *ABN* *TBN*`)`. See the appropriate documentation for these functions.

Syntax: Two forms are possible:
`(in-knowledge-base` *TBN* `&optional` *ABN*`)` or
`(in-knowledge-base` *TBN* `&key (`*init* `t))`

Arguments: *TBN* - TBox name

ABN - ABox name

init - `t` or `nil`

Remarks: If no ABox is specified an ABox with the same name as the TBox is created (or initialized if already present). The ABox is associated with the TBox. If the keyword `:init` is specified with value `nil` no new knowledge base is created but just the current TBox and ABox is set. If `:init` is specified, no ABox name may be given.

Examples:
```
(in-knowledge-base peanuts peanuts-characters)
  (in-knowledge-base peanuts)
  (in-knowledge-base peanuts :init nil)
```

racer-read-file *function*

Description: A file in RACER format (as described in this document) containing TBox and/or ABox declarations is loaded.

Syntax: `(racer-read-file` *pathname*`)`

Arguments: *pathname* - is the pathname of a file

Examples: `(racer-read-file "kbs/test.lisp")`

See also: Function `include-kb`

racer-read-document *function*

Description: A file in RACER format (as described in this document) containing TBox and/or ABox declarations is loaded.

Syntax: `(racer-read-document` *URL*`)`

Arguments: *URL* - is the URL of a text document with RACER statements.

Remarks: The URL can also be a file URL. In this case, racer-read-file is used on the pathname of the URL.

Examples:
```
(racer-read-document "http://www.fh-wedel.de/mo/test.lisp")
  (racer-read-document "file:///home/mo/kbs/test.lisp")
```

See also: Function `racer-read-file`

include-kb *function*

Description: A file in RACER format (as described in this document) containing TBox and/or ABox declarations is loaded. The function include is used for partitioning a TBox or ABox into several files.

Syntax: `(include-kb` *pathname*`)`

Arguments: *pathname* - is the pathname of a file

Examples:
```
(include-kb "project:onto-kb;my-knowledge-base.lisp")
```

See also: Function `racer-read-file`

import-kb *macro*

Description: Macro equivalent of `racer-read-file`, Page 2.

daml-read-file *function*

Description: A file in DAML format (e.g., produced OilEd) is loaded and represented as a TBox and an ABox with appropriate declarations.

Syntax: `(daml-read-file` *pathname* `&key (`*init* `t) (`*verbose* `nil) (`*kb-name* `nil)))`

Arguments: *pathname* - is the pathname of a file

init - specifies whether the kb is initialized or extended (the default is to (re-)initialize the kb.

verbose - specifies whether ignored triples are indicated (the default is to just suppress any warning).

kb-name - specifies the name of the kb (TBox and ABox). The default is the file specified in the *pathname* argument (without file type).

Examples: `(daml-read-file "oiled:ontologies;madcows.daml")` reads the file `"oiled:ontologies;madcows.daml"` and creates a TBox `madcows` and an associated ABox `madcows`.

daml-read-document *function*

Description: A text document in DAML format (e.g., produced OilEd) is loaded from a web server and represented as a TBox and an ABox with appropriate declarations.

Syntax: `(daml-read-document` *URL* `&key (`*init* `t) (`*verbose* `nil) (`*kb-name* `nil)))`

Arguments: *URL* - is the URL of a text document

init - specifies whether the kb is initialized or extended (the default is to (re-)initialize the kb.

verbose - specifies whether ignored triples are indicated (the default is to just suppress any warning).

kb-name - specifies the name of the kb (TBox and ABox). The default is the document name specified in the *URL* argument (without file type).

Examples: `(daml-read-document "http://www.fh-wedel.de/mo/madcows.daml")` reads the specified text document from the corresponding web server and creates a TBox `madcows` and an associated ABox `madcows`. A file URL may also be specified `(daml-read-document "file://mo/madcows.daml")`

owl-read-file *function*

Description: A file in OWL format (e.g., produced OilEd) is loaded and represented as a TBox and an ABox with appropriate declarations.

Syntax: `(owl-read-file` *pathname* `&key (`*init* `t) (`*verbose* `nil) (`*kb-name* `nil)))`

Arguments: *pathname* - is the pathname of a file

init - specifies whether the kb is initialized or extended (the default is to (re-)initialize the kb.

verbose - specifies whether ignored triples are indicated (the default is to just suppress any warning).

kb-name - specifies the name of the kb (TBox and ABox). The default is the file specified in the *pathname* argument (without file type).

Examples: `(owl-read-file "oiled:ontologies;madcows.owl")` reads the file `"oiled:ontologies;madcows.owl"` and creates a TBox `madcows` and an associated ABox `madcows`.

owl-read-document *function*

Description: A text document in OWL format (e.g., produced OilEd) is loaded from a web server and represented as a TBox and an ABox with appropriate declarations.

Syntax: `(owl-read-document` *URL* `&key (`*init* `t) (`*verbose* `nil) (`*kb-name* `nil)))`

Arguments: *URL* - is the URL of a text document

init - specifies whether the kb is initialized or extended (the default is to (re-)initialize the kb.

verbose - specifies whether ignored triples are indicated (the default is to just suppress any warning).

kb-name - specifies the name of the kb (TBox and ABox). The default is the document name specified in the *URL* argument (without file type).

Examples: `(owl-read-document "http://www.fh-wedel.de/mo/madcows.owl")` reads the specified text document from the corresponding web server and creates a TBox `madcows` and an associated ABox `madcows`. A file URL may also be specified `(owl-read-document "file://mo/madcows.owl")`

mirror *function*

Description: If you are offline, importing OWL or DAML ontologies may cause problems. However, editing documents and inserting local URLs for ontologies is inconvenient. Therefore, Racer provides a facility to declare local mirror URLs for ontology URLs

Syntax: `(mirror` *URL mirror − URL*`)`

Arguments: *URL* - a URL used to refer to an ontology in a DAML-OIL or OWL document

mirror − URL - a URL that refers to the same ontology. Possibly, a file URL may be supplied.

clear-mirror-table *function*

Description: Delete all mirror entries

Syntax: `(clear-mirror-table)`

Arguments:

dig-read-file *function*

Description: A file in dig format (e.g., produced OilEd) is loaded and represented as a TBox and an ABox with appropriate declarations.

Syntax: `(dig-read-file` *pathname* `&key (`*init* `t) (`*verbose* `nil) (`*kb-name* `nil)))`

Arguments: *pathname* - is the pathname of a file

init - specifies whether the kb is initialized or extended (the default is to (re-)initialize the kb.

verbose - specifies whether ignored triples are indicated (the default is to just suppress any warning).

kb-name - specifies the name of the kb (TBox and ABox). The default is the file specified in the *pathname* argument (without file type).

Examples: `(dig-read-file "oiled:ontologies;madcows.dig")` reads the file `"oiled:ontologies;madcows.dig"` and creates a TBox `madcows` and an associated ABox `madcows`.

dig-read-document *function*

Description: A text document in dig format (e.g., produced OilEd) is loaded from a web server and represented as a TBox and an ABox with appropriate declarations.

Syntax: `(dig-read-document` *URL* `&key (`*init* `t) (`*verbose* `nil) (`*kb-name* `nil)))`

Arguments: *URL* - is the URL of a text document

init - specifies whether the kb is initialized or extended (the default is to (re-)initialize the kb.

verbose - specifies whether ignored triples are indicated (the default is to just suppress any warning).

kb-name - specifies the name of the kb (TBox and ABox). The default is the document name specified in the *URL* argument (without file type).

Examples: `(dig-read-document "http://www.fh-wedel.de/mo/madcows.dig")` reads the specified text document from the corresponding web server and creates a TBox `madcows` and an associated ABox `madcows`. A file URL may also be specified `(dig-read-document "file://mo/madcows.dig")`

kb-ontologies *function*

Description: A document in DAML+OIL or OWL format can import other ontologies. With this function one can retrieve all ontologies that were imported into the specified knowledge base

Syntax: `(kb-ontologies` *KBN*`)`

Arguments: *KBN* - is the name of the knowledge base.

get-namespace-prefix *function*

Description: Returns the prefix of the default namespace of a TBox loaded from an OWL resource.

Syntax: `(get-namespace-prefix` *TBN*`)`

Arguments: *TBN* - TBox name

save-kb *function*

Description: If a pathname is specified, a TBox is saved to a file. In case a stream is specified the TBox is written to the stream (the stream must already be open) and the keywords *if-exists* and *if-does-not-exist* are ignored.

Syntax:
```
(save-kb pathname-or-stream
  &key (tbox (current-tbox)) (abox (current-abox))
  (syntax :krss) (if-exists :supersede)
  (if-does-not-exist :create)
  (uri "")
  (ns0 ""))
```

Arguments: *pathname-or-stream* - is the pathname of a file or is an output stream

tbox - TBox name or TBox object

abox - ABox name or ABox object

syntax - indicates the syntax of the KB to be generated. Possible values for the *syntax* argument are `:krss` (the default), `:xml`, or `:daml`. Note that concerning KRSS only a KRSS-like syntax is supported by RACER. Therefore, instead of `:krss` it is also possible to specify `:racer`.

if-exists - specifies the action taken if a file with the specified name already exists. All keywords for the Lisp function `with-open-file` are supported. The default is `:supersede`.

if-does-not-exist - specifies the action taken if a file with the specified name does not yet exist. All keywords for the Lisp function `with-open-file` are supported. The default is `:create`.

uri - The keyword `:uri` specifies the URI prefix for names. It is only available if syntax `:daml` is specified. This argument is useful in combination with OilEd. See the OilEd documentation.

ns0 - The keyword `:uri` is also provided for generating DAML files to be processed with OilEd. The keyword `:ns0` specifies the name of the OilEd namespace 0. This keyword is important for the ABox part. If the value of `:uri` is `/home/user/test#`, the value of `:ns0` should probably be `/home/user/`. Some experimentation might be necessary to find the correct values for `:uri` and `:ns0` to be used with OilEd.

Examples:
```
(save-kb "project:onto-kb;my-knowledge-base.krss"
  :syntax :krss
  :tbox 'family
  :abox 'smith-family)
```

```
(save-kb "family.daml" :syntax :daml

  :tbox 'family
  :abox 'smith-family
  :uri "http://www.fh-wedel.de/family.daml")
  :ns0 "http://www.fh-wedel.de/")
```

1.1 TBox Management

If RACER is started, there exists a TBox named DEFAULT, which is set to the current TBox.

---

in-tbox *macro*

Description: The TBox with the specified name is taken or a new TBox with that name is generated.

Syntax: `(in-tbox` *TBN* `&key (`*init* `t))`

Arguments: *TBN* - is the name of the TBox.

init - boolean indicating if the TBox should be initialized.

Values: TBox object named *TBN*

Remarks: Usually this macro is used at top of a file containing a TBox. This macro can also be used to create new TBoxes.

The specified TBox is the `(current-tbox)` until `in-tbox` is called again.

Examples:
```
(in-tbox peanuts)
(implies Piano-Player Character)
                ⋮
```

See also: Macro `signature` on page 12.

init-tbox *function*

Description: Generates a new TBox or initializes an existing TBox. During the initialization all user-defined concept axioms and role declarations are deleted, only the concepts `*top*` and `*bottom*` remain in the TBox.

Syntax: `(init-tbox` *tbox*`)`

Arguments: *tbox* - TBox object

Values: *tbox*

Remarks: This is the way to create a new TBox object.

signature *macro*

Description: Defines the signature for a knowledge base.

If any keyword except *individuals* or *objects* is used, the `(current-tbox)` is initialized and the signature is defined for it.

If the keyword *individuals* or *objects* is used, the `(current-abox)` is initialized. If all keywords are used, the `(current-abox)` and its TBox are both initialized.

Syntax: `(signature &key (`*atomic-concepts* `nil) (`*roles* `nil)`
`(`*transitive-roles* `nil) (`*features* `nil) (`*attributes* `nil)`
`(`*individuals* `nil) (`*objects* `nil))`

Arguments: *atomic-concepts* - is a list of all the concept names, specifying $\mathcal{C}$.

roles - is a list of role declarations.

transitive-roles - is a list of transitive role declarations.

features - is a list of feature declarations.

attributes - is a list of attributes declarations.

individuals - is a list of individual names.

objects - is a list of object names.

Remarks: Usually this macro is used at top of a file directly after the macro `in-knowledge-base`, `in-tbox` or `in-abox`.

Actually it is not necessary in RACER to specify the signature, but it helps to avoid errors due to typos.

Examples: Signature for a TBox:

```
(signature
  :atomic-concepts (Character Baseball-Player...)
  :roles ((has-pet)
    (has-dog :parents (has-pet) :domain human :range dog)
    (has-coach :feature t))
  :attributes ((integer has-age) (real has-weight)))
```

Signature for an ABox:

```
(signature
  :individuals (Charlie-Brown Snoopy ...)
  :objects (age-of-snoopy ...))
```

Signature for a TBox and an ABox:

```
(signature
  :atomic-concepts (Character Baseball-Player...)
  :roles ((has-pet)
    (has-dog :parents (has-pet) :domain human :range dog)
    (has-coach :feature t))
  :attributes ((integer has-age) (real has-weight))
  :individuals (Charlie-Brown Snoopy ...)
  :objects (age-of-snoopy ...))
```

See also: For role definitions see `define-primitive-role`, on page 35, for feature definitions see `define-primitive-attribute`, on page 36, for attribute definitions see `define-concrete-domain-attribute`, on page 44.

ensure-tbox-signature *function*

Description: Defines the signature for a TBox and initializes the TBox.

Syntax: `(ensure-tbox-signature` *tbox* `&key (`*atomic-concepts* `nil)`
`(`*roles* `nil) (`*transitive-roles* `nil) (`*features* `nil) (`*attributes* `nil))`

Arguments: *tbox* - is a TBox name or a TBox object.

atomic-concepts - is a list of all the concept names.

roles - is a list of all role declarations.

transitive-roles - is a list of transitive role declarations.

features - is a list of feature declarations.

attributes - is a list of attributes declarations.

See also: Definition of macro `signature`.

get-tbox-signature *function*

Description: Gets the signature for a TBox.

Syntax: `(get-tbox-signature &optional` *tbox*`)`

Arguments: *tbox* - is a TBox name or a TBox object.

current-tbox *function*

Description: The function returns a TBox name.

Syntax: `(current-tbox)`

Arguments:

set-current-tbox *function*

Description: The function sets the current TBox.

Syntax: `(set-current-tbox` *tbox*`)`

Arguments:

get-tbox-version *Function*

Description: Gets a version indicator for a TBox.

Syntax: `(get-tbox-version` *tbox*`)`

Arguments: *tbox* - is a TBox name or a TBox object.

save-tbox *function*

Description: If a pathname is specified, a TBox is saved to a file. In case a stream is specified the TBox is written to the stream (the stream must already be open) and the keywords *if-exists* and *if-does-not-exist* are ignored.

Syntax: `(save-tbox` *pathname-or-stream* `&optional (`*tbox* `(current-tbox))`
`&key (`*syntax* `:krss) (`*transformed* `nil) (`*if-exists* `:supersede)`
`(`*if-does-not-exist* `:create)`
`(`*uri* `""))`

Arguments: *pathname-or-stream* - is the pathname of a file or is an output stream

tbox - TBox object

syntax - indicates the syntax of the KB to be generated. Possible values for the *syntax* argument are `:krss` (the default), `:xml`, or `:daml`. Note that only a KRSS-like syntax is supported by RACER. Therefore, instead of `:krss` it is also possible to specify `:racer`.

if-exists - specifies the action taken if a file with the specified name already exists. All keywords for the Lisp function `with-open-file` are supported. The default is `:supersede`.

if-does-not-exist - specifies the action taken if a file with the specified name does not yet exist. All keywords for the Lisp function `with-open-file` are supported. The default is `:create`.

Values: TBox object

Remarks: A file may contain several TBoxes.
The usual way to load a TBox file is to use the Lisp function `load`.
If the server version is used, it must have been started with the option -u in order to have this function available.

Examples:
```
(save-tbox "project:TBoxes;tbox-one.lisp")
(save-tbox "project:TBoxes;final-tbox.lisp"
  (find-tbox 'tbox-one) :if-exists :error)
```

forget-tbox *function*

Description: Delete the specified TBox from the list of all TBoxes. Usually this enables the garbage collector to recycle the memory used by this TBox.

Syntax: `(forget-tbox` *tbox*`)`

Arguments: *tbox* - is a TBox object or TBox name.

Values: List containing the name of the removed TBox and a list of names of optionally removed ABoxes

Remarks: All ABoxes referencing the specified TBox are also deleted.

Examples: `(forget-tbox 'smith-family)`

delete-tbox *macro*

Description: Delete the specified TBox from the list of all TBoxes. Usually this enables the garbage collector to recycle the memory used by this TBox.

Syntax: `(delete-tbox` *TBN*`)`

Arguments: *TBN* - is a TBox name.

Values: List containing the name of the removed TBox and a list of names of optionally removed ABoxes

Remarks: Calls `forget-tbox`

Examples: `(delete-tbox smith-family)`

delete-all-tboxes *function*

Description: Delete all known TBoxes except the default TBox called `default`. Usually this enables the garbage collector to recycle the memory used by these TBoxes.

Syntax: `(delete-all-tboxes)`

Values: List containing the names of the removed TBoxes and a list of names of optionally removed ABoxes

Remarks: All ABoxes are also deleted.

create-tbox-clone *function*

Description: Returns a new TBox object which is a clone of the given TBox. The clone keeps all declarations from its original but it is otherwise fresh, i.e., new declarations can be added. This function allows one to create new TBox versions without the need to reload the already known declarations.

Syntax: `(create-tbox-clone` *tbox* `&key (`*new-name* `nil) (`*overwrite* `nil))`

Arguments: *tbox* - is a TBox name or a TBox object.

new-name - if bound to a symbol, this specifies the name of the clone. A new unique name based on the name of *tbox* is generated otherwise.

overwrite - if bound to `t` an existing TBox with the name given by *new-name* is overwritten. If bound to `nil` an error is signaled if a TBox with the name given by *new-name* is found.

Values: TBox object

Examples:
```
(create-tbox-clone 'my-TBox)
(create-tbox-clone 'my-TBox :new-name 'my-clone :overwrite t)
```

clone-tbox *macro*

Description: Returns a new TBox object which is a clone of the given TBox. The clone keeps all declarations from its original but it is otherwise fresh, i.e., new declarations can be added. This function allows one to create new TBox versions without the need to reload the already known declarations.

Syntax: `(clone-tbox` *TBN* `&key (`*new-name* `nil) (`*overwrite* `nil))`

Arguments: *TBN* - is a TBox name.

new-name - if bound to a symbol, this specifies the name of the clone. A new unique name based on the name of *tbox* is generated otherwise.

overwrite - if bound to `t` an existing TBox with the name given by *new-name* is overwritten. If bound to `nil` an error is signaled if a TBox with the name given by *new-name* is found.

Values: TBox object

Remarks: The function `create-tbox-clone` is called.

Examples:
```
(clone-tbox my-TBox)
(clone-tbox my-TBox :new-name my-clone :overwrite t)
```

See also: Function `create-tbox-clone` on page 16.

find-tbox *function*

Description: Returns a TBox object with the given name among all TBoxes.

Syntax: `(find-tbox` *TBN* `&optional (`*errorp* `t))`

Arguments: *TBN* - is the name of the TBox to be found.

errorp - if bound to `t` an error is signaled if the TBox is not found.

Values: TBox object

Remarks: This function can also be used to get rid of TBoxes or to rename TBoxes as shown in the examples.

set-find-tbox *function*

Description: Changes the name of an TBox.

Syntax: `(set-find-tbox` *tbox − name − 1* *tbox − name − 2*`)`

Arguments: *tbox − name − 1* - is the old name of the TBox.

tbox − name − 2 - is the new name of the TBox. This argument may be nil

Values: TBox

Remarks: This function can also be used to delete TBoxes or rename TBoxes as shown in the examples.

Examples: Get rid of an TBox, i.e. make the TBox garbage collectible:
`(set-find-tbox 'tbox1 nil)`

Renaming an TBox `tbox1` to `tbox2`:
`(set-find-tbox tbox1 'tbox2)`

clear-default-tbox *function*

Description: This function initializes the default TBox.

Syntax: `(clear-default-tbox)`

Arguments:

associated-aboxes *function*

Description: Returns a list of ABoxes or ABox names which are defined wrt. the TBox specified as a parameter.

Syntax: `(associated-aboxes` *TBN*`)`

Arguments: *TBN* - is the name of a TBox.

Values: List of ABox objects

xml-read-tbox-file *function*

Description: A file in XML format containing TBox declarations is parsed and the resulting TBox is returned.

Syntax: `(xml-read-tbox-file` *pathname*`)`

Arguments: *pathname* - is the pathname of a file

Values: TBox object

Remarks: Only XML descriptions which correspond the so-called FaCT DTD are parsed, everything else is ignored.

Examples:
```
(xml-read-tbox-file "project:TBoxes;tbox-one.xml")
```

rdfs-read-tbox-file *function*

Description: A file in RDFS format containing TBox declarations is parsed and the resulting TBox is returned. The name of the TBox is the filename without file type.

Syntax: `(rdfs-read-tbox-file` *pathname*`)`

Arguments: *pathname* - is the pathname of a file

Values: TBox object

Remarks: If the file to be read also contains RDF descriptions, use the function `daml-read-file` instead. The RDF descriptions are represented using appropriate ABox assertions. The function `rdfs-read-tbox-file` is supported for backward compatibility.

Examples:
```
(rdfs-read-tbox-file "project:TBoxes;tbox-one.rdfs")
```

1.2 ABox Management

If RACER is started, there exists a ABox named DEFAULT, which is set to the current ABox.

in-abox *macro*

Description: The ABox with this name is taken or generated. If a TBox is specified, the ABox is also initialized.

Syntax: `(in-abox` *ABN* `&optional (`*TBN* `(current-tbox)))`

Arguments: *ABN* - ABox name

TBN - name of the TBox to be associated with the ABox.

Values: ABox object named *ABN*

Remarks: If the specified TBox does not exist, an error is signaled.

Usually this macro is used at top of a file containing an ABox. This macro can also be used to create new ABoxes. If the ABox is to be continued in another file, the TBox must not be specified again.

The specified ABox is the current abox until `in-abox` is called again. The TBox of the ABox is made the `(current-tbox)`.

Examples:
```
(in-abox peanuts-characters peanuts)
(instance Schroeder Piano-Player)
                    ⋮
```

See also: Macro `signature` on page 12.

init-abox *function*

Description: Initializes an existing ABox or generates a new ABox. During the initialization all assertions and the link to the referenced TBox are deleted.

Syntax: `(init-abox` *abox* `&optional (`*tbox* `(current-tbox)))`

Arguments: *abox* - ABox object to initialize

tbox - TBox object associated with the ABox

Values: *abox*

Remarks: The *tbox* has to already exist before it can be referred to by `init-abox`.

ensure-abox-signature *function*

Description: Defines the signature for an ABox and initializes the ABox.

Syntax: `(ensure-abox-signature` *abox* `&key (`*individuals* `nil) (`*objects* `nil))`

Arguments: *abox* - ABox object

individuals - is a list of individual names.

objects - is a list of concrete domain object names.

See also: Macro `signature` on page 12 is the macro counterpart. It allows to specify a signature for an ABox and a TBox with one call.

get-abox-signature *function*

Description: Gets the signature for an ABox.

Syntax: `(get-abox-signature &optional` *ABN*`)`

Arguments: *ABN* - is an ABox name

get-kb-signature *function*

Description: Gets the signature for a knowledge base.

Syntax: `(get-kb-signature &optional` *KBN*`)`

Arguments: *KBN* - is a name for a knowledge base.

current-abox *function*

Description: Returns the current ABox.

Syntax: `(current-abox)`

Arguments:

set-current-abox *function*

Description: The function sets the current ABox.

Syntax: `(set-current-abox` *abox*`)`

Arguments:

get-abox-version *Function*

Description: Gets a version indicator for a ABox.

Syntax: `(get-abox-version` *abox*`)`

Arguments: *abox* - is a ABox name.

save-abox *function*

Description: If a pathname is specified, an ABox is saved to a file. In case a stream is specified, the ABox is written to the stream (the stream must already be open) and the keywords *if-exists* and *if-does-not-exist* are ignored.

Syntax: `(save-abox` *pathname-or-stream* `&optional (`*abox* `(current-abox))` `&key (`*syntax* `:krss) (`*transformed* `nil) (`*if-exists* `:supersede)` `(`*if-does-not-exist* `:create))`

Arguments: *pathname-or-stream* - is the name of the file or an output stream.

abox - ABox object

syntax - indicates the syntax of the TBox. Possible value for the *syntax* argument are :krss (the default), :xml, or :daml.

transformed - if bound to `t` the ABox is saved in the format it has after preprocessing by RACER.

if-exists - specifies the action taken if a file with the specified name already exists. All keywords for the Lisp function `with-open-file` are supported. The default is `:supersede`.

if-does-not-exist - specifies the action taken if a file with the specified name does not yet exist. All keywords for the Lisp function `with-open-file` are supported. The default is `:create`.

Values: ABox object

Remarks: A file may contain several ABoxes.
The usual way to load an ABox file is to use the Lisp function `load`.
If the server version is used, it must have been started with the option -u in order to have this function available.

Examples:
```
(save-abox "project:ABoxes;abox-one.lisp")
(save-abox "project:ABoxes;final-abox.lisp"
  (find-abox 'abox-one) :if-exists :error)
```

forget-abox *function*

Description: Delete the specified ABox from the list of all ABoxes. Usually this enables the garbage collector to recycle the memory used by this ABox.

Syntax: `(forget-abox` *abox*`)`

Arguments: *abox* - is a ABox object or ABox name.

Values: The name of the removed ABox

Examples: `(forget-abox 'family)`

delete-abox *macro*

Description: Delete the specified ABox from the list of all ABoxes. Usually this enables the garbage collector to recycle the memory used by this ABox.

Syntax: `(delete-abox` *ABN*`)`

Arguments: *ABN* - is a ABox name.

Values: The name of the removed ABox

Remarks: Calls `forget-abox`

Examples: `(delete-abox family)`

delete-all-aboxes *function*

Description: Delete all known ABoxes. Usually this enables the garbage collector to recycle the memory used by these ABoxes.

Syntax: `(delete-all-aboxes)`

Values: List containing the names of the removed ABoxes

create-abox-clone *function*

Description: Returns a new ABox object which is a clone of the given ABox. The clone keeps the assertions and the state from its original but new declarations can be added without modifying the original ABox. This function allows one to create new ABox versions without the need to reload (and reprocess) the already known assertions.

Syntax: `(create-abox-clone` *abox* `&key (`*new-name* `nil) (`*overwrite* `nil))`

Arguments: *abox* - is an ABox name or an ABox object.

new-name - if bound to a symbol, this specifies the name of the clone. A new unique name based on the name of *abox* is generated otherwise.

overwrite - if bound to `t` an existing ABox with the name given by *new-name* is overwritten. If bound to `nil` an error is signaled if an ABox with the name given by *new-name* is found.

Values: ABox object

Remarks: The current ABox is set to the result of this function.

Examples:
```
(create-abox-clone 'my-ABox)
(create-abox-clone 'my-ABox :new-name 'abox-clone :overwrite t)
```

clone-abox *macro*

Description: Returns a new ABox object which is a clone of the given ABox. The clone keeps the assertions and the state from its original but new declarations can be added without modifying the original ABox. This function allows one to create new ABox versions without the need to reload (and reprocess) the already known assertions.

Syntax: `(clone-abox` *ABN* `&key (`*new-name* `nil) (`*overwrite* `nil))`

Arguments: *ABN* - is an ABox name.

new-name - if bound to a symbol, this specifies the name of the clone. A new unique name based on the name of *abox* is generated otherwise.

overwrite - if bound to `t` an existing ABox with the name given by *new-name* is overwritten. If bound to `nil` an error is signaled if an ABox with the name given by *new-name* is found.

Values: ABox object

Remarks: The function `create-abox-clone` is called.

Examples:
```
(clone-abox my-ABox)
(clone-abox my-ABox :new-name abox-clone :overwrite t)
```

See also: Function `create-abox-clone` on page 25.

find-abox *function*

Description: Finds an ABox object with a given name among all ABoxes.

Syntax: `(find-abox` *ABN* `&optional (`*errorp* `t))`

Arguments: *ABN* - is the name of the ABox to be found.

errorp - if bound to `t` an error is signaled if the ABox is not found.

Values: ABox object

set-find-abox *function*

Description: Changes the name of an ABox.

Syntax: `(set-find-abox` *abox − name − 1* *abox − name − 2*`)`

Arguments: *abox − name − 1* - is the old name of the ABox.

abox − name − 2 - is the new name of the ABox. This argument may be nil

Values: ABox

Remarks: This function can also be used to delete ABoxes or rename ABoxes as shown in the examples.

Examples: Get rid of an ABox, i.e. make the ABox garbage collectible:
`(set-find-abox 'abox1 nil)`

Renaming an ABox `abox1` to `abox2`:
`(set-find-abox 'abox1 'abox2)`

tbox *function*

Description: Gets the associated TBox for an ABox.

Syntax: `(tbox` *abox*`)`

Arguments: *abox* - ABox object

Values: TBox object

Remarks: This function is provided in the Lisp version only.

associated-tbox *function*

Description: Gets the associated TBox for an ABox.

Syntax: `(associated-tbox` *abox*`)`

Arguments: *abox* - ABox object

Values: TBox object

Remarks: This function is provided in the server version only.

set-associated-tbox *function*

Description: Sets the associated TBox for an ABox.

Syntax: `(set-associated-tbox` *ABN TBN*`)`

Arguments: *ABN* - ABox name
TBN - TBox name

Values: TBox object

Remarks: This function is provided in the server version only.

Chapter 2

Knowledge Base Declarations

Knowledge base declarations include concept axioms and role declarations for the TBox and the assertions for the ABox. The TBox object and the ABox object must exist before the functions for knowledge base declarations can be used. The order of axioms and assertions does not matter because forward references can be handled by RACER.

The macros for knowledge base declarations add the concept axioms and role declarations to the `(current-tbox)` and the assertions to the `(current-abox)`.

2.1 Built-in Concepts

---

top, top *concept*

Description: The name of most general concept of each TBox, the top concept ($\top$).

Syntax: `*top*`

Remarks: The concepts `*top*` and `top` are synonyms. These concepts are elements of every TBox.

bottom, bottom *concept*

Description: The name of the incoherent concept, the bottom concept ($\bot$).

Syntax: `*bottom*`

Remarks: The concepts `*bottom*` and `bottom` are synonyms. These concepts are elements of every TBox.

2.2 Concept Axioms

This section documents the macros and functions for specifying concept axioms.

Please note that the concept axioms `define-primitive-concept`, `define-concept` and `define-disjoint-primitive-concept` have the semantics given in the KRSS specification only if they are the only concept axiom defining the concept *CN* in the terminology. This is not checked by the RACER system.

implies *macro*

Description: Defines a GCI between C_1 and C_2.

Syntax: `(implies` C_1 C_2`)`

Arguments: C_1, C_2 - concept term

Remarks: C_1 states necessary conditions for C_2. This kind of facility is an addendum to the KRSS specification.

Examples:
```
(implies Grandmother (and Mother Female))
(implies
  (and (some has-sibling Sister) (some has-sibling Twin)
    (exactly 1 has-sibling))
  (and Twin (all has-sibling Twin-sister)))
```

equivalent *macro*

Description: States the equality between two concept terms.

Syntax: `(equivalent` C_1 C_2`)`

Arguments: C_1, C_2 - concept term

Remarks: This kind of concept axiom is an addendum to the KRSS specification.

Examples:
```
(equivalent Grandmother
  (and Mother (some has-child Parent)))
(equivalent
  (and polygon (exactly 4 has-angle))
  (and polygon (exactly 4 has-edges)))
```

disjoint *macro*

Description: This axiom states the disjointness of a set of concepts.

Syntax: `(disjoint` CN_1 `...`CN_n`)`

Arguments: CN_1,..., CN_n - concept names

Examples:
```
(disjoint Yellow Red Blue)
(disjoint January February ...November December))
```

define-primitive-concept *KRSS macro*

Description: Defines a primitive concept.

Syntax: `(define-primitive-concept` CN C`)`

Arguments: CN - concept name

C - concept term

Remarks: C states the necessary conditions for CN.

Examples:
```
(define-primitive-concept Grandmother (and Mother Female))
(define-primitive-concept Father Parent)
```

define-concept *KRSS macro*

Description: Defines a concept.

Syntax: `(define-concept` *CN C*`)`

Arguments: *CN* - concept name

C - concept term

Remarks: Please note that in RACER, definitions of a concept do not have to be unique. Several definitions may be given for the same concept.

Examples:
```
(define-concept Grandmother
  (and Mother (some has-child Parent)))
```

define-disjoint-primitive-concept *KRSS macro*

Description: This axiom states the disjointness of a group of concepts.

Syntax: `(define-disjoint-primitive-concept` *CN GNL C*`)`

Arguments: *CN* - concept name

GNL - group name list, which lists all groups to which *CN* belongs to (among other concepts). All elements of each group are declared to be disjoint.

C - concept term, that is implied by *CN*.

Remarks: This function is just supplied to be compatible with the KRSS.

Examples:
```
(define-disjoint-primitive-concept January
  (Month) (exactly 31 has-days))
(define-disjoint-primitive-concept February
  (Month) (and (at-least 28 has-days) (at-most 29 has-days)))
                    ⋮
```

add-concept-axiom *function*

Description: This function adds a concept axiom to a TBox.

Syntax: `(add-concept-axiom` *tbox* C_1 C_2 `&key (`*inclusion-p* `nil))`

Arguments: *tbox* - TBox object

C_1, C_2 - concept term

inclusion-p - boolean indicating if the concept axiom is an inclusion axiom (GCI) or an equality axiom. The default is to state an inclusion.

Values: *tbox*

Remarks: RACER imposes no constraints on the sequence of concept axiom declarations with `add-concept-axiom`, i.e. forward references to atomic concepts for which other concept axioms are added later are supported in RACER.

add-disjointness-axiom *function*

Description: This function adds a disjointness concept axiom to a TBox.

Syntax: `(add-disjointness-axiom` *tbox CN GN*`)`

Arguments: *tbox* - TBox object

CN - concept name

GN - group name

Values: *tbox*

2.3 Role Declarations

Roles can be declared with the following statements.

define-primitive-role

KRSS macro (with changes)

Description: Defines a role.

Syntax: `(define-primitive-role` *RN* `&key (`*transitive* `nil) (`*feature* `nil) (`*symmetric* `nil) (`*reflexive* `nil) (`*inverse* `nil) (`*domain* `nil) (`*range* `nil) (`*parents* `nil))`

Arguments: *RN* - role name

transitive - if bound to `t` declares that the new role is transitive.

feature - if bound to `t` declares that the new role is a feature.

symmetric - if bound to `t` declares that the new role is a symmetric. This is equivalent to declaring that the new role's inverse is the role itself.

reflexive - if bound to `t` declares that the new role is reflexive (currently only supported for $\mathcal{ALCH}$). If *feature* is bound to `t`, the value of *reflexive* is ignored.

inverse - provides a name for the inverse role of *RN*. This is equivalent to `(inv` *RN*`)`. The inverse role of *RN* has no user-defined name, if *inverse* is bound to `nil`.

domain - provides a concept term defining the domain of role *RN*. This is equivalent to adding the axiom `(implies (at-least 1` *RN*`)` *C*`)` if *domain* is bound to the concept term *C*. No domain is declared if *domain* is bound to `nil`.

range - provides a concept term defining the range of role *RN*. This is equivalent to adding the axiom `(implies *top* (all` *RN* *D*`))` if *range* is bound to the concept term *D*. No range is declared if *range* is bound to `nil`.

parents - provides a list of superroles for the new role. The role *RN* has no superroles, if *parents* is bound to `nil`.
If only a single superrole is specified, the keyword `:parent` may alternatively be used, see the examples.

Remarks: This function combines several KRSS functions for defining properties of a role. For example the conjunction of roles can be expressed as shown in the first example below.

A role that is declared to be a feature cannot be transitive. A role with a feature as a parent has to be a feature itself. A role with transitive subroles may not be used in number restrictions.

Examples:
```
(define-primitive-role conjunctive-role :parents (R-1 ...R-n))
(define-primitive-role has-descendant :transitive t
  :inverse descendant-of :parent has-child)
(define-primitive-role has-children :inverse has-parents
  :domain parent :range children))
```

See also: Macro `signature` on page 12.

define-primitive-attribute

KRSS macro (with changes)

Description: Defines an attribute.

Syntax: `(define-primitive-attribute` *AN* `&key (`*symmetric* `nil) (`*inverse* `nil) (`*domain* `nil) (`*range* `nil) (`*parents* `nil))`

Arguments: *AN* - attribute name

symmetric - if bound to `t` declares that the new role is a symmetric. This is equivalent to declaring that the new role's inverse is the role itself.

inverse - provides a name for the inverse role of *AN*. This is equivalent to `(inv` *AN*`)`. The inverse role of *AN* has no user-defined name, if *inverse* is bound to `nil`.

domain - provides a concept term defining the domain of role *AN*. This is equivalent to adding the axiom `(implies (at-least 1` *AN*`)` *C*`)` if *domain* is bound to the concept term *C*. No domain is declared if *domain* is bound to `nil`.

range - provides a concept term defining the range of role *AN*. This is equivalent to adding the axiom `(implies *top* (all` *AN* *D*`))` if *range* is bound to the concept term *D*. No range is declared if *range* is bound to `nil`.

parents - provides a list of superroles for the new role. The role *AN* has no superroles, if *parents* is bound to `nil`.
If only a single superrole is specified, the keyword `:parent` may alternatively be used, see examples.

Remarks: This macro is supplied to be compatible with the KRSS specification. It is redundant since the macro `define-primitive-role` can be used with :*feature* `t`. This function combines several KRSS functions for defining properties of an attribute.

An attribute cannot be transitive. A role with a feature as a parent has to be a feature itself.

Examples:
```
(define-primitive-attribute has-mother
  :domain child :range mother :parents (has-parents))
(define-primitive-attribute has-best-friend
  :inverse best-friend-of :parent has-friends)
```

See also: Macro `signature` on page 12.

add-role-axioms *function*

Description: Adds a role to a TBox.

Syntax: `(add-role-axioms` *tbox RN* `&key (`*cd-attribute* `nil) (`*transitive* `nil) (`*feature* `nil) (`*symmetric* `nil) (`*reflexive* `nil) (`*inverse* `nil) (`*domain* `nil) (`*range* `nil) (`*parents* `nil))`

Arguments: *tbox* - TBox object to which the role is added.

RN - role name

cd-attribute - may be either `integer` or `real`.

transitive - if bound to `t` declares that *RN* is transitive.

feature - if bound to `t` declares that *RN* is a feature.

symmetric - if bound to `t` declares that *RN* is a symmetric. This is equivalent to declaring that the new role's inverse is the role itself.

reflexive - if bound to `t` declares that *RN* is reflexive (currently only supported for $\mathcal{ALCH}$). If *feature* is bound to `t`, the value of *reflexive* is ignored.

inverse - provides a name for the inverse role of *RN* (is equivalent to `(inv` *RN*`)`). The inverse role of *RN* has no user-defined name, if *inverse* is bound to `nil`.

domain - provides a concept term defining the domain of role *RN* (equivalent to adding the axiom `(implies (at-least 1` *RN*`)` *C*`)` if *domain* is bound to the concept term *C*. No domain is declared if *domain* is bound to `nil`.

range - provides a concept term defining the range of role *RN* (equivalent to adding the axiom `(implies *top* (all` *RN D*`))` if *range* is bound to the concept term *D*. No range is declared if *range* is bound to `nil`.

parents - providing a single role or a list of superroles for the new role. The role *RN* has no superroles, if *parents* is bound to `nil`.

Values: *tbox*

Remarks: For each role *RN* there may be only one call to `add-role-axioms` per TBox.

functional *macro*

Description: States that a role is to be interpreted as functional.

Syntax: `(functional` *RN*
`&optional (`*TBN* `(current-tbox)))`

Arguments: *RN* - role name
TBN - TBox name

Remarks: States that a role is to be interpreted as functional.

role-is-functional *function*

Description: States that a role is to be interpreted as functional.

Syntax: `(role-is-functional` *RN*
`&optional (`*TBN* `(current-tbox)))`

Arguments: *RN* - role name
TBN - TBox name

transitive *macro*

Description: States that a role is to be interpreted as transitive.

Syntax: `(transitive` *RN*
`&optional (`*TBN* `(current-tbox)))`

Arguments: *RN* - role name
TBN - TBox name

role-is-transitive *function*

Description: States that a role is to be interpreted as transitive.

Syntax: `(role-is-transitive` *RN*
`&optional (`*TBN* `(current-tbox)))`

Arguments: *RN* - role name
TBN - TBox name

role-is-used-as-datatype-property *function*

Description: States that a role is to be interpreted as a datatype property role.

Syntax: `(role-is-used-as-datatype-property` *RN TBN*`)`

Arguments: *RN* - role name
TBN - TBox name

role-is-used-as-annotation-property *function*

Description: States that a role is to be interpreted as an annotation property role.

Syntax: `(role-is-used-as-annotation-property` *RN TBN*`)`

Arguments: *RN* - role name
TBN - TBox name

inverse *macro*

Description: Defines a name for the inverse of a role.

Syntax: `(inverse` *RN inverse − role*
`&optional (`*TBN* `(current-tbox)))`

Arguments: *RN* - role name
inverse − role - inverse role of the Form `(inv` *RN*`)`
TBN - TBox name

inverse-of-role *function*

Description: Defines a name for the inverse of a role.

Syntax: `(inverse-of-role` *RN inverse − role*
`&optional (`*TBN* `(current-tbox)))`

Arguments: *RN* - role name
inverse − role - inverse role of the Form `(inv` *RN*`)`
TBN - TBox name

roles-equivalent *macro*

Description: Declares two roles to be equivalent.

Syntax: `(roles-equivalent` *RN1 RN1 TBN*`)`

Arguments: *RN1* - role name

RN2 - role name

TBN - TBox name

roles-equivalent-1 *function*

Description: Declares two roles to be equivalent.

Syntax: `(roles-equivalent-1` *RN1 RN2 TBN*`)`

Arguments: *RN1* - role name

RN2 - role name

TBN - TBox name

domain *macro*

Description: Declares the domain of a role.

Syntax:
```
(domain RN C
  &optional (TBN (current-tbox)))
```

Arguments: *RN* - role name

C - concept

TBN - TBox name

role-has-domain *function*

Description: Declares the domain of a role.

Syntax: `(role-has-domain` *RN C*
`&optional (`*TBN* `(current-tbox)))`

Arguments: *RN* - role name

C - concept

TBN - TBox name

attribute-has-domain *function*

Description: Declares the domain of an attribute.

Syntax: `(attribute-has-domain` *AN C*
`&optional (`*TBN* `(current-tbox)))`

Arguments: *AN* - attribute name

C - concept

TBN - TBox name

range *macro*

Description: Declares the range of a role.

Syntax: `(range` *RN C*
`&optional (`*TBN* `(current-tbox)))`

Arguments: *RN* - role name

C - concept

TBN - TBox name

role-has-range *function*

Description: Declares the range of a role.

Syntax: `(role-has-range` *RN C*
`&optional (`*TBN* `(current-tbox)))`

Arguments: *RN* - role name

C - concept

TBN - TBox name

datatype-role-has-range *function*

Description: Declares the range of a datatype property role.

Syntax: `(datatype-role-has-range` *RN type TBN* `)`

Arguments: *RN* - role name

type - either cardinal, integer, real, complex, or string

TBN - TBox name

attribute-has-range *function*

Description: Declares the range of an attribute.

Syntax: `(attribute-has-range` *AN D*
`&optional (`*TBN* `(current-tbox)))`

Arguments: *AN* - attribute name

C - concept

D - either cardinal, integer, real, complex, or string

implies-role *macro*

Description: Defines a parent of a role.

Syntax: `(implies-role` *RN1 RN2* `&optional (`*TBN* `(current-tbox)))`

Arguments: *RN1* - role name
RN2 - parent role name
TBN - TBox name

role-has-parent *function*

Description: Defines a parent of a role.

Syntax: `(role-has-parent` *RN1 RN2* `&optional (`*TBN* `(current-tbox)))`

Arguments: *RN1* - role name
RN2 - parent role name
TBN - TBox name

2.4 Concrete Domain Attribute Declaration

define-concrete-domain-attribute *macro*

Description: Defines a concrete domain attribute.

Syntax: `(define-concrete-domain-attribute` *AN* `&key` *type domain*`)`

Arguments: *AN* - attribute name
type - can be either bound to `cardinal`, `integer`, `real`, `complex`, or `string`. The type must be supplied.
domain - a concept describing the domain of the attribute.

Remarks: Calls `add-role-axioms`

Examples:
```
(define-concrete-domain-attribute has-age :type integer)
(define-concrete-domain-attribute has-weight :type real)
```

See also: Macro `signature` on page 12 and Section 2.4.

define-datatype-property *macro*

Description: Defines a role with range from a specified concrete domain. The name is reminiscent of the OWL language which calls these roles datatype properties.

Syntax: `(define-datatype-property` *RN* `&key (`*feature* `nil) (`*domain* `nil) (`*range* `nil) (`*parents* `nil))`

Arguments: *RN* - attribute name

range - can be either bound to `cardinal`, `integer`, `real`, `complex`, or `string`. The type must be supplied.

domain - a concept describing the domain of the attribute.

parents - a list of roles for the parents.

Remarks: Calls `add-role-axioms`

Examples:
```
(define-datatype-property room-number :range integer)
```

add-datatype-property *Function*

Description: Functional equivalent of `define-datatype-property`, Page 44.

2.5 Assertions

instance *KRSS macro*

Description: Builds a concept assertion, asserts that an individual is an instance of a concept.

Syntax: `(instance` *IN* *C*`)`

Arguments: *IN* - individual name

C - concept term

Examples:
```
(instance Lucy Person)
(instance Snoopy (and Dog Cartoon-Character))
```

add-concept-assertion *function*

Description: Builds an assertion and adds it to an ABox.

Syntax: `(add-concept-assertion` *abox IN C*`)`

Arguments: *abox* - ABox object

IN - individual name

C - concept term

Values: *abox*

Examples:
```
(add-concept-assertion (find-abox 'peanuts-characters)
  'Lucy 'Person)
(add-concept-assertion (find-abox 'peanuts-characters)
  'Snoopy '(and Dog Cartoon-Character))
```

forget-concept-assertion *function*

Description: Retracts a concept assertion from an ABox.

Syntax: `(forget-concept-assertion` *abox IN C*`)`

Arguments: *abox* - ABox object

IN - individual name

C - concept term

Values: *abox*

Remarks: For answering subsequent queries the index structures for the ABox will be recomputed, i.e. some queries might take some time (e.g. those queries that require the realization of the ABox).

Examples:
```
(forget-concept-assertion (find-abox 'peanuts-characters)
  'Lucy 'Person)
(forget-concept-assertion (find-abox 'peanuts-characters)
  'Snoopy '(and Dog Cartoon-Character))
```

related *KRSS macro*

Description: Builds a role assertion, asserts that two individuals are related via a role (or feature).

Syntax: `(related` IN_1 IN_2 R`)`

Arguments: IN_1 - individual name of the predecessor

IN_2 - individual name of the filler

R - a role term or a feature term.

Examples:
```
(related Charlie-Brown Snoopy has-pet)
(related Linus Lucy (inv has-brother))
```

add-role-assertion *function*

Description: Adds a role assertion to an ABox.

Syntax: `(add-role-assertion` *abox* IN_1 IN_2 R`)`

Arguments: *abox* - ABox object

IN_1 - individual name of the predecessor

IN_2 - individual name of the filler

R - role term

Values: *abox*

Examples:
```
(add-role-assertion (find-abox 'peanuts-characters)
  'Charlie-Brown 'Snoopy 'has-pet)
(add-role-assertion (find-abox 'peanuts-characters)
  'Linus 'Lucy '(inv has-brother))
```

forget-role-assertion *function*

Description: Retracts a role assertion from an ABox.

Syntax: `(forget-role-assertion` *abox* IN_1 IN_2 *R*`)`

Arguments: *abox* - ABox object

IN_1 - individual name of the predecessor

IN_2 - individual name of the filler

R - role term

Values: *abox*

Remarks: For answering subsequent queries the index structures for the ABox will be recomputed, i.e. some queries might take some time (e.g. those queries that require the realization of the ABox).

Examples:
```
(forget-role-assertion (find-abox 'peanuts-characters)
  'Charlie-Brown 'Snoopy 'has-pet)
(forget-role-assertion (find-abox 'peanuts-characters)
  'Linus 'Lucy '(inv has-brother))
```

forget-disjointness-axiom *function*

Description: This function is used to forget declarations with `define-disjoint-primitive-concept`.

Syntax: `(forget-disjointness-axiom` *tbox CN group − name*`)`

Arguments: *tbox* - TBox object

CN - concept-name

group − name - name of the disjointness group

forget-disjointness-axiom-statement *function*

Description: This function is used to forget statements of the form `(disjoint a b c)`

Syntax: `(forget-disjointness-axiom-statement` *tbox* `&rest` *concepts*`)`

Arguments: *tbox* - TBox object

concepts - List of concepts

forget-constrained-assertion *function*

Description: Forget assertions with the form `constrained`.

Syntax: `(forget-constrained-assertion` *abox IN ON attributeterm*`)`

Arguments: *abox* - ABox

IN - individual name

ON - object name

attributeterm - attribute term

forget-constraint *function*

Description: Forget assertions with the form `constraint`

Syntax: `(forget-constraint` *abox constraint*`)`

Arguments: *abox* - ABox

constraint - constraint term

define-distinct-individual *KRSS macro*

Description: This statement asserts that an individual is distinct to all other individuals in the ABox.

Syntax: `(define-distinct-individual` *IN*`)`

Arguments: *IN* - name of the individual

Values: *IN*

Remarks: Introduces IN as a name for an individual which as made distinct from all other individuals automatically.

define-individual *KRSS macro*

Description: This statement asserts that an individual is distinct to all other individuals in the ABox.

Syntax: `(define-individual` *IN*`)`

Arguments: *IN* - name of the individual

Values: *IN*

Remarks: Introduces IN as a name for an individual not necessarily distinct from other individuals.

same-as *Macro*

Description: This form declares two individuals to refer to the same domain object.

Syntax: `(same-as IN1 IN2)`

Arguments: *IN1* - an individual name

IN2 - an individual name

same-individual-as *Function*

Description: Synonym to `same-as`, Page 49.

add-same-individual-as-assertion *Function*

Description: This form declares two individuals to refer to the same domain object.

Syntax: `(add-same-individual-as-assertion ABox IN1 IN2)`

Arguments: *ABox* - ABox name

IN1 - an individual name

IN2 - an individual name

Remarks: Functional equivalent of same-as.

different-from *Macro*

Description: This form declares two individuals NOT to refer to the same domain object.

Syntax: `(different-from IN1 IN2)`

Arguments: *IN1* - an individual name

IN2 - an individual name

add-different-from-assertion *Function*

Description: This form declares two individuals NOT to refer to the same domain object.

Syntax: `(add-different-from-assertion ABox IN1 IN2)`

Arguments: *ABox* - ABox name

IN1 - an individual name

IN2 - an individual name

Remarks: Functional equivalent of different-from.

all-different *Macro*

Description: This form declares the argument individuals NOT to refer to the same domain object.

Syntax: `(all-different &rest individuals)`

Arguments: *individuals* - individual names

add-all-different-assertion *Macro*

Description: This form declares the argument individuals NOT to refer to the same domain object.

Syntax: `(all-different ABox &rest individuals)`

Arguments: *ABox* - ABox name

individuals - individual names

state *KRSS macro*

Description: This macro asserts a set of ABox statements.

Syntax: `(state &body forms)`

Arguments: *forms* - is a sequence of `instance` or `related` assertions.

Remarks: This macro is supplied to be compatible with the KRSS specification. It realizes an implicit `progn` for assertions.

forget *macro*

Description: This macro retracts a set of TBox/ABox statements. Note that statement to be forgotten must be literally identical to the ones previously asserted, i.e., only explicitly given information can be forgotten.

Syntax:
```
(forget (&key (tbox (current-tbox)) (abox (current-abox)))
  &body forms)
```

Arguments: *forms* - is a sequence of assertions.

Remarks: For answering subsequent queries the index structures for the TBox/ABox will probably be recomputed, i.e. some queries might take some time (e.g. those queries that require the reclassification of the TBox or realization of the ABox).

Examples:
```
(forget (:tbox family) (implies c d) (implies a b ))
(forget (:abox smith-family) (instance i d))
```

forget-statement *function*

Description: Functional interface for the macro `forget`

Syntax: `(forget-statement` *tbox abox* `&rest statements)`

Arguments: *tbox* - TBox

tbox - ABox

statements - statement previously asserted

2.6 Concrete Domain Assertions

add-constraint-assertion *function*

Description: Builds a concrete domain predicate assertion and adds it to an ABox.

Syntax: `(add-constraint-assertion` *abox constraint*`)`

Arguments: *abox* - ABox object

constraint - constraint form

Examples:
```
(add-constraint-assertion (find-abox 'family)
  '(= temp-eve 102.56))
```

constraints *macro*

Description: This macro asserts a set of concrete domain predicates for concrete domain objects.

Syntax: `(constraints &body forms)`

Arguments: *forms* - is a sequence of concrete domain predicate assertions.

Remarks: Calls `add-constraint-assertion`.

Examples:
```
(constraints
  (= temp-eve 102.56)
  (= temp-doris 38.5)
  (> temp-eve temp-doris))
```

add-attribute-assertion *function*

Description: Adds a concrete domain attribute assertion to an ABox. Asserts that an individual is related with a concrete domain object via an attribute.

Syntax: `(add-attribute-assertion` *abox IN ON AN*`)`

Arguments:
- *abox* - ABox object
- *IN* - individual name
- *ON* - concrete domain object name as the filler
- *AN* - attribute name

Examples:
```
(add-attribute-assertion (find-abox 'family)
  'eve 'temp-eve 'temperature-fahrenheit))
```

constrained *macro*

Description: Adds a concrete domain attribute assertion to an ABox. Asserts that an individual is related with a concrete domain object via an attribute.

Syntax: `(constrained` *IN ON AN*`)`

Arguments:
- *IN* - individual name
- *ON* - concrete domain object name as the filler
- *AN* - attribute name

Remarks: Calls `add-attribute-assertion`

Examples:
```
(constrained eve temp-eve temperature-fahrenheit)
```

set-attribute-filler *Function*

Description: Set the filler of an attribute w.r.t. an individual.

Syntax: `(set-attribute-filler` *ABox IN value AN* `)`

Arguments:
- *IN* - individual name
- *ABox* - ABox
- *value* - value
- *AN* - Attribute name

attribute-filler *Macro*

Description: Set the filler of an attribute w.r.t. an individual.

Syntax: `(attribute-filler` *IN value AN* `)`

Arguments: *IN* - individual name

value - value

AN - Attribute name

add-datatype-role-filler *Function*

Description: Adds a filler for a datatype role w.r.t. an individual.

Syntax: `(add-datatype-role-filler` *ABox IN value RN* `)`

Arguments: *IN* - individual name

ABox - ABox

value - value

RN - datatype property role name

datatype-role-filler *Macro*

Description: Adds a filler of a datatype role w.r.t. an individual.

Syntax: `(attribute-filler` *IN value RN* `)`

Arguments: *IN* - individual name

value - value

RN - datatype property role name

add-annotation-role-assertion *function*

Description: Adds an annotation role assertion to an ABox. Asserts that an individual is related with a concrete domain object via an annotation role.

Syntax: `(add-annotation-role-assertion` *abox IN value AN*`)`

Arguments:
- *abox* - ABox object
- *IN* - individual name
- *value* - concrete domain value
- *AN* - attribute name

add-annotation-concept-assertion *function*

Description: Adds an annotation concept assertion to an ABox.

Syntax: `(add-annotation-concept-assertion` *abox IN C*`)`

Arguments:
- *abox* - ABox object
- *IN* - individual name
- *C* - concept

Chapter 3

Reasoning Modes

get-racer-version *Function*

Description: Returns a string which describe the version of the Racer system.

Syntax: `(get-racer-version)`

Arguments:

Values: string

time *Macro*

Description: This macro prints some timing information

Syntax: `(time form)`

Arguments: *form* - is a Racer expression.

Values: The value is the result of processing *form*.

set-unique-name-assumption *Function*

Description: This form globally instructs Racer to make the unique name assumption if t is specified as the argument. If nil is specified, Racer will not make the unique name assumption (the default).

Syntax: `(set-unique-name-assumption boolean)`

Arguments: *boolean* - boolean

set-server-timeout *Function*

Description: Set a timeout for query answering (in seconds). If nil is provided as an argument, no timeout will be used (the default).

Syntax: `(set-server-timeout seconds)`

Arguments: *seconds* - integer or nil

get-server-timeout *Function*

Description: Returns the timeout for query answering

Syntax: `(get-server-timeout)`

Arguments:

Values: Integer (seconds) or nil (for no timeout)

parse-expression *Function*

Description: Parses a Racer expression as returns the TBox or the ABox that the expression refers plus a characterization

Syntax: `(parse-expression` *expression*`)`

Arguments: *expression* - a Racer expression

The following function provide a way for you to collect the statements sent to the RACER server.

logging-on *macro*

Description: Start logging of expressions to the Racer server.

Syntax: `(logging-on` *filename*`)`

Arguments: *filename* - filename

Values: None.

Remarks: RACER must have been started in unsafe mode (option -u) to use this facility. Logging is only available in the RACER server version.

logging-off *macro*

Description: Start logging of expressions to the Racer server.

Syntax: `(logging-off )`

Arguments:

Values: None.

Remarks: Logging is only available in the RACER server version.

compute-index-for-instance-retrieval *function*

Description: Let RACER create an index for subsequent instance retrieval queries wrt. the specified ABox.

Syntax: `(compute-index-for-instance-retrieval &optional (`*ABN* `(current-abox)))`)

Arguments: *ABN* - ABox object

Remarks: Computing an index requires the associated TBox be classified and the input ABox be realized. Thus, it may take some time for this function to complete. Use the function `abox-realized-p` to check whether index-based instance retrieval is enabled.

ensure-subsumption-based-query-answering *function*

Description: Instruct RACER to use caching strategies and to exploit query subsumption for answering instance retrieval queries.

Syntax: `(ensure-subsumption-based-query-answering &optional (`*ABN* `(current-abox)))`

Arguments: *ABN* - ABox object

Remarks: Subsumption-based query answering requires the associated TBox to be classified. Thus, the function might require computational resources that are not negligible. Instructing RACER to perform reasoning in this mode pays back if one and the same instance retrieval query might be posed several times or if the concepts in subsequent instance retrieval queries subsumes each other (in other words: if queries are more and more refined). Use the function `tbox-classified-p` to check whether index-based instance retrieval is enabled.

ensure-small-tboxes *function*

Description: Instructs Racer to try to save space by throwing away internal information. This might help if for large TBoxes memory requirements cannot be met.

Syntax: `(ensure-small-tboxes)`

Arguments:

Remarks: Use with caution. Some query functions are no longer defined on TBoxes if this option is set.

Chapter 4

Evaluation Functions and Queries

4.1 Queries for Concept Terms

---

concept-satisfiable? *macro*

Description: Checks if a concept term is satisfiable.

Syntax: `(concept-satisfiable?` *C* `&optional (`*tbox* `(current-tbox)))`

Arguments: *C* - concept term.

tbox - TBox object

Values: Returns `t` if *C* is satisfiable and `nil` otherwise.

Remarks: For testing whether a concept term is satisfiable *with respect to a TBox tbox*. If satisfiability is to be tested without reference to a TBox, `nil` can be used.

concept-satisfiable-p *function*

Description: Checks if a concept term is satisfiable.

Syntax: `(concept-satisfiable-p` *C tbox*`)`

Arguments: *C* - concept term.
tbox - TBox object

Values: Returns `t` if *C* is satisfiable and `nil` otherwise.

Remarks: For testing whether a concept term is satisfiable *with respect to a TBox tbox*. If satisfiability is to be tested without reference to a TBox, `nil` can be used.

concept-subsumes? *KRSS macro*

Description: Checks if two concept terms subsume each other.

Syntax: `(concept-subsumes?` C_1 C_2 `&optional (`*tbox* `(current-tbox)))`

Arguments: C_1 - concept term of the subsumer
C_2 - concept term of the subsumee
tbox - TBox object

Values: Returns `t` if C_1 subsumes $C2$ and `nil` otherwise.

concept-subsumes-p *function*

Description: Checks if two concept terms subsume each other.

Syntax: `(concept-subsumes-p` C_1 C_2 *tbox*`)`

Arguments: C_1 - concept term of the subsumer
C_2 - concept term of the subsumee
tbox - TBox object

Values: Returns `t` if C_1 subsumes $C2$ and `nil` otherwise.

Remarks: For testing whether a concept term subsumes the other *with respect to a TBox tbox*. If the subsumption relation is to be tested without reference to a TBox, `nil` can be used.

See also: Function `concept-equivalent-p`, on page 63, and function `atomic-concept-synonyms`, on page 93.

concept-equivalent? *macro*

Description: Checks if the two concepts are equivalent in the given TBox.

Syntax: `(concept-equivalent?` C_1 C_2 `&optional (`*tbox* `(current-tbox)))`

Arguments: C_1, C_2 - concept term

tbox - TBox object

Values: Returns `t` if C_1 and C_2 are equivalent concepts in *tbox* and `nil` otherwise.

Remarks: For testing whether two concept terms are equivalent *with respect to a TBox tbox*.

See also: Function `atomic-concept-synonyms`, on page 93, and function `concept-subsumes-p`, on page 63.

concept-equivalent-p *function*

Description: Checks if the two concepts are equivalent in the given TBox.

Syntax: `(concept-equivalent-p` C_1 C_2 *tbox*`)`

Arguments: C_1, C_2 - concept terms

tbox - TBox object

Values: Returns `t` if C_1 and C_2 are equivalent concepts in *tbox* and `nil` otherwise.

Remarks: For testing whether two concept terms are equivalent *with respect to a TBox tbox*. If the equality is to be tested without reference to a TBox, `nil` can be used.

See also: Function `atomic-concept-synonyms`, on page 93, and function `concept-subsumes-p`, on page 63.

concept-disjoint? *macro*

Description: Checks if the two concepts are disjoint, e.g. no individual can be an instance of both concepts.

Syntax: `(concept-disjoint?` C_1 C_2 `&optional (`*tbox* `(current-tbox)))`

Arguments: C_1, C_2 - concept term

tbox - TBox object

Values: Returns `t` if C_1 and C_2 are disjoint with respect to *tbox* and `nil` otherwise.

Remarks: For testing whether two concept terms are disjoint *with respect to a TBox tbox*. If the disjointness is to be tested without reference to a TBox, `nil` can be used.

concept-disjoint-p *function*

Description: Checks if the two concepts are disjoint, e.g. no individual can be an instance of both concepts.

Syntax: `(concept-disjoint-p` C_1 C_2 *tbox*`)`

Arguments: C_1, C_2 - concept term

tbox - TBox object

Values: Returns `t` if C_1 and C_2 are disjoint with respect to *tbox* and `nil` otherwise.

Remarks: For testing whether two concept terms are disjoint *with respect to a TBox tbox*. If the disjointness is to be tested without reference to a TBox, `nil` can be used.

concept-p *function*

Description: Checks if *CN* is a concept name for a concept in the specified TBox.

Syntax: `(concept-p` *CN* `&optional (`*tbox* `(current-tbox)))`

Arguments: *CN* - concept name

tbox - TBox object

Values: Returns `t` if *CN* is a name of a known concept and `nil` otherwise.

concept? *macro*

Description: Checks if *CN* is a concept name for a concept in the specified TBox.

Syntax: `(concept?` *CN* `&optional (`*TBN* `(current-tbox)))`

Arguments: *CN* - concept name

TBN - TBox name

Values: Returns `t` if *CN* is a name of a known concept and `nil` otherwise.

concept-is-primitive-p *function*

Description: Checks if *CN* is a concept name of a so-called *primitive* concept in the specified TBox.

Syntax: `(concept-is-primitive-p` *CN* `&optional (`*tbox* `(current-tbox)))`

Arguments: *CN* - concept name

tbox - TBox object

Values: Returns `t` if *CN* is a name of a known primitive concept and `nil` otherwise.

concept-is-primitive? *macro*

Description: Checks if *CN* is a concept name of a so-called *primitive* concept in the specified TBox.

Syntax: `(concept-is-primitive-p` *CN* `&optional (`*TBN* `(current-tbox)))`

Arguments: *CN* - concept name

TBN - TBox name

Values: Returns `t` if *CN* is a name of a known primitive concept and `nil` otherwise.

alc-concept-coherent *function*

Description: Tests the satisfiability of a $\mathrm{K}_{(m)}$, $\mathrm{K4}_{(m)}$ or $\mathrm{S4}_{(m)}$ formula encoded as an $\mathcal{ALC}$ concept.

Syntax: `(alc-concept-coherent` *C* `&key (`*logic* `:K))`

Arguments: *C* - concept term

logic - specifies the logic to be used.

`:K` - modal $\mathbf{K_{(m)}}$,

`:K4` - modal $\mathbf{K4_{(m)}}$ all roles are transitive,

`:S4` - modal $\mathbf{S4_{(m)}}$ all roles are transitive and reflexive.

If no logic is specified, the logic `:K` is chosen.

Remarks: This function can only be used for $\mathcal{ALC}$ concept terms, so number restrictions are not allowed.

4.2 Role Queries

role-subsumes? *KRSS macro*

Description: Checks if two roles are subsuming each other.

Syntax: `(role-subsumes?` R_1 R_2
`&optional (`*TBN* `(current-tbox)))`

Arguments: R_1 - role term of the subsuming role

R_2 - role term of the subsumed role

TBN - TBox name

Values: Returns `t` if R_1 is a parent role of R_2.

role-subsumes-p *function*

Description: Checks if two roles are subsuming each other.

Syntax: `(role-subsumes-p` R_1 R_2 *tbox*`)`

Arguments:
- R_1 - role term of the subsuming role
- R_2 - role term of the subsumed role
- *tbox* - TBox object

Values: Returns `t` if R_1 is a parent role of R_2.

role-equivalent? *KRSS macro*

Description: Checks if two roles are equivalent.

Syntax: `(role-equivalent?` R_1 R_2
`&optional (`*TBN* `(current-tbox)))`

Arguments:
- R_1 - role term of the subsuming role
- R_2 - role term of the subsumed role
- *TBN* - TBox name

Values: Returns `t` if R_1 is an equivalent of R_2.

role-equivalent-p *function*

Description: Checks if two roles are equivalent.

Syntax: `(role-equivalent-p` R_1 R_2 *tbox*`)`

Arguments:
- R_1 - role term of the subsuming role
- R_2 - role term of the subsumed role
- *tbox* - TBox object

Values: Returns `t` if R_1 is an equivalent of R_2.

role-p *function*

Description: Checks if R is a role term for a role in the specified TBox.

Syntax: `(role-p` R `&optional (`*tbox* `(current-tbox)))`

Arguments: R - role term
tbox - TBox object

Values: Returns `t` if R is a known role term and `nil` otherwise.

role? *macro*

Description: Checks if R is a role term for a role in the specified TBox.

Syntax: `(role?` R `&optional (`*TBN* `(current-tbox)))`

Arguments: R - role term
TBN - TBox name

Values: Returns `t` if R is a known role term and `nil` otherwise.

transitive-p *function*

Description: Checks if R is a transitive role in the specified TBox.

Syntax: `(transitive-p` R `&optional (`*tbox* `(current-tbox)))`

Arguments: R - role term
tbox - TBox object

Values: Returns `t` if the role R is transitive in *tbox* and `nil` otherwise.

transitive? *macro*

Description: Checks if R is a transitive role in the specified TBox.

Syntax: `(transitive?` R `&optional (`*TBN* `(current-tbox)))`

Arguments: R - role term
TBN - TBox name

Values: Returns `t` if the role R is transitive in *TBN* and `nil` otherwise.

feature-p *function*

Description: Checks if R is a feature in the specified TBox.

Syntax: `(feature-p` R `&optional (`*tbox* `(current-tbox)))`

Arguments: R - role term

tbox - TBox object

Values: Returns `t` if the role R is a feature in *tbox* and `nil` otherwise.

feature? *macro*

Description: Checks if R is a feature in the specified TBox.

Syntax: `(feature?` R `&optional (`*TBN* `(current-tbox)))`

Arguments: R - role term

TBN - TBox name

Values: Returns `t` if the role R is a feature in *TBN* and `nil` otherwise.

cd-attribute-p *function*

Description: Checks if AN is a concrete domain attribute in the specified TBox.

Syntax: `(cd-attribute-p` AN `&optional (`*tbox* `(current-tbox)))`

Arguments: AN - attribute name

tbox - TBox object

Values: Returns `t` if AN is a concrete domain attribute in *tbox* and `nil` otherwise.

cd-attribute? *macro*

Description: Checks if *AN* is a concrete domain attribute in the specified TBox.

Syntax: `(cd-attribute?` *AN* `&optional`
`(`*TBN* `(current-tbox)))`

Arguments: *AN* - attribute name

TBN - TBox name

Values: Returns `t` if the role *AN* is a concrete domain attribute in *TBN* and `nil` otherwise.

symmetric-p *function*

Description: Checks if *R* is symmetric in the specified TBox.

Syntax: `(symmetric-p` *R* `&optional (`*tbox* `(current-tbox)))`

Arguments: *R* - role term

tbox - TBox object

Values: Returns `t` if the role *R* is symmetric in *tbox* and `nil` otherwise.

symmetric? *macro*

Description: Checks if *R* is symmetric in the specified TBox.

Syntax: `(symmetric?` *R* `&optional (`*TBN* `(current-tbox)))`

Arguments: *R* - role term

TBN - TBox name

Values: Returns `t` if the role *R* is symmetric in *TBN* and `nil` otherwise.

reflexive-p *function*

Description: Checks if R is reflexive in the specified TBox.

Syntax: `(reflexive-p` R `&optional (`*tbox* `(current-tbox)))`

Arguments: R - role term

tbox - TBox object

Values: Returns `t` if the role R is reflexive in *tbox* and `nil` otherwise.

reflexive? *macro*

Description: Checks if R is reflexive in the specified TBox.

Syntax: `(reflexive?` R `&optional (`*TBN* `(current-tbox)))`

Arguments: R - role term

TBN - TBox name

Values: Returns `t` if the role R is reflexive in *TBN* and `nil` otherwise.

atomic-role-inverse *function*

Description: Returns the inverse role of role term R.

Syntax: `(atomic-role-inverse` R *tbox*)

Arguments: R - role term

tbox - TBox object

Values: Role name or term for the inverse role of R.

role-inverse *macro*

Description: Returns the inverse role of role term *R*.

Syntax: `(role-inverse` *R* `&optional (`*TBN* `(current-tbox)))`

Arguments: *R* - role term
TBN - TBox name

Values: Role name or term for the inverse role of *R*.

Remarks: This macro uses `atomic-role-inverse`.

role-domain *macro*

Description: Returns the domain of role name *RN*.

Syntax: `(role-domain` *RN* `&optional (`*TBN* `(current-tbox)))`

Arguments: *RN* - role name
TBN - TBox name

Remarks: This macro uses `atomic-role-domain`.

atomic-role-domain *function*

Description: Returns the domain of role name *RN*.

Syntax: `(atomic-role-domain` *RN* `&optional (`*TBN* `(current-tbox)))`

Arguments: *RN* - role name
TBN - TBox name

role-range *macro*

Description: Returns the range of role name *RN*.

Syntax: `(role-range` *RN* `&optional (`*TBN* `(current-tbox)))`

Arguments: *RN* - role name
TBN - TBox name

Remarks: This macro uses `atomic-role-range`.

atomic-role-range *function*

Description: Returns the range of role name *RN*.

Syntax: `(atomic-role-range` *RN* `&optional (`*TBN* `(current-tbox)))`

Arguments: *RN* - role name

TBN - TBox name

datatype-role-range *function*

Description: Returns the range of datatype property role name *RN*.

Syntax: `(datatype-role-range` *RN* *TBN*`)`

Arguments: *RN* - role name

TBN - TBox name

role-used-as-datatype-property-p *function*

Description: Returns t if the role is declared as a datatype property or nil otherwise.

Syntax: `(role-used-as-datatype-property-p` *RN* *TBN*`)`

Arguments: *RN* - role name

TBN - TBox name

role-used-as-annotation-property-p *function*

Description: Returns t if the role is declared as an annotation property or nil otherwise.

Syntax: `(role-used-as-annotation-property-p` *RN* *TBN*`)`

Arguments: *RN* - role name

TBN - TBox name

attribute-domain *macro*

Description: Returns the domain of attribute name *AN*.

Syntax: `(attribute-domain` *AN* `&optional (`*TBN* `(current-tbox)))`

Arguments: *AN* - attribute name

TBN - TBox name

attribute-domain-1 *function*

Description: Returns the domain of attribute name *AN*.

Syntax: `(attribute-domain-1` *AN* `&optional (`*TBN* `(current-tbox)))`

Arguments: *AN* - attribute name

TBN - TBox name

4.3 TBox Evaluation Functions

classify-tbox *function*

Description: Classifies the whole TBox.

Syntax: `(classify-tbox &optional (`*tbox* `(current-tbox)))`

Arguments: *tbox* - TBox object

Remarks: This function needs to be executed before queries can be posed.

check-tbox-coherence *function*

Description: This function checks if there are any unsatisfiable atomic concepts in the given TBox.

Syntax: `(check-tbox-coherence &optional (`*tbox*` (current-tbox)))`

Arguments: *tbox* - TBox object

Values: Returns a list of all atomic concepts in *tbox* that are not satisfiable, i.e. an empty list (`NIL`) indicates that there is no additional synonym to bottom.

Remarks: This function does not compute the concept hierarchy. It is much faster than `classify-tbox`, so whenever it is sufficient for your application use `check-tbox-coherence`. This function is supplied in order to check whether an atomic concept is satisfiable during the development phase of a TBox. There is no need to call the function `check-tbox-coherence` if, for instance, a certain ABox is to be checked for consistency (with `abox-consistent-p`).

tbox-classified-p *function*

Description: It is checked if the specified TBox has already been classified.

Syntax: `(tbox-classified-p &optional (`*tbox*` (current-tbox)))`

Arguments: *tbox* - TBox object

Values: Returns `t` if the specified TBox has been classified, otherwise it returns `nil`.

tbox-classified? *macro*

Description: It is checked if the specified TBox has already been classified.

Syntax: `(tbox-classified? &optional (`*TBN*` (current-tbox)))`

Arguments: *TBN* - TBox name

Values: Returns `t` if the specified TBox has been classified, otherwise it returns `nil`.

tbox-prepared-p *function*

Description: It is checked if internal index structures are already computed for the specified TBox.

Syntax: `(tbox-prepared-p &optional (`*tbox*` (current-tbox)))`

Arguments: *tbox* - TBox object

Values: Returns `t` if the specified TBox has been processed (to some extent), otherwise it returns `nil`.

Remarks: The function is used to determine whether Racer has spent some effort in processing the axioms of the TBox.

tbox-prepared? *macro*

Description: It is checked if internal index structures are already computed for the specified TBox.

Syntax: `(tbox-prepared? &optional (`*TBN*` (current-tbox)))`

Arguments: *TBN* - TBox name

Values: Returns `t` if the specified TBox has been processed (to some extent), otherwise it returns `nil`.

Remarks: The form is used to determine whether Racer has spent some effort in processing the axioms of the TBox.

tbox-cyclic-p *function*

Description: It is checked if cyclic GCIs are present in a TBox

Syntax: `(tbox-cyclic-p &optional (`*tbox*` (current-tbox)))`

Arguments: *tbox* - TBox object

Values: Returns `t` if the specified TBox contains cyclic GCIs otherwise it returns `nil`.

Remarks: Cyclic GCIs can be given either directly as a GCI or can implicitly result from processing, for instance, disjointness axioms.

tbox-cyclic? *macro*

Description: It is checked if cyclic GCIs are present in a TBox

Syntax: `(tbox-cyclic? &optional (`*tbox*` (current-tbox)))`

Arguments: *tbox* - TBox object

Values: Returns `t` if the specified TBox contains cyclic GCIs otherwise it returns `nil`.

Remarks: Cyclic GCIs can be given either directly as a GCI or can implicitly result from processing, for instance, disjointness axioms.

tbox-coherent-p *function*

Description: This function checks if there are any unsatisfiable atomic concepts in the given TBox.

Syntax: `(tbox-coherent-p &optional (`*tbox*` (current-tbox)))`

Arguments: *tbox* - TBox object

Values: Returns `nil` if there is an inconsistent atomic concept, otherwise it returns `t`.

Remarks: This function calls `check-tbox-coherence` if necessary.

tbox-coherent? *macro*

Description: Checks if there are any unsatisfiable atomic concepts in the current or specified TBox.

Syntax: `(tbox-coherent? &optional (`*TBN*` (current-tbox)))`

Arguments: *TBN* - TBox name

Values: Returns `t` if there is an inconsistent atomic concept, otherwise it returns `nil`.

Remarks: This macro uses `tbox-coherent-p`.

get-tbox-language *function*

Description: Returns a specifier indicating the description logic language used in the axioms of a given TBox.

Syntax: `(get-tbox-language &optional (`*TBN*` (current-tbox)))`

Arguments: *TBN* - TBox name

Values: The language is indicated with the quasi-standard scheme using letters. Note that the language is identified for selecting optimization techniques. Since RACER does not exploit optimization techniques for sublanguages of $\mathcal{ALC}$, the language indicator starts always with `ALC`. Then `f` indicates whether features are used, `Q` indicates qualified number restrictions, `N` indicates simple number restrictions, `H` stands for a role hierarchy, `I` indicates inverse roles, `r+` indicates transitive roles, the suffix `-D` indicates the use of concrete domain language constructs.

get-meta-constraint *function*

Description: Optimized DL systems perform a static analysis of given terminological axioms. The axioms of a TBox are usually transformed in such a way that processing promises to be faster. In particular, the idea is to transform GCIs into (primitive) concept definitions. Since it is not always possible to "absorb" GCIs completely, a so-called meta constraint might remain. The functions `get-meta-constraint` returns the remaining constraint as a concept.

Syntax: `(get-meta-constraint &optional (`*TBN*` (current-tbox)))`

Arguments: *TBN* - TBox name

Values: A concept term.

Remarks: The absorption process uses heuristics. Changes to a TBox might have dramatic effects on the value returned by `get-meta-constraint`.

get-concept-definition *macro*

Description: Optimized DL systems perform a static analysis of given terminological axioms. The axioms of a TBox are usually transformed in such a way that processing promises to be faster. In particular, the idea is to transform GCIs into (primitive) concept definitions. For a given concept name the function `get-concept-definition` returns the definition compiled by RACER during the absorption phase.

Syntax: `(get-concept-definition` *CN* `&optional (`*TBN* `(current-tbox)))`

Arguments: *CN* - concept name

TBN - TBox name

Values: A concept term.

Remarks: The absorption process uses heuristics. Changes to a TBox might have dramatic effects on the value returned by `get-concept-definition`. Note that it might be useful to test whether the definition is primitive. See the function `concept-primitive-p`. RACER does not introduce new concept names for primitive definitions.

get-concept-definition-1 *function*

Description: Functional interface for `get-concept-definition`

Syntax: `(get-concept-definition-1` *CN* `&optional (`*TBN* `(current-tbox)))`

Arguments: *CN* - concept name

TBN - TBox name

Remarks: The absorption process uses heuristics. Changes to a TBox might have dramatic effects on the value returned by `get-concept-negated-definition`. Note that it might be useful to test whether the definition is primitive. See the function `concept-primitive-p`. RACER does not introduce new concept names for primitive definitions.

Examples: Assume the following TBox:

```
(in-tbox test)
  (implies top (or a b c))
```

Then, `(get-concept-negated-definition c)` returns `(OR A B)`. Thus, RACER has transformed the GCI into the form `(implies (not C) (OR A B))` which can be handled more effectively be lazy unfolding. Note that the absorption process is heuristic. RACER could also transform the GCI into `(implies (not B) (OR A C))` or something similar depending on the current version and strategy.

get-concept-negated-definition *macro*

Description: Optimized DL systems perform a static analysis of given terminological axioms. The axioms of a TBox are usually transformed in such a way that processing promises to be faster. In particular, the idea is to transform GCIs into (primitive) concept definitions. For a given concept name the function `get-concept-negated-definition` returns the definition of the negated concept compiled by RACER during the absorption phase.

Syntax: `(get-concept-negated-definition` *CN* `&optional (`*TBN* `(current-tbox)))`

Arguments: *CN* - concept name

TBN - TBox name

get-concept-negated-definition-1 *function*

Description: Functional interface for `get-concept-negated-definition`.

Syntax: `(get-concept-negated-definition-1` *CN* `&optional (`*TBN* `(current-tbox)))`

Arguments: *CN* - concept name

TBN - TBox name

get-concept-pmodel *function*

Description: Returns a so-called pseudo model for a concept.

Syntax: `(get-concept-pmodel concept &optional (`*TBN*` (current-tbox)))`

Arguments: *concept* - concept term

TBN - TBox name

Values: Returns a list (name positive-literals negative-literals exists restricts attributes ensured-attributes unique-p).

Examples:
```
(in-knowledge-base test)
  (implies a (and e (some r c)))
  (implies b (and (not f) (all r d) ))
  (equivalent c (and a b))
  (get-concept-pmodel '(and a b) 'test)
  returns (C (C A B E) (F) (R) (R) NIL NIL T)
```

4.4 ABox Evaluation Functions

realize-abox *function*

Description: This function checks the consistency of the ABox and computes the most-specific concepts for each individual in the ABox.

Syntax: `(realize-abox &optional (`*abox*` (current-abox)))`

Arguments: *abox* - ABox object

Values: *abox*

Remarks: This Function needs to be executed before queries can be posed. If the TBox has changed and is classified again the ABox has to be realized, too.

abox-realized-p *function*

Description: Returns `t` if the specified ABox object has been realized.

Syntax: `(abox-realized-p &optional (`*abox*` (current-abox)))`

Arguments: *abox* - ABox object

Values: Returns `t` if *abox* has been realized and `nil` otherwise.

abox-realized? *macro*

Description: Returns `t` if the specified ABox object has been realized.

Syntax: `(abox-realized? &optional (`*ABN*` (current-abox)))`

Arguments: *ABN* - ABox name

Values: Returns `t` if *ABN* has been realized and `nil` otherwise.

prepare-abox *function*

Description: Compute internal data structures for processing abox assertions.

Syntax: `(prepare-abox &optional (`*abox*` (current-abox)))`

Arguments: *abox* - abox object

Remarks: This function is useful for benchmarks. You can explicitly measure the so-called preparation time (encoding of concept terms etc. in ABox assertions).

prepare-racer-engine *function*

Description: Compute internal data structures for instance retrieval.

Syntax: `(prepare-racer-engine &key (`*abox*` (current-abox))`
`(classify-tbox-p nil))`

Arguments: *abox* - abox object

classify − *tbox* − *p* - t or nil

Remarks: This function is useful for benchmarks. You can explicitly measure the time for computing index structures for answering nRQL queries.

abox-prepared-p *function*

Description: It is checked if internal index structures are already computed for the specified abox.

Syntax: `(abox-prepared-p &optional (`*abox*` (current-abox)))`

Arguments: *abox* - abox object

Values: Returns `t` if the specified abox has been processed (to some extent), otherwise it returns `nil`.

Remarks: The function is used to determine whether Racer has spent some effort in processing the assertions of the abox.

abox-prepared? *macro*

Description: It is checked if internal index structures are already computed for the specified abox.

Syntax: `(abox-prepared? &optional (`*TBN*` (current-abox)))`

Arguments: *ABN* - abox name

Values: Returns `t` if the specified abox has been processed (to some extent), otherwise it returns `nil`.

Remarks: The form is used to determine whether Racer has spent some effort in processing the assertions of the abox.

compute-all-implicit-role-fillers *function*

Description: Instruct RACER to use compute all implicit role fillers. After computing these fillers, the function all-role-assertions returns also the implicit role fillers.

Syntax: `(compute-all-implicit-role-fillers &optional (`*ABN* `(current-abox)))`)

Arguments: *ABN* - ABox name

compute-implicit-role-fillers *function*

Description: Instruct RACER to use compute all implicit role fillers for the individual specified. After computing these fillers, the function all-role-assertions returns also the implicit role fillers for the individual specified.

Syntax: `(compute-implicit-role-fillers individual &optional (`*ABN* `(current-abox)))`

Arguments: *individual* - individual name

ABN - ABox name

get-abox-language *function*

Description: Returns a specifier indicating the description logic language used in the axioms of a given ABox.

Syntax: `(get-abox-language &optional (`*ABN* `(current-abox)))`

Arguments: *ABN* - ABox name

Values: The language is indicated with the quasi-standard scheme using letters. Note that the language is identified for selecting optimization techniques. Since RACER does not exploit optimization techniques for sublanguages of $\mathcal{ALC}$, the language indicator starts always with `ALC`. Then `f` indicates whether features are used, `Q` indicates qualified number restrictions, `N` indicates simple number restrictions, `H` stands for a role hierarchy, `I` indicates inverse roles, `r+` indicates transitive roles, the suffix `-D` indicates the use of concrete domain language constructs.

4.5 ABox Queries

abox-consistent-p *function*

Description: Checks if the ABox is consistent, e.g. it does not contain a contradiction.

Syntax: `(abox-consistent-p &optional (`*abox* `(current-abox)))`

Arguments: *abox* - ABox object

Values: Returns `t` if *abox* is consistent and `nil` otherwise.

abox-consistent? *macro*

Description: Checks if the ABox is consistent.

Syntax: `(abox-consistent? &optional (`*ABN*` (current-abox)))`

Arguments: *ABN* - ABox name

Values: Returns `t` if the ABox *ABN* is consistent and `nil` otherwise.

Remarks: This macro uses `abox-consistent-p`.

abox-una-consistent-p *function*

Description: Checks if the ABox is consistent, e.g. it does not contain a contradiction if the unique name assumption is imposed.

Syntax: `(abox-una-consistent-p &optional (`*abox*` (current-abox)))`

Arguments: *abox* - ABox object

Values: Returns `t` if *abox* is consistent w.r.t. the unique name assumption and `nil` otherwise.

abox-una-consistent? *macro*

Description: Checks if the ABox is consistent if the unique name assumption is imposed.

Syntax: `(abox-una-consistent? &optional (`*ABN*` (current-abox)))`

Arguments: *ABN* - ABox name

Values: Returns `t` if the ABox *ABN* is consistent w.r.t. the unique name assumption and `nil` otherwise.

Remarks: This macro uses `abox-una-consistent-p`.

check-abox-coherence *function*

Description: Checks if the ABox is consistent. If there is a contradiction, this function prints information about the culprits.

Syntax: `(check-abox-coherence &optional (`*abox*` (current-abox))`
`(stream *standard-output*)`

Arguments: *abox* - ABox object

stream - Stream object

Values: Returns `t` if *abox* is consistent and `nil` otherwise.

individual-instance? *KRSS macro*

Description: Checks if an individual is an instance of a given concept with respect to the `(current-abox)` and its TBox.

Syntax: `(individual-instance?` *IN C*
`&optional (`*abox*` (current-abox)))`

Arguments: *IN* - individual name

C - concept term

abox - ABox object

Values: Returns `t` if *IN* is an instance of *C* in *abox* and `nil` otherwise.

individual-instance-p *function*

Description: Checks if an individual is an instance of a given concept with respect to an ABox and its TBox.

Syntax: `(individual-instance-p` *IN C abox*`)`

Arguments: *IN* - individual name

C - concept term

abox - ABox object

Values: Returns `t` if *IN* is an instance of *C* in *abox* and `nil` otherwise.

constraint-entailed? *macro*

Description: Checks a specified constraint is entailed by an ABox (and its associated TBox).

Syntax: `(constraint-entailed? constraint &optional (`*abox* `(current-abox)))`

Arguments: *constraint* - A constraint

abox - ABox object

Values: Returns `t` if *abox* the constraint and `nil` otherwise.

constraint-entailed-p *function*

Description: Checks a specified constraint is entailed by an ABox (and its associated TBox).

Syntax: `(constraint-entailed-p constraint &optional (`*abox* `(current-abox)))`

Arguments: *constraint* - A constraint

abox - ABox object

Values: Returns `t` if *abox* the constraint and `nil` otherwise.

individuals-related? *macro*

Description: Checks if two individuals are directly related via the specified role.

Syntax: `(individuals-related?` IN_1 IN_2 R `&optional (`*abox* `(current-abox)))`

Arguments: IN_1 - individual name of the predecessor

IN_2 - individual name of the role filler

R - role term

abox - ABox object

Values: Returns `t` if IN_1 is related to IN_2 via R in *abox* and `nil` otherwise.

individuals-related-p *function*

Description: Checks if two individuals are directly related via the specified role.

Syntax: `(individuals-related-p` IN_1 IN_2 R *abox*`)`

Arguments: IN_1 - individual name of the predecessor

IN_2 - individual name of the role filler

R - role term

abox - ABox object

Values: Returns `t` if IN_1 is related to IN_2 via R in *abox* and `nil` otherwise.

See also: Function `retrieve-individual-filled-roles`, on page 109,
Function `retrieve-related-individuals`, on page 108.

individuals-equal? *KRSS macro*

Description: Checks if two individual names refer to the same domain object.

Syntax: `(individuals-equal?` IN_1 IN_2 `&optional (`*abox* `(current-abox)))`

Arguments: IN_1, IN_2 - individual name

abox - abox object

individuals-equal-p *function*

Description: Functional equivalent to `individuals-equal?`, Page 88.

individuals-not-equal? *KRSS macro*

Description: Checks if two individual names do not refer to the same domain object.

Syntax: `(individuals-not-equal?` IN_1 IN_2
`&optional (`*abox* `(current-abox)))`

Arguments: IN_1, IN_2 - individual name

abox - abox object

Remarks: Because the unique name assumption holds in RACER this macro always returns `t` for individuals with different names. This macro is just supplied to be compatible with the KRSS.

individuals-not-equal-p *function*

Description: Functional equivalent to `individuals-not-equal?`, Page 89.

individual-p *function*

Description: Checks if *IN* is a name of an individual mentioned in an ABox *abox*.

Syntax: `(individual-p` *IN* `&optional (`*abox* `(current-abox)))`

Arguments: *IN* - individual name

abox - ABox object

Values: Returns `t` if *IN* is a name of an individual and `nil` otherwise.

individual? *macro*

Description: Checks if *IN* is a name of an individual mentioned in an ABox *ABN*.

Syntax: `(individual?` *IN* `&optional (`*ABN* `(current-abox)))`

Arguments: *IN* - individual name

ABN - ABox name

Values: Returns `t` if *IN* is a name of an individual and `nil` otherwise.

cd-object-p *function*

Description: Checks if *ON* is a name of a concrete domain object mentioned in an ABox *abox*.

Syntax: `(cd-object-p` *ON* `&optional (`*abox* `(current-abox)))`

Arguments: *ON* - concrete domain object name
abox - ABox object

Values: Returns `t` if *ON* is a name of a concrete domain object and `nil` otherwise.

cd-object? *macro*

Description: Checks if *ON* is a name of a concrete domain object mentioned in an ABox *ABN*.

Syntax: `(cd-object?` *ON* `&optional (`*ABN* `(current-abox)))`

Arguments: *ON* - concrete domain object name
ABN - ABox name

Values: Returns `t` if *ON* is a name of a concrete domain object and `nil` otherwise.

get-individual-pmodel *function*

Description: Returns a so-called pseudo model for an individual.

Syntax: `(get-individual-pmodel IN &optional (`*TBN* `(current-tbox)))`

Arguments: *IN* - individual name
TBN - TBox name

Values: Returns a list (name positive-literals negative-literals exists restricts attributes ensured-attributes unique-p).

Examples:
```
(in-knowledge-base test)
  (implies a (and e (some r c)))
  (implies b (and (not f) (all r d) ))
  (equivalent c (and a b))
  (get-individual-pmodel '(and a b) 'test)
   returns ((I) (E B A C) (F) (R S) (R) NIL NIL T)
```

Chapter 5

Retrieval

If the retrieval refers to concept names, RACER always returns a set of names for each concept name. A so called name set contains all synonyms of an atomic concept in the TBox.

5.1 TBox Retrieval

---

taxonomy *function*

Description: Returns the whole taxonomy for the specified TBox.

Syntax: `(taxonomy &optional (`*tbox*` (current-tbox)))`

Arguments: *tbox* - TBox object

Values: A list of triples, each of it consisting of:

a name set - the atomic concept *CN* and its synonyms

list of concept-parents name sets - each entry being a list of a concept parent of *CN* and its synonyms

---

list of concept-children name sets - each entry being a list of a concept child of *CN* and its synonyms.

Examples: `(taxonomy my-TBox)`
may yield:
```
(((*top*) () ((quadrangle tetragon)))
  ((quadrangle tetragon) ((*top*)) ((rectangle) (diamond)))
  ((rectangle) ((quadrangle tetragon)) ((*bottom*)))
  ((diamond) ((quadrangle tetragon)) ((*bottom*)))
  ((*bottom*) ((rectangle) (diamond)) ()))
```

See also: Function `atomic-concept-parents`,
function `atomic-concept-children` on page 95.

concept-synonyms *macro*

Description: Returns equivalent concepts for the specified concept in the given TBox.

Syntax: `(concept-synonyms` *CN*
`&optional (`*tbox* `(current-tbox)))`

Arguments: *CN* - concept name
tbox - TBox object

Values: List of concept names

Remarks: The name *CN* is not included in the result.

See also: Function `concept-equivalent-p`, on page 63.

atomic-concept-synonyms *function*

Description: Returns equivalent concepts for the specified concept in the given TBox.

Syntax: `(atomic-concept-synonyms` *CN* *tbox*`)`

Arguments: *CN* - concept name
tbox - TBox object

Values: List of concept names

Remarks: The name *CN* is included in the result.

See also: Function `concept-equivalent-p`, on page 63.

concept-descendants *KRSS macro*

Description: Gets all atomic concepts of a TBox, which are subsumed by the specified concept.

Syntax: `(concept-descendants` *C*
`&optional (`*TBN* `(current-tbox)))`

Arguments: *C* - concept term
TBN - TBox name

Values: List of name sets

Remarks: This macro return the transitive closure of the macro `concept-children`.

atomic-concept-descendants *function*

Description: Gets all atomic concepts of a TBox, which are subsumed by the specified concept.

Syntax: `(atomic-concept-descendants` *C tbox*`)`

Arguments: *C* - concept term
tbox - TBox object

Values: List of name sets

Remarks: Returns the transitive closure from the call of `atomic-concept-children`.

concept-ancestors *KRSS macro*

Description: Gets all atomic concepts of a TBox, which are subsuming the specified concept.

Syntax: `(concept-ancestors` *C*
`&optional (`*TBN* `(current-tbox)))`

Arguments: *C* - concept term
TBN - TBox name

Values: List of name sets

Remarks: This macro return the transitive closure of the macro `concept-parents`.

atomic-concept-ancestors *function*

Description: Gets all atomic concepts of a TBox, which are subsuming the specified concept.

Syntax: `(atomic-concept-ancestors` *C tbox*`)`

Arguments: *C* - concept term

tbox - TBox object

Values: List of name sets

Remarks: Returns the transitive closure from the call of `atomic-concept-parents`.

concept-children *KRSS macro*

Description: Gets the direct subsumees of the specified concept in the TBox.

Syntax: `(concept-children` *C*
`&optional (`*TBN* `(current-tbox)))`

Arguments: *C* - concept term

TBN - TBox name

Values: List of name sets

Remarks: Is the equivalent macro for the KRSS macro `concept-offspring`, which is also supplied in RACER.

atomic-concept-children *function*

Description: Gets the direct subsumees of the specified concept in the TBox.

Syntax: `(atomic-concept-children` *C tbox*`)`

Arguments: *C* - concept term

tbox - TBox object

Values: List of name sets

concept-parents *KRSS macro*

Description: Gets the direct subsumers of the specified concept in the TBox.

Syntax: `(concept-parents` *C*
`&optional (`*TBN* `(current-tbox)))`

Arguments: *C* - concept term

TBN - TBox name

Values: List of name sets

atomic-concept-parents *function*

Description: Gets the direct subsumers of the specified concept in the TBox.

Syntax: `(atomic-concept-parents` *C* *tbox*`)`

Arguments: *C* - concept term

tbox - TBox object

Values: List of name sets

role-descendants *KRSS macro*

Description: Gets all roles from the TBox, that the given role subsumes.

Syntax: `(role-descendants` *R*
`&optional (`*TBN* `(current-tbox)))`

Arguments: *R* - role term

TBN - TBox name

Values: List of role terms

Remarks: This macro is the transitive closure of the macro `role-children`.

atomic-role-descendants *function*

Description: Gets all roles from the TBox, that the given role subsumes.

Syntax: `(atomic-role-descendants` *R tbox*`)`

Arguments: *R* - role term

tbox - TBox object

Values: List of role terms

Remarks: This function is the transitive closure of the function `atomic-role-descendants`.

role-ancestors *KRSS macro*

Description: Gets all roles from the TBox, that subsume the given role in the role hierarchy.

Syntax: `(role-ancestors` *R*
`&optional (`*TBN* `(current-tbox)))`

Arguments: *R* - role term

TBN - TBox name

Values: List of role terms

atomic-role-ancestors *function*

Description: Gets all roles from the TBox, that subsume the given role in the role hierarchy.

Syntax: `(atomic-role-ancestors` *R tbox*`)`

Arguments: *R* - role term

tbox - TBox object

Values: List of role terms

role-children *macro*

Description: Gets all roles from the TBox that are directly subsumed by the given role in the role hierarchy.

Syntax: `(role-children` *R*
`&optional (`*TBN* `(current-tbox)))`

Arguments: *R* - role term

TBN - TBox name

Values: List of role terms

Remarks: This is the equivalent macro to the KRSS macro `role-offspring`, which is also supplied by the RACER system.

atomic-role-children *function*

Description: Gets all roles from the TBox that are directly subsumed by the given role in the role hierarchy.

Syntax: `(atomic-role-children` *R tbox*`)`

Arguments: *R* - role term

tbox - TBox object

Values: List of role terms

role-parents *KRSS macro*

Description: Gets the roles from the TBox that directly subsume the given role in the role hierarchy.

Syntax: `(role-parents` *R* `&optional (`*TBN* `(current-tbox)))`

Arguments: *R* - role term

TBN - TBox name

Values: List of role terms

atomic-role-parents *function*

Description: Gets the roles from the TBox that directly subsume the given role in the role hierarchy.

Syntax: `(atomic-role-parents` *R tbox*`)`

Arguments: *R* - role term
tbox - TBox object

Values: List of role terms

role-synonyms *KRSS macro*

Description: Gets the synonyms of a role including the role itself.

Syntax: `(role-synonyms` *R* `&optional (`*TBN* `(current-tbox)))`

Arguments: *R* - role term
TBN - TBox name

Values: List of role terms

atomic-role-synonyms *function*

Description: Gets the synonyms of a role including the role itself.

Syntax: `(atomic-role-synonyms` *R tbox*`)`

Arguments: *R* - role term
tbox - TBox object

Values: List of role terms

all-tboxes *function*

Description: Returns the names of all known TBoxes.

Syntax: `(all-tboxes)`

Values: List of TBox names

all-atomic-concepts *function*

Description: Returns all atomic concepts from the specified TBox.

Syntax: `(all-atomic-concepts &optional (`*tbox*` (current-tbox)))`

Arguments: *tbox* - TBox object

Values: List of concept names

all-equivalent-concepts *function*

Description: xx

Syntax: `(all-equivalent-concepts &optional (`*tbox*` (current-tbox)))`

Arguments: *tbox* - TBox object

Values: List of name sets

all-roles *function*

Description: Returns all roles and features from the specified TBox.

Syntax: `(all-roles &optional (`*tbox*` (current-tbox)))`

Arguments: *tbox* - TBox object

Values: List of role terms

Examples: `(all-roles (find-tbox 'my-tbox))`

all-features *function*

Description: Returns all features from the specified TBox.

Syntax: `(all-features &optional (`*tbox*` (current-tbox)))`

Arguments: *tbox* - TBox

Values: List of feature terms

all-attributes *function*

Description: Returns all attributes from the specified TBox.

Syntax: `(all-attributes &optional (`*tbox*` (current-tbox)))`

Arguments: *tbox* - TBox

Values: List of attributes names

attribute-type *function*

Description: Returns the attribute type declared for a given attribute name in a specified TBox.

Syntax: `(attribute-type` *AN* `&optional (`*tbox*` (current-tbox)))`

Arguments: *AN* - attribute name

tbox - TBox

Values: Either cardinal, integer, real, or complex.

all-transitive-roles *function*

Description: Returns all transitive roles from the specified TBox.

Syntax: `(all-transitive-roles &optional (`*tbox*` (current-tbox)))`

Arguments: *tbox* - TBox object

Values: List of transitive role terms

describe-tbox *function*

Description: Generates a description for the specified TBox.

Syntax: `(describe-tbox &optional (`*tbox*` (current-tbox)) (`*stream*` *standard-output*))`

Arguments: *tbox* - TBox object or TBox name
stream - open stream object

Values: *tbox*
The description is written to *stream*.

describe-concept *function*

Description: Generates a description for the specified concept used in the specified TBox or in the ABox and its TBox.

Syntax: `(describe-concept` *CN* `&optional (`*tbox-or-abox*` (current-tbox)) (`*stream*` *standard-output*))`

Arguments: *tbox-or-abox* - TBox object or ABox object
CN - concept name
stream - open stream object

Values: *tbox-or-abox*
The description is written to *stream*.

describe-role *function*

Description: Generates a description for the specified role used in the specified TBox or ABox.

Syntax: `(describe-role` *R* `&optional (`*tbox-or-abox*` (current-tbox)) (`*stream*` *standard-output*))`

Arguments: *tbox-or-abox* - TBox object or ABox object
R - role term (or feature term)
stream - open stream object

Values: *tbox-or-abox*
The description is written to *stream*.

5.2 ABox Retrieval

individual-direct-types *KRSS macro*

Description: Gets the most-specific atomic concepts of which an individual is an instance.

Syntax: `(individual-direct-types` *IN*
`&optional (`*ABN* `(current-abox)))`

Arguments: *IN* - individual name

ABN - ABox name

Values: List of name sets

most-specific-instantiators *function*

Description: Gets the most-specific atomic concepts of which an individual is an instance.

Syntax: `(most-specific-instantiators` *IN* *abox*`)`

Arguments: *IN* - individual name

abox - ABox object

Values: List of name sets

individual-types *KRSS macro*

Description: Gets *all* atomic concepts of which the individual is an instance.

Syntax: `(individual-types` *IN*
`&optional (`*ABN* `(current-abox)))`

Arguments: *IN* - individual name

ABN - ABox name

Values: List of name sets

Remarks: This is the transitive closure of the KRSS macro `individual-direct-types`.

instantiators *function*

Description: Gets *all* atomic concepts of which the individual is an instance.

Syntax: `(instantiators` *IN abox*`)`

Arguments: *IN* - individual name

abox - ABox object

Values: List of name sets

Remarks: This is the transitive closure of the function `most-specific-instantiators`.

concept-instances *KRSS macro*

Description: Gets all individuals from an ABox that are instances of the specified concept.

Syntax: `(concept-instances` *C*
`&optional (`*ABN* `(current-abox)) (`*candidates*`)`

Arguments: *C* - concept term

ABN - ABox name

candidates - a list of individual names

Values: List of individual names

retrieve-concept-instances *function*

Description: Gets all individuals from an ABox that are instances of the specified concept.

Syntax: `(retrieve-concept-instances` *C abox candidates*`)`

Arguments: *C* - concept term

abox - ABox object

candidates - a list of individual names

Values: List of individual names

individual-synonyms *Macro*

Description: Gets all individuals which can be proven to refer to the same domain object.

Syntax: `(individual-synonyms` *IN* `&optional (`*ABN* `(current-abox)))`

Arguments: *IN* - individual name

ABN - ABox name

Values: List of individual names

retrieve-individual-synonyms *function*

Description: Gets all individuals which can be proven to refer to the same domain object.

Syntax: `(retrieve-individual-fillers` *IN* *abox*`)`

Arguments: *IN* - individual name

abox - ABox name

Values: List of individual names

individual-fillers *KRSS macro*

Description: Gets all individuals that are fillers of a role for a specified individual.

Syntax:
```
(individual-fillers IN R
  &optional (ABN (current-abox)))
```

Arguments: *IN* - individual name of the predecessor

R - role term

ABN - ABox name

Values: List of individual names

Examples:
```
(individual-fillers Charlie-Brown has-pet)
(individual-fillers Snoopy (inv has-pet))
```

retrieve-individual-fillers *function*

Description: Gets all individuals that are fillers of a role for a specified individual.

Syntax: `(retrieve-individual-fillers` *IN R abox*`)`

Arguments: *IN* - individual name of the predecessor

R - role term

abox - ABox object

Values: List of individual names

Examples:
```
(retrieve-individual-fillers 'Charlie-Brown 'has-pet
  (find-abox 'peanuts-characters))
```

individual-attribute-fillers *macro*

Description: Gets all object names that are fillers of an attribute for a specified individual.

Syntax:
```
(individual-attribute-fillers IN AN
  &optional (ABN (current-abox)))
```

Arguments: *IN* - individual name of the predecessor

AN - attribute-name

ABN - ABox name

Values: List of object names

retrieve-individual-attribute-fillers *function*

Description: Gets all object names that are fillers of an attribute for a specified individual.

Syntax:
```
(retrieve-individual-attribute-fillers IN AN
  &optional (ABN (current-abox)))
```

Arguments: *IN* - individual name of the predecessor

AN - attribute-name

ABN - ABox name

Values: List of object names

told-value *function*

Description: Returns an explicitly asserted value for an object that is declared as filler for a certain attribute w.r.t. an individual.

Syntax: `(told-value` *ON*
`&optional (`*ABN* `(current-abox)))`

Arguments: *ON* - object name

ABN - ABox name

Values: Concrete domain value

individual-told-attribute-fillers *macro*

Description: Gets object names which are fillers for attributes.

Syntax: `(individual-told-attribute-fillers` *IN AN*
`&optional (`*ABN* `(current-abox)))`

Arguments: *IN* - individual name of the predecessor

RN - attribute name

ABN - ABox name

Values: List of object names whose type is determined by the type of the attribute.

retrieve-individual-told-attribute-fillers *Function*

Description: Functional equivalent of `individual-told-attribute-fillers`, Page 106.

individual-told-attribute-value *macro*

Description: Gets told values for attributes.

Syntax: `(individual-told-attribute-value` *IN AN* `&optional (`*ABN* `(current-abox)))`

Arguments: *IN* - individual name of the predecessor

RN - attribute name

ABN - ABox name

Values: List of values whose type is determined by the type of the attribute.

retrieve-individual-told-attribute-value *Function*

Description: Functional equivalent of `individual-told-attribute-value`, Page 107.

individual-told-datatype-fillers *function*

Description: Gets told values for datatype property roles.

Syntax: `(individual-told-datatype-fillers` *IN RN* `&optional (`*ABN* `(current-abox)))`

Arguments: *IN* - individual name of the predecessor

RN - datatype property role name

ABN - ABox name

Values: List of values whose type is determined by the range of the datatype property role.

retrieve-individual-told-datatype-fillers *Function*

Description: Functional equivalent of `individual-told-datatype-fillers`, Page 107.

retrieve-individual-annotation-property-fillers *function*

Description: Gets told values for attributes.

Syntax: `(individual-annotation-property-fillers` *IN AN*
`&optional (`*ABN* `(current-abox)))`

Arguments: *IN* - individual name of the predecessor

RN - attribute name

ABN - ABox name

Values: List of values whose type is determined by the type of the attribute.

related-individuals *macro*

Description: Gets all pairs of individuals that are related via the specified relation.

Syntax: `(related-individuals` *R*
`&optional (`*ABN* `(current-abox)))`

Arguments: *R* - role term

ABN - ABox name

Values: List of pairs of individual names

Examples:
```
(retrieve-related-individuals 'has-pet
  (find-abox 'peanuts-characters))
```
may yield:
```
((Charlie-Brown Snoopy) (John-Arbuckle Garfield))
```

See also: Function `individuals-related-p`, on page 88.

retrieve-related-individuals *function*

Description: Functional equivalents of `related-individuals`.

retrieve-individual-filled-roles *function*

Description: This function gets all roles that hold between the specified pair of individuals.

Syntax: `(retrieve-individual-filled-roles` IN_1 IN_2 *abox*`)`.

Arguments:
- IN_1 - individual name of the predecessor
- IN_2 - individual name of the role filler
- *abox* - ABox object

Values: List of role terms

Examples:
```
(retrieve-individual-filled-roles 'Charlie-Brown 'Snoopy
  (find-abox 'peanuts-characters))
```

See also: Function `individuals-related-p`, on page 88.

individual-filled-roles *macro*

Description: Equivalent to `retrieve-individual-filled-roles`, Page 109.

retrieve-direct-predecessors *function*

Description: Gets all individuals that are predecessors of a role for a specified individual.

Syntax: `(retrieve-direct-predecessors` *R IN abox*`)`

Arguments:
- *R* - role term
- *IN* - individual name of the role filler
- *abox* - ABox object

Values: List of individual names

Examples:
```
(retrieve-direct-predecessors 'has-pet 'Snoopy
  (find-abox 'peanuts-characters))
```

direct-predecessors *macro*

Description: Equivalent to `retrieve-direct-predecessors`, Page 109.

all-aboxes *function*

Description: Returns the names of all known ABoxes.

Syntax: `(all-aboxes)`

Values: List of ABox names

all-individuals *function*

Description: Returns all individuals from the specified ABox.

Syntax: `(all-individuals &optional (`*abox*` (current-abox)))`

Arguments: *abox* - ABox object

Values: List of individual names

all-concept-assertions-for-individual *function*

Description: Returns all concept assertions for an individual from the specified ABox.

Syntax: `(all-concept-assertions-for-individual` *IN*
`&optional (`*abox*` (current-abox)))`

Arguments: *IN* - individual name

abox - ABox object

Values: List of concept assertions

See also: Function `all-concept-assertions` on page 112.

all-role-assertions-for-individual-in-domain *function*

Description: Returns all role assertions for an individual from the specified ABox in which the individual is the role predecessor.

Syntax: `(all-role-assertions-for-individual-in-domain` *IN* `&optional (`*abox* `(current-abox)))`

Arguments: *IN* - individual name

abox - ABox object

Values: List of role assertions

Remarks: Returns only the role assertions explicitly mentioned in the ABox, not the inferred ones.

See also: Function `all-role-assertions` on page 112.

all-role-assertions-for-individual-in-range *function*

Description: Returns all role assertions for an individual from the specified ABox in which the individual is a role successor.

Syntax: `(all-role-assertions-for-individual-in-range` *IN* `&optional (`*abox* `(current-abox)))`

Arguments: *IN* - individual name

abox - ABox object

Values: List of assertions

See also: Function `all-role-assertions` on page 112.

all-concept-assertions *function*

Description: Returns all concept assertions from the specified ABox.

Syntax: `(all-concept-assertions &optional (`*abox* `(current-abox)))`

Arguments: *abox* - ABox object

Values: List of assertions

all-annotation-concept-assertions *function*

Description: Returns all annotation concept assertions from the specified ABox.

Syntax: `(all-annotation-concept-assertions &optional (`*abox*` (current-abox)))`

Arguments: *abox* - ABox object

Values: List of assertions

all-role-assertions *function*

Description: Returns all role assertions from the specified ABox.

Syntax: `(all-role-assertions &optional (`*abox*` (current-abox)))`

Arguments: *abox* - ABox object

Values: List of assertions

See also: Function `all-concept-assertions-for-individual` on page 110.

all-annotation-role-assertions *function*

Description: Returns all annotation role assertions from the specified ABox.

Syntax: `(all-annotation-role-assertions &optional (`*abox*` (current-abox)))`

Arguments: *abox* - ABox object

Values: List of assertions

all-constraints *function*

Description: Returns all constraints from the specified ABox which refer to a list of object names.

Syntax: `(all-constraints &optional (`*abox*` (current-abox)) `*ONs*`)`

Arguments: *abox* - ABox object

ONs - list of object names

Values: List of constraints

Remarks: If ONs is not specified, all constraints of the ABox are returned.

all-attribute-assertions *function*

Description: Returns all attribute assertions from the specified ABox.

Syntax: `(all-attribute-assertions &optional (`*abox*` (current-abox)))`

Arguments: *abox* - ABox object

Values: List of assertions

describe-abox *function*

Description: Generates a description for the specified ABox.

Syntax: `(describe-abox &optional (`*abox*` (current-abox))`
`  (`*stream*` *standard-output*))`

Arguments: *abox* - ABox object

stream - open stream object

Values: *abox*
The description is written to *stream*.

describe-individual *function*

Description: Generates a description for the individual from the specified ABox.

Syntax: `(describe-individual` *IN* `&optional (`*abox* `(current-abox))` `(`*stream* `*standard-output*))`

Arguments: *IN* - individual name

abox - ABox object

stream - open stream object

Values: *IN*
The description is written to *stream*.

Chapter 6

The API of the nRQL Query Processing Engine

In the following, each API function provided by nRQL is described. We differentiate between *functions* and *macros*.

Users of *Jracer, RacerPorter, RICE* or any other graphical front end tool which allows to post commands to the RacerPro server can completely ignore the difference.

However, if you are accessing the RacerPro server via the *LRacer* API which is implemented in Lisp, or you are using *RacerMaster* and RacerPro is running in the same Lisp environment, then you will need to know which arguments will be *evaluated* and which not.

In this case, if you want to call a *function*, then the Lisp environment will always evaluate *all* arguments. However, if you use a *macro call*, then the macro can chose not to evaluate certain arguments.

You can always prevent the evaluation of an argument provided to a function by *quoting* the argument with `" ' "`. A quoted argument always evaluates to itself. For example, in the function call `(racer-answer-query '(?x) '(?x woman) :abox 'smith-family)`. Thus, the expression `'(?x woman)` is taken as a (complex) literal - a constant list (tree). Note that the corresponding *macro call* looks as follows: `(retrieve (?x) (?x woman) :abox smith-family)`. In `retrieve`, Page 155 you will see that all arguments are marked with an asterix, thus, `(?x)`, `(?x woman)`, and `smith-family` are taken as literals.

Let us explain the format used for describing the API. Suppose the function `test-function` has the following syntax specification:

Syntax: `(test` *a* `&optional (`*b* `3) &key` *c* `(`*d* `4))`.

This function is named `test-function`, and has one required (mandatory) argument a. It has 3 optional arguments: b, c, d. The arguments c and d are called *keyword arguments*. If an optional (`&optional` or `&key`) parameter is specified like `(`b `3)`, then this means that there is a *default value* specified for this optional argument. Thus, if b is not explicitly specified in a call to `test-function`, it will take the specified default value, in this case 3. In case no default value is specified, the value will be `NIL`. If a function has `&optional` as well as `&key` parameters, and you want to pass it a keyword argument, then you will have to supply values for all arguments listed between `&optional` and `&key`. This is the usual Lisp way of handling optional and keyword arguments.

Thus, given the specification of `test` as above, the following calls are possible:

1. `(test-function 1)`, parameters will be bound to: a=1, b=3, c=`NIL`, d=4
2. `(test-function 2 2 :d 5)`, parameters will be bound to: a=2, b=2, c=`NIL`, d=5
3. `(test-function 2 nil :d 5 :c 6)`, parameters will be bound to: a=2, b=`NIL`, c=6, d=5
4. Note that you CANNOT make this call: `(test-function 2 :d 5 :c 6)`, since a correct value for b is missing (in fact, b is bound to the keyword symbol `:d`, but then a formal parameter is missing for the subsequent value "5").

Users of the *LRacer API* will find all functions and macros as described here.

Some API function might raise errors. However, under *values* we only describe the value which is returned in case no error has been raised.

6.1 Basic Commands

all-queries *Function*

Description: Returns a list of (the IDs of) all queries

Syntax: `(all-queries)`

Arguments:

all-rules *Function*

Description: This is the rule equivalent of `all-queries`, Page 117.

all-substrates *Function*

Description: Returns a list of all substrates. A substrate is the internal (ABox) representation that nRQL needs in order to answer queries. A substrate has a type and a corresponding Racer ABox. For each ABox and type there is at most one substrate

Syntax: `(all-substrate)`

Arguments:

Values: A list of (abox type-of-substrate) entries, denoting the name of the substrate (which is identical to the name of the associated ABox) as well as the type of the substrate.

Remarks: .

See also: `reset-all-substrates`, Page 122, `describe-all-substrates`, Page 118

delete-all-substrates *Function*

Description: Deletes all substrates

Syntax: `(delete-all-substrates)`

Arguments:

Values: `:okay-all-substrates-deleted`.

See also: `reset-all-substrates`, Page 122, `reset-nrql-engine`, Page 123, `all-substrates`, Page 117, `describe-all-substrates`, Page 118

describe-all-queries *Function*

Description: Applies `describe-query`, Page 119 to the result of `all-queries`, Page 117 and returns it

Syntax: `(describe-all-queries)`

Arguments: `rewritten-p` - t.

describe-all-rules *Function*

Description: This is the rule equivalent of `describe-all-queries`, Page 118.

describe-all-substrates *Function*

Description: Maps `describe-substrate`, Page 211 over `all-substrates`, Page 117

Syntax: `(describe-all-substrates)`

Arguments:

Values: A list containing the results of `describe-substrate`, Page 211 applied to the substrates in `all-substrates`, Page 117.

Remarks: .

See also: `describe-substrate`, Page 211

describe-query *Function*

Description: Returns a description of the query `id`

Syntax: `(describe-query id &optional rewritten-p)`

Arguments: `id` - the ID of the query, or `:last` or `:last-query`.

`rewritten-p` - t.

Values: A list of three items: The query identify, the current status description (see `describe-query-status`, Page 119), and either the rewritten or original syntactic query.

Remarks: This function uses `describe-query-status`, Page 119, `query-head`, Page 122 (or `original-query-head`, Page 121) and `query-body`, Page 122 (or `original-query-body`, Page 120) to create the description.

See also: `describe-all-queries`, Page 118

describe-query-status *Function*

Description: Describes the current status of the query `id` - whether the query is ready (to run), running, waiting (sleeping), or terminated

Syntax: `(describe-query-status id)`

Arguments: `id` - the ID of the query, or `:last` or `:last-query`.

Values: A list of status symbols describing the current status of the query.

See also: `describe-all-queries`, Page 118

describe-rule *Function*

Description: This is the rule equivalent of `describe-all-queries`, Page 118.

describe-rule-status *Function*

Description: This is the rule equivalent of `describe-query-status`, Page 119.

exit-server *Function*

Description: Exists the current RacerPro server. Command only works if RacerPro is running in unsafe mode.

full-reset *Function*

Description: Simply calls `(reset-nrql-engine :full-reset-p t)`, see `reset-nrql-engine`, Page 123.

get-nrql-version *Function*

Description: Returns the nRQL version number.

get-substrate-type *Function*

Description: Returns the type (class) of the substrates that nRQL creates internally on request

Syntax: `(get-substrate-type)`

Arguments:

Values: the type (class), a symbol.

See also: `set-substrate-type`, Page 124, `describe-query-processing-mode`, Page 183

in-unsafe-mode? *Function*

Description: Check whether RacerPro is running in unsafe mode..

original-query-body *Function*

Description: Like `query-body`, Page 122, but the original body is returned.

original-query-head *Function*

Description: Like `query-head`, Page 122, but the original head is returned.

original-rule-antecedence *Function*

Description: Like `rule-antecedence`, Page 123, but the original antecedence is returned. This is the rule equivalent of `original-query-body`, Page 120.

original-rule-consequence *Function*

Description: Like `rule-consequence`, Page 124, but the original consequence is returned. This is the rule equivalent of `original-query-head`, Page 121.

prepare-nrql-engine *Function*

Description: Prepares the internal index structures of the nRQL engine for query answering on the ABox argumentabox. Usually, there is no need to call this function. The function will be called automatically before the first query to argumentabox is executed. Thus, answering the first query to argumentabox might take considerably longer than subsequent queries to that ABox. For benchmarking purposes, the nRQL engine should thus be prepared using this function (or with `prepare-racer-engine`, Page 83) before the first query is executed

Syntax: `(prepare-nrql-engine abox &rest args)`

Arguments: `abox` - (current-abox).

`args` - see `with-nrql-settings`, Page 199.

Values: The (name of the) ABox `abox` is returned.

See also: `reset-nrql-engine`, Page 123, `prepare-racer-engine`, Page 83

query-body *Function*

Description: Returns the (possibly rewritten) body of the query `id`

Syntax: `(query-body id)`

Arguments: `id` - the ID of the query, or `:last` or `:last-query`.

Values: The (possibly rewritten) body of the query.

See also: `original-query-body`, Page 120, `query-head`, Page 122

query-head *Function*

Description: Returns the (possibly rewritten) head of the query `id`

Syntax: `(query-head id)`

Arguments: `id` - the ID of the query, or `:last` or `:last-query`.

Values: The (possibly rewritten) head of the query.

Remarks: Note that individuals in the original query head are usually yreplaced by representative variables.

See also: `original-query-head`, Page 121, `query-body`, Page 122

reset-all-substrates *Function*

Description: Resets all substrates

Syntax: `(reset-all-substrates)`

Arguments:

Values: `:okay-all-substrates-reset`.

Remarks: Does not delete anything from the server.

See also: `delete-all-substrates`, Page 118, `reset-nrql-engine`, Page 123, `all-substrates`, Page 117, `describe-all-substrates`, Page 118

reset-nrql-engine *Function*

Description: Aborts all (active) queries and rules using `abort-all-queries`, Page 136, `abort-all-rules`, Page 136, then resets the internal caches of the nRQL engine using `reset-all-substrates`, Page 122), and finally calls `restore-standard-settings`, Page 194.

If `full-reset-p` = `t` is given, nRQL will delete all TBoxes (as well as the associated ABoxes) using `delete-all-tboxes`, Page **??**), delete all the queries and rules using `delete-all-queries`, Page 125, `delete-all-rules`, Page 126, deletes all substrates (as well as the associated QBoxes) and associated defined queries

Syntax: `(reset-nrql-engine &key full-reset-p)`

Arguments: `full-reset-p`, default `nil` - pass `t` if you really want to reset the nRQL engine fully - note that this will delete everything from the RacerPro server.

Values: `:okay-full-reset` or `:okay-engine-reset`.

See also: `reset-nrql-engine`, Page 123, `restore-standard-settings`, Page 194

rule-antecedence *Function*

Description: Returns the (possibly rewritten) antecedence of the rule `id` This is the rule equivalent of `query-body`, Page 122.

Syntax: `(rule-antecedence id)`

Arguments: `id` - the ID of the rule, or `:last` or `:last-rule`.

Values: The (possibly rewritten) antecedence of the rule.

See also: Rule equivalent of `query-body`, Page 122. `original-rule-antecedence`, Page 121, `rule-consequence`, Page 124

rule-consequence *Function*

Description: Returns the (possibly rewritten) rule consequence of the rule `id` This is the rule equivalent of `query-head`, Page 122.

Syntax: `(rule-consequence id)`

Arguments: `(first id the ID of the rule, or :last or :last-rule)` - nil.

Values: The (possibly rewritten) consequence of the rule.

See also: Rule equivalent of `query-head`, Page 122. `original-rule-consequence`, Page 121, `rule-antecedence`, Page 123

set-substrate-type *Function*

Description: Determines the type (class) of the substrates that nRQL creates internally on request

Syntax: `(set-substrate-type type-of-substrate)`

Arguments: `type-of-substrate` - a substrate type (class), one of: `data-substrate`, `mirror-data-substrate`, `rcc-substrate`, `rcc-mirror-substrate`.

Values: `:okay` or `:ignored`.

See also: `get-substrate-type`, Page 120, `describe-query-processing-mode`, Page 183

6.2 Query Management

delete-all-queries *Function*

Description: Aborts and deletes all queries

Syntax: `(delete-all-queries)`

Arguments:

Values: `:okay-all-queries-deleted`.

See also: `abort-query`, Page 136, `delete-query`, Page 125, `abort-all-queries`, Page 136

delete-query *Function*

Description: Deletes the query `id`, enabling the garbage collector to recycle some memory

Syntax: `(delete-query id)`

Arguments: `id` - the ID of the query to be deleted, or `:last` or `:last-query`.

Values: `:okay-query-deleted` or `:not-found`.

See also: `delete-all-queries`, Page 125

6.3 Rule Management

delete-all-rules *Function*

Description: Aborts and deletes all rules

Syntax: `(delete-all-rules)`

Arguments:

Values: `:okay-all-rules-deleted`.

See also: `abort-rule`, Page 136, `delete-rule`, Page 126, `abort-all-rules`, Page 136

delete-rule *Function*

Description: Deletes the query `id`, enabling the garbage collector to recycle some memory

Syntax: `(delete-rule id)`

Arguments: `id` - the ID of the rule to be deleted, or `:last` or `:last-query`.

Values: `:okay-rule-deleted` or `:not-found`.

See also: `delete-all-rules`, Page 126

6.4 Query Life Cycle

active-queries *Function*

Description: Returns a list of (the IDs of) all queries which satisfy `query-active-p`, Page 128

Syntax: `(active-queries)`

Arguments:

See also: `ready-queries`, Page 130, `processed-queries`, Page 127

prepared-queries *Function*

Description: Synonym for `ready-queries`, Page 130.

processed-queries *Function*

Description: Returns a list of (the IDs of) all queries which satisfy `query-processed-p`, Page 128

Syntax: `(processed-queries)`

Arguments:

See also: `prepared-queries`, Page 127, `active-queries`, Page 127

query-active-p *Function*

Description: Checks whether query `id` is active. A query is active iff a corresponding query answering thread exists

Syntax: `(query-active-p id)`

Arguments: `id` - the id of the query, or `:last` or `:last-query`.

Values: `t` or `nil`.

Remarks: An active query can either be waiting (if it has been started in lazy mode, `query-waiting-p`, Page 130) or running (`query-running-p`, Page 129), until its process terminates or is manually aborted (see `abort-query`, Page 136), `query-processed-p`, Page 128.

See also: `active-queries`, Page 127, `query-waiting-p`, Page 130, `query-running-p`, Page 129

query-prepared-p *Function*

Description: Synonym for `query-ready-p`, Page 129.

query-processed-p *Function*

Description: Checks whether the query `id` is processed and terminated, i.e., its query answering process (thread) has died

Syntax: `(query-processed-p id)`

Arguments: `id` - the id of the query, or `:last` or `:last-query`.

Values: `t` or `nil`.

Remarks: Use `reprepare-query`, Page 131 to reprepare the query, `reexecute-query`, Page 131 to reexecute it.

See also: `processed-queries`, Page 127, `query-active-p`, Page 128, `query-prepared-p`, Page 128

query-ready-p *Function*

Description: Checks whether query `id` is ready for execution

Syntax: `(query-ready-p id)`

Arguments: `id` - the id of the query, or `:last` or `:last-query`.

Values: `t` or `nil`.

Remarks: Use `execute-query`, Page 150 to start the query.

See also: `ready-queries`, Page 130

query-running-p *Function*

Description: Checks whether query `id` is active and running, i.e., its query answering process (thread) is currently consuming CPU cycles

Syntax: `(query-running-p id)`

Arguments: `id` - the id of the query, or `:last` or `:last-query`.

Values: `t` or `nil`.

Remarks: Use `abort-query`, Page 136 to abort the query.

See also: `running-queries`, Page 131, `query-active-p`, Page 128, `query-waiting-p`, Page 130

query-sleeping-p *Function*

Description: Synonym for `query-waiting-p`, Page 130.

query-terminated-p *Function*

Description: Synonym for `query-processed-p`, Page 128.

query-waiting-p *Function*

Description: Checks whether the query `id` is active and waiting (sleeping), i.e., its query answering process (thread) is currently not consuming CPU cycles

Syntax: `(query-waiting-p id)`

Arguments: `id` - the id of the query, or `:last` or `:last-query`.

Values: `t` or `nil`.

Remarks: Use `abort-query`, Page 136 to abort the query.

See also: `waiting-queries`, Page 132, `query-active-p`, Page 128, `query-running-p`, Page 129

ready-queries *Function*

Description: Returns a list of (the IDs of) all queries which satisfy `query-ready-p`, Page 129

Syntax: `(ready-queries)`

Arguments:

See also: `active-queries`, Page 127, `processed-queries`, Page 127

reexecute-query *Function*

Description: Reprepares and executes an already processed query (see `query-processed-p`, Page 128)

Syntax: `(reexecute-query id &rest args)`

Arguments: `id` - the ID of the query, or `:last` or `:last-query`.

`args` - see `execute-query`, Page 150.

Values: The result of `execute-query`, Page 150.

Remarks: Note that the query cannot be altered, but can be executed using different settings, since `args` are passed to `execute-query`, Page 150.

See also: `reprepare-query`, Page 131

reprepare-query *Function*

Description: Reprepares an already processed query (see `query-processed-p`, Page 128) and makes it ready for execution (via `execute-query`, Page 150) again (see `query-ready-p`, Page 129)

Syntax: `(reprepare-query id)`

Arguments: `id` - the ID of the query, or `:last` or `:last-query`.

Values: The result of `racer-prepare-query`, Page 155.

Remarks: Note that the query cannot be altered.

rule-running-p *Function*

Description: This is the rule equivalent of `query-running-p`, Page 129.

rule-waiting-p *Function*

Description: This is the rule equivalent of `query-waiting-p`, Page 130.

running-queries *Function*

Description: Returns a list of (the IDs of) all queries which satisfy `query-running-p`, Page 129

Syntax: `(running-queries)`

Arguments:

See also: `active-queries`, Page 127, `waiting-queries`, Page 132

sleeping-queries *Function*

Description: Synonym for `waiting-queries`, Page 132.

terminated-queries *Function*

Description: Synonym for `processed-queries`, Page 127.

waiting-queries *Function*

Description: Returns a list of (the IDs of) all queries which satisfy `query-waiting-p`, Page 130

Syntax: `(waiting-queries)`

Arguments:

See also: `active-queries`, Page 127, `running-queries`, Page 131

6.5 Rule Life Cycle

active-rules *Function*

Description: Returns a list of (the IDs of) all rules which satisfy `rule-active-p`, Page 134

Syntax: `(active-rules)`

Arguments:

See also: `ready-rules`, Page 133, `processed-rules`, Page 133

prepared-rules *Function*

Description: Synonym for `ready-rules`, Page 133.

processed-rules *Function*

Description: Returns a list of (the IDs of) all rules which satisfy `rule-processed-p`, Page 134

Syntax: `(processed-rules)`

Arguments:

See also: `prepared-rules`, Page 133, `active-rules`, Page 133

ready-rules *Function*

Description: Returns a list of (the IDs of) all rules which satisfy `rule-ready-p`, Page 134

Syntax: `(ready-rules)`

Arguments:

See also: `active-rules`, Page 133, `processed-rules`, Page 133

reexecute-rule *Function*

Description: This is the rule equivalent of `reexecute-query`, Page 131.

reprepare-rule *Function*

Description: This is the rule equivalent of `reprepare-query`, Page 131.

rule-active-p *Function*

Description: This is the rule equivalent of `query-active-p`, Page 128.

rule-prepared-p *Function*

Description: Synonym for `rule-ready-p`, Page 134.

rule-processed-p *Function*

Description: This is the rule equivalent of `query-processed-p`, Page 128.

rule-ready-p *Function*

Description: This is the rule equivalent of `query-ready-p`, Page 129.

rule-sleeping-p *Function*

Description: Synonym for `rule-waiting-p`, Page 131.

rule-terminated-p *Function*

Description: Synonym for `rule-processed-p`, Page 134.

running-rules *Function*

Description: Returns a list of (the IDs of) all rules which satisfy `rule-running-p`, Page 131 This is the rule equivalent of `running-queries`, Page 131.

Syntax: `(running-rules)`

Arguments:

See also: Rule equivalent of `running-queries`, Page 131. `active-rules`, Page 133, `waiting-rules`, Page 135

sleeping-rules *Function*

Description: Synonym for `waiting-rules`, Page 135.

terminated-rules *Function*

Description: Synonym for `processed-rules`, Page 133.

waiting-rules *Function*

Description: Returns a list of (the IDs of) all rules which satisfy `rule-waiting-p`, Page 131 This is the rule equivalent of `waiting-queries`, Page 132.

Syntax: `(waiting-rules)`

Arguments:

See also: Rule equivalent of `waiting-queries`, Page 132. `active-rules`, Page 133, `running-rules`, Page 135

6.6 Execution Control

abort-all-queries *Function*

Description: Applies `abort-query`, Page 136 to `active-queries`, Page 127

Syntax: `(abort-all-queries)`

Arguments: `id` - the ID of the query, or `:last` or `:last-query`.

Values: `:okay-all-queries-aborted`.

See also: `abort-query`, Page 136, `execute-query`, Page 150

abort-all-rules *Function*

Description: This is the rule equivalent of `abort-all-queries`, Page 136.

abort-query *Function*

Description: Aborts the active query (see `query-active-p`, Page 128) `id`. The query becomes processed (see `query-processed-p`, Page 128)

Syntax: `(abort-query id)`

Arguments: `id` - nil.

Values: Either `:okay-query-aborted` or `:not-found`.

See also: `abort-all-queries`, Page 136, `query-active-p`, Page 128

abort-rule *Function*

Description: This is the rule equivalent of `abort-query`, Page 136.

accurate-queries *Function*

Description: Returns a list of (the IDs of) the queries that satisfy `query-accurate-p`, Page 142. Note that this is a subset of the set of processed queries (see `processed-queries`, Page 127

Syntax: `(accurate-queries)`

Arguments:

accurate-rules *Function*

Description: This is the rule equivalent of `accurate-queries`, Page 137.

applicable-rules *Function*

Description: Returns (a list of) all rules which satisfy `rule-applicable-p`, Page 178.

cheap-queries *Function*

Description: Returns a list of (the IDs of) the queries that satisfy `cheap-query-p`, Page 138. Note that this is a subset of the set of active queries (see `active-queries`, Page 127

Syntax: `(cheap-queries)`

Arguments:

cheap-query-p *Function*

Description: Checks whether query `id` is still in emphphase one (see User Guide). Only active queries (see `query-active-p`, Page 128) can be recognized as cheap

Syntax: `(cheap-query-p id)`

Arguments: `id` - the id of the query, or `:last` or `:last-query`.

Values: `t` or `nil`.

Remarks: A query will only produce cheap tuples if it has been started in two-phase query processing mode, see `execute-query`, Page 150, `enable-two-phase-query-processing-mode`, Page 190. Note also that query must be executed in lazy tuple-at-a-time mode.

See also: `cheap-queries`, Page 137, `expensive-query-p`, Page 140, `enable-two-phase-query-processing-mode`, Page 190, `set-nrql-mode`, Page 196, `execute-query`, Page 150

cheap-rule-p *Function*

Description: This is the rule equivalent of `cheap-query-p`, Page 138.

cheap-rules *Function*

Description: This is the rule equivalent of `cheap-queries`, Page 137.

execute-all-queries *Function*

Description: Applies `execute-query`, Page 150 to `ready-queries`, Page 130

Syntax: `(execute-all-queries &rest args)`

Arguments: `args` - see `execute-query`, Page 150.

Values: A list containing the results of `execute-query`, Page 150 applied to `ready-queries`, Page 130.

See also: `execute-query`, Page 150, `abort-query`, Page 136

execute-all-rules *Function*

Description: This is the rule equivalent of `execute-all-queries`, Page 139.

execute-applicable-rules *Function*

Description: Calls `execute-rule`, Page 174 on all `applicable-rules`, Page 137

Syntax: `(execute-applicable-rules &rest args)`

Arguments: `args` - see `execute-rule`, Page 174.

Values: The list of results returned by `execute-rule`, Page 174 on the `applicable-rules`, Page 137.

See also: `execute-rule`, Page 174, `applicable-rules`, Page 137, `rule-applicable-p`, Page 178

execute-or-reexecute-all-queries *Function*

Description: Applies `execute-or-reexecute-query`, Page 140 to `processed-queries`, Page 127

Syntax: `(execute-or-reexecute-all-queries &rest arsg)`

Arguments: `args` - see `execute-query`, Page 150 `reexecute-query`, Page 131.

Values: A list containing the results of `execute-or-reexecute-query`, Page 140 applied to `ready-queries`, Page 130 and `processed-queries`, Page 127.

Remarks: First the ready queries are executed, and then the processed queries reexecuted (however, the ready queries are not executed twice).

See also: `reexecute-query`, Page 131, `execute-query`, Page 150, `execute-all-queries`, Page 139

execute-or-reexecute-all-rules *Function*

Description: This is the rule equivalent of `execute-or-reexecute-all-queries`, Page 139.

execute-or-reexecute-query *Function*

Description: Like `execute-query`, Page 150 in case the query is already prepared (ready, see `query-ready-p`, Page 129), and like `reexecute-query`, Page 131 in case the query is already processed (see `query-processed-p`, Page 128).

execute-or-reexecute-rule *Function*

Description: This is the rule equivalent of `execute-or-reexecute-query`, Page 140.

expensive-queries *Function*

Description: Returns a list of (the IDs of) the queries that satisfy `expensive-query-p`, Page 140. Note that this is a subset of the set of active queries (see `active-queries`, Page 127

Syntax: `(expensive-queries)`

Arguments:

expensive-query-p *Function*

Description: If an active query is not cheap `cheap-query-p`, Page 138, then it is expensive; thus, emphphase 2 has started.

expensive-rule-p *Function*

Description: This is the rule equivalent of `expensive-query-p`, Page 140.

expensive-rules *Function*

Description: This is the rule equivalent of `expensive-queries`, Page 140.

get-all-answers *Function*

Description: Applies `get-answer`, Page 163 to specified lists of queries and/or rules

Syntax: `(execute-or-reexecute-all-queries &rest args &key ready-p (active-p t) (processed-p t) (queries-p t) (rules-p t) execute-p)`

Arguments: `rules-p`, default `nil` - if `t`, applies `get-answer`, Page 163 to rules.

`rules-p`, default `nil` - if `t`, applies `get-answer`, Page 163 to queries.

`ready-p`, default `nil` - if `t`, applies `get-answer` to `ready-queries`, Page 130 and/or `ready-rules`, Page 133; note that `args` is passed to `get-answer`, Page 163 and the queries are executed automatically (see `get-answer`, Page 163, `execute-p` = `t`.

`active-p`, default `nil` - if `t`, applies `get-answer` to `active-queries`, Page 127 and/or `active-rules`, Page 133. Note that this requires computational ressources, i.e., the answers have to be computed.

`processed-p`, default `nil` - if `t`, applies `get-answer` to `processed-queries`, Page 127 and/or `processed-rules`, Page 133. Note that this requires no computational ressources, since the answers are already available and stored within the query objects.

Values: A list containing the results of `get-answer`, Page 163 applied to the specified lists of queries and/or rules.

Remarks: First the ready queries are executed, and then the processed queries reexecuted (however, the ready queries are not executed twice).

See also: `reexecute-query`, Page 131, `execute-query`, Page 150, `execute-all-queries`, Page 139

inaccurate-queries *Function*

Description: Returns a list of (the IDs of) the queries that do not satisfy `query-accurate-p`, Page 142. Note that this is a subset of the set of processed queries (see `processed-queries`, Page 127

Syntax: `(inaccurate-queries)`

Arguments:

inaccurate-rules *Function*

Description: This is the rule equivalent of `inaccurate-queries`, Page 141.

query-accurate-p *Function*

Description: Determines whether the computed and stored answer of a processed query `query-processed-p`, Page 128 is still accurate. The answer resp. processed query is called emphaccurate iff the queried KB (TBox, ABox) has not changed since the query was executed. Thus, the answers of an accuarte query must not be recomputed. Inaccurate query answers can be reecomputed see `reexecute-query`, Page 131

Syntax: `(query-accurate-p id)`

Arguments: `id` - the id of the query, or `:last` or `:last-query`.

Values: `t` or `nil`.

See also: `accurate-queries`, Page 137

reexecute-all-queries *Function*

Description: Applies `reexecute-query`, Page 131 to `processed-queries`, Page 127

Syntax: `(reexecute-all-queries &rest arsg)`

Arguments: `args` - see `reexecute-query`, Page 131.

Values: A list containing the results of `reexecute-query`, Page 131 applied to `ready-queries`, Page 130.

See also: `reexecute-query`, Page 131, `execute-query`, Page 150, `execute-all-queries`, Page 139

reexecute-all-rules *Function*

Description: This is the rule equivalent of `reexecute-all-queries`, Page 142.

rule-accurate-p *Function*

Description: This is the rule equivalent of `query-accurate-p`, Page 142.

rule-unapplicable-p *Function*

Description: Negation of `rule-applicable-p`, Page 178.

run-all-queries *Function*

Description: Synonym for `execute-all-queries`, Page 139.

run-all-rules *Function*

Description: Synonym for `execute-all-rules`, Page 139. This is the rule equivalent of `run-all-rules`, Page 143.

running-cheap-queries *Function*

Description: Returns a list of (the IDs of) the queries that satisfy `cheap-query-p`, Page 138 and `query-running-p`, Page 129

Syntax: `(running-cheap-queries)`

Arguments:

running-cheap-rules *Function*

Description: This is the rule equivalent of `running-cheap-rules`, Page 143.

running-expensive-queries *Function*

Description: Returns a list of (the IDs of) the queries that satisfy `expensive-query-p`, Page 140 and `query-running-p`, Page 129

Syntax: `(running-expensive-queries)`

Arguments:

running-expensive-rules *Function*

Description: This is the rule equivalent of `running-expensive-rules`, Page 144.

sleeping-cheap-queries *Function*

Description: Synonym for `waiting-cheap-queries`, Page 145.

sleeping-cheap-rules *Function*

Description: Synonym for `waiting-cheap-rules`, Page 145.

sleeping-expensive-queries *Function*

Description: Synonym for `waiting-expensive-queries`, Page 146.

sleeping-expensive-rules *Function*

Description: Synonym for `waiting-expensive-rules`, Page 146.

unapplicable-rules *Function*

Description: Returns (a list of) all rules which satisfy `rule-unapplicable-p`, Page 143.

wait-for-queries-to-terminate *Function*

Description: Waits (i.e., blocks the API) until all queries have terminated, i.e., `active-queries`, Page 127 returns `nil`

Syntax: `(wait-for-queries-to-terminate)`

Arguments:

Values: `:okay`.

Remarks: queries which are executed in lazy tuple-at-a-time mode do not terminate automatically. Thus, in order to prevent deadlocks, this function can only be called if no such queries are active.

See also: `active-queries`, Page 127, `abort-query`, Page 136

wait-for-rules-to-terminate *Function*

Description: This is the rule equivalent of `wait-for-queries-to-terminate`, Page 145.

waiting-cheap-queries *Function*

Description: Returns a list of (the IDs of) the queries that satisfy `cheap-query-p`, Page 138 and `query-waiting-p`, Page 130

Syntax: `(waiting-cheap-queries)`

Arguments:

waiting-cheap-rules *Function*

Description: Returns a list of (the IDs of) the rules that satisfy `cheap-rule-p`, Page 138 and `rule-waiting-p`, Page 131 This is the rule equivalent of `waiting-cheap-queries`, Page 145.

Syntax: `(waiting-cheap-rules)`

Arguments:

waiting-expensive-queries *Function*

Description: Returns a list of (the IDs of) the queries that satisfy `expensive-query-p`, Page 140 and `query-waiting-p`, Page 130

Syntax: `(waiting-expensive-queries)`

Arguments:

waiting-expensive-rules *Function*

Description: Returns a list of (the IDs of) the rules that satisfy `expensive-rule-p`, Page 140 and `rule-waiting-p`, Page 131 This is the rule equivalent of `waiting-expensive-queries`, Page 146.

Syntax: `(waiting-expensive-rules)`

Arguments:

6.7 ABox Queries

answer-query *Function*

Description: Synonym for `racer-answer-query`, Page 152.

answer-query-under-premise *Function*

Description: Synonym for `racer-answer-query-under-premise`, Page 152.

answer-query-under-premise1 *Function*

Description: Synonym for `racer-answer-query-under-premise1`, Page 152.

answer-query1 *Function*

Description: Synonym for `racer-answer-query1`, Page 153.

execute-query *Function*

Description: Sets up and starts a query answering process (thread) for the prepared (ready) query argumentid. The query answering process prepares the substrate for query answering if it has not been prepared yet (see `prepare-nrql-engine`, Page 121) before query answering starts on that substrate. In set-at-a-time mode, automatically calls `get-answer`, Page 163 and returns the answer

Syntax:
```
(execute-query id &rest args &key how-many
exclude-permutations-p use-individual-synonyms-p
tuple-at-a-time-p deliver-kb-has-changed-warning-tokens-p
proactive-tuple-computation-p told-information-reasoning-p
check-abox-consistency-p ensure-tbox-classification-p
initial-abox-mirroring-p initial-role-assertion-mirroring-p
two-phase-processing-p deliver-phase-two-warning-tokens-p)
```

Arguments:

`id` - the id of the query to be executed, or `:last` or `:last-query`.

`how-many`, default *by environment* - the number of tuples to be computed; `nil` means unbounded; see also `set-max-no-of-tuples-bound`, Page 195.

`exclude-permutations-p`, default *by environment* - if `t` is specified, then permutations will be exluded from the query answer; see also funrefexclude-permutations.

`tuple-at-a-time-p`, default *by environment* - if `t` is specified, then the tuple-at-a-time mode will be used, otherwise set-at-a-time mode; see also `process-tuple-at-a-time`, Page 193.

`deliver-kb-has-changed-warning-tokens-p`, default *by environment* - if `t` is specified, and the query is execute in tuple-at-a-time mode, then a warning token will be delivered if the KB has changed during query execution; see also `enable-kb-has-changed-warning-tokens`, Page 186.

`proactive-tuple-computation-p`, default *by environment* - if `t` is specified, then the eager mode will be used in tuple-at-a-time mode, otherwise the lazy mode; in set-at-a-time mode, tuple computation is always eager. See also `enable-eager-tuple-computation`, Page 185.

`deliver-phase-two-warning-tokens-p`, default *by environment* - if `t` is specified, then a warning token will be delivered if the query is executed in two-phase processing mode before phase two starts; see also `enable-phase-two-starts-warning-tokens`, Page 188.

execute-query *Function*

Description: continued

Arguments: `told-information-reasoning-p`, default *by environment* - if `t` is specified, then only the information in the nRQL caches (resp. the current substrate) is used for query answering. Calls to RacerPro basic ABox retrieval functions are avoided, but query answering is incomplete. Note that the amount of information in the caches depends on the nRQL mode which was active at the time the substrate was prepared (see `prepare-nrql-engine`, Page 121); see also `enable-told-information-querying`, Page 189.

`check-abox-consistency-p`, default *by environment* - if `t` is specified, the ABox to be queried is checked for consistency before querying starts; see also `check-abox-consistency-before-querying`, Page 182.

`ensure-tbox-classification-p`, default *by environment* - if the substrate for the ABox has not been prepared yet (see `prepare-nrql-engine`, Page 121), then the new query answering process will do that before query answering starts. If `t` is specified for this argument, then the substrate will be set-up in nRQL mode 1. See also `enable-smart-abox-mirroring`, Page 188, `set-nrql-mode`, Page 196.

execute-query *Function*

Description: continued

Arguments: `classify-concepts-in-instance-assertions`, default *by environment* - if nRQL shall use incomplete mode 2, then `t` must be specified. See also `enable-very-smart-abox-mirroring`, Page 191, `set-nrql-mode`, Page 196.

`two-phase-processing-p`, default *by environment* - if `t` is specified, then two-phase processing is enabled for this query; see also `enable-two-phase-query-processing-mode`, Page 190.

`use-individual-synonyms-p`, default *by environment* - if `t` is specified, then nRQL considers two individuals as being the same if they are synonyms. Note that this also changes the semantics of `same-as` from syntactic name identiy to semantic individual equivalence. See also `use-individual-synonym-equivalence-classes`, Page 196.

`args`, default - *in set-at-a-time mode*, `get-answer`, Page 163 is called automatically; thus, the keyword arguments of `get-answer`, Page 163 are accepted here as well and passed to `get-answer`, Page 163.

Values: If the query is executed in set-at-a-time mode, then the answer to that query since `get-answer`, Page 163 is called automatically, otherwise a list like `(:QUERY-466 :RUNNING)`, where `:QUERY-466` is the query ID and `:RUNNING` indicates the current status of the query.

Remarks: The query has to be prepared (ready) before it can be executed, see `query-ready-p`, Page 129, `prepared-queries`, Page 127.

If no values for the listed keyword arguments are specified, then either the lexical settings established by a surrounding `with-nrql-settings`, Page 199 or the currently active global settings (see `describe-query-processing-mode`, Page 183) will be used; this is documents as "default by environment" in the argument lists.

Note that also the keyword arguments accepted by `get-answer`, Page 163 are accepted and passed through to `get-answer`, Page 163 with argumentargs.

Examples:
```
(racer-prepare-query '(?x) '(?x woman))
(execute-query :last)
```

See also: `racer-prepare-query`, Page 155, `get-answer`, Page 163, `prepare-nrql-engine`, Page 121

prepare-query *Function*

Description: Synonym for `racer-prepare-query`, Page 155.

prepare-query1 *Function*

Description: Synonym for `racer-prepare-query1`, Page 155.

prepare-rule *Function*

Description: Synonym for `racer-prepare-rule`, Page 177.

prepare-rule1 *Function*

Description: Synonym for `racer-prepare-rule1`, Page 178.

racer-answer-query *Function*

Description: Prepares an ABox query using `racer-prepare-query`, Page 155 and then executes it with `execute-query`, Page 150 See also corresponding macro `retrieve`, Page 155.

Syntax: `(racer-answer-query head body &rest args)`

Arguments: `head body` - see `racer-prepare-query`, Page 155.

`args`, default `nil` - the union of the keyword arguments accepted by `racer-prepare-query`, Page 155 and `execute-query`, Page 150. If the query is executed in set-at-a-time mode, also `get-answer`, Page 163 is called automatically by `execute-query`, Page 150; thus, the keyword arguments of `get-answer`, Page 163 are accepted as well.

Values: Conceptually, `racer-answer-query` first calls `racer-prepare-query`, Page 155 and then `execute-query`, Page 150. If query is executed in set-at-a-time mode, then the result of `execute-query`, Page 150 is returned (the query answer). If the query is executed in tuple-at-a-time mode, then a query status description is returned.

Examples:
```
(racer-answer-query '(?x) '(and (?x woman) (?x ?y has-child)))
(racer-answer-query '(?x) '(and (?x woman) (?x ?y has-child))
:abox smith-family :id test :how-many 2)
```

See also: Corresponding macro: `retrieve`, Page 155. `racer-prepare-query`, Page 155, `execute-query`, Page 150, `get-answer`, Page 163

racer-answer-query-under-premise *Function*

Description: Like `racer-answer-query`, Page 152, but a query `premise` is added to the ABox before the query is answered (this can also be achieved with the `premise` keyword argument of `racer-answer-query`, Page 152). See also corresponding macro `retrieve-under-premise`, Page 155.

racer-answer-query-under-premise1 *Function*

Description: Like `racer-answer-query-under-premise`, Page 152, but with flipped argument positions for `head` and `body`. See also corresponding macro `retrieve-under-premise1`, Page 155.

racer-answer-query1 *Function*

Description: Like `racer-answer-query`, Page 152, but with flipped argument positions for `head` and `body`. See also corresponding macro `retrieve1`, Page 155.

racer-prepare-query *Function*

Description: Prepares (i.e., parses and compiles) a nRQL ABox query but does not execute (start) it yet See also corresponding macro `prepare-abox-query`, Page 155.

Syntax: `(racer-prepare-query head body &rest args &key id abox execute-p prepare-now-p premise type-of-substrate rewrite-defined-concepts-p)`

Arguments: `head` - the head of the query, see `<query-head>`, Section 6.1.8 in the User Guide.

`body` - the body of the query, see `<query-body>`, Section 6.1.8 in the User Guide.

`id`, default `query-xxx` - the id (name) of the query.

`abox`, default `(current-abox)` - the (name of the) ABox to be queried.

`execute-p`, default `nil` - if `t`, then the query is automatically executed; the `args` are passed to `execute-query`, Page 150.

`premise`, default `nil` - the query premise, a list of ABox assertions, see `<query-premise>`, Section 6.1.8 in the User Guide.

`type-of-substrate`, default *'racer-dummy-substrate* - a symbol naming a substrate type, one of: `racer-dummy-substrate, data-substrate, mirror-data-substrate, rcc-substrate, racer-tbox-mirror-substrate`.

`execute-p`, default `nil` - if `t` is specfied, the query is automatically started (executed); `execute-query`, Page 150 will be called. The `args` will be passed through to `execute-query`, Page 150.

`prepare-now-p`, default `nil` - if `t`, then the substrate will be prepared immediately (see `prepare-nrql-engine`, Page 121), otherwise later when the query is about to be executed.

`args` - see `execute-query`, Page 150.

Values: A list like `(:QUERY-466 :READY-TO-RUN)`, where `:QUERY-466` is the query ID and `:READY-TO-RUN` indicates the current status of the query.

Remarks: To start the query, use `execute-query`, Page 150 (or `execute-p` = `t`).

Examples: `(racer-prepare-query '(?x) '(and (?x woman) (?x ?y has-child)))`

See also: Corresponding macro: `prepare-abox-query`, Page 155. `execute-query`, Page 150, `get-answer`, Page 163

racer-prepare-query1 *Function*

Description: Like `racer-prepare-query`, Page 155, but with flipped argument positions for `head` and `body`. See also corresponding macro `prepare-abox-query1`, Page 155.

prepare-abox-query *Macro*

Description: See also corresponding function `racer-prepare-query`, Page 155.

prepare-abox-query1 *Macro*

Description: See also corresponding function `racer-prepare-query1`, Page 155.

retrieve *Macro*

Description: See also corresponding function `racer-answer-query`, Page 152.

retrieve-under-premise *Macro*

Description: See also corresponding function `racer-answer-query-under-premise`, Page 152.

retrieve-under-premise1 *Macro*

Description: See also corresponding function `racer-answer-query-under-premise1`, Page 152.

retrieve1 *Macro*

Description: See also corresponding function `racer-answer-query1`, Page 153.

with-bindings *Macro*

Description: If a nRQL query is executed in a `with-bindings`, Page 156 lexical environment, then the variables in the query is considered to be bound as established here

Syntax: `(with-bindings binding-list)`

Arguments: `binding-list` - nil.

Examples: `(with-bindings ((?x ind-123) (?z ind-456)) (retrieve (?y) (?x ?y r)))`
`(:see-also with-bindings-evaluated with-future-bindings)`

with-bindings-evaluated *Macro*

Description: Like `with-bindings`, Page 156, but now the individual forms (entries) in `binding-list` are evaluated to produce the individual (variable value) pairs, e.g., use `(with-bindings-evaluated ( (list '?x 'ind-123) ) ...)`.

with-future-bindings *Macro*

Description: Sometimes, a query must shall be prepared (i.e., parsed and compiled) with a promsie that at execution time, binding to certain variables in that query will be eastablished, this means, if `execute-query`, Page 150 is called in a lexical environment where certain variables are bound in advance with `with-bindings`, Page 156. During prepartion time, the query optimizer must thus be informed that these variables are in fact treated as individuals. This is what `with-future-bindings`, Page 157 does: It declares the `variables` to be individuals, and for query execution promisses that these variables will be bound with `with-bindings`, Page 156 priorily

Syntax: `(with-future-bindings variables)`

Arguments: `variables` - a list of variables.

Values: Like `progn` in Common LISP, so the value of the last embedded form is returned.

Examples: `(with-future-bindings (?x) (prepare-abox-query (?x ?y) (?x ?y r)))`

See also: `with-future-bindings-evaluated`, Page 197

6.8 TBox Queries

answer-tbox-query *Function*

Description: Synonym for `racer-answer-tbox-query`, Page 158.

answer-tbox-query1 *Function*

Description: Synonym for `racer-answer-tbox-query`, Page 158.

racer-answer-tbox-query *Function*

Description: TBox query equivalent of `racer-answer-query`, Page 152. See also corresponding macro `tbox-retrieve`, Page 160.

racer-answer-tbox-query1 *Function*

Description: Like `racer-answer-tbox-query`, Page 158, but with flipped argument positions for `head` and `body`. See also corresponding macro `tbox-retrieve1`, Page 160.

racer-prepare-tbox-query *Function*

Description: Prepares (i.e., parses and compiles) a nRQL TBox query but does not execute (start) it yet See also corresponding macro `prepare-tbox-query`, Page 160.

Syntax: `(racer-prepare-tbox-query head body &key tbox id prepare-now-p)`

Arguments: `head` - the head of the TBox query. Only variables, individuals and lambda operators are allowed.

`body` - the body of the TBox query. Only concept and role query atoms are allowed. Only concept names can be used in these concept query atoms, and only the role names `has-child, has-parent, has-descendant, has-ancestor` in the role query atoms.

`id`, default `query-xxx` - the id (name) of the query.

`tbox`, default `(current-tbox)` - the (name of the) TBox to be queried.

`execute-p`, default `nil` - if `t` is specified, then the query is automatically started with `execute-query`, Page 150; the `args` are passed to `execute-query`, Page 150.

`prepare-now-p`, default `nil` - if `t`, then the substrate will be prepared immediately, otherwise later when the query is about to be executed.

`args` - see `execute-query`, Page 150.

Values: A list like `(:QUERY-466 :READY-TO-RUN)`, where `:QUERY-466` is the query ID and `:READY-TO-RUN` indicates the current status of the query.

Remarks: To start the query, use `execute-query`, Page 150 (or `execute-p` = `t`).

Examples:
```
(racer-prepare-tbox-query '(?x) '(and (?x woman) (?x ?y
has-child)))
```

See also: Corresponding macro: `prepare-tbox-query`, Page 160. `execute-query`, Page 150, `get-answer`, Page 163

racer-prepare-tbox-query1 *Function*

Description: Like `racer-prepare-tbox-query`, Page 159, but with flipped argument positions for `head` and `body`. See also corresponding macro `prepare-tbox-query1`, Page 160.

prepare-tbox-query *Macro*

Description: See also corresponding function `racer-prepare-tbox-query`, Page 159.

prepare-tbox-query1 *Macro*

Description: See also corresponding function `racer-prepare-tbox-query1`, Page 159.

tbox-retrieve *Macro*

Description: See also corresponding function `racer-answer-tbox-query`, Page 158.

tbox-retrieve1 *Macro*

Description: See also corresponding function `racer-answer-tbox-query1`, Page 158.

6.9 Getting Answers

get-all-remaining-sets-of-rule-consequences *Function*

Description: Like `get-next-n-remaining-sets-of-rule-consequences`, Page 164 with `n` = `nil`.

get-all-remaining-tuples *Function*

Description: Like `get-next-n-remaining-tuples`, Page 164 with `n` = `nil`.

get-answer *Function*

Description: Gets (resp. forces the computation of) the complete answer set (result) of a query or the set of conclusions of a rule, independently if the query resp. rule had been in tuple- or set-at-a-time mode. The query or rule must be active (see `query-active-p`, Page 128, `rule-active-p`, Page 134, `active-queries`, Page 127, `active-rules`, Page 133) or already processed (see `query-processed-p`, Page 128, `rule-processed-p`, Page 134, `processed-queries`, Page 127, `processed-rules`, Page 133)

Syntax: `(get-answer id &rest args &key execute-p verbose-p dont-show-variables dont-show-head-projection-operators-p dont-show-lambdas-p)`

Arguments: `id` - nil.

`execute-p`, default `nil` - if `t` is specfied, the query is automatically started (executed); `execute-query`, Page 150 will be called. The `args` will be passed through to `execute-query`, Page 150.

`verbose-p`, default `t` - if `t` is specfied, also head projection operators are shown literally in the result tuples.

`dont-show-variables`, default `nil` - a list of variables. Usually, a variable binding is shown as a (variable value) entry in a result tuple. If the variable is a member in the list `dont-show-variables`, then only value will be included in the result tuples.

`dont-show-head-projection-operators-p`, default `nil` - the results of projection operators are usually included in the form (operator operator-result) in the result tuples. Specify `t` if only the operator result shall appear in the result tuples.

`dont-show-lambdas-p`, default `nil` - same as `dont-show-head-projection-operators-p`, but for solely for lambda operators.

`args` - see `execute-query`, Page 150.

Values: A list of tuples, or `t` or `nil`, or a list of lists of ABox assertions (the rule consequences), or `:NOT-FOUND`.

Remarks: Can be called an arbitrary number of times on a query or rule. The answer is stored in the query resp. rule object and is thus not recomputed if `get-answer` is called. You can check with `query-accurate-p`, Page 142 (resp. `rule-accurate-p`, Page 143) whether the stored answer is still valid. See also the value of `describe-query`, Page 119. In case of a rule, the rule consequences can be added with `add-chosen-sets-of-rule-consequences`, Page 172.

Note that the query or rule named `id` must be on the list of active or processed queries (see `active-queries`, Page 127, `processed-queries`, Page 127), otherwise `:NOT-FOUND` is returned.

The tuples are actually computed by repeated calls to `get-next-tuple`, Page 166. Thus, also special tokens (markers) returned by `get-next-tuple`, Page 166 might appear in the answer.

See also: `get-next-tuple`, Page 166, `get-next-n-remaining-tuples`, Page 164, `add-chosen-sets-of-rule-consequences`, Page 172, `active-queries`, Page 127, `processed-queries`, Page 127

get-answer-size *Function*

Description: Like `get-answer`, Page 163, but only returns the number of result tuples (resp. number of sets of rule conclusions).

get-current-set-of-rule-consequences *Function*

Description: Returns the result of the last call to `get-next-set-of-rule-consequences`, Page 165 on the rule `id`.

get-current-tuple *Function*

Description: Returns the result of the last call to `get-next-tuple`, Page 166 on the query `id`

Syntax: `(get-current-tuple id)`

Arguments: `id` - the ID of the query, or `:last`, or `:last-query`.

Values: See `get-next-tuple`, Page 166. Moreover, `nil` is returned if there was no previous call to `get-next-tuple`.

See also: `get-next-tuple`, Page 166

get-next-n-remaining-sets-of-rule-consequences *Function*

Description: Like `get-next-set-of-rule-consequences`, Page 165, but now the next `n` remaining sets are requested. Pass `nil` if you want all remaining sets; see also `get-all-remaining-sets-of-rule-consequences`, Page 161.

get-next-n-remaining-tuples *Function*

Description: Like `get-next-tuple`, Page 166, but now the next `n` remaining tuples are requested. Pass `nil` if you want all remaining tuples; see also `get-all-remaining-tuples`, Page 161.

get-next-set-of-rule-consequences *Function*

Description: Gets the next set of rule consequences of the rule `id`. The rule must be on the list of `active-rules`, Page 133 or `processed-rules`, Page 133. A rule can also be started (executed); use `execute-p` argument

Syntax: `(get-next-tuple id &rest args)`

Arguments: `id` - the ID of the rule, or `:last`, or `:last-rule`.

`execute-p`, default `nil` - specify `t` if the rule is ready to also fire it. The query will be started / executed then. The `args` are passed to `execute-query`, Page 150 (however, the query is always executed in tuple-at-a-time mode, this cannot be overridden).

`args` - see `execute-query`, Page 150.

Values: The tuple, or `:exhausted` in case there are no more tuples, or special tokens such as `:not-found`, `:inconsistent`, etc.

Remarks: If the query had been started in lazy tuple-at-a-time mode, then computation of the next tuple might eventually take some time and thus the function might not return immediately.

However, if the query had been started in eager mode, then there is a chance that the next tuple (and probably some more tuples not yet requested) have already been pre-computed, and are thus already available. The function `next-tuple-available-p`, Page 166 can be used to check for the availability of such (immediately available) tuples.

Note that a query might still have tuples available, even if the query process (thread) has already terminated and thus the query is no longer active (the query already appears on the list of `processed-queries`, Page 127). This happens in the eager tuple-at-a-time mode.

See also: `next-tuple-available-p`, Page 166, `get-current-tuple`, Page 163, `get-next-n-remaining-tuples`, Page 164

get-next-tuple *Function*

Description: Gets the next tuple from query `id`. The query must be on the list of `active-queries`, Page 127 or `processed-queries`, Page 127. A ready (prepared) query can also be started (executed), see `execute-p` argument. For rules, `get-next-set-of-rule-consequences`, Page 165 must be used

Syntax: `(get-next-tuple id &rest args)`

Arguments: `id` - the ID of the query, or `:last`, or `:last-query`.

`execute-p`, default `nil` - specify `t` to start (execute) a prepared (ready) query. The `args` are passed to `execute-query`, Page 150 (however, the query is always executed in tuple-at-a-time mode, this cannot be overridden).

`args` - see `execute-query`, Page 150.

Values: The tuple, or `:exhausted` in case there are no more tuples, or special tokens such as `:not-found`, `:inconsistent`, etc.

Remarks: If the query had been started in lazy tuple-at-a-time mode, then computation of the next tuple might eventually take some time and thus the function might not return immediately.

However, if the query had been started in eager mode, then there is a chance that the next tuple (and probably some more tuples not yet requested) have already been pre-computed, and are thus already available. The function `next-tuple-available-p`, Page 166 can be used to check for the availability of such (immediately available) tuples.

Note that a query might still have tuples available, even if the query process (thread) has already terminated and thus the query is no longer active (the query already appears on the list of `processed-queries`, Page 127). This happens in the eager tuple-at-a-time mode.

See also: `next-tuple-available-p`, Page 166, `get-current-tuple`, Page 163, `get-next-n-remaining-tuples`, Page 164

next-set-of-rule-consequences-available-p *Function*

Description: Checks for the availability of the next set of rule consequences This is the rule equivalent of `next-tuple-available-p`, Page 166.

Syntax: `(next-set-of-rule-consequences-available-p id)`

Arguments: `id` - the ID of the rule, or `:last`, or `:last-rule`.

Values: `t` or `nil` (or `:not-found`).

Remarks: If this function returns `t`, then `get-next-set-of-rule-consequences`, Page 165 returns that set immediately (without computation delay; the API does not block).

See also: Rule equivalent of `next-tuple-available-p`, Page 166. `get-next-set-of-rule-consequences`, Page 165

next-tuple-available-p *Function*

Description: Checks for the availability of the next answer tuple from a query

Syntax: `(next-tuple-available-p id)`

Arguments: `id` - the ID of the query, or `:last`, or `:last-query`.

Values: `t` or `nil` (or `:not-found`).

Remarks: If this function returns `t`, then `get-next-tuple`, Page 166 returns that tuple immediately (without computation delay; the API does not block).

See also: `get-next-tuple`, Page 166

6.10 Defined Queries

allow-overloaded-definitions *Function*

Description: Allow multiple defined queries with same name and arity. Inverse of `dont-allow-overloaded-definitions`, Page 169.

define-and-execute-query *Function*

Description: Like `define-query`, Page 168 with `keep-p` = `t` and `execute-p` = `t`. See also corresponding macro `def-and-exec-query`, Page 170.

define-and-prepare-query *Function*

Description: Like `define-query`, Page 168 with `keep-p` = `t` and `execute-p` = `nil`. See also corresponding macro `def-and-prep-query`, Page 171.

define-query *Function*

Description: Associates a query head and body with a name which is the name of the definition. This defined query can be reused by means of codesubstitute query atoms. The definitions are local to argumenttbox See also corresponding macro `defquery`, Page 171.

Syntax: `(defquery name head body &key keep-p tbox)`

Arguments: `name` - the name of the definition, see `<query-name>`, Section 6.1.8 in the User Guide.

`head` - the head of the query, see `<def-query-head>`, Section 6.1.8 in the User Guide. Projection operators are not allowed here.

`body` - the body of the query, see `<query-body>`, Section 6.1.8 in the User Guide.

`tbox`, default `(current-tbox)` - the TBox to which this definition is local.

`keep-p`, default `nil` - The query is prepared (and thus parsed) for syntax checking purposes, but the compiled query is discarded. The definition is registered and `name` returned. However, if `t` is specified for this argument, then the prepared query is not discarded and returned instead of `name`. Since `racer-prepare-query`, Page 155 is called and the arguments `args` are passed through, the query can automatically be executed (see `execute-p` in `racer-prepare-query`, Page 155).

Values: The `name` of the defined query if `keep-p` = `nil`, and the result of `racer-prepare-query`, Page 155 otherwise.

Remarks: The argumentbody can reference other defined queries as well, but cyclic definitions are not possible. Note that defined queries can also be used in a rule antecedence.

Examples:
```
(define-query 'is-a-mother '(?x) '(and (?x woman) (?x ?y
has-child)))
(define-query '(?a) '(substitute (is-a-mother ?a)))
(retrieve (?a) (?a is-a-mother))
```

See also: Corresponding macro: `defquery`, Page 171.

delete-all-definitions *Function*

Description: Deletes all defined queries local to `tbox`

Syntax: `(delete-all-definitions &key tbox)`

Arguments: `tbox`, default `(current-tbox)` - the TBox to which this definition is local.

Values: `:okay-all-definitions-deleted`.

See also: `undefine-query`, Page 170

describe-all-definitions *Function*

Description: Returns a list containing all definitions (see `describe-definition`, Page 169) local to `tbox`

Syntax: `(describe-all-definitions &tbox tbox)`

Arguments: `tbox`, default `(current-tbox)` - the TBox whose definitions are to be described.

Values: A list containing all definitions local to that TBox.

describe-definition *Function*

Description: Describes the definition `name` local to `tbox`

Syntax: `(describe-definition name &key tbox tbox)`

Arguments: `name` - the name of the definition.

`arity`, default `nil` - the arity of the requested definition.

`tbox`, default `(current-tbox)` - the TBox to which this definition is local.

Values: A defquery expression.

See also: `define-query`, Page 168

dont-allow-overloaded-definitions *Function*

Description: Dont allow multiple defined queries with same name and arity. Inverse of `allow-overloaded-definitions`, Page 167.

dont-prefer-defined-queries *Function*

Description: If a unary (binary) query is referenced in a body and there are corresponding concept (role) with the same name as the defined query, then the ambiguity is resolved in favor of the concept (role).. Inverse of `prefer-defined-queries`, Page 170.

prefer-defined-queries *Function*

Description: If a unary (binary) query is referenced in a body and there are corresponding concept (role) with the same name as the defined query, then the ambiguity is resolved in favor of the defined query.. Inverse of `dont-prefer-defined-queries`, Page 170.

undefine-query *Function*

Description: Deletes a defined query See also corresponding macro `undefquery`, Page 171.

Syntax: `(undefine-query name &key tbox)`

Arguments: `name` - the name of the definition, see `<query-name>`, Section 6.1.8 in the User Guide.

`tbox`, default `(current-tbox)` - the TBox to which this definition is local.

`arity`, default `nil` - the arity of the defined query to be delete; tt nil deletes all queries with that `name`.

Values: The names of the remaining definitions (local to argumenttbox).

Examples: `(undefine-query 'is-a-mother)`

See also: Corresponding macro: `undefquery`, Page 171. `define-query`, Page 168

def-and-exec-query *Macro*

Description: See also corresponding function `define-and-execute-query`, Page 167.

def-and-prep-query *Macro*

Description: See also corresponding function `define-and-prepare-query`, Page 167.

defquery *Macro*

Description: See also corresponding function `define-query`, Page 168.

undefquery *Macro*

Description: See also corresponding function `undefine-query`, Page 170.

6.11 Rules

add-chosen-sets-of-rule-consequences *Function*

Description: Adds the chosen sets of rule consequences (see `get-current-set-of-rule-consequences`, Page 163, `add-chosen-sets-of-rule-consequences`, Page 172) to the ABox (this is the ABox on which the rule has fired) produced by the rule `id`. See `get-chosen-sets-of-rule-consequences`, Page 175 to learn which assertions will be added

Syntax: `(add-chosen-sets-of-rule-consequences id)`

Arguments: `(first id the ID of the rule, or :last or :last-rule)` - nil.

`The added rule consequences resp. ABox assertions` - nil. This function can only be called if the rule has terminated (see `rule-processed-p`, Page 134). Note that an active rule (see `rule-active-p`, Page 134) can be aborted (see `abort-rule`, Page 136)

`get-chosen-sets-of-rule-consequences` - get-current-set-of-rule-consequences.

apply-rule *Function*

Description: Synonym for `racer-apply-rule`, Page 175.

apply-rule-under-premise *Function*

Description: Synonym for `racer-apply-rule-under-premise`, Page 176.

apply-rule-under-premise1 *Function*

Description: Synonym for `racer-apply-rule-under-premise1`, Page 176.

choose-current-set-of-rule-consequences *Function*

Description: Consequences of a rule are not added to the ABox as long as the rule is still active (see `rule-active-p`, Page 134). Some of the generated consequences are added when the rule has terminated.

If a rule is executed (see `execute-rule`, Page 174) in tuple-at-a time mode, then rule consequences are requested and computed lazily via `get-next-set-of-rule-consequences`, Page 165. The *current set of rule consequences,* see funrefget-current-set-of-rule-consequences, can thus be selected and memoized in the rule object for addition to the ABox (after termination of the rule) with this function. Rule consequences are added using the function `add-chosen-sets-of-rule-consequences`, Page 172. Note that in set-at-time-mode, all produced sets of consequences are chosen automatically for addition.

Using either `add-rule-consequences-automatically`, Page 182 (`dont-add-rule-consequences-automatically`, Page 184) resp. the `add-rule-consequences-p` argument of `execute-rule`, Page 174, you can determine whether (the selected) rule consequences will be added automatically with `add-chosen-sets-of-rule-consequences`, Page 172 when the rule terminates or not. In the latter case, `add-chosen-sets-of-rule-consequences`, Page 172 can be called manually later (see also `get-chosen-sets-of-rule-consequences`, Page 175)

Syntax: `(choose-current-set-of-rule-consequences id)`

Arguments: `id` - the ID of the rule, or `:last` or `:last-rule`.

Values: The current set of rule consequences (a list of ABox assertions).

See also: `add-chosen-sets-of-rule-consequences`, Page 172, `get-chosen-sets-of-rule-consequences`, Page 175

execute-rule *Function*

Description: See `execute-query`, Page 150 This is the rule equivalent of `execute-query`, Page 150.

Syntax: `(execute-rule id &rest args &key add-rule-consequences-p)`

Arguments: `id` - the id of the rule to be executed.

`add-rule-consequences-p`, default *by environment* - if `t` is specified and the rule is executed, the rule consequences are added to the ABox and returned, otherwise they are only returned but not added. Note that in set-at-a-time mode, this applies to all generated rule consequences, whereas in tuple-at-a-time mode this applies to the selected set of consequences (see `get-next-set-of-rule-consequences`, Page 165, `choose-current-set-of-rule-consequences`, Page 173).

`args` - see `execute-query`, Page 150 `answer-query`, Page 147.

Values: If the rule is executed in set-at-a-time mode, then the ABox assertions generated by the rule consequence since `get-answer`, Page 163 is called automatically, otherwise a list like `(:RULE-466 :RUNNING)`, where `:RULE-466` is the query ID and `:RUNNING` indicates the current status of the rule.

Remarks: The rule has to be prepared (ready) before it can be executed, see `rule-ready-p`, Page 134, `prepared-rules`, Page 133.

Note that rules cannot be execute in eager tuple-at-a-time mode.

See also: Rule equivalent of `execute-query`, Page 150. `racer-prepare-rule`, Page 177, `get-answer`, Page 163, `prepare-nrql-engine`, Page 121

get-chosen-sets-of-rule-consequences *Function*

Description: Returns the chosen (selected) sets of rule consequences of the rule `id`. These assertions will be added to the ABox if `add-chosen-sets-of-rule-consequences`, Page 172 is called on that rule

Syntax: `(get-chosen-sets-of-rule-consequences id)`

Arguments: `id` - the ID of the rule, or `:last` or `:last-rule`.

Values: the chosen sets of ABox assertions to be added.

See also: `choose-current-set-of-rule-consequences`, Page 173, `add-chosen-sets-of-rule-consequences`, Page 172

racer-apply-rule *Function*

Description: Prepares a rule using `racer-prepare-rule`, Page 177 and then executes it with `execute-rule`, Page 174 See also corresponding macro `apply-abox-rule`, Page 178. This is the rule equivalent of `racer-answer-query`, Page 152.

Syntax: `(racer-apply-rule antecedence consequence &rest args)`

Arguments: `head body` - see `racer-prepare-rule`, Page 177.

`args`, default `nil` - the union of the keyword arguments accepted by `racer-prepare-rule`, Page 177 and `execute-rule`, Page 174 (see there).

Values: Conceptually, `racer-apply-rule` first calls `racer-prepare-rule`, Page 177 and then `execute-rule`, Page 174. Thus, the result of `execute-rule`, Page 174 is returned. However, in case the rule is not executed (for example, if it has been recognized as inconsistent), then the result of `racer-prepare-rule`, Page 177 is returned.

See also: Corresponding macro: `apply-abox-rule`, Page 178. Rule equivalent of `racer-answer-query`, Page 152. `racer-prepare-rule`, Page 177, `execute-rule`, Page 174, `get-answer`, Page 163

racer-apply-rule-under-premise *Function*

Description: Like `racer-apply-rule`, Page 175, but with argument list `(premise antecedence consequence &rest args)`. See also corresponding macro `apply-abox-rule-under-premise`, Page 178. This is the rule equivalent of `racer-answer-query-under-premise`, Page 152.

racer-apply-rule-under-premise1 *Function*

Description: Like `racer-apply-rule-under-premise`, Page 176, but with argument list `(premise antecedence consequence &rest args)`. See also corresponding macro `apply-abox-rule-under-premise1`, Page 179. This is the rule equivalent of `racer-answer-query-under-premise1`, Page 152.

racer-apply-rule1 *Function*

Description: Like `racer-apply-rule`, Page 175, but with flipped argument positions for `antecedence` and `consequence`. See also corresponding macro `apply-abox-rule1`, Page 179. This is the rule equivalent of `racer-answer-query1`, Page 153.

racer-prepare-rule *Function*

Description: Prepares (i.e., parses and compiles) a nRQL ABox query but does not execute (start) it yet See also corresponding macro `prepare-abox-rule`, Page 180. This is the rule equivalent of `racer-prepare-query`, Page 155.

Syntax: `(racer-prepare-rule antecedence consequence &rest args &key id abox execute-p premise type-of-substrate prepare-now-p)`

Arguments: `antecedence` - the antecedence of the rule, see `<rule-antecedence>`, Section 6.1.8 in the User Guide.

`consequence` - the consequence of the rule, see `<rule-consequence>`, Section 6.1.8 in the User Guide.

`id`, default `rule-xxx` - the id (name) of the query.

`abox`, default `(current-abox)` - the (name of the) ABox to which the rule is applied.

`execute-p`, default `nil` - if `t`, then the rule is automatically executed; the `args` are passed to `execute-rule`, Page 174.

`premise`, default `nil` - the premise of the rule, a list of ABox assertions, see `<query-premise>`, Section 6.1.8 in the User Guide.

`type-of-substrate`, default *'racer-dummy-substrate* - a symbol naming a substrate type, one of: `racer-dummy-substrate`, `data-substrate`, `mirror-data-substrate`, `rcc-substrate`, `racer-tbox-mirror-substrate`.

`prepare-now-p`, default `nil` - if `t`, then the substrate will be prepared immediately, otherwise later when the rule is about to be applied.

`args` - see `execute-rule`, Page 174.

Values: A list like `(:RULE-XXX :READY-TO-RUN)`, where `:RULE-XXX` is the rule ID and `:READY-TO-RUN` indicates the current status of the rule.

Remarks: To fire (start, apply) the rule, use `execute-rule`, Page 174.

See also: Corresponding macro: `prepare-abox-rule`, Page 180. Rule equivalent of `racer-prepare-query`, Page 155. `prepare-abox-query`, Page 155, `execute-query`, Page 150

racer-prepare-rule1 *Function*

Description: Like `racer-prepare-rule`, Page 177, but with flipped argument positions for `antecedence` and `consequence`. See also corresponding macro `prepare-abox-rule1`, Page 180. This is the rule equivalent of `racer-prepare-query1`, Page 155.

rule-applicable-p *Function*

Description: Checks whether rule `id` is applicable, i.e. its antecedence is true. Thus, its consequence might produce new ABox assertions (or delete existing ABox assertions)

Syntax: `(rule-applicable-p id)`

Arguments: `id` - the ID of the rule, or `:last` or `:last-rule`.

Values: `t` or `nil` (or `:not-found`).

Remarks: A rule can only be applicable if it is either ready (see `rule-ready-p`, Page 134) or processed (see `rule-processed-p`, Page 134). Rules which are already active (see `rule-active-p`, Page 134) are not applicable.

If an already processed rule is found to be applicable, then it is also automatically reprepared, see funrefreprepare-rule so it can immediately be fired again (see `execute-rule`, Page 174).

See also: `rule-unapplicable-p`, Page 143, `applicable-rules`, Page 137, `unapplicable-rules`, Page 144, `execute-rule`, Page 174

apply-abox-rule *Macro*

Description: See also corresponding function `racer-apply-rule`, Page 175. This is the rule equivalent of `retrieve`, Page 155.

apply-abox-rule-under-premise *Macro*

Description: See also corresponding function `racer-apply-rule-under-premise`, Page 176. This is the rule equivalent of `retrieve-under-premise`, Page 155.

apply-abox-rule-under-premise1 *Macro*

Description: See also corresponding function `racer-apply-rule-under-premise1`, Page 176. This is the rule equivalent of `retrieve-under-premise1`, Page 155.

apply-abox-rule1 *Macro*

Description: See also corresponding function `racer-apply-rule1`, Page 176. This is the rule equivalent of `retrieve1`, Page 155.

firerule *Macro*

Description: See also corresponding function `racer-apply-rule`, Page 175. Synonym for `apply-abox-rule`, Page 178.

firerule-under-premise *Macro*

Description: See also corresponding function `racer-apply-rule-under-premise`, Page 176. Synonym for `apply-abox-rule-under-premise`, Page 178.

firerule-under-premise1 *Macro*

Description: See also corresponding function `racer-apply-rule-under-premise1`, Page 176. Synonym for `apply-abox-rule-under-premise1`, Page 179.

firerule1 *Macro*

Description: See also corresponding function `racer-apply-rule1`, Page 176. Synonym for `apply-abox-rule1`, Page 179.

prepare-abox-rule *Macro*

Description: See also corresponding function `racer-prepare-rule`, Page 177. This is the rule equivalent of `prepare-abox-query`, Page 155.

prepare-abox-rule1 *Macro*

Description: See also corresponding function `racer-prepare-rule1`, Page 178.

preprule *Macro*

Description: See also corresponding function `racer-prepare-rule`, Page 177. Synonym for `prepare-abox-rule`, Page 180.

preprule1 *Macro*

Description: See also corresponding function `racer-prepare-rule1`, Page 178. Synonym for `prepare-abox-rule1`, Page 180.

6.12 Querying Modes

add-role-assertions-for-datatype-properties *Function*

Description: Constraint query atoms referring OWL datatype properties only work on OWL KBs if some additional auxiliary ABox assertions are added to the ABox created from the OWL file. Use this function to ensure that nRQL adds these additional assertions. Note that this function must be called before the first nRQL query to that OWL KB is posed Inverse of `dont-add-role-assertions-for-datatype-properties`, Page 184.

Syntax: `(add-role-assertions-for-datatype-properties)`

Arguments:

Values: `:okay-adding-role-assertions-for-datatype-properties`.

See also: Inverse of `dont-add-role-assertions-for-datatype-properties`, Page 184.`describe-query-processing-mode`, Page 183, `set-nrql-mode`, Page 196, `with-nrql-settings`, Page 199

add-rule-consequences-automatically *Function*

Description: Rule consequences / ABox assertions generated by a rule application can be added automatically after the rule has terminated. Use this function to put nRQL (globally) into that mode. Note that in tuple-at-a-time mode this applies to the chosen sets of rule consequences (see `choose-current-set-of-rule-consequences`, Page 173), whereas in set-at-a-time mode this applies to all sets of rule consequences. In case rule consequences are not automatically, the functino `add-chosen-sets-of-rule-consequences`, Page 172 can be called manually, but only once. See also argument `add-rule-consequences-p` of function `execute-rule`, Page 174 Inverse of `dont-add-rule-consequences-automatically`, Page 184.

Syntax: `(add-rule-consequences-automatically)`

Arguments:

Values: `:okay-adding-rule-consequences-automatically`.

See also: Inverse of `dont-add-rule-consequences-automatically`, Page 184.`describe-query-processing-mode`, Page 183, `with-nrql-settings`, Page 199, `process-tuple-at-a-time`, Page 193, `process-set-at-a-time`, Page 193, `execute-rule`, Page 174, `get-current-set-of-rule-consequences`, Page 163, `choose-current-set-of-rule-consequences`, Page 173, `add-chosen-sets-of-rule-consequences`, Page 172

check-abox-consistency-before-querying *Function*

Description: Advises nRQL to check the ABox consistency before a query is executed; see also argument `check-abox-consistency-p` of function `execute-query`, Page 150. Queries on inconsistent ABoxes are not meaningful

Syntax: `(check-abox-consistency-before-querying)`

Arguments:

Values: `:okay-checking-abox-consistency-before-querying`.

See also: `describe-query-processing-mode`, Page 183, `with-nrql-settings`, Page 199

describe-query-processing-mode *Function*

Description: Returns a description of the current (global) nRQL settings resp. query processing mode

Syntax: `(describe-query-processing-modes)`

Arguments:

Values: The description, a structured list.

See also: `describe-query`, Page 119, `describe-current-substrate`, Page 210, `set-nrql-mode`, Page 196, `with-nrql-settings`, Page 199, `execute-query`, Page 150

disable-abox-mirroring *Function*

Description: Inverse of `enable-abox-mirroring`, Page 185.

disable-kb-has-changed-warning-tokens *Function*

Description: Inverse of `enable-kb-has-changed-warning-tokens`, Page 186.

disable-nrql-warnings *Function*

Description: Inverse of `enable-nrql-warnings`, Page 187.

disable-phase-two-starts-warning-tokens *Function*

Description: Inverse of `enable-phase-two-starts-warning-tokens`, Page 188.

disable-query-optimization *Function*

Description: Inverse of `enable-query-optimization`, Page 188.

disable-told-information-querying *Function*

Description: Inverse of `enable-told-information-querying`, Page 189.

disable-two-phase-query-processing-mode *Function*

Description: Inverse of `disable-two-phase-query-processing-mode`, Page 184.

dont-add-role-assertions-for-datatype-properties *Function*

Description: Inverse of `add-role-assertions-for-datatype-properties`, Page 181.

dont-add-rule-consequences-automatically *Function*

Description: Inverse of `add-rule-consequences-automatically`, Page 182.

dont-check-abox-consistency-before-querying *Function*

Description: Inverse of `check-abox-consistency-before-querying`, Page 182.

dont-use-individual-synonym-equivalence-classes *Function*

Description: Inverse of `dont-use-individual-synonym-equivalence-classes`, Page 184.

dont-use-injective-variables-by-default *Function*

Description: Inverse of `use-injective-variables-by-default`, Page 197.

enable-abox-mirroring *Function*

Description: Instructs nRQL (globally) to mirror the asserted content of an ABox (the ABox assertions) into its internal data caches before querying starts. Note that the amount of information in the substrate resp. caches determines the degree of query answering completness in the incomplete modes (see `enable-told-information-querying`, Page 189). See also argument `told-information-reasoning-p` of function `execute-query`, Page 150. Inverse of `disable-abox-mirroring`, Page 183.

enable-eager-tuple-computation *Function*

Description: Advises nRQL to precompute answer tuples in tuple-at-a-time mode, even if these tuples have not yet been requested (see `get-next-tuple`, Page 166); see also argument `proactive-tuple-computation-p` of function `execute-query`, Page 150. A query started in eager mode will never appear on `waiting-queries`, Page 132. The inverse tuple-at-a-time mode is called lazy tuple-at-a-time mode. In this mode, the next answer tuple will not be computed by the query answering process (thread) until it is really requested; in the meantime, such a query appears on the list of `waiting-queries`, Page 132 Inverse of `enable-lazy-tuple-computation`, Page 186.

Syntax: `(enable-eager-tuple-computation)`

Arguments:

Values: `:okay-eager-mode-enabled` or `:ignored-not-in-tuple-at-a-time-mode`.

Remarks: Is only effective if nRQL is in tuple-at-a-time-mode.

See also: Inverse of `enable-lazy-tuple-computation`, Page 186.`describe-query-processing-mode`, Page 183, `set-nrql-mode`, Page 196, `with-nrql-settings`, Page 199

enable-kb-has-changed-warning-tokens *Function*

Description: Enables (global) delivery of `:warning-kb-has-changed` tokens in tuple-at-a-time query processing mode. Such a token is delivered iff the query ABox / TBox changes during query answering (in the time the query is still active). See also argument `deliver-kb-has-changed-warning-tokens-p` of function `execute-query`, Page 150 Inverse of `disable-kb-has-changed-warning-tokens`, Page 183.

Syntax: `(enable-kb-has-changed-warning-tokens)`

Arguments:

Values: `:okay-kb-has-changed-warning-tokens-enabled` or `:ignored-not-in-tuple-at-a-time-mode`.

Remarks: Can only be called if nRQL is in tuple-at-a-time mode.

See also: Inverse of `disable-kb-has-changed-warning-tokens`, Page 183.`set-nrql-mode`, Page 196, `with-nrql-settings`, Page 199, `describe-query-processing-mode`, Page 183

enable-lazy-tuple-computation *Function*

Description: Inverse of `enable-eager-tuple-computation`, Page 185.

enable-nrql-warnings *Function*

Description: Advises nRQL to print out warnings on STDOUT in certain circumstances and enables delivery of warning tokens (see `enable-kb-has-changed-warning-tokens`, Page 186, `enable-phase-two-starts-warning-tokens`, Page 188) Inverse of `disable-nrql-warnings`, Page 183.

Syntax: `(enable-nrql-warnings)`

Arguments:

Values: `:okay-warnings-enabled`.

See also: Inverse of `disable-nrql-warnings`, Page 183.`describe-query-processing-mode`, Page 183, `enable-kb-has-changed-warning-tokens`, Page 186, `with-nrql-settings`, Page 199, `disable-kb-has-changed-warning-tokens`, Page 183

enable-phase-two-starts-warning-tokens *Function*

Description: Enables (global) delivery of `:warning-expensive-phase-two-starts` tokens in two-phase query processing modes, denoting the transition between cheap (see `cheap-query-p`, Page 138) and expensive answer tuples (see `expensive-query-p`, Page 140). See also argument `deliver-phase-two-warning-tokens-p` of function `execute-query`, Page 150 Inverse of `disable-phase-two-starts-warning-tokens`, Page 183.

Syntax: `(enable-phase-two-starts-warning-tokens)`

Arguments:

Values: `:okay-phase-two-warning-tokens-enabled` or `:ignored-not-in-two-phase-processing-mode`.

Remarks: Can only be called if nRQL is in two phase processing mode (see `enable-two-phase-query-processing-mode`, Page 190).

See also: Inverse of `disable-phase-two-starts-warning-tokens`, Page 183.`enable-two-phase-query-processing-mode`, Page 190, `disable-two-phase-query-processing-mode`, Page 184, `set-nrql-mode`, Page 196, `with-nrql-settings`, Page 199, `describe-query-processing-mode`, Page 183

enable-query-optimization *Function*

Description: Enables the cost-based query optimizer Inverse of `disable-query-optimization`, Page 183.

Syntax: `(enable-query-optimization)`

Arguments:

Values: `:okay-query-optimization-enabled`.

Remarks: Note that queries must be brought into DNF (Disjunctive Normal Form). Thus, query optimization might be expensive.

See also: Inverse of `disable-query-optimization`, Page 183.`optimizer-use-cardinality-heuristics`, Page 193, `describe-query-processing-mode`, Page 183

enable-smart-abox-mirroring *Function*

Description: Enables ABox mirroring, see funrefenable-abox-mirroring, but in a smarter way. Not only are the ABox assertion mirrored and put into the substrate caches and index structures, but also the TBox information is exploited. In case of a concept assertion such as `(instance i C)` with atomic concept `C`, not only `C` is added as told information for `i` to the ABox mirror resp. substrate caches and index structures, but also the set of concept synonyms and concept ancestors from the TBox is computed and added as well. The same applies for `related` role membership assertions in the presence of role hierarchies, etc. Please consult the User Guide for more details. See also argument `told-information-reasoning-p` of function `execute-query`, Page 150

Syntax: `(enable-smart-abox-mirroring)`

Arguments:

Values: `:okay-smart-abox-mirroring-enabled`.

See also: `enable-very-smart-abox-mirroring`, Page 191, `disable-abox-mirroring`, Page 183, `enable-abox-mirroring`, Page 185, `set-nrql-mode`, Page 196, `with-nrql-settings`, Page 199, `describe-query-processing-mode`, Page 183

enable-told-information-querying *Function*

Description: Puts nRQL (globally) into told information querying mode; see also `told-information-reasoning-p` of `execute-query`, Page 150. Told information querying means means that calls to RacerPro's ABox retrieval functions are avoided and only the information in the substrate caches is used for query answering. It is recommended to use `set-nrql-mode`, Page 196 instead of this function. See also argument `told-information-reasoning-p` of function `execute-query`, Page 150 Inverse of `disable-told-information-querying`, Page 184.

Syntax: `(enable-told-information-querying)`

Arguments:

Values: `:okay-told-information-querying-enabled`.

See also: Inverse of `disable-told-information-querying`, Page 184.`set-nrql-mode`, Page 196, `describe-query-processing-mode`, Page 183, `with-nrql-settings`, Page 199

enable-two-phase-query-processing-mode *Function*

Description: Enables (global) two phase query processing. In this mode, nRQL can distinguish between cheap (see `cheap-query-p`, Page 138) and expensive answer tuples (see `expensive-query-p`, Page 140) of a query. Please consult the User Guide for more information. See also argument `two-phase-processing-p` of function `execute-query`, Page 150 Inverse of `disable-two-phase-query-processing-mode`, Page 184.

Syntax: `(enable-two-phase-query-processing-mode)`

Arguments:

Values: `:okay-two-phase-query-processing-mode-enabled`.

Remarks: Before the first expensive tuple is computed, nRQL can be advised to deliver a so-called warning token, see `enable-phase-two-starts-warning-tokens`, Page 188.

See also: Inverse of `disable-two-phase-query-processing-mode`, Page 184.`disable-two-phase-query-processing-mode`, Page 184, `enable-phase-two-starts-warning-tokens`, Page 188, `set-nrql-mode`, Page 196, `with-nrql-settings`, Page 199, `describe-query-processing-mode`, Page 183

enable-very-smart-abox-mirroring *Function*

Description: Like `enable-smart-abox-mirroring`, Page 188, but now not only atomic concept assertions will be used for augmenting the information in the mirror resp. substrate caches and index structures, but also non-atomic concepts in ABox assertions. Thus, if `(instance i C)` is present for a non-atomic concepts `C`, then also the set of concept synonyms and concept ancestors is computed and added to the mirror. See also argument `told-information-reasoning-p` of function `execute-query`, Page 150

Syntax: `(enable-very-smart-abox-mirroring)`

Arguments:

Values: `:okay-very-smart-abox-mirroring-enabled`.

Remarks: Might be expensive, since concepts in ABox concept assertion must be classified in order to compute the synonyms and ancestors.

See also: `enable-smart-abox-mirroring`, Page 188, `disable-abox-mirroring`, Page 183, `enable-abox-mirroring`, Page 185, `set-nrql-mode`, Page 196, `with-nrql-settings`, Page 199, `describe-query-processing-mode`, Page 183

exclude-permutations *Function*

Description: Advises nRQL to (globally) exclude permutations of anwers tuples. See also argument `exclude-permutations-p` of function `execute-query`, Page 150 Inverse of `include-permutations`, Page 192.

Syntax: `(exclude-permutations)`

Arguments:

Values: `:okay-exluding-permuatation`.

See also: Inverse of `include-permutations`, Page 192.`describe-query-processing-mode`, Page 183, `execute-query`, Page 150, `with-nrql-settings`, Page 199

get-initial-size-of-process-pool *Function*

Description: Inverse of `set-initial-size-of-process-pool`, Page 194.

get-max-no-of-tuples-bound *Function*

Description: Inverse of `set-max-no-of-tuples-bound`, Page 195.

get-maximum-size-of-process-pool *Function*

Description: Inverse of `set-maximum-size-of-process-pool`, Page 195.

get-process-pool-size *Function*

Description: Returns the current number of processes in the process pool (the process pool size)

Syntax: `(set-process-pool-size)`

Arguments:

Values: the pool size, a natural number.

See also: `set-initial-size-of-process-pool`, Page 194,
`get-initial-size-of-process-pool`, Page 191,
`set-maximum-size-of-process-pool`, Page 195,
`get-maximum-size-of-process-pool`, Page 192,
`set-initial-size-of-process-pool`, Page 194,
`get-initial-size-of-process-pool`, Page 191,
`describe-query-processing-mode`, Page 183

include-permutations *Function*

Description: Inverse of `exclude-permutations`, Page 191.

optimizer-dont-use-cardinality-heuristics *Function*

Description: Inverse of `optimizer-use-cardinality-heuristics`, Page 193.

optimizer-use-cardinality-heuristics *Function*

Description: Advises the cost-based optimizer to use ABox statistics for enhanced query optimization Inverse of `optimizer-dont-use-cardinality-heuristics`, Page 192.

Syntax: `(optimizer-use-cardinality-heuristics)`

Arguments:

Values: Also turns on the optimizer (calls `enable-query-optimization`, Page 188).

See also: Inverse of `optimizer-dont-use-cardinality-heuristics`, Page 192.`enable-query-optimization`, Page 188, `describe-query-processing-mode`, Page 183

process-set-at-a-time *Function*

Description: Inverse of `process-tuple-at-a-time`, Page 193.

process-tuple-at-a-time *Function*

Description: Puts nRQL (globally) into tuple-at-time-mode. See also argument `tuple-at-a-time-p` of function `execute-query`, Page 150 Inverse of `process-set-at-a-time`, Page 193.

Syntax: `(process-tuple-at-a-time)`

Arguments:

Values: `:okay-processing-tuple-at-a-time`.

See also: Inverse of `process-set-at-a-time`, Page 193.`describe-query-processing-mode`, Page 183

restore-standard-settings *Function*

Description: Restores the standard nRQL settings

Syntax: `(restore-standard-settings)`

Arguments:

See also: `set-nrql-mode`, Page 196, `with-nrql-settings`, Page 199, `full-reset`, Page 120, `reset-nrql-engine`, Page 123, `describe-query-processing-mode`, Page 183

set-initial-size-of-process-pool *Function*

Description: nRQL uses a process (thread) pool for the query answering processes. The initial (minimal) size of the pooled processes can be specified with that function. This specifies the lower bound of concurrent queries Inverse of `get-initial-size-of-process-pool`, Page 191.

Syntax: `(set-initial-size-of-process-pool n)`

Arguments: `n` - A natural number, the size of the pool (the number of processes in the pool).

Values: `n`.

Remarks: note that setting the initial process pool causes the pool to reinitialize; all active queries (and rules) are aborted.

See also: Inverse of `get-initial-size-of-process-pool`, Page 191.`get-initial-size-of-process-pool`, Page 191, `get-process-pool-size`, Page 192, `set-maximum-size-of-process-pool`, Page 195, `get-maximum-size-of-process-pool`, Page 192, `describe-query-processing-mode`, Page 183

set-max-no-of-tuples-bound *Function*

Description: Sets a (global) bound on the number of answer tuples that are computed. Use `nil` to set to unbounded (infinite). See also argument `how-many` of function `execute-query`, Page 150 Inverse of `get-max-no-of-tuples-bound`, Page 192.

Syntax: `(set-max-no-of-tuples-bound n)`

Arguments: `n` - A natural number (the bound) or `nil` (means unbounded).

Values: `n`.

See also: Inverse of `get-max-no-of-tuples-bound`, Page 192.`describe-query-processing-mode`, Page 183, `get-max-no-of-tuples-bound`, Page 192, `with-nrql-settings`, Page 199

set-maximum-size-of-process-pool *Function*

Description: Like `set-initial-size-of-process-pool`, Page 194, but now the maximum number of processes in the process pool is specified. In case a new query answering process is needed from a pool and the pool is currently empty (all processes are accquired by different queries and/or rules), nRQL will create an additional process. This new process is added to the pool, thus, the process pool can grow up to an upper bound which is specified here Inverse of `get-maximum-size-of-process-pool`, Page 192.

Syntax: `(set-maximum-size-of-process-pool n)`

Arguments: `n` - A natural number (the upper bound) or `nil` (which means unbounded).

Values: `n`.

See also: Inverse of `get-maximum-size-of-process-pool`, Page 192.`set-initial-size-of-process-pool`, Page 194, `get-initial-size-of-process-pool`, Page 191, `get-process-pool-size`, Page 192, `describe-query-processing-mode`, Page 183

set-nrql-mode *Function*

Description: Puts nRQL globally into mode `n`. See User Guide for a description of the different querying modes

Syntax: `(set-nrql-mode n)`

Arguments: `n` - 3.

Values: the mode.

See also: `describe-query-processing-mode`, Page 183, `with-nrql-settings`, Page 199

use-individual-synonym-equivalence-classes *Function*

Description: Usually, nRQL's query variables are bound to ABox individuals. However, sometimes RacerPro can infer that a set of differently named individuals must represent the same (identical) domain object. In this case, the different individuals are called *individual synonyms*. Sometimes it is meaningful to bind query variables not to single individuals, but to *synonym equivalence classes*. This can be achieved by enabling this mode. If this mode is enabled, then variables will not be bound to single ABox individuals, but to representative individuals from the synonym equivalence classes Inverse of `dont-use-individual-synonym-equivalence-classes`, Page 184.

Syntax: `(use-individual-synonym-equivalence-classes)`

Arguments:

Values: `:okay-using-individual-equivalence-classes` .

Remarks: see also `use-individual-synonyms-p` in `execute-query`, Page 150.

See also: Inverse of `dont-use-individual-synonym-equivalence-classes`, Page 184.`dont-use-individual-synonym-equivalence-classes`, Page 184, `set-nrql-mode`, Page 196, `with-nrql-settings`, Page 199, `describe-query-processing-mode`, Page 183

use-injective-variables-by-default *Function*

Description: nRQL offers injective variables. Usually, all variables are non-injective. That means two different variables can be bound to the same ABox individual – the mapping from variable to ABox individuals must not be injective, unlike for injective variables, for which the mapping must be injective. By default, all variables with `?`-prefix are non-injective, and injective variables get a `$?`-prefix. In the older nRQL, all variables were injective by default; thus, `?`-prefix denoted an injective, and `$?`-prefix a non-injective variable. This function allows you to switch to the old nRQL mode. Note that you can also use negated `same-as` query atoms to enforce injective bindings Inverse of `use-injective-variables-by-default`, Page 197.

Syntax: `(use-injective-variables-by-default)`

Arguments:

Values: `:okay-using-?-prefix-for-injective-variables`.

See also: Inverse of `use-injective-variables-by-default`, Page 197.`dont-use-injective-variables-by-default`, Page 184, `set-nrql-mode`, Page 196, `with-nrql-settings`, Page 199, `describe-query-processing-mode`, Page 183

with-future-bindings-evaluated *Macro*

Description: Like `with-future-bindings`, Page 157, but now `list-of-variables` is evaluated to produce the list of variables.

with-nrql-settings *Macro*

Description: Establishes a lexical local environment shaddowing the global environment in which the query answering switches resp. the corresponding variables are rebound to specified values. The argument forms are not evaluated (see also `with-nrql-settings-evaluated`, Page 199

Syntax:
```
(with-nrql-settings (&key mode dont-show-variables
dont-show-lambdas dont-show-head-projection-operators
abox-mirroring query-optimization optimizer-use-cardinality-heuristics
how-many-tuples timeout warnings add-rule-consequences-automatically
two-phase-query-processing-mode phase-two-starts-warning-tokens
kb-has-changed-warning-tokens told-information-querying
tuple-computation-mode exclude-permutations query-repository
report-inconsistent-queries report-tautological-queries
query-realization bindings check-abox-consistency
use-individual-equivalence-classes rewrite-to-dnf
type-of-substrate abox tbox))
```

Arguments:

`mode`, default `3` - a natural number from 0 to 6, see `set-nrql-mode`, Page 196.

`dont-show-variables`, default `nil` - a list of variables, see also `get-answer`, Page 163 .

`dont-show-lambdas`, default `nil` - `t` or `nil`.

`dont-show-head-projection-operators`, default `nil` - `t` or `nil`.

`abox-mirroring`, default `nil` - `nil, t, :smart,` or `:very-smart`.

`query-optimization`, default `nil` - `t` or `nil`.

`optimizer-use-cardinality-heuristics`, default `t` - `t` or `nil`.

`how-many-tuples`, default `nil` - a natural number, or `nil` (unbounded).

`timeout`, default *current server timeout* - a natural number (the timeout in milliseconds).

`warnings`, default `t` - `t` or `nil`.

`add-rule-consequences-automatically`, default `t` - `t` or `nil`.

`two-phase-query-processing-mode`, default `nil` - `t` or `nil`.

`phase-two-starts-warning-tokens`, default `nil` - `t` or `nil`.

`kb-has-changed-warning-tokens`, default `t` - `t` or `nil`.

`told-information-querying`, default `nil` - `t` or `nil`.

`tuple-computation-mode`, default `set-at-a-time` - `:set-at-a-time` or `:tuple-at-a-time`.

`exclude-permutations`, default `nil` - `t` or `nil`.

`query-repository`, default `nil` - `t` or `nil`.

`report-inconsistent-queries`, default `nil` - `t` or `nil`.

`report-tautological-queries`, default `nil` - `t` or `nil`.

`query-realization`, default `nil` - `t` or `nil`.

`bindings`, default `nil` - a list of (variable value) bindings for variables. These variables will be prebound and treated as individuals, see

with-nrql-settings-evaluated *Macro*

Description: Like `with-nrql-settings`, Page 199, but now the argument forms are evaluated.

6.13 Inference

classify-query *Function*

Description: Classifies the query `id`, i.e., computed its correct position in the current QBox

Syntax: `(classify-query id)`

Arguments: `id` - the ID of the query, or `:last` or `:last-query`.

Values: `:classfied` or `:dont-know`.

See also: `query-entails-p`, Page 202, `query-equivalent-p`, Page 202, `query-parents`, Page 202, `query-equivalents`, Page 202, `query-children`, Page 201, `query-ancestors`, Page 200, `query-descendants`, Page 201

disable-query-realization *Function*

Description: Disables query realization. Inverse of `enable-query-realization`, Page 200.

dont-report-inconsistent-queries-and-rules *Function*

Description: Inverse of `report-inconsistent-queries-and-rules`, Page 203.

enable-query-realization *Function*

Description: Enables query realization. Inverse of `disable-query-realization`, Page 200.

query-ancestors *Function*

Description: Like `query-parents`, Page 202, but the ancestors (i.e., all subsuming resp. entailed queries) from the QBOx are returned.

query-children *Function*

Description: Returns the IDs of the children queries of the query `id` from the QBox. See Section 6.2.8 in the User Guide

Syntax: `(query-children id)`

Arguments: `id1` - the ID of a query, or `:last` or `:last-query`.

Values: A list of query IDs – the children of the query.

See also: `query-parents`, Page 202, `query-equivalents`, Page 202, `query-ancestors`, Page 200, `query-descendants`, Page 201

query-consistent-p *Function*

Description: Checks the consistency of the query `id`

Syntax: `(query-consistent-p id)`

Arguments: `id` - the ID of the query, or `:last` or `:last-query`.

Values: `t`, n̈il or `:dont-know`.

query-descendants *Function*

Description: Like `query-parents`, Page 202, but the ancestors (i.e., all subsuming resp. entailed queries) from the QBOx are returned.

query-entails-p *Function*

Description: Checks whether query `id1` entails (is more specific than) query `id2`

Syntax: `(query-entails-p id1 id2)`

Arguments: `id1` - the ID of a query, or `:last` or `:last-query`.

`id2` - the ID of a query, or `:last` or `:last-query`.

Values: `t`, n̈il or `:dont-know`.

See also: `query-consistent-p`, Page 201, `rule-consistent-p`, Page 203

query-equivalent-p *Function*

Description: Checks whether the two queries `id1` and `id2` mutually subsumes each other

Syntax: `(query-equivalent-p id1 ide2)`

Arguments: `id1` - the ID of a query, or `:last` or `:last-query`.

`id2` - the ID of a query, or `:last` or `:last-query`.

Values: `t`, n̄il or `:dont-know`.

See also: `query-entails-p`, Page 202

query-equivalents *Function*

Description: Returns the IDs of the synonym / equivalent queries of the query `id` from the QBox. See Section 6.2.8 in the User Guide

Syntax: `(query-children id)`

Arguments: `id1` - the ID of a query, or `:last` or `:last-query`.

Values: A list of query IDs – the synonym / equivalent queries.

See also: `query-parents`, Page 202, `query-equivalents`, Page 202, `query-ancestors`, Page 200, `query-descendants`, Page 201

query-parents *Function*

Description: Returns the IDs of the parent queries of the query `id` from the QBox. See Section 6.2.8 in the User Guide

Syntax: `(query-parents id)`

Arguments: `id1` - the ID of a query, or `:last` or `:last-query`.

Values: A list of query IDs – the parents of the query.

See also: `query-children`, Page 201, `query-equivalents`, Page 202, `query-ancestors`, Page 200, `query-descendants`, Page 201

report-inconsistent-queries-and-rules *Function*

Description: Advises nRQL (globally) to automatically check freshly prepared queries and/or rules using `query-consistent-p`, Page 201 and report inconsistent queries and/or rules. Inconsistent queries return no answers and are thus a waste of CPU cycles Inverse of `dont-report-inconsistent-queries-and-rules`, Page 200.

Syntax: `(report-inconsistent-queries-and-rules)`

Arguments:

Values: `:okay-reporting-inconsistent-queries-and-rules`.

Remarks: A query / rule is check for consistency when it is prepared, see `racer-prepare-query`, Page 155 (resp. `racer-prepare-rule`, Page 177).

See also: Inverse of `dont-report-inconsistent-queries-and-rules`, Page 200.`enable-query-repository`, Page 204, `enable-query-realization`, Page 200, `describe-query-processing-mode`, Page 183

rule-consistent-p *Function*

Description: This is the rule equivalent of `query-consistent-p`, Page 201.

6.14 Query Repository

disable-query-repository *Function*

Description: Disables the QBox query repository facility. Inverse of `enable-query-repository`, Page 204.

enable-query-repository *Function*

Description: Enables the QBox query repository facility. Inverse of `disable-query-repository`, Page 204.

get-dag-of-qbox-for-abox *Function*

Description: Returns the DAG of the QBox for the `abox` as a structured list

Syntax: `(get-dag-of-qbox-for-abox abox &optional abox)`

Arguments: `abox` - (current-abox).

Values: the DAG as a structured list.

See also: `show-qbox-for-abox`, Page 205, `get-nodes-in-qbox-for-abox`, Page 204

get-nodes-in-qbox-for-abox *Function*

Description: Returns the DAG nodes (queries) of the QBox for the `abox`

Syntax: `(get-nodes-in-qbox-for-abox &optional abox)`

Arguments: `abox` - (current-abox).

Values: a list of the nodes (queries) from the QBox `(:remarks)` or `(:examples)` or `(:see-also get-dag-of-qbox-for-abox show-qbox-for-abox)`.

show-qbox-for-abox *Function*

Description: Prints the DAG of the QBox for the `abox` as a tree on STDOUT

Syntax: `(show-qbox-for-abox &optional abox definitions-p)`

Arguments: `abox` - (current-abox).

`nil` - if `t`, then the query bodies will be shown.

Values: `:see-output-on-stdout`.

See also: `get-dag-of-qbox-for-abox`, Page 204, `get-nodes-in-qbox-for-abox`, Page 204

6.15 The Substrate Representation Layer

create-data-edge *Function*

Description: Creates an edge between nodes `from` and `to` in the data substrate of type `type-of-substrate` for ABox `abox`. If the nodes do not exists they are created. If the edge does not exist it is created. The label `descr` is added as a conjunct to the label of the edge. If `racer-descr` is specified, then also a role assertion `(related from to racer-descr)` is added to `abox` Inverse of `delete-data-edge`, Page 209.

Syntax: `(create-data-edge from to &key abox type-of-substrate descr racer-descr)`

Arguments: `from` - the name of the from node.

`to` - the name of the to node.

`abox`, default `(current-abox)` - the (name of the) ABox.

`type-of-substrate`, default *'racer-dummy-substrate* - the type of the substrate.

`descr`, default `nil` - the label.

`racer-descr`, default `nil` - a RacerPro concept.

Values: The name of the node.

See also: Inverse of `delete-data-edge`, Page 209.`create-data-node`, Page 207, `get-data-edge-label`, Page 214

create-data-node *Function*

Description: Creates a node named `name` in the data substrate of type `type-of-substrate` for ABox `abox` if the node does not exists yet. The label `descr` is added as a conjunct to the label of `name`. If `racer-descr` is specified, then also a concept assertion `(instance name racer-descr)` is added to `abox` Inverse of `delete-data-node`, Page 209.

Syntax: `(create-data-node name &key abox type-of-substrate descr racer-descr)`

Arguments: `name` - the name of the node.

`abox`, default `(current-abox)` - the (name of the) ABox.

`type-of-substrate`, default *'racer-dummy-substrate* - the type of the substrate.

`descr`, default `nil` - the label.

`racer-descr`, default `nil` - a RacerPro concept.

Values: The name of the node.

See also: Inverse of `delete-data-node`, Page 209.`create-data-edge`, Page 206, `get-data-node-label`, Page 214

create-rcc-edge *Function*

Description: Synonym for `create-data-edge`, Page 206.

create-rcc-node *Function*

Description: Synonym for `create-data-node`, Page 207.

data-edge1 *Function*

Description: Like `create-data-edge`, Page 206, but with signature `(from to &optional descr racer-descr abox type-of-substrate)`. See also corresponding macro `data-edge`, Page 218.

data-node1 *Function*

Description: Like `create-data-node`, Page 207, but with the signature `(name &optional descr racer-descr abox type-of-substrate)`. See also corresponding macro `data-node`, Page 218.

del-data-edge1 *Function*

Description: Like `delete-data-edge`, Page 209, but with the signature `(from to &optional abox type-of-substrate)`. See also corresponding macro `del-data-edge`, Page 218.

del-data-node1 *Function*

Description: Like `delete-data-node`, Page 209, but with the signature `(name &optional abox type-of-substrate)`. See also corresponding macro `del-data-node`, Page 218.

del-rcc-edge1 *Function*

Description: See also corresponding macro `del-rcc-edge`, Page 219. Synonym for `del-data-edge1`, Page 208.

del-rcc-node1 *Function*

Description: See also corresponding macro `del-rcc-node`, Page 219. Synonym for `del-data-node1`, Page 208.

delete-data-edge *Function*

Description: Deletes the edge between the nodes `from` and `to` from the data substrate of type `type-of-substrate` for the ABox `abox` Inverse of `create-data-edge`, Page 206.

Syntax: `(delete-data-edge from to &key abox type-of-substrate)`

Arguments: `from` - the name of the from node.

`from` - the name of the to node.

`abox`, default `(current-abox)` - the (name of the) ABox.

`type-of-substrate`, default *'racer-dummy-substrate* - the type of the substrate.

Values: `:okay-deleted` or `:not-found`.

See also: Inverse of `create-data-edge`, Page 206.`create-data-edge`, Page 206, `get-data-edge-label`, Page 214

delete-data-node *Function*

Description: Deletes the node named `name` from the data substrate of type `type-of-substrate` for the ABox `abox` Inverse of `create-data-node`, Page 207.

Syntax: `(delete-data-node name &key abox type-of-substrate)`

Arguments: `name` - the name of the node.

`abox`, default `(current-abox)` - the (name of the) ABox.

`type-of-substrate`, default *'racer-dummy-substrate* - the type of the substrate.

Values: `:okay-deleted` or `:not-found`.

See also: Inverse of `create-data-node`, Page 207.`create-data-node`, Page 207, `get-data-edge-label`, Page 214

delete-rcc-synonyms *Function*

Description: Deletes all registered RCC synonyms (see `register-rcc-synonym`, Page 217.

describe-all-edges *Function*

Description: Like `describe-all-nodes`, Page 210, but for the edges.

describe-all-nodes *Function*

Description: Returns a list containing the result of applying `get-data-node-description`, Page 214 on `get-substrate-nodes`, Page 215

Syntax: `(describe-all-nodes &key abox type-of-substrate)`

Arguments: `abox`, default `(current-abox)` - the (name of the) ABox.

`type-of-substrate`, default *'racer-dummy-substrate* - the type of the substrate.

Values: A list of node descriptions.

See also: `describe-all-edges`, Page 210

describe-current-substrate *Function*

Description: Returns a description of the current substrate used for query answering

Syntax: `(describe-current-substrate)`

Arguments:

Values: The description, a structured list.

See also: `describe-query-processing-mode`, Page 183, `describe-query`, Page 119, `describe-rule`, Page 119

describe-substrate *Function*

Description: Returns a description of the substrate for ABox `abox` of type `type-of-substrate`

Syntax: `(describe-substrate &optional abox type-of-substrate)`

Arguments: `abox`, default `(current-abox)` - the (name of the) ABox.

`type-of-substrate`, default *'racer-dummy-substrate* - the type of the substrate.

Values: A description of the substrate, a structured list.

Remarks: .

See also: `describe-all-substrates`, Page 118, `describe-substrate`, Page 211

description-implies-p *Function*

Description: Checks whether label `a` implies label `b` See also corresponding macro `description-implies?`, Page 219.

Syntax: `(description-implies-p a b)`

Arguments: `a` - a label.

`b` - a label.

Values: `t` or `nil`.

Examples: `(description-implies-p 'a '((a b)))`

See also: Corresponding macro: `description-implies?`, Page 219.

disable-data-substrate-mirroring *Function*

Description: Inverse of `enable-data-substrate-mirroring`, Page 212.

disable-rcc-substrate-mirroring *Function*

Description: Inverse of `enable-rcc-substrate-mirroring`, Page 213.

edge-description1 *Function*

Description: Like `get-data-edge-description`, Page 213, but with the signature `(from to &optional abox type-of-substrate)`. See also corresponding macro `edge-description`, Page 219.

edge-label1 *Function*

Description: Like `get-data-edge-label`, Page 214, but with the signature `(from to &optional abox type-of-substrate)`. See also corresponding macro `edge-label`, Page 219.

enable-data-substrate-mirroring *Function*

Description: Advises nRQL (globally) to create substrates of type `mirror-data-substrate`. Additional retrieval facilities (especially for OWL) are provided on this kind of substrate. Please refer to the User Guide. See also argument `type-of-substrate` of function `racer-prepare-query`, Page 155 Inverse of `disable-data-substrate-mirroring`, Page 211.

Syntax: `(enable-data-substrate-mirroring)`

Arguments:

Remarks: If you want to exploit the additional retrieval facilities offered by the (mirror) data substrate, then make sure that this function is called before the first nRQL query is posed.

See also: Inverse of `disable-data-substrate-mirroring`, Page 211.`disable-data-substrate-mirroring`, Page 211, `set-nrql-mode`, Page 196, `with-nrql-settings`, Page 199, `describe-query-processing-mode`, Page 183

enable-rcc-substrate-mirroring *Function*

Description: Like `enable-data-substrate-mirroring`, Page 212, but substrates of type `rcc-mirror-substrate` are created. This is a `mirror-data-substrate` as well as a `rcc-substrate`. Inverse of `disable-rcc-substrate-mirroring`, Page 211.

get-data-edge-description *Function*

Description: Returns the edge label of the edge between the nodes `from` and `to` from the data substrate of type `type-of-substrate` for the ABox `abox`

Syntax: `(get-data-edge-label from to &key abox type-of-substrate)`

Arguments: `from` - the name of the from node.

`to` - the name of the to node.

`abox`, default `(current-abox)` - the (name of the) ABox.

`type-of-substrate`, default *'racer-dummy-substrate* - the type of the substrate.

Values: the label of the node, a structured list.

See also: `get-data-edge-label`, Page 214

get-data-edge-label *Function*

Description: Returns a description of the edge between the nodes `from` and `to` from the data substrate of type `type-of-substrate` for the ABox `abox`

Syntax: `(get-data-edge-description from to &key abox type-of-substrate)`

Arguments: `from` - the name of the from node.

`to` - the name of the to node.

`abox`, default `(current-abox)` - the (name of the) ABox.

`type-of-substrate`, default *'racer-dummy-substrate* - the type of the substrate.

Values: a description of the edge, a structured list.

See also: `get-data-edge-description`, Page 213

get-data-node-description *Function*

Description: Returns a description of the node `name` from the data substrate of type `type-of-substrate` for the ABox `abox`

Syntax: `(get-data-node-description name &key abox type-of-substrate)`

Arguments: `name` - the name of the node.

`abox`, default `(current-abox)` - the (name of the) ABox.

`type-of-substrate`, default *'racer-dummy-substrate* - the type of the substrate.

Values: the description of the node, a structured list.

See also: `get-data-node-label`, Page 214

get-data-node-label *Function*

Description: Returns the node label of the node `name` from the data substrate of type `type-of-substrate` for the ABox `abox`

Syntax: `(get-data-node-label name &key abox type-of-substrate)`

Arguments: `name` - the name of the node.

`abox`, default `(current-abox)` - the (name of the) ABox.

`type-of-substrate`, default *'racer-dummy-substrate* - the type of the substrate.

Values: the label of the node, a structured list.

See also: `get-data-node-description`, Page 214

get-substrate-edges *Function*

Description: Like `get-substrate-nodes`, Page 215, but for the edges.

get-substrate-nodes *Function*

Description: Returns all nodes in the substrate of type `type-of-substrate` for ABox `abox`

Syntax: `(get-substrate-nodes &key abox type-of-substrate)`

Arguments: `abox`, default `(current-abox)` - the (name of the) ABox.

`type-of-substrate`, default *'racer-dummy-substrate* - the type of the substrate.

Values: A list of node names.

See also: `get-substrate-edges`, Page 214

node-description1 *Function*

Description: Like `get-data-node-description`, Page 214, but with the signature `(name &optional abox type-of-substrate)`. See also corresponding macro `node-description`, Page 220.

node-label1 *Function*

Description: Like `get-data-node-label`, Page 214, but with the signature `(name &optional abox type-of-substrate)`. See also corresponding macro `node-label`, Page 220.

rcc-consistent-p *Function*

Description: Checks the RCC substrate for relational RCC consistency See also corresponding macro `rcc-consistent?`, Page 220.

Syntax: `(rcc-consistent-p &optional abox type-of-substrate)`

Arguments: `abox` - (current-abox).

`type-of-substrate` - 'rcc-substrate.

Values: `t` or `nil`.

See also: Corresponding macro: `rcc-consistent?`, Page 220.

rcc-edge-description1 *Function*

Description: See also corresponding macro `rcc-edge-description`, Page 220. Synonym for `node-description1`, Page 215.

rcc-edge-label1 *Function*

Description: See also corresponding macro `rcc-edge-label`, Page 220. Synonym for `node-label1`, Page 215.

rcc-edge1 *Function*

Description: See also corresponding macro `rcc-edge`, Page 220. Synonym for `data-edge1`, Page 207.

rcc-instance1 *Function*

Description: See also corresponding macro `rcc-instance`, Page 220. Synonym for `data-node1`, Page 208.

rcc-node-description1 *Function*

Description: See also corresponding macro `rcc-node-description`, Page 221. Synonym for `node-description1`, Page 215.

rcc-node-label1 *Function*

Description: See also corresponding macro `rcc-node-label`, Page 221. Synonym for `node-label1`, Page 215.

rcc-node1 *Function*

Description: See also corresponding macro `rcc-node`, Page 220. Synonym for `data-node1`, Page 208.

rcc-related1 *Function*

Description: See also corresponding macro `rcc-related`, Page 221. Synonym for `data-edge1`, Page 207.

register-rcc-synonym *Function*

Description: Registers the role name `role` as a synonym for the RCC relation `rcc-relation` See also corresponding macro `rcc-synonym`, Page 221.

Syntax: `(register-rcc-synoym role rcc-relation)`

Arguments: `role` - the role.

`rcc-relation` - the RCC relation.

Values: the synonym mapping function (a list).

See also: Corresponding macro: `rcc-synonym`, Page 221.

set-data-box *Function*

Description: Creates a data substrate for ABox `abox` See also corresponding macro `in-data-box`, Page 219.

Syntax: `(set-data-box abox)`

Arguments: `abox` - the (name of the) ABox.

Values: the `abox`.

See also: Corresponding macro: `in-data-box`, Page 219. `set-mirror-data-box`, Page 218, `set-rcc-box`, Page 218

set-mirror-data-box *Function*

Description: Like `set-data-box`, Page 217, but a mirror data box is created. See also corresponding macro `in-mirror-data-box`, Page 219.

set-rcc-box *Function*

Description: Creates an RCC substrate for ABox `abox` See also corresponding macro `in-rcc-box`, Page 219.

Syntax: `(set-rcc-box abox &optional rcc-type)`

Arguments: `abox` - the (name of the) ABox.

`rcc-type` - rcc8.

Values: the `abox`.

See also: Corresponding macro: `in-rcc-box`, Page 219. `set-data-box`, Page 217, `enable-data-substrate-mirroring`, Page 212, `set-substrate-type`, Page 124

data-edge *Macro*

Description: See also corresponding function `data-edge1`, Page 207.

data-node *Macro*

Description: See also corresponding function `data-node1`, Page 208.

del-data-edge *Macro*

Description: See also corresponding function `del-data-edge1`, Page 208.

del-data-node *Macro*

Description: See also corresponding function `del-data-node1`, Page 208.

del-rcc-edge *Macro*

Description: See also corresponding function `del-rcc-edge1`, Page 208.

del-rcc-node *Macro*

Description: See also corresponding function `del-rcc-node1`, Page 208.

description-implies? *Macro*

Description: See also corresponding function `description-implies-p`, Page 211.

edge-description *Macro*

Description: See also corresponding function `edge-description1`, Page 212.

edge-label *Macro*

Description: See also corresponding function `edge-label1`, Page 212.

in-data-box *Macro*

Description: See also corresponding function `set-data-box`, Page 217.

in-mirror-data-box *Macro*

Description: See also corresponding function `set-mirror-data-box`, Page 218.

in-rcc-box *Macro*

Description: See also corresponding function `set-rcc-box`, Page 218.

node-description *Macro*

Description: See also corresponding function `node-description1`, Page 215.

node-label *Macro*

Description: See also corresponding function `node-label1`, Page 215.

rcc-consistent? *Macro*

Description: See also corresponding function `rcc-consistent-p`, Page 216.

rcc-edge *Macro*

Description: See also corresponding function `rcc-edge1`, Page 216.

rcc-edge-description *Macro*

Description: See also corresponding function `rcc-edge-description1`, Page 216.

rcc-edge-label *Macro*

Description: See also corresponding function `rcc-edge-label1`, Page 216.

rcc-instance *Macro*

Description: See also corresponding function `rcc-instance1`, Page 216.

rcc-node *Macro*

Description: See also corresponding function `rcc-node1`, Page 217.

rcc-node-description *Macro*

Description: See also corresponding function `rcc-node-description1`, Page 216.

rcc-node-label *Macro*

Description: See also corresponding function `rcc-node-label1`, Page 216.

rcc-related *Macro*

Description: See also corresponding function `rcc-related1`, Page 217.

rcc-synonym *Macro*

Description: See also corresponding function `register-rcc-synonym`, Page 217.

6.16 The nRQL Persistency Facility

restore-all-substrates *Function*

Description: Restores the substrates from the files `<filename>.SUB.IMG` as well as they KBs they reference from `<filename>.KBS.IMG` Inverse of `store-all-substrates`, Page 223.

Syntax: `(restore-substrate filename)`

Arguments: `filename` - the filename.

Values: The names of the restored substrates.

Remarks: Note that the referenced KBs (ABox, TBox) are restored from the file `<filename>.KB.IMG` using `restore-kbs-image`, Page 233.

See also: Inverse of `store-all-substrates`, Page 223.`restore-substrate`, Page 223, `restore-server-image`, Page 222

restore-server-image *Function*

Description: Restores a server image from the files `<filename>.SUB.IMG` and `<filename>.KBS.IMG` Inverse of `store-server-image`, Page 224.

Syntax: `(restore-server-image filename)`

Arguments: `filename` - the filename.

Values: `:done`.

See also: Inverse of `store-server-image`, Page 224.`restore-all-substrates`, Page 222, `restore-substrates`, Page **??**

restore-substrate *Function*

Description: Restores the substrate from the files `<filename>.SUB.IMG` and `<filename>.KBS.IMG` Inverse of `store-substrate-for-abox`, Page 224.

Syntax: `(restore-substrate filename)`

Arguments: `filename` - the filename.

Values: The name of the restored substrate.

Remarks: Note that the referenced KB (ABox, TBox) is also restored from the file `<filename>.KB.IMG` using `restore-kb-image`, Page 233.

See also: Inverse of `store-substrate-for-abox`, Page 224.`restore-all-substrates`, Page 222, `restore-server-image`, Page 222

store-all-substrates *Function*

Description: Stores all substrates in a file `filename` Inverse of `restore-all-substrates`, Page 222.

Syntax: `(store-all-substrates filename)`

Arguments: `filename` - the filename.

Values: `:done`.

Remarks: Note that RacerPro must be running in unsafe mode (i.e., file io must be allowed). RacerPro also stores all KBs (ABoxes, TBoxes) referenced by the substrates. Thus, RacerPro creates two files: `<filename>.SUB.IMG` contains the substrates, and `<filename>.SUB.IMG` contains the KBs (this image is store with `store-kbs-image`, Page 233).

See also: Inverse of `restore-all-substrates`, Page 222.`store-substrate-for-abox`, Page 224, `store-server-image`, Page 224

store-server-image *Function*

Description: Stores the server state in the files `<filename>.SUB.IMG` and `<filename>.KBS.IMG` Inverse of `restore-server-image`, Page 222.

Syntax: `(store-server-image filename)`

Arguments: `filename` - the filename.

Values: `:done`.

Remarks: This is like `store-all-substrate`, Page **??**, but all KBs (not only the KBs reference by the substrates) are stored in the image.

See also: Inverse of `restore-server-image`, Page 222.`store-all-substrate`, Page **??**, `store-substrate-for-abox`, Page 224

store-substrate-for-abox *Function*

Description: Stores the substrate for ABox `abox` of type (class) `type-of-substrate` Inverse of `restore-substrate`, Page 223.

Syntax: `(store-substrate-for-abox filename &optional for-abox type-of-substrate)`

Arguments: `filename` - the filename.

`for-abox` - (current-abox).

`type-of-substrate` - 'racer-dummy-substrate.

Values: `:done`.

Remarks: Note that RacerPro must be running in unsafe mode (i.e., file io must be allowed). Also note that the KB (ABox, TBox) is also stored in the file; RacerPro creates two files: `<filename>.SUB.IMG` for the substrate, and `<filename>.SUB.IMG` for the KB using `store-kb-image`, Page 233.

See also: Inverse of `restore-substrate`, Page 223.`store-all-substrates`, Page 223, `store-server-image`, Page 224

WARNING Unclassified: (get-individual-successors set-rewrite-defined-concepts enable-abduction disable-abduction)

Chapter 7

Publish and Subscribe Functions

In the following the functions offered by the publish-subscribe facility are explained in detail.

publish *macro*

Description: Publish an ABox individual.

Syntax: `(publish` *IN*
`&optional (`*ABN* `(current-abox)))`

Arguments: *IN* - individual name

ABN - ABox name

Values: A list of tuples consisting of subscriber and individuals names.

publish-1 *macro*

Description: Functional interface for `publish`.

Syntax: `(publish-1` *IN*
`&optional (`*ABN* `(current-abox)))`

Arguments: *IN* - individual name

ABN - ABox name

unpublish *macro*

Description: Withdraw a publish statement.

Syntax:
```
(unpublish IN
  &optional (ABN (current-abox)))
```

Arguments: *IN* - individual name

ABN - ABox name

unpublish-1 *function*

Description: Functional interface for `unpublish`.

Syntax:
```
(unpublish-1 IN
  &optional (ABN (abox-name (current-abox))))
```

Arguments: *IN* - individual name

ABN - ABox name

subscribe *macro*

Description: Subscribe to an instance retrieval query.

Syntax:
```
(subscribe subscriber-name C
  ABN
  (:notification-method tcp host port))
```

Arguments: *IN* - individual name

ABN - ABox name

host - host name, a string

port - port, an integer

Values: A list of tuples consisting of subscriber and individuals names.

subscribe-1 *function*

Description: Functional interface for `subscribe`.

Syntax: `(subscribe-1` *subscriber-name C ABN* `(:notification-method tcp` *host port*`))`

Arguments: *IN* - individual name

ABN - ABox name

host - host name, a string

port - port, an integer

unsubscribe *macro*

Description: Retract a subscription.

Syntax: `(unsubscribe` *subscriber-name* `&optional` *C* `(`*ABN* `(current-abox)))`

Arguments: *subscriber-name* - subscriber name

C - concept term

ABN - ABox name

unsubscribe-1 *function*

Description: Functional interface for `unsubscribe`.

Syntax: `(unsubscribe` *subscriber-name* `&optional` *C* `(`*ABN* `(current-abox)))`

Arguments: *subscriber-name* - subscriber name

C - concept term

ABN - ABox name

init-subscriptions *macro*

Description: Initialize the subscription database.

Syntax: `(init-subscriptions &optional (`*ABN*` (current-abox)))`

Arguments: *ABN* - ABox name

init-subscriptions-1 *function*

Description: Functional interface for `init-subscriptions`

Syntax: `(init-subscriptions-1 &optional (`*ABN*` (current-abox)))`

Arguments: *ABN* - ABox name

init-publications *macro*

Description: Initialize the set of published individuals.

Syntax: `(init-publications &optional (`*ABN*` (current-abox)))`

Arguments: *ABN* - ABox name

init-publications-1 *function*

Description: Functional interface for `init-subscription`.

Syntax: `(init-publications-1 &optional (`*ABN*` (current-abox)))`

Arguments: *ABN* - ABox name

check-subscriptions *macro*

Description: Explicitly check for new instance retrieval results w.r.t. the set of subscriptions.

Syntax: `(check-subscriptions` *ABN*`)`

Arguments: *ABN* - ABox name

Values: A list of tuples consisting of subscriber and individuals names.

Chapter 8

The Racer Persistency Services

The following functions define the Racer Persistency Services.

store-tbox-image *function*

Description: Store an image of a TBox.

Syntax: `(store-tbox-image` *filename* `&optional (`*TBN* `(current-tbox))`

Arguments: *filename* - filename
TBN - tbox name

store-tboxes-image *function*

Description: Store an image of a list of TBoxes.

Syntax: `(store-tboxes-image` *tboxes* *filename*`)`

Arguments: *tboxes* - a list of TBox names
filename - filename

restore-tbox-image *function*

Description: Restore an image of a TBox.

Syntax: `(restore-tbox-image` *filename*`)`

Arguments: *filename* - filename

restore-tboxes-image *function*

Description: Restore an image of a set of TBoxes.

Syntax: `(restore-tboxes-image` *filename*`)`

Arguments: *filename* - filename

store-abox-image *function*

Description: Store an image of an Abox.

Syntax: `(store-abox-image` *filename* `&optional (`*ABN* `(current-abox)))`

Arguments: *filename* - filename

ABN - abox name

store-aboxes-image *function*

Description: Store an image of a list of Aboxes.

Syntax: `(store-aboxes-image` *aboxes filename*`)`

Arguments: *aboxes* - a list of abox names

filename - filename

restore-abox-image *function*

Description: Restore an image of an Abox.

Syntax: `(restore-abox-image` *filename*`)`

Arguments: *filename* - filename

restore-aboxes-image *function*

Description: Restore an image of a set of aboxes.

Syntax: `(restore-aboxes-image` *filename*`)`

Arguments: *filename* - filename

store-kb-image *function*

Description: Store an image of an kb.

Syntax: `(store-kb-image` *filename* `&optional (`*KBN* `(current-tbox)))`

Arguments: *filename* - filename
KBN - kb name

store-kbs-image *function*

Description: Store an image of a list of kbs.

Syntax: `(store-kbs-image` *kbs filename*`)`

Arguments: *kbs* - a list of knowledge base names
filename - filename

restore-kb-image *function*

Description: Restore an image of an kb.

Syntax: `(restore-kb-image` *filename*`)`

Arguments: *filename* - filename

restore-kbs-image *function*

Description: Restore an image of a set of kbs.

Syntax: `(restore-kbs-image` *filename*`)`

Arguments: *filename* - filename

Index