RacerPro Reference Manual
Version 1.9

Racer Systems GmbH & Co. KG

October 12, 2010

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

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

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. Arguments which will not get evaluated by a macro are indicated with an asterix ("*").

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 144 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

get-nrql-version *Function*

Description: Returns the current version number of nRQL.

Syntax: `(get-nrql-version)`

Arguments:

Values: The current nRQL version number.

enable-nrql-warnings *Function*

Description: Advises nRQL to print out warnings on STDOUT. Moreover, *warning tokens* will be delivered in some circumstances, see `enable-kb-has-changed-warning-tokens`, Page 167, `enable-phase-two-starts-warning-tokens`, Page 167.

Syntax: `(enable-nrql-warnings)`

Arguments:

Values: `:OKAY-WARNINGS-ENABLED`

See also: `disable-nrql-warnings`, Page 117, `enable-kb-has-changed-warning-tokens`, Page 167, `enable-phase-two-starts-warning-tokens`, Page 167

disable-nrql-warnings *Function*

Description: Inverse function of `enable-nrql-warnings`, Page 117.

restore-standard-settings *Function*

Description: Resets nRQL into default query processing mode. Nothing is deleted.

Syntax: `(restore-standard-settings)`

Arguments:

Values: `:OKAY-STANDARD-SETTINGS-RESTORED`

Examples:
```
> (restore-standard-settings)
:OKAY-STANDARD-SETTINGS-RESTORED
> (describe-query-processing-mode)
((:CREATING-SUBSTRATES-OF-TYPE :RACER-DUMMY-SUBSTRATE)
:CHECK-ABOX-CONSISTENCY :QUERY-OPTIMIZATION-ENABLED
:OPTIMIZER-USES-CARDINALITY-HEURISTICS
:AUTOMATICALLY-ADDING-RULE-CONSEQUENCES :WARNINGS
:COMPLETE-MODE :MODE-3 :SET-AT-A-TIME-MODE
:DELIVER-KB-HAS-CHANGED-WARNING-TOKENS)
```

See also: `reset-nrql-engine`, Page 119

reset-nrql-engine *Function*

Description: Aborts all currently active queries and rules. Resets the internal caches of the nRQL engine and then calls `restore-standard-settings`, Page 118. If *full-reset-p* `T` is used, nRQL will also *delete* all TBoxes and all ABoxes, delete all queries, all rules, all substrates (as well as QBoxes) and all associated query definitions.

Syntax: `(reset-nrql-engine &key` *full-reset-p*`)`

Arguments: *full-reset-p* - pass `T` if you really want to reset the nRQL engine fully - note that this will delete everything from the RacerPro server.

Values: `:OKAY-ENGINE-RESET`

Remarks: It should not be necessary to call this function. If a TBox and/or ABox gets changed, nRQL will be notified by RacerPro in order to invalidate its caches automatically.

See also: `restore-standard-settings`, Page 118, `reset-nrql-engine`, Page 119

full-reset *Function*

Description: Simply calls `reset-nrql-engine`, Page 119 with *full-reset-p* = `T`.

prepare-nrql-engine *Function*

Description: Prepares the internal index structures of the nRQL engine for query answering on the ABox *abox*. Usually, there is no need to call this function explicitly, since nRQL will automatically prepare and compute its index structures from a RacerPro ABox if needed.

If the nRQL engine is not explicitly prepared with this function for query answering on ABox *abox*, then the first call to `retrieve`, Page 144 or `racer-answer-query`, Page 145 will prepare the nRQL engine. This means that answering the first query for an ABox takes usually considerably longer than answering subsequent queries if the engine has not been prepared explicitly.

Syntax: `(prepare-nrql-engine` *abox* `&rest` *args*`)`

Arguments: *abox* - the name of the ABox for which the engine is prepared.

args - a list of optional keyword-value arguments, see `with-nrql-settings`, Page 178 for a description of the valid keyword arguments, with the exception of the *abox* argument (since this is the mandatory first argument of this function).

Values: The name of the ABox.

See also: `reset-nrql-engine`, Page 119, `prepare-racer-engine`, Page 83

6.2 Query / Rule Management

all-queries *Function*

Description: Returns all queries, including queries which are ready to run (have been prepared), which are currently running, or have already been processed (and are thus terminated).

Syntax: `(all-queries)`

Arguments:

Values: A list of query IDs

Remarks: As long as a query is on this list, API (functions and macros) will "know" the Id of that query. However, certain API functions (macros) can only be applied if a query is in a certain state.

See also: `all-rules`, Page 120

all-rules *Function*

Description: Equivalent of `all-queries`, Page 120 for rules.

accurate-queries *Function*

Description: Returns all queries which are still accurate (see `query-accurate-p`, Page 154). A query is *accurate* iff the referenced ABox has not changed since the parsing of the query.

Syntax: `(accurate-queries)`

Arguments:

Values: A list of query IDs

See also: `accurate-rules`, Page 121, `inaccurate-rules`, Page 121

inaccurate-queries *Function*

Description: See `accurate-queries`, Page 120.

accurate-rules *Function*

Description: Equivalent of `accurate-queries`, Page 120 for rules.

inaccurate-rules *Function*

Description: See `accurate-rules`, Page 121.

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`.

Values: `:OKAY-QUERY-DELETED` or `:NOT-FOUND`

See also: `delete-all-queries`, Page 121

delete-rule *Function*

Description: Equivalent of `delete-query`, Page 121 for rules.

delete-all-queries *Function*

Description: Deletes all queries.

Syntax: `(delete-all-queries)`

Arguments:

Values: `:OKAY-ALL-QUERIES-DELETED`

See also: `delete-all-rules`, Page 122

delete-all-rules *Function*

Description: Equivalent of `delete-all-queries`, Page 121 for rules.

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`.

Values: A list of status symbols describing the current status of the query. Returned status symbols are: `:READY-TO-RUN`, `:RUNNING`, `:WAITING-FOR-GET-NEXT-TUPLE`, `:PROCESSED`, `:ACCURATE`, `:NOT-ACCURATE`, `:PHASE-ONE`, `:PHASE-TWO`, `:PROCESSED`

See also: `describe-all-queries`, Page 124

describe-rule-status *Function*

Description: Equivalent of `describe-query-status`, Page 122 for rules.

query-head *Function*

Description: Retrieves the (possibly rewritten) head of a query.

Syntax: `(query-head` *id*`)`

Arguments: *id* - the ID of the query, or `:last`.

Values: The (possibly rewritten) head of the query.

Remarks: Usually, individuals in the original query head are replaced by representative variables.

See also: `original-query-head`, Page 123

rule-head *Function*

Description: Equivalent of `query-head`, Page 122 for rules.

original-query-head *Function*

Description: Like `query-head`, Page 122, but the *original,* non-rewritten head is returned.

original-rule-head *Function*

Description: Equivalent of `original-query-head`, Page 123 for rules.

query-body *Function*

Description: Retrieves the (possibly rewritten) body of a query.

Syntax: `(query-body` *id*`)`

Arguments: *id* - the ID of the query, or `:last`.

Values: The (potentially rewritten) body of the query.

Remarks: The original body is usually rewritten (optimized, brought into DNF, etc.)

See also: `original-query-body`, Page 123

rule-body *Function*

Description: Equivalent of `query-body`, Page 123 for rules.

original-query-body *Function*

Description: Like `query-body`, Page 123, but the original, non-rewritten body is returned.

original-rule-body *Function*

Description: Equivalent of `original-query-body`, Page 123 for rules.

describe-query *Function*

Description: Returns a description of the query *id*.

Syntax: `(describe-query` *id* `&optional (`*rewritten-p* `T))`

Arguments: *id* - the ID of the query, or `:last`.

rewritten-p - if set to `NIL` (`T` is the default value), then the *original* query will be returned, otherwise the internally rewritten query.

Values: The internally rewritten or original syntactic description of the query.

Remarks: This function uses `describe-query-status`, Page 122, `query-head`, Page 122 (or `original-query-head`, Page 123) as well as `query-body`, Page 123 (or `original-query-body`, Page 123).

See also: `describe-rule`, Page 124

describe-rule *Function*

Description: Equivalent of `describe-query`, Page 124 for rules.

describe-all-queries *Function*

Description: Returns a list of descriptions of all queries.

Syntax: `(describe-all-queries &optional (`*rewritten-p* `T))`

Arguments: *rewritten-p* - see `describe-query`, Page 124.

Values: A list of query descriptions.

Remarks: Simply "maps" `describe-query`, Page 124 over `all-queries`, Page 120.

See also: `describe-query`, Page 124, `all-queries`, Page 120

describe-all-rules *Function*

Description: Equivalent of `describe-all-queries`, Page 124 for rules.

6.3 Query / Rule Life Cycle

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`.

Values: `T` or `NIL`

Remarks: Use `execute-query`, Page 139 to run (start) the query.

See also: `ready-queries`, Page 130

rule-ready-p *Function*

Description: Equivalent of `query-ready-p`, Page 125 for rules.

query-prepared-p *Function*

Description: Equivalent to `query-ready-p`, Page 125

rule-prepared-p *Function*

Description: Equivalent of `query-prepared-p`, Page 125 for rules.

query-active-p *Function*

Description: Checks whether query *id* is active. A query is active iff a corresponding query answering thread exists.

If query *id* had been started in lazy incremental mode, then this thread will be put to sleep after a new tuple has been computed until the next tuple is requested (see `query-waiting-p`, Page 126).

Syntax: `(query-active-p` *id*`)`

Arguments: *id* - the ID of the query, or `:last`.

Values: `T` or `NIL`

See also: `active-queries`, Page 131

rule-active-p *Function*

Description: Equivalent of `query-active-p`, Page 126 for rules.

query-waiting-p *Function*

Description: Checks whether query *id* is waiting (sleeping). An active query is waiting iff the corresponding query answering thread is currently sleeping (waiting) until the next tuple is requested via `get-next-tuple`, Page 150.

Syntax: `(query-waiting-p` *id*`)`

Arguments: *id* - the ID of the query, or `:last`.

Values: `T` or `NIL`

Remarks: Only active queries can be waiting. Otherwise a query is prepared (ready) or processed (terminated).

See also: `waiting-queries`, Page 134

rule-waiting-p *Function*

Description: Equivalent of `query-waiting-p`, Page 126 for rules.

query-processed-p *Function*

Description: Checks whether query *id* has terminated. Thus, the query answering thread of this query has died. This is the case if all tuples have been computed, if the query has been aborted, or a timeout occurred, or if the maximal number of requested tuples bound has been reached.

Syntax: `(query-processed-p` *id*`)`

Arguments: *id* - the ID of the query, or `:last`.

Values: `T` or `NIL`

See also: `processed-queries`, Page 135

rule-processed-p *Function*

Description: Equivalent of `query-processed-p`, Page 127 for rules.

query-inactive-p *Function*

Description: Equivalent of `query-processed-p`, Page 127.

rule-inactive-p *Function*

Description: Equivalent of `query-inactive-p`, Page 127 for rules.

cheap-query-p *Function*

Description: Checks whether query *id* is still in *phase one* (see User Guide), thus still producing "cheap tuples".

Syntax: `(cheap-query-p` *id*`)`

Arguments: *id* - the ID of the query, or `:last`.

Values: `T` or `NIL`

Remarks: Not only active queries can be cheap. Also prepared (ready) queries which have not yet been started will be cheap if two-phase processing is enabled. In contrast, a query can only be expensive if it is active, `active-expensive-query-p`, Page 128.

See also: `cheap-queries`, Page 129, `active-cheap-queries`, Page 131

cheap-rule-p *Function*

Description: Equivalent of `cheap-query-p`, Page 128 for rules.

active-expensive-query-p *Function*

Description: Checks whether query *id* is already in *phase two* (see User Guide), and thus can no longer produce "cheap tuples". The subsequently produced tuples are therefore "expensive tuples".

Syntax: `(active-expensive-query-p` *id*`)`

Arguments: *id* - the ID of the query, or `:last`.

Values: `T` or `NIL`

Remarks: Note that a query can only be expensive if it is also active. See also `cheap-query-p`, Page 128.

See also: `expensive-queries`, Page 129

active-expensive-rule-p *Function*

Description: Equivalent of `active-expensive-query-p`, Page 128 for rules.

cheap-queries *Function*

Description: Returns the list of all cheap queries.

Syntax: `(cheap-queries)`

Arguments:

Values: A list of query IDs.

Remarks: Each of these queries satisfies `cheap-query-p`, Page 128. Note that these queries are either prepared (ready) or active.

See also: `cheap-query-p`, Page 128

cheap-rules *Function*

Description: Equivalent of `cheap-queries`, Page 129 for rules.

expensive-queries *Function*

Description: Returns the list of all expensive queries.

Syntax: `(expensive-queries)`

Arguments:

Values: A list of query IDs.

Remarks: Each of these queries satisfies `active-expensive-query-p`, Page 128. Only active queries can be expensive.

See also: `active-expensive-query-p`, Page 128

expensive-rules *Function*

Description: Equivalent of `expensive-queries`, Page 129 for rules.

ready-queries *Function*

Description: Returns the list of all ready (prepared) queries ("ready to run").

Syntax: `(ready-queries)`

Arguments:

Values: A list of query IDs.

Remarks: Each of these queries satisfies `query-ready-p`, Page 125.

See also: `query-ready-p`, Page 125

ready-rules *Function*

Description: Equivalent of `ready-queries`, Page 130 for rules.

prepared-queries *Function*

Description: Equivalent of `ready-queries`, Page 130

prepared-rules *Function*

Description: Equivalent of `prepared-queries`, Page 130 for rules.

active-queries *Function*

Description: Returns the list of all active queries.

Syntax: `(active-queries)`

Arguments:

Values: A list of query IDs.

Remarks: Each of these queries satisfies `query-active-p`, Page 126. Note that this list is further partitioned into running and waiting (sleeping) queries, `running-queries`, Page 132, `waiting-queries`, Page 134.

See also: `query-active-p`, Page 126

active-rules *Function*

Description: Equivalent of `active-queries`, Page 131 for rules.

active-cheap-queries *Function*

Description: Returns the list of all cheap active queries.

Syntax: `(active-cheap-queries)`

Arguments:

Values: A list of query IDs.

Remarks: Each of these queries satisfies `query-active-p`, Page 126 and `cheap-query-p`, Page 128. Note that also ready (prepared) queries can be cheap. Thus, this function returns a sublist of the queries returned by `cheap-queries`, Page 129.

See also: `query-active-p`, Page 126, `cheap-query-p`, Page 128, `cheap-queries`, Page 129

active-cheap-rules *Function*

Description: Equivalent of `active-cheap-queries`, Page 131 for rules.

active-expensive-queries *Function*

Description: Returns the list of all expensive active queries.

Syntax: `(active-expensive-queries)`

Arguments:

Values: A list of query IDs.

Remarks: Each of these queries satisfies `active-expensive-query-p`, Page 128. Note that only active queries can be expensive.

See also: `query-active-p`, Page 126, `active-expensive-query-p`, Page 128

active-expensive-rules *Function*

Description: Equivalent of `active-expensive-queries`, Page 132 for rules.

running-queries *Function*

Description: Returns the list of all running queries.

Syntax: `(running-queries)`

Arguments:

Values: A list of query IDs.

Remarks: Each of these queries satisfies `query-active-p`, Page 126 and does NOT satisfy `query-waiting-p`, Page 126. Note that this is a sublist of the queries returned by `active-queries`, Page 131. You can get the running queries with `waiting-queries`, Page 134.

See also: `query-active-p`, Page 126, `query-waiting-p`, Page 126

running-rules *Function*

Description: Equivalent of `running-queries`, Page 132 for rules.

running-cheap-queries *Function*

Description: Returns the list of all cheap running queries.

Syntax: `(running-cheap-queries)`

Arguments:

Values: A list of query IDs.

Remarks: Only the `running-queries`, Page 132 are returned which satisfy `cheap-query-p`, Page 128. Note that no sleeping queries are returned!

See also: `running-queries`, Page 132, `cheap-query-p`, Page 128

running-cheap-rules *Function*

Description: Equivalent of `running-cheap-queries`, Page 133 for rules.

running-expensive-queries *Function*

Description: Returns the list of all expensive running queries.

Syntax: `(running-expensive-queries)`

Arguments:

Values: A list of query IDs.

Remarks: Only the `running-queries`, Page 132 are returned which satisfy `cheap-query-p`, Page 128. Note that no sleeping queries are returned!

See also: `running-queries`, Page 132, `active-expensive-query-p`, Page 128

running-expensive-rules *Function*

Description: Equivalent of `running-expensive-queries`, Page 133 for rules.

waiting-queries *Function*

Description: Returns the list of all waiting (sleeping) queries.

Syntax: `(waiting-queries)`

Arguments:

Values: A list of query IDs.

Remarks: Each of these queries satisfies `query-waiting-p`, Page 126 (and thus `query-active-p`, Page 126). Note that this is a sublist of the queries returned by `active-queries`, Page 131. You can get the running queries with `running-queries`, Page 132.

See also: `query-waiting-p`, Page 126

waiting-rules *Function*

Description: Equivalent of `waiting-queries`, Page 134 for rules.

waiting-cheap-queries *Function*

Description: Returns the list of all cheap waiting queries.

Syntax: `(waiting-cheap-queries)`

Arguments:

Values: A list of query IDs.

Remarks: Each of these queries satisfies `query-waiting-p`, Page 126 and `cheap-query-p`, Page 128. Note that this is a sublist of the queries returned by `waiting-queries`, Page 134, and of the list of queries returned by `cheap-queries`, Page 129.

See also: `query-waiting-p`, Page 126, `cheap-query-p`, Page 128

waiting-cheap-rules *Function*

Description: Equivalent of `waiting-cheap-queries`, Page 134 for rules.

waiting-expensive-queries *Function*

Description: Returns the list of all expensive waiting queries.

Syntax: `(waiting-expensive-queries)`

Arguments:

Values: A list of query IDs.

Remarks: Each of these queries satisfies `query-waiting-p`, Page 126 and `active-expensive-query-p`, Page 128. Note that this is a sublist of the queries returned by `waiting-queries`, Page 134, and of the list of queries returned by `active-expensive-queries`, Page 132.

See also: `query-waiting-p`, Page 126, `active-expensive-query-p`, Page 128

waiting-expensive-rules *Function*

Description: Equivalent of `waiting-expensive-queries`, Page 135 for rules.

processed-queries *Function*

Description: Returns the list of queries which have been processed (are terminated).

Syntax: `(processed-queries)`

Arguments:

Values: A list of query IDs.

Remarks: Each of these queries satisfies `query-processed-p`, Page 127. Each query whose query answering thread has died (for whatever reason) is put on this list. Note that this must not be the end of the life cycle of a query - queries can be reprepared and reexecuted, `reprepare-query`, Page 141, `reexecute-query`, Page 142.

See also: `query-processed-p`, Page 127

processed-rules *Function*

Description: Equivalent of `processed-queries`, Page 135 for rules.

inactive-queries *Function*

Description: Equivalent to `processed-queries`, Page 135

inactive-rules *Function*

Description: Equivalent of `inactive-queries`, Page 135 for rules.

terminated-queries *Function*

Description: Equivalent to `processed-queries`, Page 135

terminated-rules *Function*

Description: Equivalent of `terminated-queries`, Page 135 for rules.

6.4 Execution Control

wait-for-queries-to-terminate *Function*

Description: This function is useful if a bunch of queries had been started in parallel, but the application program wants to block the execution until all queries have been answered (processed). Thus, this function does not return until all queries have terminated.

Syntax: `(wait-for-queries-to-terminate)`

Arguments:

Values: `:OKAY` or `:DENIED-DUE-TO-DEADLOCK-PREVENTION`

Remarks: Note that, if queries have been started in lazy incremental mode, they will not terminate automatically. Thus, in order to prevent a deadlock, nRQL will not allow you to call this function if such a query is found on the list of `active-queries`, Page 131. You will get a warning such as

```
*** NRQL WARNING: DENIED DUE TO DEADLOCK PREVENTION! THE
FOLLOWING QUERIES WILL NOT TERMINATE AUTOMATICALLY, SINCE THEY
HAVE BEEN STARTED IN LAZY INCREMENTAL MODE: (QUERY-1).
```

on STDOUT as well as the return value

`:DENIED-DUE-TO-DEADLOCK-PREVENTION`.

See also: `active-queries`, Page 131, `execute-query`, Page 139, `abort-query`, Page 138

wait-for-rules-to-terminate *Function*

Description: Analog to `wait-for-queries-to-terminate`, Page 137, but for rules.

abort-query *Function*

Description: Aborts (terminates) the query *id*; thus, the query answering thread is killed.

Syntax: `(abort-query` *id*`)`

Arguments: *id* - the ID of the query, or `:last`.

Values: `:OKAY-QUERY-ABORTED` or `:NOT-FOUND`

Remarks: Note that you can only abort queries which satisfy `query-active-p`, Page 126 - `:NOT-FOUND` will be returned instead. A query which has been aborted is put on the list of `processed-queries`, Page 135.

See also: `abort-all-queries`, Page 138, `processed-queries`, Page 135

abort-rule *Function*

Description: Equivalent of `abort-query`, Page 138 for rules.

abort-all-queries *Function*

Description: Simply maps `abort-query`, Page 138 over `active-queries`, Page 131.

Syntax: `(abort-all-queries)`

Arguments:

Values: `:OKAY-ALL-QUERIES-ABORTED`

See also: `active-queries`, Page 131, `abort-query`, Page 138

abort-all-rules *Function*

Description: Equivalent of `abort-all-queries`, Page 138 for rules.

execute-query *Function*

Description: Sets up and starts a query answering thread for the query *id*. The query has to be ready before it can be executed, see `query-ready-p`, Page 125.

Syntax: `(execute-query` *id*`)`

Arguments: *id* - the ID of the query, or `:last`.

Values: Either `:NOT-FOUND` (in this case the query was not found on `ready-queries`, Page 130), or, if nRQL is in *set at a time mode*, then the answer to this query is returned. Otherwise, if nRQL is in *tuple at a time mode,* then you will get an answers such as `(:QUERY-32 :RUNNING)`, describing the current status of the query. Other possible return values are `(:QUERY-32 :DENIED-DUE-TO-DEADLOCK-PREVENTION)`, or `(:QUERY-32 :ACQUIRE-PROCESS-FAILED-POOL-SIZE-EXCEEDED)`.

Remarks: The query *id* is also put on the list of `active-queries`, Page 131. To put queries on `ready-queries`, Page 130, use `prepare-abox-query`, Page 146 and related functions.

Examples:

```
> (process-tuple-at-a-time)

:OKAY-PROCESSING-TUPLE-AT-A-TIME

> (prepare-abox-query (?x) (?x woman))

(:QUERY-32 :READY-TO-RUN)

> (execute-query :last)

(:QUERY-32 :RUNNING)
```

See also: `ready-queries`, Page 130, `abort-query`, Page 138

execute-rule *Function*

Description: Equivalent of `execute-query`, Page 139 for rules. Note that rules can be used in set at a time as well as in tuple at a time mode. In set at a time mode, unlike `execute-query`, Page 139 which returns a query answer, `execute-rule` returns a list of lists of ABox assertions (statements) which are added to (executed on) the ABox.

execute-all-queries *Function*

Description: Simply maps `execute-query`, Page 139 over `ready-queries`, Page 130.

Syntax: `(execute-all-queries)`

Arguments:

Values: The list containing the values returned by `execute-query`, Page 139 for the individual queries on `ready-queries`, Page 130.

Examples:

```
> (process-set-at-a-time)
:OKAY-PROCESSING-SET-AT-A-TIME
> (prepare-abox-query (?x) (?x man))
(:QUERY-25 :READY-TO-RUN)
> (prepare-abox-query (?x) (?x woman))
(:QUERY-26 :READY-TO-RUN)
> (execute-all-queries)
((((?X ALICE))) (((?X JAMES))))
```

See also: `ready-queries`, Page 130, `execute-query`, Page 139

execute-all-rules *Function*

Description: Equivalent of `execute-all-queries`, Page 140 for rules.

run-all-queries *Function*

Description: Equivalent to `execute-all-queries`, Page 140

run-all-rules *Function*

Description: Equivalent of `run-all-queries`, Page 140 for rules.

reexecute-all-queries *Function*

Description: Simply maps `reexecute-query`, Page 142 over `processed-queries`, Page 135.

Syntax: `(reexecute-all-queries)`

Arguments:

Values: See `execute-all-queries`, Page 140.

Remarks: Note that only the queries on `processed-queries`, Page 135 will be reexecuted.

See also: `reexecute-all-rules`, Page 141

reexecute-all-rules *Function*

Description: Equivalent of `reexecute-all-queries`, Page 141 for rules.

reprepare-query *Function*

Description: Puts a query which has already been processed, thus being on the list of `processed-queries`, Page 135, back onto the list of `ready-queries`, Page 130. Thus, the query can be executed again, see `execute-query`, Page 139. This is especially useful for rules, see `reprepare-rule`, Page 142.

Syntax: `(reprepare-query` *id*`)`

Arguments: *id* - the ID of the query, or `:last`.

Values: A tuple like `(:QUERY-32 :READY-TO-RUN)`, describing the current status of the query (`prepare-abox-query`, Page 146), or `:NOT-FOUND` in case the query was not found on the list of `processed-queries`, Page 135.

Remarks: The query *id* is again put on the list of `ready-queries`, Page 130. Instead of using `repreapare-query` and `execute-query`, Page 139, you can also use the shortcut `reexecute-query`, Page 142.

See also: `reprepare-rule`, Page 142

reprepare-rule *Function*

Description: Equivalent of `reprepare-query`, Page 141 for rules. This is how you can "fire" (apply) a rule more than once! See also `execute-rule`, Page 139.

reexecute-query *Function*

Description: First applies `reprepare-query`, Page 141 to a query and than calls `execute-query`, Page 139 on that query.

Syntax: `(reexecute-query` *id*`)`

Arguments: *id* - the ID of the query, or `:last`.

Values: See `execute-query`, Page 139.

See also: `reexecute-rule`, Page 142

reexecute-rule *Function*

Description: Equivalent of `reexecute-query`, Page 142 for rules.

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`.

Values: `T`, `NIL` or `:NOT-FOUND`

Remarks: Note that a rule can only be applicable if it is found on (`ready-rules`, Page 130) or `processed-rules`, Page 135. Rules which are already on the list of `active-rules`, Page 131 are not applicable.

As a side effect, if a rule on `processed-rules`, Page 135 is recognized as applicable, then it is also reprepared, see `reprepare-rule`, Page 142 and thus put back onto `ready-rules`, Page 130. It can then be fired again, see `execute-rule`, Page 139.

See also: `execute-applicable-rules`, Page 143

applicable-rules *Function*

Description: Returns all `ready-rules`, Page 130 and `processed-rules`, Page 135 that satisfy `rule-applicable-p`, Page 143 and are thus "ready to fire". Simply maps `rule-applicable-p`, Page 143 over `ready-rules`, Page 130 and `processed-rules`, Page 135.

Syntax: `(applicable-rules)`

Arguments:

Values: A list of IDs of applicable rules.

Remarks: As a side effect of checking rule applicability with `rule-applicable-p`, Page 143, the applicable rules on `processed-rules`, Page 135 are put back onto the list of `ready-rules`, Page 130.

See also: `rule-applicable-p`, Page 143, `unapplicable-rules`, Page 143

unapplicable-rules *Function*

Description: Returns all rules from `all-rules`, Page 120 which DO NOT satisfy `rule-applicable-p`, Page 143, see also `applicable-rules`, Page 143.

execute-applicable-rules *Function*

Description: Simply maps `execute-rule`, Page 139 over `applicable-rules`, Page 143.

Syntax: `(execute-applicable-rules)`

Arguments:

Values: A list containing the values returned by `execute-rule`, Page 139 for the individual rules on `applicable-rules`, Page 143.

See also: `rule-applicable-p`, Page 143

6.5 ABox Queries

retrieve *Macro*

Description: Prepares and starts a nRQL ABox query.

Syntax: `(retrieve` *head body* `&key (`*abox* `(current-abox)) `*id* `...)`

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.

abox(∗) - the ABox to be queried - optional keyword argument. Default value is the `(current-abox)`.

id (*) - the ID of the query - optional keyword argument. In case a query with the given *id* already exists, an error is raised. If not specified, nRQL will create a query ID such as `:QUERY-2`.

... - see `with-nrql-settings`, Page 178 for even more arguments!

Values: In set at a time mode: The answer to this query – a list of tuples, or `NIL` or `T`, or `:INCONSISTENT` (see `report-inconsistent-queries`, Page 179).

In tuple at a time mode: A tuple like `(:QUERY-466 :RUNNING)`, where `:QUERY-466` is the ID of the query used for referencing the query, `:RUNNING` indicating that the query answering thread has been started. You might also get `(:QUERY-466 :DENIED-DUE-TO-DEADLOCK-PREVENTION)`, or `(:QUERY-466 :ACQUIRE-PROCESS-FAILED-POOL-SIZE-EXCEEDED)`.

Remarks: Conceptually, `retrieve` first calls `prepare-abox-query`, Page 146, and then `execute-query`, Page 139. Thus, the result of `execute-query`, Page 139 is returned. However, in case the query is not executed (for example, if it has been recognized as inconsistent), then the result of `prepare-abox-query`, Page 146 will be returned.

Examples:

```
(retrieve (?x) (and (?x woman) (?x ?y has-child)))

(retrieve (?x) (and (?x woman) (?x ?y has-child)) :abox
smith-family :how-many 2)
```

See also: `racer-answer-query`, Page 145, `with-nrql-settings`, Page 178

racer-answer-query *Function*

Description: Functional equivalent of `retrieve`, Page 144.

retrieve-under-premise *Macro*

Description: Like `retrieve`, Page 144, but a query premise is added to the queried ABox prior to query execution.

Syntax: `(retrieve-under-premise` *premise head body* `&key ...)`

Arguments: *premise* (*) - the premise of the query, see `<query-premise>`, Section 6.1.8 in the User Guide. This is simply a list of ordinary RacerPro ABox assertions.

head (*), *body* (*) - see `retrieve`, Page 144.

... - see `retrieve`, Page 144.

Values: See `retrieve`, Page 144.

Remarks: The premise is only added temporarily to the ABox. The ABox will only temporarily be modified. However, the ABox must be changed for that, and will thus be exclusively locked for the time of execution of this query. Other queries cannot access the ABox until the query is processed and the lock released.

Examples:
```
(retrieve-under-premise ( (instance betty mother) (related
betty doris has-child) ) (?x) (and (?x mother) (?x ?y
has-child)))
```

See also: `racer-answer-query-under-premise`, Page 145

racer-answer-query-under-premise *Function*

Description: Functional equivalent of `retrieve-under-premise`, Page 145.

prepare-abox-query *Macro*

Description: Prepares but does not start a nRQL ABox query.

Syntax: `(prepare-abox-query ...)`

Arguments: See `retrieve`, Page 144.

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 139.

Examples: `(prepare-abox-query (?x) (and (?x woman) (?x ?y has-child)))`

See also: `racer-prepare-query`, Page 146

racer-prepare-query *Function*

Description: Functional equivalent of `prepare-abox-query`, Page 146.

6.6 TBox Queries

tbox-retrieve *Macro*

Description: Prepares and starts a nRQL TBox query.

Syntax: `(tbox-retrieve` *head body* `&key (`*tbox* `(current-tbox))` *id* `...)`

Arguments: *head* (*) - the head of the query, see `<query-head>`, Section 6.1.8 in the User Guide, and `retrieve`, Page 144.

Projection operators are not meaningful.

body (*) - the body of the query, see `<query-body>`, Section 6.1.8 in the User Guide, and `retrieve`, Page 144.

Constraint query atoms are not meaningful. Only the concept names from *tbox* as well as the roles `has-child, has-parent, has-ancestor, has-descendant` are meaningful.

tbox (*) - the TBox to be queried - optional keyword argument. Default value is the `(current-abox)`.

id (*) - see `retrieve`, Page 144.

... - see also `with-nrql-settings`, Page 178 for even more arguments.

Values: See `retrieve`, Page 144.

Remarks: Conceptually, `tbox-retrieve` first calls `prepare-tbox-query`, Page 148, and then `execute-query`, Page 139.

Examples:
```
(tbox-retrieve (?x ?y) (and (top ?x) (?x ?y has-child)))
```

```
(tbox-retrieve (?x ?y) (and (?x woman) (?x ?y has-descendant))
:tbox family-1)
```

See also: `racer-answer-tbox-query`, Page 147

racer-answer-tbox-query *Function*

Description: Functional equivalent of `tbox-retrieve`, Page 147.

prepare-tbox-query *Macro*

Description: Prepares but does not start a nRQL TBox query.

Syntax: `(prepare-tbox-query ...)`

Arguments: See `tbox-retrieve`, Page 147.

Values: A tuple like `(:QUERY-466 :READY-TO-RUN)`, where `:QUERY-466` is the identifier used for referencing the query, `:READY-TO-RUN` indicating the current status of the query.

Remarks: To start the query, use `execute-query`, Page 139.

Examples: `(prepare-tbox-query (?x) (and (?x woman) (?x ?y has-child)))`

See also: `racer-prepare-tbox-query`, Page 148

racer-prepare-tbox-query *Function*

Description: Functional equivalent of `prepare-tbox-query`, Page 148.

6.7 Getting Answers

next-tuple-available-p *Function*

Description: Checks for the availability of yet another tuple from query *id*.

Syntax: `(next-tuple-available-p` *id*`)`

Arguments: *id* - the ID of the query, or `:last`.

Values: `T`, `NIL`, or `:NOT-FOUND`.

Remarks: If this function returns `T`, then the next tuple of this query can be retrieved without further delay using `get-next-tuple`, Page 150. The tuple is already available. This function is useful if query *id* is running in eager tuple at a time mode, and the client wants to know whether `get-next-tuple`, Page 150 can be called without blocking the nRQL API.

See also: `get-next-tuple`, Page 150

next-set-of-rule-consequences-available-p *Function*

Description: Equivalent of `next-tuple-available-p`, Page 149, but for rules.

get-next-tuple *Function*

Description: Gets the next tuples from query *id*. The query must be on the list of `active-queries`, Page 131 or `processed-queries`, Page 135.

Syntax: `(get-next-tuple` *id*`)`

Arguments: *id* - the ID of the query, or `:last`.

Values: The tuple, or `:INCONSISTENT` (see `report-inconsistent-queries`, Page 179), or `:WARNING-KB-HAS-CHANGED` if the referenced KB has been changed in the meantime, or `:EXHAUSTED` in case there are no more tuples left, or `:WARNING-EXPENSIVE-PHASE-TWO-STARTS` in case the query has been started in two-phase query processing mode and phase one is over, or `:NOT-FOUND` in case the query is not on the list of active or terminated queries.

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.

If the query had been started in eager mode, then there is a chance that the next tuple (and probably some future tuples) have already been computed, and are thus already available. See `next-tuple-available-p`, Page 149 to check for the availability of such tuples.

Note that, even if the query thread has already terminated and thus the query is found on the list of `processed-queries`, Page 135, still there might be still some tuples available which have not been requested by the user yet. This happens in the eager tuple at a time mode.

See also: `next-tuple-available-p`, Page 149

get-current-tuple *Function*

Description: Returns the result of the previous call to `(get-next-tuple` *id*`)`, see `get-next-tuple`, Page 150.

Syntax: `(get-current-tuple` *id*`)`

Arguments: *id* - the ID of the query, or `:last`.

Values: See `get-next-tuple`, Page 150. Moreover, `NIL` is returned if there was no previous call `(get-next-tuple` *id*`)`, and `:NOT-FOUND` in case the query is not on the list of active or processed queries, as usual.

See also: `get-current-set-of-rule-consequences`, Page 151

get-next-set-of-rule-consequences *Function*

Description: If the rule named *id* has been started in (incremental) tuple at a time query processing mode, then this function gets you the "next" set of rule consequences that this rule has produced. Note that only the LAZY incremental mode is available for rules! nRQL will automatically use the lazy mode if rules are fired in tuple at a time mode. Moreover, the rule named *id* must be on the list of active or processed (terminated) rules.

The set of rule consequences is a set of ABox assertions which has been derived from the rule consequence in which the variables have been replaced by their current bindings. Applications can look at this current set of rule consequences and decide whether to add these assertions to the ABox or not. See `choose-current-set-of-rule-consequences`, Page 163. This is how you can implement you own rule application strategy.

Syntax: `(get-next-set-of-rule-consequences` *id*`)`

Arguments: *id* - the ID of the rule, or `:last`.

Values: A set of ABox assertions (and statements), or `:INCONSISTENT` (see `report-inconsistent-queries`, Page 179), or `:WARNING-KB-HAS-CHANGED` if the referenced KB had been changed in the meantime, or `:EXHAUSTED` in case there are no more binding possibilities left, or `:WARNING-EXPENSIVE-PHASE-TWO-STARTS` in case the rule had been started in two-phase query processing mode and phase one is over, or `:NOT-FOUND` in case the rule is not on the list of active or terminated rules.

Remarks: This function is the equivalent of `get-next-tuple`, Page 150, but for rules. Please also refer to `get-next-tuple`, Page 150!

If the rule had been started in lazy incremental mode, then computation of the next set of rule consequences might eventually take some time.

If the rule had been started in eager mode, then there is a chance that the next set of ABox assertions (and probably some future sets) have already been computed, and are thus already available. See `next-set-of-rule-consequences-available-p`, Page 149 to check for the availability of such sets.

See also: `next-set-of-rule-consequences-available-p`, Page 149

get-current-set-of-rule-consequences *Function*

Description: Equivalent of `get-current-tuple`, Page 150 for rules.

get-next-n-remaining-tuples *Function*

Description: Gets the next n tuples from the query *id* (or all tuples if n is not specified).

Syntax: `(get-next-n-remaining-tuples` *id* `&optional` *n*`)`

Arguments: *id* - the ID of the query, or `:last`.

n - the number of requested tuples. Note that this parameter is optional. Default value is `NIL`. If $n =$ `NIL`, then *all tuples* are requested.

Values: A list of (maximal n) tuples, or `:NOT-FOUND`.

Remarks: This function repeatedly calls `get-next-tuple`, Page 150.

If nRQL is in two-phase query processing mode and delivery of the "phase two starts" warning token is enabled (see `enable-phase-two-starts-warning-tokens`, Page 167), then the `:WARNING-EXPENSIVE-PHASE-TWO-STARTS` token as delivered by `get-next-tuple`, Page 150 does *not* appear in the list of tuples returned by this function.

Instead, if `:WARNING-EXPENSIVE-PHASE-TWO-STARTS` is encountered, `get-next-n-remaining-tuples` stops requesting additional tuples with `get-next-tuple`, and returns the list of tuples accumulated so far immediately.

Now, in order to get the remaining tuples (the tuples of phase 2), simply call `get-next-n-remaining-tuples`, Page 152 again.

Note that this behavior can be changed either by not using the two-phase query processing mode at all, or by instructing nRQL not to deliver the `:WARNING-EXPENSIVE-PHASE-TWO-STARTS` token (see `enable-phase-two-starts-warning-tokens`, Page 167).

See also: `next-tuple-available-p`, Page 149, `get-next-tuple`, Page 150, `get-all-remaining-tuples`, Page 153

get-next-n-remaining-sets-of-rule-consequences *Function*

Description: Equivalent of `get-next-n-remaining-tuples`, Page 152 for rules.

get-all-remaining-tuples *Function*

Description: Similar to `get-next-n-remaining-tuples`, Page 152 if n = `NIL` is specified. However, the function really returns *all* tuples. If the `:WARNING-EXPENSIVE-PHASE-TWO-STARTS` token is encountered, then, unlike `get-next-n-remaining-tuples`, Page 152, it does not stop.

Syntax: `(get-all-remaining-tuple` *id*`)`

Arguments: *id* - the ID of the query, or `:last`.

Values: The list of *all remaining* tuples, or `:NOT-FOUND`.

See also: `get-next-n-remaining-tuples`, Page 152, `get-answer`, Page 154, `get-all-remaining-sets-of-rule-consequences`, Page 153

get-all-remaining-sets-of-rule-consequences *Function*

Description: The equivalent of `get-all-remaining-tuples`, Page 153 for rules.

get-answer *Function*

Description: Similar to `get-all-remaining-tuples`, Page 153. However, not only the *remaining tuples*, but *all* tuples are returned. Thus, this function can be called an arbitrary number of times on a query *id*, in contrast to `get-all-remaining-tuples`, Page 153, which returns `NIL` if it is called the 2nd time. This function can also be used on rules. In this case, the set of sets of rule consequences is returned.

Syntax: `(get-answer` *id*`)`

Arguments: *id* - the ID of the query or rule, or `:last`.

Values: A list of tuples, or `T` or `NIL`, or a list of list of ABox assertions (the list of rule consequences). See `retrieve`, Page 144, `firerule`, Page 160.

Remarks: Can be called an arbitrary number of times on a query (rule) *id*. The answer to a query is stored in the query object representing the query (rule) and is thus not recomputed.

Note that the query or rule named *id* must be on the list of active or processed queries (see `active-queries`, Page 131, `processed-queries`, Page 135).

See also: `get-all-answers`, Page 154

get-answer-size *Function*

Description: Counts the number of answer tuples that `get-answer`, Page 154 retrieves for the specified query or rule *id*. Thus, this function can be used to find out how many tuples a query has produced. If used on a rule, this function returns the number of sets of rule consequences which have been produced by firing this rule.

get-all-answers *Function*

Description: Simply maps `get-answer`, Page 154 over the list of `active-queries`, Page 131, `processed-queries`, Page 135, `active-rules`, Page 131, and `processed-rules`, Page 135.

Syntax: `(get-all-answers)`

Arguments:

Values: A list of tuples of the form `(<id> <answer>)`, where `<answer>` is the answer to query (or rule) `<id>`, see `get-answer`, Page 154.

See also: `get-answer`, Page 154, `processed-queries`, Page 135

query-accurate-p *Function*

Description: Determines whether a query is still accurate. A query is *accurate* iff the referenced ABox has not changed since the parsing of this query. Thus, if an answer set has been computed for the query, and the query is still accurate, then reexecuting this query will still produce the same answer. See `reexecute-query`, Page 142.

Syntax: `(query-accurate-p` *id*`)`

Arguments: *id* - the ID of the query, or `:last`.

Values: `T` or `NIL`.

See also: `rule-accurate-p`, Page 155

rule-accurate-p *Function*

Description: Equivalent of `query-accurate-p`, Page 154 for rules.

6.8 Defined Queries

defquery *Macro*

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 `substitute` query atoms. The definitions are local to *tbox*.

Syntax: `(defquery` *name head body* `&key (`*tbox* `(current-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 as head entries.

body (*) - the body of the query, see `<query-body>`, Section 6.1.8 in the User Guide.

tbox (*) - the TBox to which this definition is local.

Values: The *name* of the defined query.

Remarks: The query is neither answered nor prepared. Cyclic definitions are not possible, but *body* can reference other defined queries as well.

Examples:
```
(defquery is-a-mother (?x) (and (?x woman) (?x ?y has-child)))

(retrieve (?a) (substitute (is-a-mother ?a)))
```

or

```
(retrieve (?a) (?x is-a-mother))
```

See also: `define-query`, Page 158

undefquery *Macro*

Description: Deletes a local definition.

Syntax: `(undefquery` *name* `&key (`*tbox* `(current-tbox)))`

Arguments: *name* (*) - the name of the definition.

tbox (*) - the TBox to which this definition is local.

Values: The names of the remaining definitions (local to *tbox*).

Examples: `(undefquery mother)`

See also: `undefine-query`, Page 158

def-and-prep-query *Macro*

Description: Defines a query local to *tbox* and prepares it for execution.

Syntax: See `defquery`, Page 156.

Arguments: See `defquery`, Page 156 and `prepare-abox-query`, Page 146 for optional arguments.

Values: See `prepare-abox-query`, Page 146.

Remarks: Conceptually, first the defined query is created (`defquery`, Page 156), and then this defined query is prepared (`prepare-abox-query`, Page 146).

Examples: `(def-and-prep-query is-a-mother (?x) (and (?x woman) (?x ?y has-child)) :tbox family)`

See also: `define-and-prepare-query`, Page 158

def-and-exec-query *Macro*

Description: Defines, prepares and executes a query.

Syntax: See `defquery`, Page 156.

Arguments: See `defquery`, Page 156, and `prepare-abox-query`, Page 146 and `execute-query`, Page 139 for optional arguments.

Values: See `retrieve`, Page 144, `execute-query`, Page 139.

Remarks: Conceptually, the query is defined and prepared (`def-and-prep-query`, Page 157), and then executed (`execute-query`, Page 139).

Examples:
```
(def-and-exec-query is-a-mother (?x) (and (?x woman) (?x ?y
has-child)) :tbox family)
```

See also: `define-and-execute-query`, Page 158

define-query *Function*

Description: Functional equivalent of `defquery`, Page 156.

undefine-query *Function*

Description: Functional equivalent of `undefquery`, Page 157

define-and-prepare-query *Function*

Description: Functional equivalent of `def-and-prep-query`, Page 157.

define-and-execute-query *Function*

Description: Functional equivalent of `def-and-exec-query`, Page 158.

describe-definition *Function*

Description: Describes the definition named *name* which is local to *tbox*.

Syntax: `(describe-definition` *name* `&key (`*tbox* `(current-tbox)))`

Arguments: *name* - the name of the definition.
tbox - the TBox to which this definition is local to.

Values: The definition named *name*.

Examples:
```
(describe-definition 'mother :tbox 'family)
```

See also: `describe-all-definitions`, Page 159

describe-all-definitions *Function*

Description: Describes all definitions which are local to *tbox*.

Syntax: `(describe-all-definitions &key (`*tbox* `(current-tbox)))`

Arguments: *tbox* - the TBox to which the definitions are local.

Values: All definitions local to *Tbox*.

Examples:
```
(describe-all-definitions)
(describe-all-definitions :tbox 'family)
```

See also: `describe-definition`, Page 159

delete-all-definitions *Function*

Description: Deletes all definitions local to *tbox*.

Syntax: `(delete-all-definitions &key (`*tbox* `(current-tbox)))`

Arguments: *tbox* - the TBox to which this definition is local.

Values: `:OKAY-ALL-DEFINITIONS-DELETED`

Examples:
```
(delete-all-definitions)
```

See also: `undefquery`, Page 157

6.9 Rules

firerule *Macro*

Description: Prepares a rule and applies (fires) it to *abox*. `firerule` is the equivalent of `retrieve`, Page 144 for rules.

Syntax: `(firerule` *antecedence consequence* `&key (`*abox* `(current-abox))` *premise* `...))`

Arguments: *antecedence* (*) - the antecedence of the rule, see `<rule-antecedence>`, Section 6.1.8 in the User Guide.

consequence (*) - the consequence of the rule. This is a set of generalized ABox assertions, see `<rule-consequence>`, Section 6.1.8 in the User Guide.

Note that you can also put in `forget` statements into *consequence*.

abox (*) - the ABox to which the rule shall be applied - an optional keyword argument whose default value is the `(current-abox)`.

premise (*) - the premise of the rule, see `<query-premise>`, Section 6.1.8 in the User Guide.

... - see also `with-nrql-settings`, Page 178 for even more arguments.

Values: In set at a time mode: The set of rule consequences (a set of ABox assertions and possibly `forget` statements) the rule has created.

In tuple at a time mode: A rule status description.

See also `retrieve`, Page 144.

Remarks: There are no "TBox rules".

Note that the produced ABox assertions may not be "new", i.e. the generated axioms are eventually already syntactically present in the ABox. However, due to the presence of NAF, non-monotonic rules can be written!

See also: `racer-apply-rule`, Page 161

apply-abox-rule *Macro*

Description: Same as `firerule`, Page 160.

racer-apply-rule *Function*

Description: Functional equivalent of `firerule`, Page 160.

prepare-abox-rule *Macro*

Description: Prepares but does not fire a nRQL rule. `prepare-abox-rule` is the equivalent of `prepare-abox-query`, Page 146, but for rules.

Syntax: `(prepare-abox-rule ...)`

Arguments: See `firerule`, Page 160.

Values: A tuple like `(:RULE-466 :READY-TO-RUN)`, where `:RULE-466` is the identifier used for referencing the rule, `:READY-TO-RUN` indicating the current status of the rule. See also `prepare-abox-query`, Page 146.

Remarks: To fire the rule, use `execute-rule`, Page 139.

See also: `racer-prepare-rule`, Page 161

preprule *Macro*

Description: Same as `prepare-abox-rule`, Page 161.

racer-prepare-rule *Function*

Description: Functional equivalent of `prepare-abox-rule`, Page 161.

add-rule-consequences-automatically *Function*

Description: Advises nRQL to add rule consequences produced (by rule firing) automatically to the ABox. Usually, you don't want this, but implement your own rule application strategy. See `choose-current-set-of-rule-consequences`, Page 163, `add-chosen-sets-of-rule-consequences`, Page 163.

Syntax: `(add-rule-consequences-automatically)`

Arguments:

Values: `:OKAY-ADDING-RULE-CONSEQUENCES-AUTOMATICALLY`

See also: `dont-add-rule-consequences-automatically`, Page 162

dont-add-rule-consequences-automatically *Function*

Description: Disables automatic addition of rule consequences, see `add-rule-consequences-automatically`, Page 162.

choose-current-set-of-rule-consequences *Function*

Description: Rule consequences of a rule are never added to an ABox as long as the rule that produces them is still running. The rule must terminate, only then can the computed set of rule consequences be added to the ABox. If rules are fired in the tuple at a time mode, then rule consequences are requested and computed lazily one after the other via `get-next-set-of-rule-consequences`, Page 151, in an incremental fashion. The current set of rule consequences, `get-current-set-of-rule-consequences`, Page 151, can be memorized by nRQL with a call to `choose-current-set-of-rule-consequences` such that this chosen current set of rule consequences can later be added to the ABox, after the rule has terminated.

If RacerPro is in `add-rule-consequences-automatically`, Page 162 mode, then the chosen sets of rule consequences will be added automatically. However, if RacerPro is in `dont-add-rule-consequences-automatically`, Page 162 mode, then the chosen sets of rule consequences will not be added automatically, but instead the application (user) has to call `add-chosen-sets-of-rule-consequences`, Page 163 explicitly.

Syntax: `(choose-current-set-of-rule-consequences` *id*`)`

Arguments: *id* - the ID of the rule, or `:last`.

Values: The chosen current set of rule consequences (if not `NIL`), or `:NOT-FOUND`.

Remarks: Note that *id* must be on the list of `active-rules`, Page 131 or `processed-rules`, Page 135.

See also: `firerule`, Page 160, `get-next-set-of-rule-consequences`, Page 151, `get-current-set-of-rule-consequences`, Page 151, `add-chosen-sets-of-rule-consequences`, Page 163.

add-chosen-sets-of-rule-consequences *Function*

Description: Adds the sets of rule consequence which have been produced by rule *id* and selected with calls to `choose-current-set-of-rule-consequences`, Page 163 to the ABox. Note that you can apply this function only once to a rule (and only to a rule for which rule consequences have been chosen).

Syntax: `(add-chosen-set-of-rule-consequences` *id*`)`

Arguments: *id* - the ID of the rule, or `:last`.

Values: The added ABox assertions, or `:NOT-FOUND`.

Remarks: Note that *id* must be on the list of `processed-rules`, Page 135.

See also: `firerule`, Page 160, `get-next-set-of-rule-consequences`, Page 151, `get-current-set-of-rule-consequences`, Page 151, `add-chosen-sets-of-rule-consequences`, Page 163.

6.10 Configuring the Querying Modes of nRQL

describe-query-processing-mode *Function*

Description: Returns a description of the current settings of the nRQL engine.

Syntax: `(describe-query-processing-mode)`

Arguments:

Values: A list of descriptive tokens and attribute-value pairs.

Examples:
```
> (describe-query-processing-mode)

((:CREATING-SUBSTRATES-OF-TYPE :RACER-DUMMY-SUBSTRATE)
:CHECK-ABOX-CONSISTENCY :QUERY-OPTIMIZATION-ENABLED
:OPTIMIZER-USES-CARDINALITY-HEURISTICS
:AUTOMATICALLY-ADDING-RULE-CONSEQUENCES :WARNINGS
:COMPLETE-MODE :MODE-3 :SET-AT-A-TIME-MODE
:DELIVER-KB-HAS-CHANGED-WARNING-TOKENS)
```

See also: `with-nrql-settings`, Page 178, `set-nrql-mode`, Page 165

describe-current-substrate *Function*

Description: Returns a description of the current substrate used by the nRQL engine.

Syntax: `(describe-current-substrate)`

Arguments:

Values: A list of attribute-value pairs.

Examples:
```
> (describe-current-substrate)

((:NAME SMITH-FAMILY) (:TYPE THEMATIC-SUBSTRATE::RACER-DUMMY-SUBSTRATE)
(:ASSOCIATED-ABOX SMITH-FAMILY) (:ASSOCIATED-TBOX FAMILY))
```

See also: `with-nrql-settings`, Page 178, `set-nrql-mode`, Page 165

set-nrql-mode *Function*

Description: Sets the level of completeness of nRQL query answering and determines whether set at a time or tuple at a time processing will be used. See Section 6.2.5 in the User Guide. The modes are:

Mode 0: Incomplete, told information reasoning, no exploited TBox information. No RacerPro ABox retrieval functions will be used.

Mode 1: Incomplete, told information reasoning, exploited

TBox information for atomic concept assertions in the ABox will be exploited. No RacerPro ABox retrieval functions will be used. TBox should be classified before using this mode.

Mode 2: Incomplete, told information reasoning, exploited TBox information for *all* (also complex) concept membership assertions in the ABox. No RacerPro ABox retrieval functions will be used.

Mode 3: Complete RacerPro + nRQL querying, Racer's ABox retrieval functions will be used. Can be expensive.

Mode 4: Complete RacerPro + nRQL querying, incremental tuple at a time, lazy, two-phase query processing mode. Tuples from phase one will be computed according to mode 1.

Mode 5: Like Mode 4, but tuples from phase one will be computed according to mode 2.

Mode 6: Like Mode 3, but internally, a two-phase tuple computation will be exploited. Compared to mode 3, this will probably result in a reduced number of calls to Racer's expensive ABox retrieval functions.

Syntax: `(set-nrql-mode` *mode*`)`

Arguments: *mode* - a cardinal number from 0 to 6.

Values: `:OKAY-MODE-`*mode*

See also: `with-nrql-settings`, Page 178, `describe-query-processing-mode`, Page 164

enable-query-optimization *Function*

Description: Enables the cost-based query optimizer.

Syntax: `(enable-query-optimization)`

Arguments:

Values: `:OKAY-QUERY-OPTIMIZATION-ENABLED`

See also: `disable-query-optimization`, Page 166, `enable-query-realization`, Page 182

disable-query-optimization *Function*

Description: Disables the cost-based query optimizer. See `enable-query-optimization`, Page 166.

optimizer-use-cardinality-heuristics *Function*

Description: Advises the optimizer to exploit statistical information about concept extension cardinalities from the ABox.

Syntax: `(optimizer-use-cardinality-heuristics)`

Arguments:

Values: `:OKAY-USING-CARDINALITY-HEURISTICS` or `:IGNORED-OPTIMIZER-IS-DISABLED`

Remarks: The optimizer must be enabled, see `enable-query-optimization`, Page 166.

See also: `optimizer-dont-use-cardinality-heuristics`, Page 166

optimizer-dont-use-cardinality-heuristics *Function*

Description: Advises the optimizer not to exploit cardinality heuristics. See `optimizer-use-cardinality-heuristics`, Page 166.

enable-phase-two-starts-warning-tokens *Function*

Description: Enables delivery of `:WARNING-EXPENSIVE-PHASE-TWO-STARTS` tokens in two-phase query processing modes.

Syntax: `(enable-phase-two-starts-warning-tokens)`

Arguments:

Values: `:IGNORED-NOT-IN-TWO-PHASE-PROCESSING-MODE` or `:OKAY-PHASE-TWO-WARNING-TOKENS-ENABLED`.

See also: `set-nrql-mode`, Page 165, `disable-phase-two-starts-warning-tokens`, Page 167

disable-phase-two-starts-warning-tokens *Function*

Description: Disables delivery of `:WARNING-EXPENSIVE-PHASE-TWO-STARTS` tokens, see `enable-phase-two-starts-warning-tokens`, Page 167

enable-kb-has-changed-warning-tokens *Function*

Description: Enables delivery of `:WARNING-KB-HAS-CHANGED` tokens in incremental query processing modes.

Syntax: `(enable-kb-has-changed-warning-tokens)`

Arguments:

Values: `:IGNORED-NOT-IN-TUPLE-AT-A-TIME-MODE` or `:OKAY-KB-HAS-CHANGED-WARNING-TOKENS-ENABLED`.

Remarks: This token is delivered if an ABox is changed while the query was still active. Thus, the answer might be incomplete (or wrong). For these reasons, the token is also included in the query answer.

See also: `disable-kb-has-changed-warning-tokens`, Page 167

disable-kb-has-changed-warning-tokens *Function*

Description: Disables delivery of `:WARNING-KB-HAS-CHANGED` tokens, see `enable-kb-has-changed-warning-tokens`, Page 167.

enable-eager-tuple-computation *Function*

Description: Configures nRQL to precompute tuples (when in tuple at a time mode), even if these tuples have not yet been requested via calls to `get-next-tuple`, Page 150 (or related functions) yet. A query which is started in eager mode will never appear on `waiting-queries`, Page 134.

Syntax: `(enable-eager-tuple-computation)`

Arguments:

Values: `:IGNORED-NOT-IN-TUPLE-AT-A-TIME-MODE` or `:OKAY-EAGER-MODE-ENABLED`.

Remarks: The complement mode is called lazy tuple at a time mode. Thus, there is no `disable-eager-tuple-computation`, only `enable-lazy-tuple-computation`, Page 168.

See also: `enable-lazy-tuple-computation`, Page 168

enable-lazy-tuple-computation *Function*

Description: Configures nRQL NOT to precompute tuples in tuple at time mode. Thus, the query answering process goes to sleep (see `query-waiting-p`, Page 126) until the next tuple is requested via `get-next-tuple`, Page 150.

Syntax: `(enable-lazy-tuple-computation)`

Arguments:

Values: `:IGNORED`, if not in tuple-at-a-time query processing mode, otherwise `:OKAY-LAZY-MODE-ENABLED`.

Remarks: The complement mode is called eager tuple-at-a-time mode. Thus, there is no `disable-lazy-tuple-computation`, only `enable-eager-tuple-computation`, Page 168.

See also: `enable-eager-tuple-computation`, Page 168

check-abox-consistency-before-querying *Function*

Description: Configures nRQL to always check the consistency of the ABox to be queried before querying starts. Querying an inconsistent ABox is not meaningful, but checking an ABox for consistency can be very expensive!

Syntax: `(check-abox-consistency-before-querying)`

Arguments:

Values: `:OKAY-CHECKING-ABOX-CONSISTENCY-BEFORE-QUERYING`

See also: `dont-check-abox-consistency-before-querying`, Page 169

dont-check-abox-consistency-before-querying *Function*

Description: Configures nRQL NOT to check the consistency of the ABox to be queried before querying starts. See also `check-abox-consistency-before-querying`, Page 169.

add-role-assertions-for-datatype-properties *Function*

Description: If an OWL file is read into RacerPro, then constraint query atoms referring OWL datatype properties can only be answered if some auxiliary ABox assertions are added to the ABox resulting from reading in that OWL file.

Syntax: `(add-role-assertions-for-datatype-properties)`

Arguments:

Values: `:OKAY-ADDING-ROLE-ASSERTIONS-FOR-DATATYPE-PROPERTIES`

Remarks: You must call `add-role-assertions-for-datatype-properties` before you pose the first nRQL query to the ABox (for that OWL file).

See also: `dont-add-role-assertions-for-datatype-properties`, Page 169

dont-add-role-assertions-for-datatype-properties *Function*

Description: Disables addition of auxiliary ABox assertions to ABoxes produced from OWL files, see `add-role-assertions-for-datatype-properties`, Page 169.

get-max-no-of-tuples-bound *Function*

Description: Gets the current maximal number of tuples bound. If this bound is non-`NIL`, then query answer sets can not contain more tuples than specified by this bound.

Syntax: `(get-max-no-of-tuples-bound)`

Arguments:

Values: The current bound (a cardinal), or `NIL` if no bound is active.

Remarks: Usually, you should not set a bound. Thus, `NIL` is the default value.

Examples: `(get-max-no-of-tuples-bound)`

See also: `set-max-no-of-tuples-bound`, Page 170

set-max-no-of-tuples-bound *Function*

Description: Sets the maximal number of tuples bound to n. Thus, query answers cannot contain more than n tuples. Pass `NIL` to set to unbounded. Note that this bound also affects the rules - thus, if set to n, nRQL will not produce more than n sets of rule consequences.

Syntax: `(set-max-no-of-tuples-bound &optional` n`)`

Arguments: n - the bound, a cardinal.

Values: The n.

Remarks: Use `NIL` to set to unbounded (reset the bound).

Examples: `(set-max-no-of-tuples-bound 1)`

`(set-max-no-of-tuples-bound)`

See also: `get-max-no-of-tuples-bound`, Page 170

get-process-pool-size *Function*

Description: The nRQL query processing engine maintains a *pool of threads (Lisp processes).* Instead of creating and starting a fresh thread for each new query, nRQL tries to acquire a thread from a pool of available threads (Lisp processes) in order to save some memory.

This function returns the current number of available (free) threads in the pool.

Syntax: `(get-process-pool-size)`

Arguments:

Values: The current number of available threads in the pool.

See also: `get-maximum-size-of-process-pool`, Page 171, `get-initial-size-of-process-pool`, Page 172

get-maximum-size-of-process-pool *Function*

Description: This function returns the maximum number n of threads (Lisp processes) which nRQL will acquire as entries for the pool. This means, there cannot be more than n concurrently running queries. If `NIL` is returned, then there is no bound on the number of threads which will be acquired.

If a query cannot acquire a free thread from that pool, then a new thread will be created unless the bound as specified by `get-maximum-size-of-process-pool` is reached. In this case the `:ACQUIRE-PROCESS-FAILED-POOL-SIZE-EXCEEDED` token is returned. Then you must increase the size of this pool via `set-maximum-size-of-process-pool`, Page 173.

Syntax: `(get-maximum-size-of-process-pool)`

Arguments:

Values: The maximum number of processes in the pool, or `NIL` in case there is no bound on the number of pool entries.

See also: `set-maximum-size-of-process-pool`, Page 173

get-initial-size-of-process-pool *Function*

Description: This function returns the *initial available* number of threads (Lisp processes) which nRQL has acquired as entries for the pool.

Syntax: `(get-initial-size-of-process-pool)`

Arguments:

Values: The initial number of entries in the pool.

See also: `set-initial-size-of-process-pool`, Page 172

set-initial-size-of-process-pool *Function*

Description: Sets the initial number of threads (Lisp processes) in the pool. The pool is also reinitialized.

Syntax: `(set-initial-size-of-process-pool` n`)`

Arguments: n - a cardinal, the number of initial processes in the process pool.

Values: The n.

Remarks: The pool will also be reinitialized; i.e., n fresh threads (Lisp processes) will be created.

See also: `get-initial-size-of-process-pool`, Page 172

set-maximum-size-of-process-pool *Function*

Description: Sets the maximum number of threads (Lisp processes) for the pool.

If a query cannot acquire a free thread from the pool, then a new thread will be created unless the bound as specified by `get-maximum-size-of-process-pool`, Page 171 is reached. In this case the `:ACQUIRE-PROCESS-FAILED-POOL-SIZE-EXCEEDED` token is returned. Then you must increase the size of the pool using this function.

If `NIL` is specified, then there is no bound on the number of threads.

Syntax: `(set-maximum-size-of-process-pool` *n*`)`

Arguments: *n* - a cardinal, or `NIL` (no bound).

Values: The *n*.

See also: `get-maximum-size-of-process-pool`, Page 171

process-set-at-a-time *Function*

Description: Switches nRQL into set at a time mode. This means, the answer to a query will be delivered in one big bunch (the answer set). Functions such as `retrieve`, Page 144 work synchronously then.

Syntax: `(process-set-at-a-time)`

Arguments:

Values: `:OKAY-PROCESSING-SET-AT-A-TIME`

Remarks: This is the default mode.

See also: `process-tuple-at-a-time`, Page 173, `with-nrql-settings`, Page 178, `set-nrql-mode`, Page 165

process-tuple-at-a-time *Function*

Description: Configures nRQL to deliver the answer set in an incremental *tuple after tuple mode*. Functions such as `retrieve`, Page 144 work asynchronously then.

Syntax: `(process-tuple-at-a-time)`

Arguments:

Values: `:OKAY-PROCESSING-TUPLE-AT-A-TIME`

Remarks: See `get-next-tuple`, Page 150 as well as related functions: `enable-lazy-tuple-computation`, Page 168, `enable-eager-tuple-computation`, Page 168, ...

See also: `process-set-at-a-time`, Page 173

exclude-permutations *Function*

Description: Configures nRQL to filter out permutations from the answer set. Thus, if the answer contains `((?x a) (?y b))`, then it will not also contain `((?x b) (?y a))`.

Syntax: `(exclude-permutations)`

Arguments:

Values: `:OKAY-EXCLUDING-PERMUTATIONS`

Remarks: Filtering out permutations slows down the nRQL engine and consumes some memory!

See also: `include-permutations`, Page 174

include-permutations *Function*

Description: Disables filtering of permutations; see `exclude-permutations`, Page 174.

enable-abox-mirroring *Function*

Description: Instructs nRQL to mirror the asserted content of an ABox (the ABox assertions) into its internal data caches before querying starts.

Syntax: `(enable-abox-mirroring)`

Arguments:

Values: `:OKAY-ABOX-MIRRORING-ENABLED`

See also: `disable-abox-mirroring`, Page 174, `enable-smart-abox-mirroring`, Page 175, `enable-very-smart-abox-mirroring`, Page 175

disable-abox-mirroring *Function*

Description: Instructs nRQL to disable its ABox mirroring facility, see `enable-abox-mirroring`, Page 174.

enable-smart-abox-mirroring *Function*

Description: Enables ABox mirroring, see `enable-abox-mirroring`, Page 174, but in a smarter way: In case of a atomic concept assertion such as `(instance i C)`, so `C` is a concept name, not only `C` is added as told information for `i` to the ABox mirror, but also the set of concept synonyms and concept ancestors is computed and added to the mirror object for `i` as well. The same applies for `related` role membership assertions in the presence of role hierarchies, etc.

Syntax: `(enable-smart-abox-mirroring)`

Arguments:

Values: `:OKAY-SMART-ABOX-MIRRORING-ENABLED`

See also: `disable-abox-mirroring`, Page 174, `enable-abox-mirroring`, Page 174, `enable-very-smart-abox-mirroring`, Page 175

enable-very-smart-abox-mirroring *Function*

Description: Enables smart ABox mirroring (see `enable-smart-abox-mirroring`, Page 175, but in a smarter way: In this case, smart abox mirroring is also exploited for non-atomic concepts in concept assertions `(instance i C)`. Thus, also for non-atomic concepts `C` the set of concept synonyms and concept ancestors is computed and added to the mirror. The `related` axioms are mirrored as if `enable-smart-abox-mirroring`, Page 175 were used.

Syntax: `(enable-very.smart-abox-mirroring)`

Arguments:

Values: `:OKAY-VERY-SMART-ABOX-MIRRORING-ENABLED`

See also: `disable-abox-mirroring`, Page 174, `enable-abox-mirroring`, Page 174, `enable-smart-abox-mirroring`, Page 175

with-nrql-settings *Macro*

Description: Use this macro to change the settings of nRQL temporarily. This is the preferred way to alter the settings of nRQL, since neither the global state of RacerPro nor the global state of nRQL must be changed if you make your nRQL API calls within the scope of this macro. Using this macro, you can run different queries concurrently with different nRQL settings. Note that default values are indicated like this: (*mode* `3`); so 3 is the default for *mode*.

Syntax: `(with-nrql-settings (&key`

(*mode* `3`) (*warnings* `t`)

(*check-abox-consistency* `t`)

abox-mirroring

(*query-optimization* `t`) (*optimizer-use-cardinality-heuristics* `t`)

how-many-tuples *timeout*

(*add-rule-consequences-automatically* `t`)

phase-two-starts-warning-tokens (*kb-has-changed-warning-tokens* `t`)

report-inconsistent-queries *report-tautological-queries* *query-realization*

query-repository

exclude-permutations

(*abox* `(current-abox)`) (*tbox* `(current-tbox)`))

`&body` *body*)

Arguments: (*mode* `3`) -sets the nRQL query processing mode, see `set-nrql-mode`, Page 165. Default is 3.

abox-mirroring -enables ABox mirroring, see `enable-abox-mirroring`, Page 174. Value can be `NIL`, `:smart` or `:very-smart`. See `disable-abox-mirroring`, Page 174, `enable-smart-abox-mirroring`, Page 175, `enable-very-smart-abox-mirroring`, Page 175.

(*warnings* t) - see `enable-nrql-warnings`, Page 117

report-inconsistent-queries - see `report-inconsistent-queries`, Page 179

report-tautological-queries - see `report-tautological-queries`, Page 180

query-realization - see `enable-query-realization`, Page 182

(*add-rule-consequences-automatically* t) - see `add-rule-consequences-automatically`, Page 162

query-repository - see `enable-query-repository`, Page 183

(*query-optimization* `t`) - see `enable-query-optimization`, Page 166

(*optimizer-use-cardinality-heuristics* `t`) - see `optimizer-use-cardinality-heuristics`, Page 166

how-many-tuples - see `set-max-no-of-tuples-bound`, Page 170

timeout - a timeout, specified in milliseconds

phase-two-starts-warning-tokens - see `enable-phase-two-starts-warning-tokens`, Page 167

(*kb-has-changed-warning-tokens* t) - see `enable-kb-has-changed-warning-tokens`, Page 167

exclude-permutations - see `exclude-permutations`, Page 174

(*check-abox-consistency* t) - see `check-abox-consistency-before-querying`, Page 169. The default value is `t`, but only for the complete modes (3,4,5,6). The incomplete modes will use default value `NIL`.

(*abox* `(current-abox)`) - the ABox to be queried. Note that the `(current-abox)` of RacerPro will not be changed. However, the query definition mechanism of nRQL is aware of this change and correctly puts definitions which are made in the scope of the macro into the specified ABox.

(*tbox* `(current-tbox)`) - the TBox to be queried, for TBox queries

&body *body* - the body of the macro

Remarks: For most of these keyword arguments a corresponding pair of API functions called `enable-.../ disable-...` exists. See their documentations.

Examples:

```
(with-nrql-settings (:mode 1 :abox 'smith-family)
```

```
(retrieve (?x) (?x woman))
(describe-query-processing-mode))
```

See also: `describe-query-processing-mode`, Page 164, `set-nrql-mode`, Page 165

6.11 Query Inference

report-inconsistent-queries *Function*

Description: Advises nRQL to check newly prepared queries and rules automatically for consistency and produce a warning if an inconsistent query or rule is encountered. A call of `execute-query`, Page 139 on such a query will return `:inconsistent` .

Syntax: `(report-inconsistent-queries)`

Arguments:

Values: `:OKAY-REPORTING-INCONSISTENT-QUERIES`

Remarks: The consistency checker is incomplete. See Section 6.2.7 in the User Guide. For rules, also the consequence of the rule is taken into account.

See also: `report-tautological-queries`, Page 180, `dont-report-inconsistent-queries`, Page 179

dont-report-inconsistent-queries *Function*

Description: Advises nRQL no longer to report inconsistent queries, see `report-inconsistent-queries`, Page 179.

report-tautological-queries *Function*

Description: Advises nRQL to check newly prepared queries and rules automatically for being a tautology. If a tautological query or rule is encountered, a warning will be printed on STDOUT.

Syntax: `(report-tautological-queries)`

Arguments:

Values: `:OKAY-REPORTING-TAUTOLOGICAL-QUERIES`

Remarks: The tautology checker is currently very incomplete. See Section 6.2.7 in the User Guide for more info.

See also: `report-inconsistent-queries`, Page 179, `dont-report-tautological-queries`, Page 180

dont-report-tautological-queries *Function*

Description: Advises nRQL no longer to report tautological queries, see `report-tautological-queries`, Page 180.

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`.

Values: `T` or `NIL`.

Remarks: Note that only `NIL` answers can be trusted. The answer `T` does not mean consistent, but *unknown*.

See also: `query-tautological-p`, Page 180, `query-inconsistent-p`, Page 180, `query-entails-p`, Page 181

query-inconsistent-p *Function*

Description: See `query-consistent-p`, Page 180. Note that only `T` answers can be trusted. The answer `NIL` does not mean consistent, but *unknown*. See also `query-consistent-p`, Page 180, `query-tautological-p`, Page 180, `query-entails-p`, Page 181.

query-tautological-p *Function*

Description: See `query-consistent-p`, Page 180. Checks whether the query with specified ID is tautological. Note that also `T` can be trusted. The answer `NIL` does not mean that the query is tautological, but means *unknown*. See also `query-consistent-p`, Page 180, `query-inconsistent-p`, Page 180, `query-entails-p`, Page 181.

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 the first query, or `:last`.

id2 - the ID of the second query, or `:last`.

Values: `T` or `NIL`.

Remarks: Note that `T` can be trusted, and `NIL` means *unknown*. See Section 6.2.7 in the User Guide. We are working on more complete algorithms.

See also: `query-consistent-p`, Page 180, `query-inconsistent-p`, Page 180, `query-tautological-p`, Page 180, `query-equivalent-p`, Page 181

query-equivalent-p *Function*

Description: Checks whether query *id1* is equivalent to query *id2*. Simply checks whether `(query-entails-p` *id1 id2*`)` and `(query-entails-p` *id2 id1*`)` both return `T`, see `query-entails-p`, Page 181.

enable-query-realization *Function*

Description: Configures nRQL to automatically add logically implied conjuncts to newly prepared queries. The resulting query will be equivalent to the original one, but "more informed".

Syntax: `(enable-query-realization)`

Arguments:

Values: `:OKAY-QUERY-REALIZATION-ENABLED`

Remarks: This might be called a "semantic optimization" technique. See Section 6.2.9 in the User Guide. Adding logically implied conjuncts to a query enhances the degree of informdness of the query answering search process. This is still experimental, as the whole query reasoning API.

See also: `disable-query-realization`, Page 182

disable-query-realization *Function*

Description: Disables the addition of logically implied conjuncts to queries. See `enable-query-realization`, Page 182.

6.12 Query Repository

enable-query-repository *Function*

Description: Configures nRQL to use the Query Repository, also called the QBox. See Section 6.2.8 in the User Guide. Each new query is automatically classified into the current QBox. The stored answer sets of the queries in the QBox are used as caches to speed up query answer computations.

Syntax: `(enable-query-repository)`

Arguments:

Values: `:OKAY-QUERY-REPOSITORY-ENABLED`

Remarks: Automatically classifying each new query into the QBox is a potentially expensive operation. Thus, currently it may not pay off to use the QBox. We are working on more efficient algorithms.

See also: `disable-query-repository`, Page 183, `show-current-qbox`, Page 184

disable-query-repository *Function*

Description: Configures nRQL NOT to use the Query Repository, see `enable-query-repository`, Page 183.

show-qbox-for-abox *Function*

Description: Prints the DAG of the QBox for the given ABox as a tree.

Syntax: `(show-qbox-for-abox` *abox* `&optional` *show-definitions-p*`)`

Arguments: *abox* - the ABox whose QBox shall be printed.

show-definitions-p - if `T`, not only the query IDs will be printed, but also the bodies of the queries stored in the QBox.

Values: A graphical representation of the QBox for the ABox *abox* on STDOUT.

This function either returns `:see-output-on-stdout`, or `:NOT-FOUND`.

Remarks: This function only returns `:SEE-OUTPUT-ON-STDOUT` or `:NOT-FOUND` as a return value. However, the graphical representation of the QBox is printed to STDOUT. RacerPorter will display this output.

See also: `get-dag-of-qbox-for-abox`, Page 184, `show-current-qbox`, Page 184.

show-current-qbox *Function*

Description: Simply calls `show-qbox-for-abox`, Page 184 on `(current-abox)`.

get-dag-of-qbox-for-abox *Function*

Description: Returns the DAG of the QBox for the given ABox as a list of triples in the format "(<equivalent queries>, <query parents>, <query children>)".

Syntax: `(get-dag-of-qbox-for-abox` *abox*`)`

Arguments: *abox* - the ABox whose QBox shall be returned.

Values: The DAG as a list of triples, or `:NOT-FOUND`.

See also: `get-dag-of-current-qbox`, Page 184, `show-qbox-for-abox`, Page 184.

get-dag-of-current-qbox *Function*

Description: Simply calls `get-dag-of-qbox-for-abox`, Page 184 on `(current-abox)`.

get-abox-of-current-qbox *Function*

Description: Returns the ABox which is associated to the current QBox.

Syntax: `(get-abox-of-current-qbox)`

Arguments:

Values: The name of the ABox, or `:NOT-FOUND` in case there is not current QBox.

get-nodes-in-qbox-for-abox *Function*

Description: Returns the IDs of the queries (nodes) in the QBox for the specified ABox *abox*.

Syntax: `(get-nodes-in-qbox-for-qbox` *abox*`)`

Arguments: *abox* - the ABox specifying the QBox whose nodes (queries) shall be returned.

Values: A list of query IDs in this QBox, or `:NOT-FOUND` in case there is no such QBox.

get-nodes-in-current-qbox *Function*

Description: Simply calls `get-nodes-in-qbox-for-abox`, Page 185 on `(current-abox)`.

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: *id* - the ID of the query, or `:last`.

Values: A list of query IDs – the parents of the query *id*.

Remarks: Works only if query repository is enabled. Otherwise, the query *id* was not classified. See `enable-query-repository`, Page 183.

See also: `query-ancestors`, Page 186

query-children *Function*

Description: Returns the IDs of the child queries of the query *id* from the QBox. See Section 6.2.8 in the User Guide.

Syntax: `(query-children` *id*`)`

Arguments: *id* - the ID of the query, or `:last`.

Values: A list of query IDs – the children of the query *id*.

Remarks: Works only if query repository is enabled. Otherwise, the query *id* was not classified. See `enable-query-repository`, Page 183.

See also: `query-descendants`, Page 186

query-ancestors *Function*

Description: Returns the query ancestors. See `query-parents`, Page 185.

query-descendants *Function*

Description: Returns the query descendants. See `query-children`, Page 186.

query-equivalents *Function*

Description: Returns the IDs of the equivalent queries of the query *id* from the QBox. See Section 6.2.8 in the User Guide.

Syntax: `(query-equivalents` *id*`)`

Arguments: *id* - the ID of the query, or `:last`

Values: A list of query IDs – the queries which are equivalent to query *id*.

Remarks: Works only if query repository is enabled. Otherwise, the query *id* was not classified. See `enable-query-repository`, Page 183.

See also: `query-parents`, Page 185, `query-children`, Page 186

6.13 The Substrate Representation Layer

create-data-node *Function*

Description: Creates a data substrate node with appropriate name, label, and optionally also an associated ABox individual. See Section 6.1.7 in the User Guide.

Syntax: `(create-data-node` *name* `&key` *abox type-of-substrate*

descr

racer-descr`)`

Arguments: *abox* - the name of the associated ABox of the substrate in which the node is to be created.

type-of-substrate - the type of the substrate which is associated with the ABox *abox*.

descr - the label of the node. See `<data-substrate-label>`, Section 6.1.8 in the User Guide.

racer-descr - if supplied, a corresponding ABox individual is created in *abox*, and `(instance` *name racer-descr*`)` is asserted.

Values: The name of the node.

See also: `data-node`, Page 187

data-node *Macro*

Description: Corresponding macro for `create-data-node`, Page 187.

Syntax: `(data-node` *name* `(*) &optional (`*descr* `nil) (*)`

`(`*racer-descr* `nil) (*)`

abox `(*)` *type-of-substrate* `(*))`

Arguments: See `create-data-node`, Page 187.

Remarks: None of the arguments is evaluated.

See also: `create-data-node`, Page 187

delete-data-node *Function*

Description: Delete a data substrate node.

Syntax: `(delete-data-node` *name* `&key` *abox type-of-substrate*`)`

Arguments: *name* - the name of the node which shall be deleted.

abox - the name of the associated ABox of the substrate in which the node shall be deleted.

type-of-substrate - the type of the substrate which is associated with the ABox *abox*.

Values: `:OKAY-DELETED` or `:NOT-FOUND`

See also: `del-data-node`, Page 188

del-data-node *Macro*

Description: Corresponding macro for `delete-data-node`, Page 188.

Syntax: `(del-data-node` *name* `(*) &optional` *abox* `(*)` *type-of-substrate* `(*))`

Arguments: See `delete-data-node`, Page 188.

Remarks: None of the arguments is evaluated.

See also: `delete-data-node`, Page 188

create-data-edge *Function*

Description: Creates a labeled data substrate edge, and optionally also a corresponding role membership assertion in the ABox .

Syntax: `(create-data-edge` *from to* `&key` *abox type-of-substrate*

(*racer-descr* `nil`))

Arguments: *from*, *to* - the names of the two substrate nodes between which the edge is to be created.

abox - the name of the associated ABox of the substrate in which the edge is to be created.

type-of-substrate - the type of the substrate which is associated with the ABox *abox*.

descr - the label of the edge, if supplied.

racer-descr - if supplied, the ABox assertion `(related` *from to racer-descr*`)` is asserted to *abox*.

Values: The pair (*from to*).

See also: `data-edge`, Page 189

data-edge *Macro*

Description: Corresponding macro for `create-data-edge`, Page 189.

Syntax: `(data-edge` *from* `(*)` *to* `(*)` *descr* `(*) &optional (`*racer-descr* `nil) (*)` *abox* `(*)` *type-of-substrate* `(*))`

Arguments: See `create-data-edge`, Page 189.

Remarks: None of the arguments is evaluated.

See also: `create-data-edge`, Page 189

delete-data-edge *Function*

Description: Deletes a data substrate edge.

Syntax: `(delete-data-edge` *from to* `&key` *abox type-of-substrate*`)`

Arguments: *from*, *to* - the names of the nodes between which the edge shall be deleted.

abox - the name of the associated ABox of the substrate in which the node shall be deleted.

type-of-substrate - the type of the substrate which is associated with the ABox *abox*.

Values: `:OKAY-DELETED` or `:NOT-FOUND`

See also: `del-data-edge`, Page 190

del-data-edge *Macro*

Description: Corresponding macro for `delete-data-node`, Page 188.

Syntax: `(del-data-edge` *from* `(*)` *to* `(*) &optional` *abox* `(*)` *type-of-substrate* `(*))`

Arguments: See `delete-data-edge`, Page 190.

Remarks: None of the arguments is evaluated.

See also: `delete-data-node`, Page 188

get-data-node-label *Function*

Description: Gets the label of a data substrate node.

Syntax: `(get-data-node-label` *name* `&key` *abox* *type-of-substrate*`)`

Arguments: *name* - the name of the node

abox - the name of the associated ABox of the substrate

type-of-substrate - the type of the substrate which is associated with the ABox *abox*.

Values: The label of the node, or `:NOT-FOUND`

See also: `node-label`, Page 190

node-label *Macro*

Description: Corresponding macro for `get-data-node-label`, Page 190.

Syntax: `(node-label` *name* `(*) &optional` *abox* `(*)` *type-of-substrate* `(*))`

Arguments: See `get-data-node-label`, Page 190

Remarks: None of the arguments is evaluated.

See also: `get-data-node-label`, Page 190

get-data-edge-label *Function*

Description: Gets the label of a data substrate edge.

Syntax: `(get-data-edge-label` *from to* `&key` *abox type-of-substrate*`)`

Arguments: *from*, *to* - the names of the nodes of the edge

abox - the name of the associated ABox of the substrate

type-of-substrate - the type of the substrate which is associated with the ABox *abox*.

Values: The label of the edge, or `:NOT-FOUND`

See also: `edge-label`, Page 191

edge-label *Macro*

Description: Corresponding macro for `get-data-node-label`, Page 190.

Syntax: `(edge-label` *from* `(*)` *to* `(*) &optional` *abox* `(*)` *type-of-substrate* `(*))`

Arguments: See `get-data-edge-label`, Page 191

Remarks: None of the arguments is evaluated.

See also: `get-data-edge-label`, Page 191

in-data-box *Macro*

Description: Sets up a data substrate for an ABox.

Syntax: `(in-data-box` *abox* `(*))`

Arguments: *abox* (*) - the name of the associated ABox of the substrate

Values: The name of the substrate.

See also: `in-mirror-data-box`, Page 192, `in-rcc-box`, Page 193, `data-node`, Page 187, `data-edge`, Page 189, `del-data-node`, Page 188, `del-data-edge`, Page 190, `edge-label`, Page 191, `node-label`, Page 190

set-data-box *Function*

Description: Functional equivalent of `in-data-box`, Page 191.

in-mirror-data-box *Macro*

Description: Sets up a mirror data substrate for an ABox.

Syntax: `(in-mirror-data-box` *abox* `(*))`

Arguments: See `in-data-box`, Page 191

See also: `in-data-box`, Page 191, `enable-abox-mirroring`, Page 174, `enable-smart-abox-mirroring`, Page 175, `enable-very-smart-abox-mirroring`, Page 175

set-mirror-data-box *Function*

Description: Functional equivalent of `in-mirror-data-box`, Page 192.

enable-data-substrate-mirroring *Macro*

Description: Advises nRQL to create substrates of type `mirror-data-substrate` instead of substrates of type `racer-dummy-substrate` for Racer ABoxes. Additional retrieval facilities are then provided, e.g., for OWL files. Please refer to the User Guide.

Syntax: `(enable-data-substrate-mirroring)`

Arguments:

Remarks: If you want to exploit the additional retrieval facilities offered by the data substrate for OWL or Racer KBs, then you must call `enable-data-substrate-mirroring` before the first nRQL query is made.

See also: `disable-data-substrate-mirroring`, Page 192

disable-data-substrate-mirroring *Function*

Description: See `enable-data-substrate-mirroring`, Page 192.

in-rcc-box *Macro*

Description: Sets up a RCC data substrate for an ABox.

Syntax: `(in-rcc-box` *abox* `(*) &optional` *RCC-type* `(*))`

Arguments: *abox* (*) - the name of the associated ABox of the substrate
RCC-type (*) - must be `:RCC5` or `:RCC8`

See also: `in-data-box`, Page 191

set-rcc-box *Function*

Description: Functional equivalent of `in-rcc-box`, Page 193.

rcc-instance *Macro*

Description: Syntactic sugar - same as `data-node`, Page 187.

rcc-node *Macro*

Description: Syntactic sugar - same as `data-node`, Page 187.

create-rcc-node *Function*

Description: Syntactic sugar - same as `create-data-node`, Page 187.

rcc-related *Macro*

Description: Syntactic sugar - same as `data-edge`, Page 189.

rcc-edge *Macro*

Description: Syntactic sugar - same as `data-edge`, Page 189.

create-rcc-edge *Function*

Description: Syntactic sugar - same as `create-data-edge`, Page 189.

rcc-node-label *Macro*

Description: Syntactic sugar - same as `node-label`, Page 190.

rcc-edge-label *Macro*

Description: Syntactic sugar - same as `edge-label`, Page 191.

del-rcc-node *Macro*

Description: Syntactic sugar - same as `del-data-node`, Page 188.

del-rcc-edge *Macro*

Description: Syntactic sugar - same as `del-data-edge`, Page 190.

rcc-consistent-p *Function*

Description: Checks the consistency of an RCC network.

Syntax: `(rcc-consistent-p &optional` *abox type-of-substrate*`)`

Arguments: *abox* - the name of the associated ABox of the RCC substrate
type-of-substrate - the type of the substrate which is associated with the ABox *abox*

Values: `T` or `NIL`

rcc-consistent? *Macro*

Description: Corresponding macro for `rcc-consistent-p`, Page 194.

6.14 The nRQL Persistency Facility

store-substrate-for-abox *Function*

Description: Stores a binary dump of the specified substrate into a file.

Syntax: `(store-substrate-for-abox` *filename* `&optional (`*for-abox* `(current-abox))` *type-of-substrate*`)`

Arguments: *filename* - the name of the file.

for-abox - the name of the associated ABox of the substrate which shall be stored.

type-of-substrate - the type of the substrate in case there is more than one substrate associated to this ABox. Must be one of: `racer-dummy-substrate`, `data-substrate`, `mirror-data-substrate`, `rcc-substrate`.

Values: The name of the substrate which has been stored, or `:NOT-FOUND` in case nRQL cannot find a substrate with the specified name and/or type.

Remarks: Note that also the associated ABox, TBox, QBox, as well as defined queries are stored into the dump.

See also: `restore-substrate`, Page 195

restore-substrate *Function*

Description: Restores a substrate from the specified file. Note that `current-abox` is set to the restored ABox, as well as `current-tbox` to the associated TBox. An eventually restored QBox and the definitions of the restored substrate are made "current" as well.

Syntax: `(restore-substrate` *filename*`)`

Arguments: *filename* - the name of the file.

See also: `store-substrate-for-abox`, Page 195

store-substrate-for-current-abox *Function*

Description: Simply calls `store-substrate-for-abox`, Page 195 on the `(current-abox)`.

store-all-substrates *Function*

Description: Stores all available substrates into the specified file *filename*.

restore-all-substrates *Function*

Description: Restores all substrates form the specified file *filename*. Note that changes to the state of RacerPro and nRQL are made.

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
  &optional (ABN (current-abox))
  host port)
```

Arguments: *subscriber-name* - subscriber name

C - concept term

ABN - ABox name

host - ip number of the host to which results are to be sent as a string

port - port number (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*
`&optional (`*ABN* `(current-abox))`
`host port)`

Arguments: *subscriber-name* - subscriber name

C - concept term

ABN - ABox name

host - ip number of the host to which results are to be sent as a string

port - port number (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