

Scone User's Guide

**Scott E. Fahlman
Carnegie Mellon University
Language Technologies Institute &
Computer Science Department**

Contact: sef@cs.cmu.edu

**This document corresponds to Scone version 1.0.0,
released August, 2014.**

Copyright Notice:

This manual is copyright © 2003-2014 by Scott E. Fahlman.

Carnegie Mellon University holds the copyright to the core Scone software. The Scone software is made available to the public under the Apache 2.0 open source license. A copy of this license is distributed with the software. The license can also be found at <http://www.apache.org/licenses/LICENSE-2.0>.

Unless required by applicable law or agreed to in writing, software distributed under the License is distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied. See the License for the specific language governing permissions and limitations under the License.

Acknowledgments:

Development of Scone since February 2013 has been supported in part by the U.S. Office of Naval Research under award number N000141310224.

Development of Scone from January 2012 through August 2013 was supported in part by the Intelligence Advanced Research Projects Activity (IARPA) via Department of Defense US Army Research Laboratory contract number W911NF-12-C-0020.

Development of Scone from 2003 through 2008 was supported in part by the Defense Advanced Research Projects Agency (DARPA) under contract numbers NBCHD030010 and FA8750-07-D-0185.

Additional support for Scone development has been provided by research grants from Cisco Systems Inc. and from Google Inc.

The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright annotation thereon.

Disclaimer: The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of IARPA, DoD/ARL, ONR, DARPA, the U.S. Government, or any of our other sponsors.

Table of Contents

1. Overview.....	6
Note on Common Lisp Syntax.....	7
Running Scone.....	7
2. Loading, Saving, and Checkpointing KB Files.....	8
Functions and Variables.....	8
3. Scone Elements	12
3.1 Structure of Scone Elements.....	12
Functions and Variables.....	13
3.2 Element Properties.....	14
Functions.....	15
4. Referring to Scone Elements.....	16
4.1 Element Internal Names.....	16
Functions and Variables.....	18
4.2 Element External (English) Names.....	19
Functions and Variables.....	20
5. Markers.....	22
Functions and Variables.....	23
6. Contexts	26
Functions and Variables.....	26
7. Adding Elements to the Scone KB	27
7.1 Creating New Elements	27
The :context Keyword Argument	27
Internal Names and the :iname Argument	27
The :english Keyword Argument.....	28
Negation Links.....	28
7.2 Indv Nodes.....	29
Functions.....	29
7.3 Type Nodes	30
Functions.....	31
7.4 Map Nodes.....	32

Functions.....	32
7.5 IS-A Links.....	32
Functions.....	32
7.6 EQ Links	33
Functions.....	33
7.7 Has Links	33
Functions.....	33
7.8 Cancel Links	33
Functions.....	34
7.9 Relations	34
Functions.....	35
7.10 Statement Links	35
Functions.....	35
7.11 Split and Complete-Split Links.....	35
Functions.....	36
7.12 Functions to Create Multiple Nodes	36
Functions.....	37
7.13 Roles	37
8. Removing Elements from the Scone KB.	39
Functions and Variables.....	39
9. Queries and Predicates	40
9.1 Element-Type Predicate Functions	40
9.2 Predicates on the Is-A Hierarchy	41
Functions and Variables.....	41
9.3 Mark, List, and Show Functions	41
Functions and Variables.....	42
9.4 Query Functions for Roles and Relations	44
9.5 Miscellaneous Show Functions.....	46
10. Marker Scans.....	47
Functions and Variables.....	47
11. Miscellaneous Functions and Variables.....	49
11.1 Commentary, Printing, and Naming	49

11.2 Variables Linking to Essential Score Elements	49
12. Structure-Creation Functions from the Core KB	51
Functions and Variables.....	51
13. Alphabetical Index of Functions and Variables	52

1. Overview

Scone is a *knowledge-base system* (KBS) intended for use as a component in a wide range of software applications. The Scone system comprises a search/inference engine and a “core” *knowledge base* (KB) of common knowledge. Application-specific KBs and interfaces to other software applications can be built on top of this common base.

This manual is intended as a *programmer-level* guide to the operation and functions of the Scone software. It is not intended to be a tutorial on knowledge representation concepts used in Scone. A book-length tutorial document is needed, and it is currently under development. Some of the ideas in Scone are discussed in essay-length posts on Scott Fahlman’s “Knowledge Nuggets” blog: <http://www.cs.cmu.edu/~nuggets/>. Scone is currently under active development. We will include an updated version of this manual with each release, along with other release notes as appropriate.

The greatest problem for users of knowledge-base systems has been the difficulty of adding new knowledge to the system and making that knowledge fully effective. This gives rise to spotty coverage of common-sense domains. Scone attempts to ease this burden by relatively clean design, expressiveness of the representation system, and by insulating the user from most efficiency concerns. In addition, the Scone group has developed a number of tools for converting knowledge from other structured or semi-structured formats (existing ontologies such as SUMO and WordNet, tables, databases, and information mined from the Internet) to Scone format. Extraction of Scone knowledge from natural-language text, such as Wikipedia or newswire stories, is a long-term goal and is the subject of active research in our group.

Scone is written in Common Lisp.¹ The primary vehicle for Scone development is [Steel Bank Common Lisp](#) (SBCL), which is a derivative of [CMU Common Lisp](#) (CMUCL). SBCL is a free, high-performance implementation of Common Lisp running on Linux and some other operating systems. The SBCL system makes use of the 64-bit address space on Intel and AMD processors, making it possible to run very large knowledge bases on those machines.

Scone also has been run successfully on CMU Common Lisp and on several Windows-based Common Lisp systems, including CLISP, LispWorks, and Allegro Common Lisp. However, as of now, we do not actively test or maintain Scone in those environments.

Scone can be run as a server process on an Intel/Linux system, communicating with other software applications via character-stream I/O using the TCP/IP socket mechanism. The other software applications may be written in any computer language and may be running on the same machine or a different machine on the Internet.

Scone is designed for efficiency of search and inference on conventional high-end workstations. However, the marker-passing inference algorithms used in Scone were originally designed with parallelism in mind. Scone is therefore well suited for future implementation on a massively parallel machine or perhaps a computing grid.

¹ It would be possible to re-implement the Scone engine in some other language such as Java, perhaps with certain performance-critical code in C for efficiency. However, we believe that Common Lisp is by far the best language for the *development* of Scone, and (with careful coding) the Lisp version is nearly as efficient as it would be in C. The downside is that experienced Lisp programmers are becoming an endangered species.

Note on Common Lisp Syntax

This manual assumes that the reader has some general familiarity with Common Lisp² syntax. As a quick reminder, if this document indicates that a function's syntax looks like this

some-function (a b &optional c) [FUNCTION]

then **some-function** is the name of the function, arguments **a** and **b** are required, while argument **c** is optional, with some default value. Unless otherwise specified, the default default value is **nil** – Lisp's representation for both the empty list and "false".

If a function's syntax specification looks like this

some-function (a b &key :cat :dog) [FUNCTION]

then **a** and **b** are required arguments, while **:dog** and **:cat** are optional “keyword” arguments, which the caller can pass in any order or not at all. Some syntactically legal calls to this function would include the following

```
(some-function 27 'my-arg)
(some-function 27 'my-arg :cat "Felix" :dog "Fido")
(some-function 27 'my-arg :dog "Fido" :cat "Felix")
```

In Common Lisp, **'foo** represents a quoted constant value (in this case the symbol **foo**), and **"foo"** represents a string object. Symbols prefixed with the colon character are called *keywords*, and do not have to be quoted; these are used for named tokens or enumerations and for labeling the keyword/value pairs in a function call.

In Common Lisp, functions may return more than one value via the **values** form. This is useful when a function generates more than one useful value – for example, to return both a quotient and a remainder from a division function. If the function is called normally, return values beyond the first one are discarded. However, if the function is called under **multiple-value-bind** or similar functions, the additional return values are captured.

The curly-brace **{something}** syntax that Scone uses for element internal names is not standard Common Lisp syntax; it is a Scone-specific extension created using Lisp's character macro facility.

Running Scone

Instructions for running Scone will vary, depending on the operating system, details of the installation, the version of Common Lisp, and the version of Scone. Please see the "README" file in the Scone distribution for detailed instructions on loading and starting the version of Scone you are using.

In our typical setup, start Steel Bank Common Lisp, then type **(scone)** or **(scone "current")** to start the system, where the string **"current"** may be replaced with some other Scone version you want to run. In most cases you will then want to call **(load-kb "core")** to load the core knowledge base.

² For a good introduction to Common Lisp, we suggest [Practical Common Lisp](#) by Peter Seibel, available for free online or in hardcopy form at booksellers. For excellent examples of how to use Lisp in AI programming, including some knowledge representation examples, see [Paradigms of AI Programming](#) by Peter Norvig. The definitive online reference for Common Lisp is the [Common Lisp Hyperspec](#), produced by Kent M. Pitman for Lispworks Ltd., based on the ANSI X3J13 language standard.

2. Loading, Saving, and Checkpointing KB Files

The knowledge-base (or “KB”) files in Scone are just files of Common Lisp expressions and comments, following the syntax of Common Lisp. Users can easily modify these files offline, using the text editor of their choice. These files have the extension “.lisp”. Since these text-format KB files are just Common Lisp source files, any Common Lisp expressions you like may be included in these files, mixed with the Scone-specific expressions that create new elements.

Functions and Variables

Loading KB Files

load-kb (filename &key :verbose) [FUNCTION]

Loads the named file, with missing parts of the pathname filled in from the default in ***default-kb-pathname***. This is just a wrapper on Lisp's built-in **load** function, but **load-kb** does some internal bookkeeping so that the most recent loads can be rolled back if necessary, and so that the system knows it is in non-interactive mode during the load.

Often the KB file will effectively be a script that loads a list of other KB files. If the **:verbose** argument is **t**, the result of loading and executing each form in the file will be printed.

Example of usage: **(load-kb “project1/geopolitics” :verbose t)**

default-kb-pathname [VARIABLE]

A string. The default location for text-format KB files. This is initialized by the **scone** function, which starts some version of Scone.

loading-kb-file [VARIABLE]

If reading a KB file, this will be **t**. Certain interactive dialogues may be suppressed, etc.

loaded-files [VARIABLE]

A list of files loaded in the current KB. The head of the list is the file loaded most recently. Files go on this list when they have been loaded completely.

currently-loading-files [VARIABLE]

A list of the files currently being loaded. One file may load another, and so on, so there may be a number of files on this list. The first file on the list is the most recently started (i.e. the load that is most deeply nested), while the last file is generally the one for which the user called **load-kb**.

verbose-loading [VARIABLE]

If set, loading files will produce printout showing progress. Normally set by the **:verbose** keyword argument in **load-kb**.

disambiguate-policy *[VARIABLE]*
This global variable controls what happens when an ambiguous element name is encountered during input. **:error** says to signal an error. **:ask** says to query the user interactively.

Variables Controlling Inference and Checking During KB Modification

reloading-kb-file *[VARIABLE]*
Normally when loading forms into the KB, Scone will do extra work to see if the new knowledge is legal and consistent, and whether Scone must create derived elements implied by the new knowledge. If we are re-loading a KB file that was dumped from Scone, we set ***reloading-kb-file*** to **t**, indicating that all this checking has already been done and does not need to be repeated. This can greatly speed up the re-loading. Note that if other KB files have changed significantly since the dump occurred, it is best to load, rather than re-load, the dump – that is, we want to do the checking after all.

no-kb-error-checking *[VARIABLE]*
If **t**, suppress most error checking when creating KB elements. Use this only when loading a file that you've loaded successfully before. This speeds up loading, but not as much as ***reloading-kb-file***. The latter also skips looking for derived information, assuming that this is all present in the dump file.

check-defined-types *[VARIABLE]*
If **t** (the default), check whether newly defined elements fit the definition of some defined type, and, if so, make it an instance or subtype of that defined-type. If **nil**, don't bother.

create-undefined-elements *[VARIABLE]*
If **t** (the default), check whether newly defined elements fit the definition of some defined type. If **nil**, don't bother.

deduce-owner-type-from-role *[VARIABLE]*
If **nil**, a call to **x-is-the-y-of-z** (or a variant) signals an error if **Z** does not have or inherit a **Y**-role. If **t** (the default), try to create an **is-a** link from **Z** to the owner of role-node **Y**. If successful, **Y** is now inherited and we can go ahead and give it value **X**. If not, due to a type mismatch or some other error, return **NIL** and do not modify the KB.

comment-on-element-creation *[VARIABLE]*
If **t** (the default), whenever Scone creates inferred KB structures (nodes or link) not specifically requested by the user, it prints a comment about this via the commentary stream. This can become verbose if, for example, we are loading a file that defines types in some random order, rather than top-down, but it also catches lots of errors and typos.

comment-on-defined-types *[VARIABLE]*
If **t**, whenever Scone detects that some element satisfies the definition of a defined type and adds the element to that type, it prints a comment about this via the commentary stream. This is useful for debugging, but can be verbose when loading files that are known to be correct, so the default is **nil**.

Deferred Connections

Note: The deferred-connection mechanism has been largely superseded by the newer mechanism that creates an **{unknown thing}** element whenever the user makes reference to an element that has not yet been explicitly defined. The deferred-connection mechanism is now used only for bootstrapping of the root elements of the knowledge base, and we hope to remove it altogether in the near future.

Normally we build a Scone knowledge base in orderly layers. A new element will be connected to other elements that have already been created. But in some situations it is necessary to create forward references or circular references. In such cases, we use the “deferred connections” mechanism. This facility is enabled by placing **(setq *defer-unknown-connections* t)** at the start of a file that contains forward references. Normally this variable is **nil**, and references to unknown elements signal an error.

When ***defer-unknown-connections*** is **t** and an element-creation routine is told to connect a wire to some element, specified by its **iname**, and if no such element exists at present in the KB, we push an entry onto the ***deferred-connections*** list so that we can try to create the connection later, after additional elements have been created or loaded. This allows us to handle situations with forward or circular references, even if the circularities span multiple file boundaries.

defer-unknown-connections *[VARIABLE]*

If **t**, this enables the deferred-connection mechanism. This allows for references to elements that have not yet been defined, but that is not always a good thing: many such references are errors. So this is **nil** by default, and we only activate it for files that actually need this.

deferred-connections *[VARIABLE]*

This is the list of connections and element-names that we should try to set up later, after the named elements have been defined.

process-deferred-connections nil *[FUNCTION]*

The **process-deferred-connections** function says that we should scan the ***deferred-connections*** list now to see if any of the deferred connections can now be created. Any connections successfully processed are removed from the ***deferred-connections*** list; the rest remain on the list. This function is called after loading each new KB file, but users may occasionally want to call it directly.

Manual Checkpointing of the KB

When a KB is growing interactively, we probably want to checkpoint it from time to time so that knowledge won't be lost if the Scone process dies or is killed. One way to checkpoint is to save the entire Lisp core image, but that may not be practical because the core-image may be multiple gigabytes in size. The functions in this section provide an alternative.

checkpoint-kb (&optional filename) *[FUNCTION]*

Dumps all the elements in the current KB into a Lisp (i.e. text) file with name **filename**. Any missing components of the filename are filled in from ***default-kb-pathname***. If not supplied, the filename defaults to **"checkpoint"**. This does not alter the running Scone, so it is safe (though perhaps not efficient) to checkpoint often. The elements created by the **bootstrap.lisp** file are not dumped.

To create a clone of the checkpointed Scone process, start a new Scone process (with **bootstrap.lisp** already loaded) and then use **load-kb** to load the checkpoint file.

checkpoint-new (&optional filename) *[FUNCTION]*

Like **checkpoint-kb**, but dumps only the elements created since the completion of the most recent file load by **load-kb**. The default output filename is **"checkpoint-new"**.

To create a clone of the checkpointed Scone process, start a new Scone, load the same files as in the original Scone, and then load the checkpoint file. At the start of the checkpoint file, there is a test to ensure that the expected files are in fact present in the Lisp. (This checks only for files of the correct filename, not for identical versions.)

check-loaded-files (file-list) *[FUNCTION]*

Check whether all the files in **file-list** are currently loaded in Scone. Check only the filename itself, not the directory or extension. Don't worry about load order or any extra files that have been loaded. If we fail the test, signal a continuable error.

3. Scone Elements

3.1 Structure of Scone Elements

The Scone knowledge base is best thought of as a collection of interconnected *elements*. Each element is represented in the Scone software as a data structure – a Common Lisp "defstruct".³ An element in Scone can be a *node*, a *link*, or a *relation*.⁴

A *node* represents a conceptual entity such as “George Washington” or “the typical elephant” or “the mother of Clyde”. Note that the node represents a specific *concept* or *meaning* in each case, not a word or word-definition. The connection between Scone elements and natural-language words or phrases is a many-to-many association. It is the job of external software (an English-language front-end to Scone – under construction) to resolve any ambiguity.

A *link* represents a relation or statement of some kind, such as “Clyde is an elephant” or “The wife of George is Martha” or “Every elephant hates P. T. Barnum.”

A *split* is a special kind of link with an unusual structure: it connects to any number of elements rather than just two or three. It is used to state that these connected elements – types and individuals – are mutually distinct and disjoint.

A *relation* is an element that represents the definition of a relation in Scone. The set of relations is open-ended. Once it has been created, the relation can be instantiated for specific cases by creating statement links. A relation is in some respects like a type-node: it can have other relations below it in the is-a hierarchy – effectively these are sub-types of the relation – and statement-links serve as the instances.

A Scone knowledge base represents its long-term knowledge as a set of elements and the pattern of interconnections among these elements. In general, nodes are connected by links. Each link has a few distinct *wires* by which it can be connected to other elements in the knowledge base. These are designated *A-wire*, *B-wire*, *C-wire*, *parent-wire*, *context-wire*, and (for split-links only) any number of *split-wires*.

The A and B wires are used by links to indicate the two elements that the link is saying something about. For example, if a link L represents the statement “Clyde hates Jumbo”, the A wire of element L would go to {**Clyde**}, the B wire to {**Jumbo**}, the parent wire to {**hates**}, and the context wire to the currently-active context, which is always held in the Lisp variable ***context***.

Some relations and statements in Scone also use the *C-wire*. This allows us to easily represent trinary relations and statements, such as "the distance from A to B is C".

³ Scone elements are not currently implemented as objects in the Common Lisp Object System (CLOS) because element structures are heavily used in inner loops of the Scone engine, and some CLOS implementations are inefficient. We may revisit that decision in future releases of Scone.

⁴ At present, we use a single structure type to represent all types of Scone elements. That is a simple solution, but is a bit wasteful, since a given element-type will not use all the wires and other fields in this structure, though only a relatively small fraction of the allocated storage is wasted. We may revisit this decision later, and use multiple structure types to more closely fit the needs of each element.

In general, the *parent wire* of a link is used to indicate one superior class of this link in the type hierarchy. In other words, the parent wire tells us what kind of link this is. (A few particularly important link types, such as *is-a*, *eq*, and *cancel*, are represented as special built-in element types, and are directly recognizable by the software.) The *context wire* is used for links to indicate the context (represented by a node) in which a given statement is considered to be true.

Every element has a *terminal point* to which any number of incoming wires from other elements may be attached.

We think of a link as connecting two or three nodes (A and B and maybe C), and representing some statement about the relationship between these nodes: “A is a B” or “A hates B” or whatever. However, the link also is an entity in the knowledge base, representing the statement itself, and we can make statements (sometimes referred to as “meta-statements”) about this statement: who told us this piece of information, how strongly we believe it, whether exceptions are possible, and so on. So each link element really has two parts: the five wires, which are used to implement the statement the link is making, plus the marker bits and terminal point of a node that represents the statement itself. We sometimes refer to this built-in node as the *handle node* of the link. (Note that in many cases it is preferable to put meta-information on a context node containing a collection of other nodes and links, rather than on the handle node of an individual link.)

Here is another complication: We think of a node as representing an entity, about which we can make various statements by attaching link-wires to it. But every node (except the top-level **{thing}** node) will be a member of at least one superior class. So to reduce the total number of elements and to speed up some common inferences, we equip each node with a *parent wire* that serves as a virtual is-a link to one superior class. So the **{George Washington}** individual-node may have a parent-wire going to the **{person}** type-node, rather than an is-a link to **{person}**.

While this use of a node’s parent-wire significantly reduces the number of elements required in a Scone knowledge base, there is a cost: because a parent-wire is not a full-fledged link with an integral handle node, we cannot attach meta-information to this virtual is-a link or tie it to a context. If it becomes necessary to do that, we must detach the parent wire and, replace it with a full-fledged is-a link. This is handled by the **convert-parent-wire-to-link** function.

Functions and Variables

n-elements [VARIABLE]

The number of elements currently in use in the KB.

first-element [VARIABLE]

First element in the chain of all elements in the KB.

last-element [VARIABLE]

Last element in the chain of all elements in the KB.

do-elements [Complex Macro Body] [MACRO]

This macro iterates over the set of all elements. A call to this macro looks like this:

(do-elements (var) ... body forms ...)
or
(do-elements (var after) ... body forms ...)

Each element in turn is bound to **var** and then the body is executed. Returns **nil**. If **after** is supplied, it must be a form that evaluates to an element. In this case we execute the body only for elements created after the **after** element.

next-element (e) [FUNCTION]
Given an element **e**, get the next one in the chain.

previous-element (e) [FUNCTION]
Given an element **e**, get the previous element in the chain. Note: This is inefficient, since we don't keep back-pointers in this chain. It is used in infrequent tasks such as removing an element from the KB.

a-element (e) [FUNCTION]
b-element (e) [FUNCTION]
c-element (e) [FUNCTION]
parent-element (e) [FUNCTION]
context-element (e) [FUNCTION]
Given an element **e**, return the element connected to the specified wire of **e** (the A, B, C, parent, or context wire), or **nil** if that wire is not connected.

split-elements (e) [FUNCTION]
Given a split element **e**, return the list of elements connected to the split wires of **e**, or **nil** if there are none.

incoming-a-elements (e) [FUNCTION]
incoming-b-elements (e) [FUNCTION]
incoming-c-elements (e) [FUNCTION]
incoming-parent-elements (e) [FUNCTION]
incoming-context-elements (e) [FUNCTION]
incoming-split-elements (e) [FUNCTION]
Given an element **e**, return a list of all elements whose specified wire (the A, B, C, parent, context, or split wire) is connected to **e**'s terminal point. Return **nil** if there are none.

convert-parent-wire-to-link (e &key :context) [FUNCTION]
The parent-wire of element **e** is disconnected and replaced by an equivalent is-a link, which can then be modified or tied to a context. If the user does not supply a **:context** for the new is-a link, use the current ***context***. Returns the new is-a link.

3.2 Element Properties

Every Scone element has a *property list*, storing attribute-value pairs. The set of attributes is open-ended. Each attribute is represented by a Common Lisp keyword. Some properties are defined and used by the Scone implementation, while others may be defined by users or by

software external to Scone. These properties provide an inexpensive way to store meta-information about an element in situations where the information is static and where high-speed access is not required; information that may be cancelled or altered, or that may change from one context to another, is better represented using full-fledged Scone elements.

Functions

get-element-property (e property) *[MACRO]*

Find and return the specified property of element **e**, or **nil** if this property is not present.

The **property** argument is typically a Lisp keyword.

set-element-property (e property &optional value) *[FUNCTION]*

Set the specified property of element **e** to **value**. If **value** is not supplied, the default value is **t**. Returns **value**.

clear-element-property (e property) *[FUNCTION]*

Remove the specified **property** of element **e**. If **e** doesn't have this property, do nothing.

push-element-property (e property value) *[FUNCTION]*

The **property** of **e** should be a list. Push **value** onto this list. Create the property if it doesn't already exist.

4. Referring to Scone Elements

Many of the Scone functions described in this document take one or more *elements* as arguments. Within the running Scone system, each element is represented as a Common Lisp structure (sometimes called a “defstruct”). If you have a pointer to one of these element-objects, perhaps returned by `lookup-element` or some other Scone function, you can supply this directly wherever an element is required.⁵

Alternatively, you can refer to an element by its name. There are two kinds of element names in Scone: *internal names* (sometimes called *inames*) and *external names* (sometimes called “English names”, though other natural languages will eventually be supported).

An internal name such as `{George Washington}` refers uniquely and unambiguously to a specific Scone element. External names are not so constrained: an element may have many external names, and an external name such as `"mouse"` may refer to several distinct Scone elements – in this case, one representing the animal mouse and one representing a computer pointing device. A knowledge-base file can contain both internal and external names, but if a reference is made to an external name with more than one possible meaning, the system may interactively request a clarification while the file is being loaded. So it is generally best to use internal names within KB files.

4.1 Element Internal Names

Internal names are surrounded by “curly braces”. So, for example, the internal name of the node at the top of the type hierarchy can be referred to as `{thing}`. This curly-brace notation can be used wherever a Scone element is required.

An internal name is an arbitrary character string that serves as the label for a specific element. We tend to use names like `{elephant}` that will be meaningful to a human reader, but that is just for convenience in developing and understanding the system’s knowledge base. As far as Scone is concerned, the internal name for the `{elephant}` node could just as well be `{elephant (the animal)}`, `{P123456}`, `{xyzyzy}`, or `{aardvark}` – as long as the chosen name is unique.

These iname strings are matched in a case-insensitive way; that is, `{Clyde}`, `{clyde}`, and `{CLYDE}` all refer to the same element. However, Scone remembers the exact string, including case, that was used when an element was first created, and that string is used whenever the system must print out a reference to the element.

Internal name strings may contain spaces and other punctuation characters, but not the colon character, which is reserved for another use (see below). So, for example, `{mother of Clyde}` is a legal internal name, and names with spaces are indeed fairly common.

⁵ In Lisp, almost all data objects are allocated in dynamic storage, and we pass around pointers to these objects. So it is common in Lisp culture to say that we are “passing Scone element X to function Y”, when in fact we are really passing a pointer to the heap-allocated structure representing Scone element X. If all pointers to the element-X structure are dropped, the Lisp garbage collector will eventually reclaim the storage. But note that a chain of pointers runs through all the Scone elements, so no element structure will be reclaimed unless we explicitly snip it out of this chain, perhaps with `remove-element`.

Internal names are organized into *namespaces*. Within its namespace, an internal name must be unique. That is, it refers to only one element. Any attempt to create a second node with the same iname in a given namespace is an error.

Namespaces are designated by Common Lisp strings such as "**common**" or "**astronomy**". These *namespace names* must be globally unique so that accidental collisions do not occur if we try to combine separately developed knowledge bases. An informal registry of Scone namespace names currently in use is maintained by the author. Contact the author if you want to register a new global namespace. (Scone may eventually have to move to a more complex universal namespace scheme, of the kind found in XML-based knowledge representations, but we want to delay that move as long as possible – the resulting names are long and confusing for human readers.)

To designate an internal name without danger of ambiguity we use its *full name*. This is a Common Lisp string divided into two parts by a colon. The part of the string before the colon designates the namespace, while the part after the colon is the unique internal name within that namespace. So **{biology:mouse}** and **{computer:mouse}** may co-exist without creating a conflict. There may be only one colon in an internal name string.

There is a function called **in-namespace** that takes a string as an argument. That string is converted to a namespace object, which becomes the value of the ***namespace*** variable. Whenever the system is given an iname without a colon in it (referred to as a *short name*), the value of the ***namespace*** variable is used as the default namespace. Typically a KB file will have an **in-namespace** form at the start and will use short-names internally, except when it is necessary to refer to an iname in another namespace.

A namespace may optionally *include* one other namespace, which may include another, and so on, forming a chain. If an attempt is made to look up an internal name in a given namespace and the iname is not found there, the system will look in the included namespace (if any) and then in its included namespace, and so on until the end of the chain is reached. If the iname is not found anywhere in this chain of namespaces, the name is undefined.

If we create a new iname, it will be placed directly into the current ***namespace***, but an error will be signaled if the name conflicts with the same namestring in any included namespace. This namespace-inclusion mechanism is often used to include the "**common**" namespace in some more specialized namespace that the user is creating, so that elements like **{common:thing}** and **{common:person}** can be referred to simply as **{thing}** and **{person}**.

Links normally do not have meaningful internal names. However, if you believe that you will want to refer to a specific link, you can give it an internal name via the **:iname** keyword argument of the "new-whatever" function that creates the link. The ***generate-long-element-names*** control variable, if **t**, tells the system to create longer, more meaningful names for links. That requires more space, but can be useful for development and debugging.

For elements that represent primitive Lisp data types, such as numbers, functions, or strings, the Lisp object itself is used as the iname and there is never a namespace designator. (These objects are kept in a special universal namespace.) So **{123}** designates an element representing an

integer, and `{12.34}` represents a real number. The Scone element `{#'query-user}` represents a Lisp function named `query-user`.

A set of brackets containing a quoted string, such as `{"http://www.cmu.edu"}`, is a string element, representing the Lisp string itself. We use these string elements to represent words, names, URLs, command-line commands, and so on. Note that any colon within a string element is part of the string, not a namespace designator.

Functions and Variables

namespace *[VARIABLE]*
The structure representing the current default namespace.

namespaces *[VARIABLE]*
A hash table containing an entry for each namespace currently loaded in the KB. Associates a string (the name of the namespace) with a namespace structure.

list-namespaces nil *[FUNCTION]*
Return a list of all the Scone namespaces that are currently loaded.

lookup-element (name &key :syntax-tags) *[FUNCTION]*
This takes an element-designator **name** (an element object, an internal name in curly braces, or an external name string) and returns the corresponding element object. If the argument does not correspond to an existing element in the KB, return `nil`.

The optional **:syntax-tag** argument, if present, is used in the resolution of external (English) name strings to element objects. (See the discussion of the **disambiguate** function in the English Names section of this manual). If **lookup-element** is given a string argument and successfully finds an element, it returns the element's syntax-tag as the second return value.

lookup-element-test (name &key :syntax-tags) *[FUNCTION]*
Like **lookup-element**, but if the named element does not (yet) exist, signal an error.

lookup-element-or-defer (name &key :syntax-tags) *[FUNCTION]*
Like **lookup-element**, but if the named element does not (yet) exist, return the argument unchanged.

in-namespace (namespace-name &key :include) *[FUNCTION]*
The namespace designated by the **namespace-name** string becomes the new default namespace (the value of the ***namespace*** variable). If the specified namespace does not already exist, it is created. If the **:include** argument is supplied, this indicates an existing namespace that is included in the new one, if a new one is actually created. The **:include** argument may be either a namespace object or a string designating a namespace.

4.2 Element External (English) Names

A long-term goal for Scone is to interface it to a natural-language input-output system so that users can augment or query the knowledge base in English (or in the human language of their choice). At present, the system has only a simple facility for associating words or phrases (represented as Common Lisp strings) with various elements in the knowledge base. We refer to these words and phrases collectively as *external names* to distinguish them from the internal element-names described in the previous section.

While most of the machinery for natural-language input and output is external to the Scone inference engine, it makes sense (for now, at least) to maintain the dictionary mapping names to elements and elements to names as part of the core Scone system. When a KB file containing new concepts is loaded into the knowledge base, the external names for these concepts are loaded at the same time, as part of the information in this file.

The mapping between external names and Scone elements is many-to-many. Scone provides a simple interface by which an external natural-language system can ask for all the Scone elements (if any) that correspond to a given name. Similarly, the user can ask for a list of all the names (if any) that correspond to a given element.

Note that the name strings are matched exactly, except that the match is case-insensitive. At present there is no spelling correction or “find an approximate match” capability. In general, external name strings should be entered in lower-case, except in the case of proper nouns like “Obama”. There should be no whitespace surrounding the word or phrase. In multiple-word strings (representing idiomatic phrases such as “**kick the bucket**”) the words should be separated by a single space.

Each mapping of a name-string to an element has an associated *syntax-tag*. This is a Lisp keyword that corresponds roughly to the part of speech associated with the name-string. This allows the NL machinery to return only those Scone meanings that correspond to the part of speech it has found – for example, the action “fall” rather than a noun representing a season.

Note: This syntax-tag mechanism is something of a place-holder for now. Once the natural-language interface to Scone is further developed, we will have a much better idea of what syntactic machinery really is needed. We will probably end up with a complex hierarchy of word-types and grammatical roles, represented in Scone itself, but more or less distinct from the conceptual knowledge that the linguistic forms refer to.

For now, the following syntax-tags are defined:

: noun

Either a common name or a proper name for the entities represented by the element. "Clyde" or "elephant".

: adj

An adjective like "blue", which represents a unary predicate of some kind, and is generally associated with a Scone type-node. To describe a type, you normally combine this adjective with some noun labeling a supertype of the type-node in question. So we would normally say "a blue object" or "a blue automobile" rather than just "a blue".

: adj-noun

A word like "solid" that can be used either as a noun or as an adjective. These are very common in English.

:role

A word like "population" that designates a role under some type node. Normally this used with "of" or a possessive form: "the population of India" or "India's population".

:inverse-role

Name given to the inverse of a role relation. For example, if the role name is "parent", the inverse-role may be "child". So if we are told that "John is the child of Mary", the natural-language machinery can convert this to "Mary is the parent of John".

:relation

A word like "employs" or "hates" or a phrase like "taller than" representing some relation that is (grammatically) not a role.

:inverse-relation

Name given to the inverse of a relation. For example, if the relation name is "taller than", the inverse-relation may be "shorter than". So if we are told that "John shorter than Mary", the natural-language machinery can convert this to "Mary is taller than John".

:action

An action verb such as "hit".

Functions and Variables

legal-syntax-tags*[VARIABLE]*

Each external name has an associated syntax-tag. The tags currently defined are stored in this variable.

english (element &rest r)*[FUNCTION]*

Define one or more English names for the specified **element**. The remaining arguments after element are strings or syntax tags (keywords). Initially, we assume that each string becomes an English name of element with syntax-tag **:noun**, but the tag changes in a sticky way whenever a different syntax-tag is encountered. If the special syntax-tag **:iname** is encountered, that indicates that the element's internal name is used as one of the English names. However, if the argument list contains **:no-iname**, that takes precedence and any **:iname** in the **r** list is ignored.

For example, the form

```
(english {swamp} :iname "wetland" "marsh" :adj "soggy" "marshy")
```

says that the element **{swamp}** has English names "swamp", "wetland", and "marsh" (all nouns), plus the adjectives "soggy" and "marshy" (as in "soggy area" or "marshy area").

lookup-definitions (string &optional syntax-tags)*[FUNCTION]*

Returns a list containing all the meanings of the **string** argument (if any). Each meaning in the list will be of the form (**element . syntax-tag**). If the **syntax-tag** argument is supplied, construct and return a list containing only meanings with the specified syntax tag. This function works only for external name strings and is called by the other functions of the lookup-element family.

mark-named-elements (string m &optional syntax-tag) *[FUNCTION]*
All the elements associated with the name **string** are marked with marker **m**. If **syntax-tag** is supplied, only meanings of the specified type are marked.

get-english-names (element &optional syntax-tag) *[FUNCTION]*
Returns a list of all the names associated with **element** (if any). Each name will be of the form (**string . syntax-tag**). If a **syntax-tag** argument is supplied, construct and return a list containing only names of the specified type.

disambiguate (name definition-list &optional syntax-tags) *[FUNCTION]*
Given a list of possible definitions (element.tag pairs) for an English word, try to disambiguate it based on the value of **disambiguate-policy**. If **:ERROR**, just signal an error. If **:ASK**, ask the user. If **:HEURISTIC**, use SYNTAX-TAGS to choose the best definition, or if that fails just pick one.

list-synonyms (string &optional syntax-tag) *[FUNCTION]*

show-synonyms (string &optional syntax-tag) *[FUNCTION]*
The **list-synonyms** function finds all the elements that are registered as a meaning of **string** in the KB. For each of these it returns a list, each element of which has a Scone element first, followed by any number of alternative names for that element. If **syntax-tag** is non-null, we look only at elements whose connection to **string** is of this type. The **show-synonyms** function is similar, but it prints out the synonyms it finds instead of returning a list.

list-dictionary-entries nil *[FUNCTION]*
List all entries in the dictionary. This is primarily for debugging.

See also the functions **lookup-element**, **lookup-element-predicate**, and **lookup-element-test** in the previous section. These can be used to convert a string (and optional syntax-tag) to a single Scone element. If the string is ambiguous – that is, if it is associated with more than one Scone element via the given syntax-tag – these functions call **disambiguate**, which asks the user to pick one of the possible meanings.

5. Markers

Every Scone element has a set of bits representing permanent state information about that element. These bits are known as the element's *flag bits*. Some of the flag bits are used to encode the type of the node or link: individual vs. type node, "is a" versus "eq" link, and so on. Other flag bits encode properties of the element, such as whether a statement link is an instance of a transitive relation. As a rule, flag bits are intended for internal use by the Scone engine, and are not directly manipulated by users of the Scone system, so we will not describe them in detail in this manual.

Every Scone element also has storage for a number of *marker bits*, which are set and cleared by the Scone engine as part of various search and inference operations. Operations on marker bits generally refer to a specific marker using a numerical index, starting at 0. The constant **n-markers** indicates the maximum number of marker bits a Scone element can hold: in the current implementation for 32-bit SBCL, **n-markers** is 28, so legal markers are designated by integers 0 through 27. On a 64-bit Common Lisp, there are more markers available – check **n-markers** to see how many you have.

Markers are normally allocated and used in pairs: a *positive-marker*, used to mark some set of elements, and an associated *cancel-marker*. The cancel-marker is placed on elements *definitely not* in the set marked with the positive marker and also on links that are cancelled in the course of marking this set of elements. We speak of a marker and its cancel marker together as a *marker-pair*. Effectively, these pairs implement a 3-valued logic: yes, definitely no, and maybe or "don't know".

The number of possible marker pairs, as well as the number of positive markers, is indicated by the constant **n-marker-pairs**, which will be half of **n-markers** (rounded down). In the 32-bit SBCL implementation **n-marker-pairs** is 14. So markers numbered 0-13 are positive markers, and markers 14-27 are the corresponding cancel markers. When given a positive marker **m**, the function (**cancel-marker m**) will return the corresponding cancel marker.

Two of these marker-pairs are currently reserved for use by the Scone engine. The *context marker* (normally marker-pair 0) is used to mark all the nodes representing the currently active context and its superiors. The global variable ***context-marker*** always holds the index of the context marker, so users should refer to the marker via that variable. Similarly, the ***activation-marker*** (normally marker-pair 1) is used to mark all the elements that are active in the current context. The remaining marker-pairs are available for search and inference operations initiated by the user.

Scone provides functions for allocating and freeing markers (or marker-pairs) for use by various software operations. The variable ***available-markers*** indicates the number of marker-pairs available for allocation at any given time. The **get-marker** function selects and reserves one of the available markers and returns its index; the corresponding cancel-marker is also reserved by this operation. The **free-marker** function removes the specified marker and its cancel-marker from all Scone elements and returns this marker-pair to the available-markers pool. The **with-temp-markers** macro is a convenient way to allocate a temporary marker-pair, use it while executing any number of operations, and then to reliably free it.

For a detailed description of how Scone uses these markers for search and inference, please see the paper "Marker-Passing Inference in the Scone knowledge-Base System", available from the Scone website: <http://www.cs.cmu.edu/~sef/scone/publications/>.

Functions and Variables

Define the Range of Legal Markers and Marker-Pairs

n-markers [*CONSTANT*]

The maximum number of marker bits allowed in this Scone engine. This number is set when the Scone engine is compiled for a particular hardware architecture, so it cannot be changed dynamically.

n-marker-pairs [*CONSTANT*]

The maximum number of marker pairs.

legal-marker? (m) [*MACRO*]

legal-marker-pair? (m) [*MACRO*]

Predicates to determine whether a number **m** represents a legal marker in the current Scone.

The **legal-marker?** predicate accepts any integer greater than 0 and less than **n-markers**; the **legal-marker-pair?** predicate accepts only those integers less than **n-marker-pairs?**.

check-legal-marker (m) [*MACRO*]

check-legal-marker-pair (m) [*MACRO*]

Check whether **m** is a legal user marker. If not, signal an error.

marker-bit (m) [*MACRO*]

Get the marker bit mask associated with marker **m**.

get-cancel-marker (m) [*MACRO*]

Get the cancel-marker corresponding to marker **m**.

Basic Marker Functions

mark (e m) [*FUNCTION*]

Marks element **e** with marker **m**. If **e** is already marked with **m**, do nothing. Return the number of elements marked with **m** after this operation.

unmark (e m) [*FUNCTION*]

Removes marker **m** from element **e**. If **e** is not marked with **m**, do nothing. Return the number of elements marked with **m** after this operation.

marker-count (m) [*FUNCTION*]

Returns the number of elements currently marked with marker **m**.

marker-on? (e m) [FUNCTION]
Predicate to determine whether element **e** has marker **m** turned on.

marker-off? (e m) [FUNCTION]
Predicate to determine whether element **e** has marker **m** turned off.

clear-marker (m) [FUNCTION]
Clear marker **M** from all elements, but do not free it.

clear-marker-pair (m) [FUNCTION]
Clear marker **M** and the associated cancel marker from all elements, but do not free it.

clear-all-markers nil [FUNCTION]
Clear and free all markers from all elements in the KB. Note that this clears the context and activation markers, so you should follow this with an **in-context** or **refresh-context** call.

do-marked (var m &optional after) ... body forms ... [MACRO]
This macro iterates over the set of all elements marked with marker **m**. A call to this macro looks like this:

(do-marked (var m) ... any number of body forms ...)
or
(do-marked (var m after) ... any number of body forms...)

Each **m**-marked element in turn is bound to **var** and then the body is executed. Returns **nil**. If **after** is supplied, it must be a form that evaluates to an element marked with **m**. Only iterate over elements after this one in the chain of **m**-marked elements. *NOTE: It is OK to mark new elements with **m** in the body of this loop, but don't unmark any elements.*

Boolean Marker Functions

mark-boolean (m must-be-set must-be-clear &optional flags-set flags-clear) [FUNCTION]
The **m** argument is a marker. The **must-be-set** and **must-be-clear** arguments are lists of markers. Conceptually, we scan every element in the KB. If all of the **must-be-set** markers are on and none of the **must-be-clear** markers are on, turn on marker **m** for this element. Returns the number of elements marked with **m** after this operation.

So, for example, **(mark-boolean 3 '(0 1) '(2))** places marker 3 on every element that is currently marked with 0 and 1 and that is not marked with 2.

The **flags-set** and **flags-clear** arguments, if supplied, can control which kinds of elements will potentially be marked. This option is for internal use, and is not documented here.

unmark-boolean (m must-be-set must-be-clear &optional flags-set flags-clear) [FUNCTION]
Like **mark-boolean**, but clears marker **m** from the selected elements.

convert-marker (m1 m2) *[FUNCTION]*
For every element marked with **m1**, mark it with **m2** (if it was not marked with **m2** already) and clear **m1** from that element. This could be done with **mark-boolean**, but this specialized version is faster.

Marker Allocation

Note: It is a Very Bad Idea to just grab a marker and start using it without informing the system that this marker is now in use. This can lead to bugs that are very hard to diagnose. So always use **with-markers** or **get-marker** and **free-marker** when you want a marker to play with.

n-available-markers *[VARIABLE]*
The number of markers currently available for allocation.

get-marker nil *[FUNCTION]*
Allocate an available marker, returning the integer index. If no available markers remain, return **nil**.

free-marker (m) *[FUNCTION]*
Return Marker **m** to the pool of available markers and clear the marker. If the marker is already available or is reserved by the system, do nothing.

free-markers nil *[FUNCTION]*
Return all markers to the free pool and clear them, except for those such as the context marker that are permanently allocated to the Scone engine.

with-markers (marker-vars body-form &optional fail-form) *[MACRO]*
The **marker-vars** argument is a list of variables. Each is locally bound to the index of a freshly allocated marker. We then execute the body form, returning whatever value or values that form returns. But before leaving this form, either in the normal way or via an error or non-local exit, we de-allocate and clear these markers.

If there are not enough markers available to fill all the **marker-vars**, we don't allocate any new markers. Instead we execute the **fail-form** and return whatever it returns. The default action, if there is no **fail-form**, is to signal an error.

A call to this macro might look like this:

```
(with-markers (v1 v2 v3)
  (progn
    (some-action-using-marker v1)
    (another-action-using-markers v1 v2 v3))
  (deal-with-an-allocation-failure))
```

6. Contexts

Every Scone element has a context-wire. This is used to tie a node to the context in which it exists, and it is used to tie a link to the context in which it is true. A context is simply a Scone node. When a new context-node is first created, it inherits the contents (nodes and links) of its superior nodes in the is-a hierarchy. You can then add additional KB structure in the new context. You can also cancel or negate nodes and links that would otherwise be inherited.

The care and feeding of contexts, and all the uses to which they can be put, is a big topic, beyond the scope of this manual. Here we will just document that functions and variables that implement the context machinery.

Functions and Variables

context *[VARIABLE]*

The element representing the current context..

in-context (e) *[FUNCTION]*

Argument **e** is an existing element representing a context. **e** becomes the value of ***context***. Then we propagate ***context-marker*** to all the contexts whose contents are inherited by **e**, and we propagate ***activation-marker*** to all the elements active in this set of contexts. If the ***context*** is already **e**, do nothing. This can be a relatively expensive operation – we have chosen to do more work when changing contexts in order to make within-context operations as fast as possible.

new-context (iname &optional parents) *[FUNCTION]*

This is a convenience function for creating a new context node as a descendant of one or more pre-existing context-nodes. It is actually a specialized form of **new-indv** (see below). The **parents** argument may either be a single parent node or a list of parents. If **parents** is unsupplied or **nil**, assume that there is a single parent, **{general}**.

refresh-context nil *[FUNCTION]*

Keep the current value of ***context***, but update the context and activation markers. This is used when we think that the context and activation markers may somehow have been messed up.

context-marker *[VARIABLE]*

Marker permanently assigned to mark the current ***context*** node and its superiors.

context-cancel-marker *[VARIABLE]*

Cancellation marker corresponding to ***context-marker***.

activation-marker *[VARIABLE]*

Marker permanently assigned to mark every element active or present in the current context.

activation-cancel-marker *[VARIABLE]*

Cancellation marker corresponding to ***activation-marker***.

7. Adding Elements to the Scone KB

7.1 Creating New Elements

Each type of Scone element has a “new” function that creates it: **new-indv**, **new-type**, **new-is-a**, and so on. These new-functions also check whether the element about to be added conflicts with other information currently in the KB.

All of these new-functions have certain features in common. All will create at least one new element in the KB, and the new element object created (or the *primary* new element created, if there are more than one) is returned as the value of executing the new-function. A knowledge-base (or KB) file is just a collection of such new-functions, with perhaps some comments and assorted other functions mixed in.

New-functions that create a node, such as **new-indv**, generally take two required arguments, the internal name of the new node and the parent node. So a typical call might be something like

```
(new-indv {Clyde} {elephant}).
```

Most new-functions that create a link take two required arguments, the two elements that the link connects, called the A and B arguments. So a typical call might be something like

```
(new-is-a {Clyde} {elephant}).
```

The new-function for general statements takes three required arguments, as in

```
(new-statement {Clyde} {hates} {Ernie}).
```

In this case, **{Clyde}** is attached to the A-wire of the new statement link, **{Ernie}** is attached to the B-wire, and **{hates}** is attached to the parent-wire of the link.

The **:context** Keyword Argument

All new-functions take optional keyword arguments. Normally, all new elements are created in the current context (as indicated by the current value of the ***context*** variable). The new elements’ context-wires are connected to that context node. However, you can over-ride this for a particular new element by providing a **:context** argument pointing to some other context-node.

Internal Names and the **:iname** Argument

iname is not a required argument for links, splits, and certain other elements. The system normally just makes up a suitable new name for these elements. However, you can provide a specific iname for these elements if you like via the **:iname** keyword argument.

Where an **iname** argument is required, a value of **nil** tells Scone to make up a unique name for the new element. For example, if we say

```
(new-indv nil {elephant})
```

the new individual node will be given a name like **{elephant (0-27)}**. Numbers are appended to these made-up-names to ensure that they are unique.

For convenience, the `iname` argument can be a quoted string instead of an internal name in curly braces, but the effect is the same: that string is converted into an iname, and it must be unique within its namespace. So `(new-indv "Clyde" {elephant})` is functionally equivalent to `(new-indv {Clyde} {elephant})`.

The `:english` Keyword Argument

Most of the new-functions take an `:english` keyword argument. This takes a list of names and syntax tags that is interpreted as in the `english` function. So, for example, we might see a form like this:

```
(new-type {swamp} {geographical area}
  :english '(:iname "wetland" "marsh" :adj "soggy" "marshy"))
```

This creates new type of land area with internal name `{swamp}` and English names "swamp", "wetland", "marsh", and adjective names "soggy" and "marshy".

By default, the new-element's internal name becomes its English name (or one of them), with a syntax-tag that is the default for elements of this type: `:noun` for most `indv` and `type` nodes, `:role` for new roles, and `:relation` for new relations. This is in addition to whatever names are given by the `:english` keyword argument. However, if the first element of the `:english` argument is a syntax-tag, the iname gets that syntax tag instead of the default. If you don't want the internal name to become an English name at all, put the tag `:no-iname` in the `:english` list. As a rule, internal names that the Scone system constructs itself are not saved as English names.

So, for example, if we execute the form

```
(add-indv {gizmo} {frobnitz} :english ("foo" "bar"))
```

the new `{gizmo}` node would have English names "gizmo", "foo" and "bar", all with the syntax-tag `:noun`. But for the form

```
(add-indv {noisy} {sound level} :english (:adj "loud"))
```

the new `{noisy}` node has English names "noisy" and "loud", both with syntax-tag `:adj`.

Negation Links

For each of the link-types in the system (except for the `cancel-link`), there is a negated version: `is-not-a`, `not-eq`, and so on. There is also a negated form of the generic statement link: `not-statement`. If there is a negated link from A to B, that will have the effect of over-riding any positive link of the corresponding type from A to B. That is, the negated relation is always stronger than the positive form of the relation whether the positive form is inherited or stated directly. If you want to re-assert the positive relation between A and B, you must first cancel the negative link using a `cancel-link`.

Cancellation and negation in Scone is another topic that is beyond the scope of this manual. It interacts in complex ways with transitive relations. A quick rule of thumb is that a non-transitive statement can either be cancelled or over-ridden by a negation; a transitive statement can only be over-ridden, since there may be multiple paths; a negation can only be cancelled (which is allowed, because negation links are never transitive).

7.2 Indv Nodes

An *individual node* (or *indv-node*) represents an individual entity rather than a type. It is not necessarily a physical object: it can be something more abstract, such as a number or a specific date. Individuals are normally the leaf-nodes of Scone's is-a hierarchy.

An indv-node may be *proper* or *generic*. A proper individual node represents a specific, first-class individual such as `{George Washington}` or `{Paraguay}` or `{21 March 1948}`. Normally, proper individuals are assumed to be distinct from one another, and it is an error to equate two of them. However, in special cases we may choose to over-ride this prohibition, for example to equate `{Clark Kent}` to `{Superman}`.

A generic individual node represents some role or defined individual, such as "Clyde's mother" or "the first elephant on the moon". Generic individuals can be given properties and relations, just like proper individuals: we can say things like "Clyde's mother is beautiful" or "the first elephant on the moon was an African elephant". However, generic individuals are not assumed to be distinct from one another – in fact, we may think of a generic individual as a stand-in for an proper individual to be named later. It is legal to equate `{mother of Clyde}` to the proper indv-node `{Bertha T. Elephant}` or to another generic indv-node like `{the first elephant on the moon}` or to both of these at once.

Some individual nodes are *primitive individual nodes*. These Scone elements represent Common Lisp objects such as numbers (integers, ratios, or real numbers), strings, or functions. There are new-functions for the various types of primitive nodes, but you normally create them at read-time by putting the desired Lisp object within curly brackets: `{3}`, `{22/7}`, `{3.14159}`, `{"xyzy"}`, or `{#'append}`. These primitive nodes have a default parent based on their type – a Scone type-node such as `{string}` or `{ratio}`, but the user may optionally provide a more specific parent node via the `:parent` argument.

Functions

`new-indv`

[FUNCTION]

(iname parent &key :context :proper :may-have :definition :english :english-default)

This creates a new individual with the specified **iname** and **parent**. (As noted above, if the **iname** argument is **nil** the system will make up a unique name.)

The user may optionally specify a **:context** argument (defaults to the value of ***context***). By default, a has-link is also created, connecting the context to the new indv-node.

If **:proper** is supplied and is explicitly **nil**, this is a generic individual; otherwise assume the new node represents a proper individual.

The **:may-have** argument, if T, indicates that the element may or may not exist in the specified context. In this case, no has-link is created.

If a **:definition** is supplied, this is a defined indv node, and the definition is saved. This is intended for internal use.

The **:english** argument is used to supply one or more English names for the new element, as described above. The default syntax-tag for these English names is normally **:noun**, but a different tag can be supplied via the **:english-default** argument.

new-number (value &key :parent)	[FUNCTION]
new-integer (value &key :parent)	[FUNCTION]
new-ratio (value &key :parent)	[FUNCTION]
new-float (value &key :parent)	[FUNCTION]
new-string (value &key :parent)	[FUNCTION]
new-function (value &key :parent)	[FUNCTION]
new-struct (value &key :parent)	[FUNCTION]

Each of these functions create a new “primitive” individual node representing some Common Lisp data object: a number (of any kind), an integer, ratio, floating-point number, string, functions, or a structure (also known as a “defstruct”). The **value** argument must be a Lisp object of the proper type. If another Scone object with this same (eql) Lisp value already exists, we may return the existing object instead of creating a new one. These objects are all created in the **{universal}** context.

node-value (element)	[FUNCTION]
-----------------------------	------------

If **element** is a primitive individual node, extracts the value from the iname and returns it as a Lisp object. Otherwise, returns **nil**.

7.3 Type Nodes

A *type-node* represents the *typical member* of some class or set: **{elephant}**, **{rainbow}**, **{weekday}**, and so on. You can attach properties and class-memberships to a type-node just as if it were an individual: "The typical elephant is a mammal", "the color of the typical elephant is gray", and so on. In the way they are used, type-nodes and indv-nodes are very similar, except that individual nodes are normally leaf-nodes in the is-a hierarchy, while type nodes are intended to have subtypes and instances.

Note that a type node represents the typical member of a set and not the set itself. For every type node, there is an implicit *set-node* that does represent the set. However, we don't actually create the set node until/unless we have something to say about the set. The **get-set-node** and **get-type-node** functions allow us to access the set-node of a given type-node, and vice-versa. It is perhaps confusing that the set-node of a type-node is an individual node – it represents an *individual instance* of a set.

As with **new-indv**, the **new-type** function optionally takes **:context** and **:english** arguments. The **:n** argument is used to indicate the number of instances of this type that exist in the specified (or default) context.

Most type-nodes represent description-based types such as “elephant”. For these, we have a description of the typical member, but not a precise definition. However, Scone is also able to represent *defined-types*, which are normally created by the **new-defined-type** function, described below. As new elements are created and as the KB is modified, the Scone engine looks for members and subtypes of each defined type. When one is found, the engine creates a new *derived is-a link*, placing the element under the defined type in the is-a hierarchy. Once that

is done, the new member may inherit properties and relations that are true of the defined class, but not part of the definition.⁶

Functions

new-type (iname parent &key :context :definition :may-have :n :english :english-default) [FUNCTION]

Creates and returns a new type-node with the specified **iname** (which will be created if the iname is **nil**) and **parent**. By default, this also creates a has-link from the specified context to the new type-node.

The user may optionally specify a **:context** argument (defaults to the value of ***context***). By default, a has-link is also created, connecting the context to the new indiv-node.

The **:may-have** argument, if T, indicates that the element may or may not exist in the specified context. In this case, no has-link is created.

If a has-link is created, the **:n** argument is placed on the has-link to indicate how many elements of this type exist (or typically exist) in the context.

If a **:definition** is supplied, this is a defined indiv node, and the definition is saved. This is intended for internal use.

The **:english** argument is used to supply one or more English names for the new element, as described above. The default syntax-tag for these English names is normally **:noun**, but a different tag can be supplied via the **:english-default** argument.

get-set-node (e &optional parent) [FUNCTION]
Returns the set-node associated with type-node **e**, creating it if necessary. The parent of the new set node is assumed to be **{set}**, but an alternative value can be supplied as the optional second argument.

get-type-node (e) [FUNCTION]
Element **e** should be a set-node associated with some type-node. Returns that type-node, if it exists, or **nil**.

new-defined-type (iname supertype-list predicate &key :context :english) [FUNCTION]
Creates and returns a new defined-type node with the specified **iname**. The definition is a predicate in two parts: **supertype-list** is a list of elements that must all be superiors of the candidate element. There must be at least one element in this list, and for efficiency it

⁶ Because Scone's reasoning is not logically complete, we cannot guarantee that *every* member of this defined class will be identified every the time the KB is modified. The computation involved may be intractable or unbounded. However, we will catch the vast majority of these. We plan to add additional inference mechanisms to the Scone engine to identify certain defined-class members at query time.

should be as specific as possible. A single object may be passed in instead of a list; **predicate**, if non-nil, is a Lisp predicate function of one argument that must evaluate to **t** for members of this defined class. The predicate should not include testing the **supertype-list** -- that is done separately.

new-intersection-type (iname parents &key :context :english) [FUNCTION]
parents should be a list of elements, usually type-nodes. Create and return a new defined-type element, with name **iname**, defined to be the intersection of all of the supertypes on the list.

new-union-type (iname parent subtypes &key :context :english) [FUNCTION]
subtypes is a list of type-nodes or individuals. Creates and returns a new type that is, by definition, the union of these subtypes. All of these subtypes become members of a complete-split, indicating that a member of the union-type must be a member of one – and only one – of the subtypes.

7.4 Map Nodes

A map-node represents, by definition, “the x of y” – for example, the **{mother}** of **{Clyde}**. Users do not normally deal with map-nodes directly – they are created and accessed by the functions in section 7.13 below.

If the **role** argument is an individual role, the map-node is an individual map node; if the role is a type-role, the map-node is a type map node.

Functions

new-map (role owner &key :iname :english :no-supplement) [FUNCTION]
Make and return a new map-node representing the **role** of the **owner**. The role and owner must already exist – no deferrals are allowed here. Optionally provide an **:iname** and **:english** names. If no **:iname** is supplied, we make up a suitable one. The **:no-supplement** arg, if T, turns off some internal housekeeping that is normally done.

7.5 IS-A Links

An is-a link says that element A "is a" B in the specified context, and that A should (by default) inherit all the properties and type-memberships of B. We speak of B as being "above" A in the is-a hierarchy.

If A and B are both type-nodes, this is a subtype relationship; if A is an individual and B is a type, it is an instance-of relationship. The type hierarchy branches in both directions (multiple inheritance), so a node may have any number of incoming and outgoing is-a links. As noted earlier, a node's parent wire may take the place of one of these outgoing is-a links.

Functions

new-is-a (a b &key :negate :dummy :derived :iname :parent :context :english) [FUNCTION]
Creates and returns an is-a link from **a** to **b**.

new-is-not-a (a b &key :iname :dummy :parent :context :english) [FUNCTION]
Make and return a new is-not-a link with the specified elements on the A and B wires.

7.6 EQ Links

An **eq** link says that A and B are equivalent – they represent the same entity – in the specified context. A inherits all the properties of B and B inherits all the properties of A. It is common to equate a proper individual with a generic individual, or two generic individuals to one another. The system will complain if you try to equate two proper individuals, though you can ignore the warning and force this.

Functions

new-eq (a b &key :negate :dummy :iname :parent :context :english :no-supplement) [FUNCTION]
Creates and returns an eq link from **a** to **b**.

new-not-eq (a b &key :iname :dummy :parent :context :english) [FUNCTION]
Make and return a new not-eq link with the specified elements on the **a** and **b** wires.

7.7 Has Links

Note: The details of the has-link and of quantification in general are under active review, and probably will be changing very significantly in future releases of Scone.

Functions

new-has (a b &key :n :negate :dummy :iname :parent :context :english) [FUNCTION]
Make and return a new HAS-LINK with the specified elements on the A and B wires.
Optionally provide :PARENT and :CONTEXT. If :N is specified, that is the cardinality, stored on the C-wire.

new-has-no (a b &key :iname :dummy :parent :context :english) [FUNCTION]
Make and return a new HAS-NO-LINK with the specified elements on the A and B wires.
Optionally provide :PARENT and :CONTEXT. This is equivalent to NEW-HAS with the :NEGATE flag.

7.8 Cancel Links

Suppose the knowledge base has a **{hates}** statement from **{elephant}** to **{snake}**, indicating that elephants hate snakes – or, more precisely, that the typical elephant hates the typical snake. Suppose that Clyde is an elephant, but an atypical one: he doesn't hate snakes. We can represent this kind of exception using a cancel link from **{Clyde}** to the **{hates}** statement inherited from **{elephant}**. The effect of the cancel link is to turn off the **{hates}** statement – to render it inoperative – when Scone is considering the properties of **{Clyde}**. The statement is still effective for other elephants.

We could also cancel snake-hating for an entire sub-class of elephants, such as **{African elephant}** by connecting the cancel-link from that type-node to the hates statement. Or we

could tie the cancel-link to a context: perhaps in `{snake fantasy world}`, everything is the same as in `{general}` – the real-world context – except that elephants do not hate snakes.

Cancel links are intended to cancel non-transitive statements, but not class-memberships. For that, use an is-not-a link.

Functions

new-cancel (a b &key :iname :dummy :parent :context :english) [FUNCTION]
Creates and returns a cancel link from **a** to **b**.

7.9 Relations

Scone allows the user to create new *relations* and then to instantiate these using *statement* links. A relation element is similar in form to a link – it has A, B, and (sometimes) C wires – but it doesn't really make an assertion of any kind. It is simply a definition or a template for statement links that do make an assertion.

A relation has an internal name or *iname* such as `{taller than}` or `{employs}`, naming the relation that holds from A to B.

The **new-relation** function creates a new relation-element whose wires are connected to the specified **:a**, **:b**, and (optionally) **:c** arguments. The **:a** element represents the domain of the relation, or rather the typical member of the domain set; the **:b** element represents the typical member of the range; the **:c** element, if present, is some value that is a function of a specific A and B. For example, a relation might represent the distance between A and B, with the C role being the value.

A very common situation is that the caller of **new-relation** will want to create a new individual with a specific parent to be used as the A element of the new relation. The most convenient way to do this is to supply an **:a-inst-of** keyword followed by the parent element, instead of the **:a** keyword. That indicates that the new-relation function should create a new individual role node with the specified parent and attach this to the new relation's a-wire. The **:b-inst-of** and **:c-inst-of** keywords perform the same function for the relation's other wires.

So, for example the user might create a new relation as follows:

```
(new-relation {reports to} :a-inst-of {person} :b-inst-of {person})
```

This says that when we create a `{reports to}` statement, both the A and B elements must be instances of `{person}`. But not every `{person}` is involved in a `{reports to}` relation.

The parent wire of the relation element may connect to a more general relation, forming a type hierarchy of relations. For example, the parent of the `{hates}` relation might be `{dislikes}`, since we think of "hate" as a more specific kind of "dislike" – a particularly intense dislike.

Relations may be marked as transitive and/or symmetric, and these attributes are inherited by the relation's subtypes and instances (statements). If we have a symmetric relation such as `{touches}`, and we know that A touches B, it is not necessary to also state that B touches A. Scone deduces that. Similarly, if we have a transitive relation, such as `{taller than}`, Scone will deduce that if A is taller than B and B is taller than C, then A is taller than C.

Functions

new-relation

[FUNCTION]

(iname &key :a :a-inst-of :a-type-of :b :b-inst-of :b-type-of :c :c-inst-of :c-type-of
:parent :symmetric :transitive :english :inverse)

Define a new relation element that can be instantiated by a statement link. The **iname** is the name of the relation from **a** to **b**. This relation may have another relation as a parent, or a type-node. Normally the caller supplies elements as **:a**, **:b**, and **:c** arguments. However, for the common case where you want to create a new generic node with a specified parent, you can instead supply **:a-inst-of** or **:a-type-of**, and similarly for **b** and **c**. Optionally provide a **:parent** element for the relation itself. Optionally set the **:symmetric** and **:transitive** properties of the relation. (These properties refer to the relation from **a** to **b**.) The **:english** argument, as usual, is the name of the relation (or list of names) in the forward direction. The **:inverse** argument is similar in form, but in the reverse direction.

7.10 Statement Links

Once a relation has been defined, we can create statement links that instantiates that relation.

Functions

new-statement (a rel b &key :c :context :negate :dummy :iname :english) [FUNCTION]

Creates and returns a statement link representing the statement "a rel b". For example, we might say something like this:

```
(new-statement {Clyde} {taller than} {Ernie})
```

The **rel** argument must be a relation element or another statement. The **a** and **b** arguments must be compatible with the **a** and **b** elements of **rel**, respectively. If the superior **rel** has a **c**-node, we can supply a **:c** argument to the statement.

new-not-statement (a rel b &key :c :context :iname :dummy :english) [FUNCTION]

Make and return a new not-statement with the specified elements on the **a** and **b** wires.

7.11 Split and Complete-Split Links

A *split* is a special kind of link: it can have any number of *split-wires* instead of individual A, B, and C wires. The meaning is that the type nodes connected by the split wires are mutually disjoint: no individual can be an instance of more than one of these types. If some of the connected nodes are individuals, they are disjoint from one another and are not instances of any types in the split. Scone has efficient machinery for determining whether a new or proposed link causes a split to be violated.

So, for example, we might say

```
(new-split '{bird} {reptile} {mammal}))
```

to indicate that these are disjoint types with no common members. However, there is no claim that the split types exhaust all the possibilities. There may be other subtypes of **{vertebrate}** that are not listed here.

A *complete split* is like a split, but it makes the additional claim that the split classes, taken together, cover all the members of some supertype. Put another way, every member of the supertype must belong to exactly one of the types in the complete split. So we might say something like this:

```
(new-complete-split {vertebrate}
  '{fish} {amphibian} {reptile} {bird} {mammal}))
```

A given type may have multiple splits and complete splits under it:

```
(new-complete-split {person} '{male} {female}))
(new-complete-split {person} '{child} {adolescent} {adult}))
```

Once a split is in place, an attempt to violate the split will be detected. However, it is possible to use a cancel-link to turn off the split for a particular individual if, for example, we want to create an individual who is both a male and a female. (However, if we do this, we may have to use a number of additional cancellations to deal with conflicting properties of this hybrid entity.)

Functions

new-split (members &key :iname :dummy :parent :context :english) [FUNCTION]

The **members** argument is a list containing any number of elements, usually type and individual nodes. Create and return a split element stating that all of the members are distinct and disjoint.

new-complete-split [FUNCTION]

(supertype members &key :iname :dummy :parent :context :english)

Like **new-split**, but also states that any instance of the **supertype** must belong to exactly one of the **members**.

add-to-split (split new-item) [FUNCTION]

Add one **new-item** to the members of the specified **split**, which may be a complete split.

7.12 Functions to Create Multiple Nodes

These functions create multiple new subtypes or members in a given class.

In each case, the **members** argument is a list of items. An item may be an internal name for one of the new elements in the group, or a list whose first element is the internal name; in that case, the remainder of the list is treated as an **:english** argument for this new element. By default, the **iname** becomes one of the English names for the element, with syntax-tag **:noun**. If the first element of the English list is a syntax-tag, the **iname** gets that tag instead. If you don't want the **iname** to be used as an English name at all, include **:no-iname** in the list.

Functions

new-split-subtypes (parent members &key :iname :context :english) *[FUNCTION]*

Create a set of disjoint subtypes under **parent**. First we create a new split under **parent**. The **members** argument is a list of internal names, one for each new type-node that is to be created, as described above. Optionally, provide the **:context**, **:iname**, and **:english** names for the new split element. Returns the new split element.

new-complete-split-subtypes (parent members &key :iname :context :english) *[FUNCTION]*

Like **new-split-subtypes**, but creates a complete split of the elements under **parent**.

new-members (parent members &key :iname :context :english) *[FUNCTION]*

Like **new-split-subtypes**, but the items become new proper individual nodes rather than subtypes of **parent**. Creates and returns a split element stating that all these new indv-nodes are distinct.

new-complete-members (parent members &key :iname :context :english) *[FUNCTION]*

Like **new-complete-split-subtypes**, but the items become new indv nodes rather than subtypes of **parent**. In other words, this states that the items list contains all the individual members of the parent set. Returns the new split element.

7.13 Roles

Roles are associated with type-nodes by the **new-indv-role** and **new-type-role** functions. Once a role has been created for some type, other functions described in this section can be used to attach roles and values to instances of that type.

Note: Quantification machinery (**:n** and **:may-have** arguments) for **new-indv-role** and **new-type-role** is messed up, and will soon be replaced. For now, this correctly describes what the code does.

new-indv-role *[FUNCTION]*

(iname owner parent &key :may-have :transitive :symmetric :english :inverse)

Creates a new generic indv-node representing some role of the **owner**. Also creates a **has** statement indicating that for every instance of **owner**, there exists one instance of this role. The **iname** is the name of the role-node. The **parent** argument is the parent of the new role-node. The **:english** argument is used to register English names for the role, with the default syntax-type of **:role**. By default, the iname becomes one of the English names.

If **:may-have** is present and non-null, that indicates that not every instance of **owner** has this **role**. In this case, we do not create the has-link.

The **:symmetric** and **:transitive** flags refer to the implicit relation created between the owner and the role-node. The **:english** argument, as usual, is the name of the role (or set of names) in the forward direction. The **:inverse** argument is similar in form, but in the reverse direction.

new-type-role *[FUNCTION]*

(iname owner parent &key :n :may-have :transitive :symmetric :english :inverse)

Like **new-indv-role**, but this function creates a type-role. The value of **:n** is the number of instances of this role that the owner typically has. If **:may-have** is present and non-null, it indicates that not every instance of owner has this role, so we do not create the has-link. Only one of **:n** and **:may-have** should be supplied.

x-is-the-y-of-z (x y z) *[FUNCTION]*

The function **x-is-the-y-of-z** finds or creates the node representing "the y of z". Check whether **x** can be equated to this node without causing any errors. If so, create an eq-link between the role node and **x**. Return the new eq-link.

x-is-a-y-of-z (x y z) *[FUNCTION]*

The function **x-is-a-y-of-z** finds or creates the node representing "the y of z". This should be a type-node. Check whether **x** can be of this type. If so, create and return an is-a link from **x** to this node.

the-x-of-y-is-a-z (x y z) *[FUNCTION]*

The function **the-x-of-y-is-a-z** finds or creates the node representing "the x of y". Check whether this node can be of type **z**. If so, create an is-a link from this node to type-node **z**. Return the new is-a link.

x-is-not-the-y-of-z (x y z) *[FUNCTION]*

x-is-not-a-y-of-z (x y z) *[FUNCTION]*

the-x-of-y-is-not-a-z (x y z) *[FUNCTION]*

These functions are like their positive counterparts, but they create a not-eq or not-is-a link that will over-ride any existing relationship and will prevent a new one from being formed (unless this not-link is later cancelled).

8. Removing Elements from the Scone KB.

Beware: Wholesale and random removal of elements can leave the knowledge base in a bad state, with lots of dangling connections that could give rise to anomalous behavior. However, it is sometimes necessary to remove a few elements to repair some error or problem in the knowledge base. It is usually safe to remove the last N elements created, since these refer to earlier elements but earlier elements usually don't refer to them, except in rare cases where the deferred-connection mechanism has been used.

Functions and Variables

remove-element (e) *[FUNCTION]*
Disconnect element **e** from the network and remove it.

remove-last-element nil *[FUNCTION]*
Remove the last element created.

remove-elements-after (e) *[FUNCTION]*
Given some element **e**, remove all elements added later than **e**, starting with the last one.

remove-last-loaded-file nil *[FUNCTION]*
Remove all the elements, starting with the last-created, until we have removed all elements created by the last loaded file (plus any created interactively after that). Pops the last file off **LOADED-FILES** and **LAST-LOADED-ELEMENTS**. Returns the number of elements remaining in the KB.

remove-currently-loading-file nil *[FUNCTION]*
Remove all the elements, starting with the last-created, until we have removed all elements created by the file that is currently being loaded. This is useful for backing out of a file-load if an error aborts the load. Returns the number of elements remaining in the KB.

remove-context (c) *[FUNCTION]*
The **c** argument is an element representing a context. Remove **c**, all its sub-contexts, and all their contents. Return the number of elements that remain in the KB after this operation.

9. Queries and Predicates

9.1 Element-Type Predicate Functions

These functions all take a single argument **e**. They return **t** if **e** is an element of the specified type, and **nil** otherwise. The indentation indicates a subtype relation. For example, if **number-node?** is true of an element, **primitive-node?**, **indv-node?**, and **node?** will be true as well.

```
node? (e)
  indv-node? (e)
    proper-indv-node? (e)
    generic-indv-node? (e)
    defined-indv-node? (e)
    primitive-node? (e)
      number-node? (e)
        integer-node? (e)
        ratio-node? (e)
        float-node? (e)
      string-node? (e)
      function-node? (e)
      struct-node? (e)
    type-node? (e)
  map-node? (e)
    indv-map-node? (e)
    type-map-node? (e)
  role-node? (e)
    indv-role-node? (e)
    type-role-node? (e)

link? (e)
  is-a-link? (e)
  is-not-a-link? (e)
  eq-link? (e)
  not-eg-link? (e)
  has-link? (e)
  has-no-link? (e)
  cancel-link? (e)
  statement? (e)
  not-statement? (e)
  split? (e)
  complete-split? (e)

relation? (e) defined-type-node?
```

The following predicates do not test the basic type of the element **e**, but test some other feature of the element.

defined? (e) *[FUNCTION]*
This predicate tests whether **e** is an element (type-node or indv-node) with a specific definition, rather than just a description.

derived? (e) *[FUNCTION]*
This predicate tests whether **e** is a derived element – that is, one created by the Scone engine rather than specifically by the user.

killed? (e) *[FUNCTION]*
The killed flag on element **e** indicates that it is present but inoperative.

symmetric? (e) *[FUNCTION]*
The symmetric flag on a relation or statement indicates that the relation holds in both directions.

transitive? (e) [FUNCTION]
The transitive flag on a role or statement indicates that this is a transitive relation.

9.2 Predicates on the Is-A Hierarchy

These predicates test for subtype, instance, and equality relations in the is-a hierarchy.

Functions and Variables

is-x-a-y? (x y) [FUNCTION]
The **x** and **y** arguments are elements. This is a predicate to determine whether **x** is known to be an instance or subtype of **y** in the current context. A return of **:yes** means **x** is definitely a **y**. **:no** means that **x** is definitely not a **y**, and stating otherwise would cause an error. **:maybe** means that we have no definite knowledge either way.

A **:no** response will return an element as second value. This is the particular split or negation link that produces a conflict, or one of these elements if there are multiple conflicts.

simple-is-x-a-y? (x y) [FUNCTION]
The **x** and **y** arguments are elements. This is a predicate to determine whether **x** is definitely known to be an instance or subtype of **y** in the current context. Returns T or NIL.

is-x-eq-y? (x y) [FUNCTION]
The **x** and **y** arguments are elements. This is a predicate to determine whether **x** is known to be equivalent to **y** in the current context. A return of **:yes** means **x** is definitely equivalent to **y**. **:no** means that **x** is definitely not equivalent to **y**, and stating otherwise would cause an error. **:maybe** means that we have no definite knowledge either way.

A **:no** response will return an element as second value. This is the particular split or negation link that produces a conflict, or one of these elements if there are multiple conflicts.

simple-is-x-eq-y? (x y) [FUNCTION]
The **x** and **y** arguments are elements. This is a predicate to determine whether **x** is definitely known to be equivalent to **y** in the current context. Returns T or NIL.

9.3 Mark, List, and Show Functions

Most of these query functions have three versions: a *mark* version, which marks all the members of some class (such as "instances of mammal") with a specified marker; a *list* version, which returns a list of such elements, suitable for processing by an external Lisp function; and a *show* version that prints the set of elements in a human-friendly format.

The "show" functions all take some additional keyword arguments that control the printing:

The **:n** argument, an integer, is the maximum number of items that will be printed. The default is 100.

An **:all** argument of **t** over-rides the **:n** argument and forces printing of all the elements in the result set. The default is **nil**.

The **:last** argument, an integer, says to print the last **n** arguments in the set rather than the first **n**.

Normally you call a show function with only one of **:n**, **:all**, or **:last**.

Functions and Variables

list-elements nil *[FUNCTION]*

show-elements (&key :n :all :last) *[FUNCTION]*

List or print all the elements in the KB, in the order in which the elements were defined.

list-marked (m) *[FUNCTION]*

show-marked (m &key :n :all :last :heading) *[FUNCTION]*

List or print all the elements marked with marker **m**, in the order in which the markers were placed.

list-not-marked (m) *[FUNCTION]*

show-not-marked (m &key :n :all :last) *[FUNCTION]*

List or print all the elements not marked with marker **m**.

mark-supertypes (e m) *[FUNCTION]*

list-supertypes (e) *[FUNCTION]*

show-supertypes (e &key :n :all :last) *[FUNCTION]*

Mark/list/print all the types above **e** in the is-a hierarchy (in the current context).

mark-subtypes (e m) *[FUNCTION]*

list-subtypes (e) *[FUNCTION]*

show-subtypes (e &key :n :all :last) *[FUNCTION]*

Mark/list/print all the types below **e** in the is-a hierarchy (in the current context).

mark-instances (e m) *[FUNCTION]*

list-instances (e) *[FUNCTION]*

show-instances (e &key :n :all :last) *[FUNCTION]*

Mark/list/print all the instances (indv nodes) under **e** in the current context.

mark-parents (e m) *[FUNCTION]*

list-parents (e) *[FUNCTION]*

show-parents (e &key :n :all :last) *[FUNCTION]*

Mark/list/print the immediate superiors of element **e** in the is-a hierarchy (in the current context).

mark-children (e m) *[FUNCTION]*

list-children (e) *[FUNCTION]*

show-children (e &key :n :all :last) *[FUNCTION]*

Mark/list/print the immediate descendants of element **e** in the is-a hierarchy (in the current context).

mark-superiors (e m) [FUNCTION]
list-superiors (e) [FUNCTION]
show-superiors (e &key :n :all :last) [FUNCTION]
 Mark/list/print all the elements, regardless of type, above **e** in the is-a hierarchy (in the current context).

mark-inferiors (e m) [FUNCTION]
list-inferiors (e) [FUNCTION]
show-inferiors (e &key :n :all :last) [FUNCTION]
 Mark/list/print all the elements, regardless of type, below **e** in the is-a hierarchy (in the current context).

mark-equals (e m) [FUNCTION]
list-equals (e) [FUNCTION]
show-equals (e &key :n :all :last) [FUNCTION]
 Mark/list/print all the elements, regardless of type, that are equivalent to **e** (i.e. connected by an EQ-link or chain of them) in the is-a hierarchy (in the current context).

mark-intersection (superiors m) [FUNCTION]
list-intersection (superiors) [FUNCTION]
show-intersection (superiors &key :n :all :last) [FUNCTION]
 Given list of **superiors**, which should all be type-nodes, mark/list/print every element that is in the intersection of all these superior types. Return the count.

mark-lowermost (m1 m2) [FUNCTION]
list-lowermost (m) [FUNCTION]
show-lowermost (m &key :n :all :last) [FUNCTION]
 The **mark-lowermost** function examines the set of elements currently marked with **m1** and places **m2** on the lowermost elements of that set – that is, on all elements marked with **m1** that do not have an immediate descendant that is also marked with **m1**. The **list-lowermost** and **show-lowermost** functions list/print the lowermost elements in the set marked with **m**.

mark-uppermost (m1 m2) [FUNCTION]
list-uppermost (m) [FUNCTION]
show-uppermost (m &key :n :all :last) [FUNCTION]
 The **mark-uppermost** function examines the set of elements currently marked with **m1** and places **m2** on the uppermost elements of that set – that is, on all elements marked with **m1** that do not have an immediate superior that is also marked with **m1**. The **list-uppermost** and **show-uppermost** functions list/print the uppermost elements in the set marked with **m**.

mark-proper (m1 m2) [FUNCTION]
list-proper (m) [FUNCTION]
show-proper (m &key :n :all :last) [FUNCTION]
 From the set of elements currently marked with **M** (or **M1**), mark/list/print any proper individual nodes that are lowermost in the type hierarchy.

mark-most-specific (m1 m2) [FUNCTION]
list-most-specific (m) [FUNCTION]
show-most-specific (m &key :n :all :last :heading) [FUNCTION]

The **mark-most-specific** function examines the set of elements currently marked with **m1** and places **m2** on the most specific elements of that set – that is, any proper individual nodes and, if none of those are present, the lowermost type or generic nodes.. The **list-most-specific** and **show-most-specific** functions list/print the most specific elements in the set marked with **m**.

acceptable-role-filler (m) [FUNCTION]

From the set of nodes currently marked with **M**, find the one that would serve best as an answer for **THE-X-OF-Y** and similar functions. Currently we accept lowermost proper individuals, types, and statements. If more than one exists, pick one at random; if none, return **NIL**.

9.4 Query Functions for Roles and Relations

mark-role (role owner m &key :downscan :augment) [FUNCTION]

OWNER is any element. **ROLE** is a node that is a role of **OWNER**, directly or by inheritance. Place marker **M** on all nodes **X** such that '**X** is the **ROLE** of **OWNER**'. If **:AUGMENT**, don't clear **M** before marking new nodes. Downscan the **M** markers unless **:DOWNSCAN** is explicitly **NIL**. Returns the number of elements ultimately marked with **M**.

mark-role-inverse (role player m &key :downscan :augment) [FUNCTION]

Place marker **M** on all nodes **X** such that '**PLAYER** is the **ROLE** of **X**'. If **:AUGMENT**, don't clear **M** before marking new nodes. Downscan the **M** markers unless **:DOWNSCAN** is explicitly **NIL**. Returns the number of elements ultimately marked with **M**.

mark-the-x-of-y (x y m) [FUNCTION]

Mark with **M** all elements **E** that collectively represent the **X** of **Y**. Do not propagate **M** down to subtypes and instances of this role.

mark-all-x-of-y (x y m) [FUNCTION]

list-all-x-of-y (x y) [FUNCTION]

show-all-x-of-y (x y &key :n :all :last) [FUNCTION]

Print out all the **X** of **Y**.

the-x-of-y (x y) [FUNCTION]

Find the most specific proper node representing the **X** of **Y**, and return that node if it exists. Else return **NIL**.

mark-the-x-inverse-of-y (x y m) [FUNCTION]

Mark with **M** all elements **E** such that the **X** of **E** is **Y**. Do not propagate **M** down to subtypes and instances of this role.

mark-all-x-inverse-of-y (x y m) [FUNCTION]
list-all-x-inverse-of-y (x y) [FUNCTION]
show-all-x-inverse-of-y (x y &key :n :all :last) [FUNCTION]
 Mark with **m**, list, or print out all nodes **e** such that the **x** of **e** is **y**.

the-x-inverse-of-y (x y) [FUNCTION]
 Find the most specific proper node E such that the X of E is Y. If no such E exists, return NIL. If there are more than one E, return one of them.

mark-roles (e m) [FUNCTION]
list-roles (e) [FUNCTION]
show-roles (e &key :n :all :last) [FUNCTION]
 Mark with **m**, list, or print out all the roles defined for element **e** in the current context, whether or not those roles are filled.

the-x-role-of-y (x y) [FUNCTION]
 Find and return the role-node or map-node directly representing the X-role of Y. If this does not exist, create it.

can-x-have-a-y? (x y) [FUNCTION]
 Predicate to determine whether element X can have a Y role in the current context. That is, return T if Y is a role of X or some node above X, else NIL.

can-x-be-the-y-of-z? (x y z) [FUNCTION]
 Predicate to determine whether it would be legal to state that X is the Y of Z. That is, would any splits or other constraints be violated? If it is OK, return T. If it is not, return NIL with the unhappy split or constraint as the second return value. If there are multiple problems, we just return the first one we hit.

can-x-be-a-y-of-z? (x y z) [FUNCTION]
 Predicate to determine whether it would be legal to state that X is a Y of Z. That is, would any splits or other constraints be violated? If it is OK, return T. If it is not, return NIL with the unhappy split or constraint as the second return value. If there are multiple problems, we just return the first one we hit.

statement-true? (a rel b) [FUNCTION]
 A predicate that tests whether "**a rel b**" is true in the current context. If so, the second return value is the link that actually states this fact (if there is a single link that does this). The third return value is that link's c-node, if any. If the predicate is false, return only the single value **nil**.

mark-rel (rel a m &key :downscan :augment :recursion-allowance) [FUNCTION]
list-rel (rel a) [FUNCTION]
show-rel (rel a &key :n :all :last) [FUNCTION]
 In **mark-rel**, the **a** argument is an element. The **rel** argument is a relation-node. The

mark-rel function places marker **m** on all elements **b** for which the statement "**a rel b**" is known to be true in the current context. Then downscan marker **m** unless the **:downscan** argument is explicitly **nil**. **:recursion-allowance** controls the number of levels of descriptions we will explore. Returns the number of elements marked with **m** after this operation. The **list-rel** and **show-rel** functions list and print the set of elements that **mark-rel** would mark.

mark-rel-inverse (rel a m &key :downscan :augment :recursion-allowance) *[FUNCTION]*

list-rel-inverse (rel b) *[FUNCTION]*

show-rel-inverse (rel b &key :n :all :last) *[FUNCTION]*

Like **mark-rel** and friends, but here we cross the statement links in the opposite direction. That is, we mark/list/print all elements **a** such that "**a rel b**".

9.5 Miscellaneous Show Functions

show-element-counts nil *[FUNCTION]*

Print a table showing the number of elements of each type.

show-marker-counts nil *[FUNCTION]*

Prints the number of elements marked with each marker.

show-ontology (&key :under :n) *[FUNCTION]*

Print every node in the ontology beneath **start**. This output is designed to be read by humans, not programs. If the **:under** argument (a Scone element) is supplied, print only the part of the ontology under this element. If **:n** is supplied, stop after N elements are printed.

10. Marker Scans

The various marker scans are powerful and rather complex internal functions that perform the pseudo-parallel search and inference in Scone. Once Scone is mature, users will seldom have to deal directly with these scans, but for now they provide the means for writing any search and inference functions that Scone does not supply.

Functions and Variables

upscan (start-element *m* &key :mode :m1 :augment :restrict-m) [FUNCTION]

Mark **start-element** with marker *m*. Propagate this marker upwards, across parent wires and is-a links that are active in the current context, while respecting is-not-a links and cancellations. Eq-links are crossed in both directions. Places the cancel-marker corresponding to *m* on nodes that explicitly are not considered to be superiors of start-element.

Returns the number of elements marked with *m* after the **upscan**

Normally **upscan** clears the *m* marker and its cancel-marker before start-element is marked and the scan is performed. However, if the **:augment** argument is *t*, we leave any pre-existing *m* markers in place and they are propagated upward as well.

The **mode** argument is one of **:cross-map**, **:gate-map**, or **:no-map**. The default is **:cross-map**. The others are specialized scan types with respect to propagating across map connections. For some modes, we need an additional marker, which is supplied as **:m1**.

The **:restrict-m** argument, if present, is a marker already present on some elements in the network. In this scan we place *m* only on elements that already have **restrict-m**. Returns the number of elements marked with *m* after the scan.

The operation of upscan and the other scans may be modified by the ***one-step***, ***ignore-context***, and ***default-recursion-allowance*** variables.

ignore-context [VARIABLE]

If *t*, marker-propagations pay no attention to whether or not the nodes and links are in an active context. This variable is used for upscanning while activating a context, and may also be useful finding whether some entity exists in *any* context.

one-step [VARIABLE]

If *t*, marker propagations go only one level in the desired direction. Used in functions such as **mark-parents**, where you don't want the full transitive closure.

default-recursion-allowance [VARIABLE]

In functions such as **mark-rel**, we must individually activate and explore each description instance in which the source node plays a role, and this process may recurse. This is the default allowance for recursion depth that we will explore.

downscan (start-element *m* &key :mode :ml :augment :restrict-m) *[FUNCTION]*
Like upscan, but propagates marker *m* and its cancel-marker downward from **start-element**. That is, the direction is from a type to its subtypes and instances.

eq-scan (start-element *m* &key :mode :ml :augment :restrict-m) *[FUNCTION]*
Like upscan, but propagates marker *m* only to elements that are equivalent to **start-element**. That is, this does not propagate upward or downward in the type hierarchy, but only crosses eq links.

lowermost? (*e m*) *[FUNCTION]*
Assume element *e* is marked with marker *m*. Determine whether *e* is one of the lowermost nodes in this marked set – that is, it has no immediate children also marked with *m*.

uppermost? (*e m*) *[FUNCTION]*
Like **lowermost?**, but determines whether *e* is an uppermost member of the set marked with *m*.

splits-ok? (*m*) *[FUNCTION]*
Some set of elements has been marked with *m* and upscanned. Look for splits that have *m* on more than one of the disjoint items. If there are none, return **t**, meaning OK. If we find an unhappy split, return **nil**, returning the unhappy split element (or one of them, if there are more than one) as the second return value.

which-split-member? (*e split*) *[FUNCTION]*
Given an element *e* and a split-node **split**, finds which of the split type nodes *e* is an inferior of. If none of the types is a supertype of *e*, returned **nil**.

11. Miscellaneous Functions and Variables

11.1 Commentary, Printing, and Naming

commentary (&rest args) *[FUNCTION]*

This function is identical in form to the Common Lisp **format** function, but without a stream argument. The **commentary** function prints to ***commentary-stream***, which may or may not be equivalent to ***standard-output***. If that variable is **nil**, then **commentary** doesn't print at all.

The intent is that **commentary** should be used to inform the user when the system has done something interesting, but with the capability to turn off all such comments in non-interactive situations or if the user doesn't want to see all this information.

commentary-stream *[VARIABLE]*

Stream to which **commentary** prints its output.

print-namespace-in-elements *[VARIABLE]*

If **t**, always include the namespace, followed by a colon, in element inames. This ensures that what we print can be read back in without any namespace ambiguity. If **nil**, do not include the namespace. This is better for human readability. If **:maybe**, print the namespace if it differs from the current ***namespace***. This is usually safe.

generate-long-element-names *[VARIABLE]*

If **t**, generate long names for certain element types. For example, a statement link will have an internal name that is an English description of arguments and the relation. This is good for development, but significantly increases the storage requirement for large knowledge bases.

print-keylist *[VARIABLE]*

If **t**, dumping an element pattern includes the list of keyword/value pairs as part of the pattern. That is necessary if we want to read that pattern back in, but we set this to **nil** for some human-readable dumps.

show-ontology-indent *[VARIABLE]*

An integer, the number of spaces to indent each level in **show-ontology**.

show-stream *[VARIABLE]*

Stream to which we print the output of **show** functions. If **nil**, don't print these at all.

11.2 Variables Linking to Essential Scone Elements

There are a number of elements set up in the **bootstrap.lisp** KB file that are referred to by code in the Scone engine. For convenience, we store each of these elements in a global variable so that it is easy for engine and user code to reference these elements. So, for example, a pointer to the element **{thing}** is stored in the variable ***thing***.

thing

The uppermost node in the is-a hierarchy.

universal

The pre-defined {**universal**} context.

general

The pre-defined {**general**} context.

set

empty-set

non-empty-set

Pre-defined type nodes related to the concept of {**set**}.

cardinality

The {**cardinality**} role for the {**set**} node.

zero

one

two

Some pre-defined number nodes.

number

integer

ratio

float

string

function

struct

relation

is-a-link

is-not-a-link

eq-link

not-eq-link

cancel-link

has-link

has-no-link

split

complete-split

statement-link

not-statement-link

These type-nodes represent the various element-types built into Scone.

12. Structure-Creation Functions from the Core KB

These commonly-used functions for creating new network structure are not defined in the Scone engine file, but instead are defined in the files contained in the “core” knowledge base. They must be defined as part of the knowledge base because they build upon element structures created as the KB files are being loaded.

Functions and Variables

new-measure (magnitude unit) [FUNCTION]

This function, defined in the “units” file, creates and returns an individual of type **{measure}**, which is the combination of a **magnitude** (i.e. a number) and a **unit**, which should be an instance of **{unit}**. So the measure represents something like “16.0 ton”. The **magnitude** may be an integer, ratio, or a float, or a Scone object of type **{number}**. A specific sub-type of **{measure}** is chosen based on the type of **unit**.

Note that the **unit** should be a singular form: “**ton**” rather than “**tons**”. Scone does not yet have the machinery to deal smoothly with plurals and other natural-language morphology.

new-quantity (magnitude unit stuff) [FUNCTION]

This is like **new-measure**, but takes a third argument **stuff**, which should be a subtype of the type **{stuff}**, which is defined in the “units” file. It creates and returns an element representing something like “16.0 ton of coal” or “10 megabytes of census-data”. Note that **{stuff}** represents any quality measurable by a unit, and not necessarily a physical substance, which in Scone is a **{material}**.

13. Alphabetical Index of Functions and Variables

activation-cancel-marker	26	*n-available-markers*	25
activation-marker	26	*n-elements*	13
cancel-link	50	*no-kb-error-checking*	9
cardinality	50	*non-empty-set*	50
check-defined-types	9	*not-eq-link*	50
commentary-stream	49	*not-statement-link*	50
comment-on-defined-types	9	*number*	50
comment-on-element-creation	9	*one*	50
complete-split	50	*one-step*	47
context	26	*print-keylist*	49
context-cancel-marker	26	*print-namespace-in-elements*	49
context-marker	26	*ratio*	50
create-undefined-elements	9	*relation*	50
currently-loading-files	8	*reloading-kb-file*	9
deduce-owner-type-from-role	9	*set*	50
default-kb-pathname	8	*show-ontology-indent*	49
default-recursion-allowance	47	*show-stream*	49
deferred-connections	10	*split*	50
defer-unknown-connections	10	*statement-link*	50
disambiguate-policy	9	*string*	50
empty-set	50	*struct*	50
eq-link	50	*thing*	50
first-element	13	*two*	50
float	50	*universal*	50
function	50	*verbose-loading*	8
general	50	*zero*	50
generate-long-element-names	49	acceptable-role-filler	44
has-link	50	add-to-split	36
has-no-link	50	a-element	14
ignore-context	47	b-element	14
integer	50	cancel-link?	40
is-a-link	50	can-x-be-a-y-of-z?	45
is-not-a-link	50	can-x-be-the-y-of-z?	45
last-element	13	can-x-have-a-y?	45
legal-syntax-tags	20	c-element	14
loaded-files	8	check-legal-marker	23
loading-kb-file	8	check-legal-marker-pair	23
namespace	18		
namespaces	18		

check-loaded-files	11	incoming-split-elements.....	14
checkpoint-kb	11	in-context	26
checkpoint-new	11	indv-map-node?	40
clear-all-markers.....	24	indv-node?	40
clear-element-property	15	indv-role-node?.....	40
clear-marker	24	in-namespace	18
clear-marker-pair.....	24	integer-node?.....	40
commentary	49	is-a-link?	40
complete-split?.....	40	is-not-a-link?	40
context-element.....	14	is-x-a-y?	41
convert-marker.....	25	killed?	40
convert-parent-wire-to-link.....	14	legal-marker?	23
defined?.....	40	legal-marker-pair?.....	23
defined-indv-node?	40	link?	40
derived?.....	40	list-all-x-inverse-of-y.....	45
disambiguate	21	list-all-x-of-y	44
do-elements	13	list-children	42
do-marked	24	list-dictionary-entries	21
downscan	48	list-elements.....	42
english	20	list-equals	43
eq-link?	40	list-inferiors	43
eq-scan	48	list-instances.....	42
free-marker	25	list-intersection	43
free-markers.....	25	list-lowermost	43
function-node?	40	list-marked	42
generic-indv-node?	40	list-most-specific	44
get-cancel-marker	23	list-namespaces.....	18
get-element-property.....	15	list-not-marked.....	42
get-english-names	21	list-parents	42
get-marker.....	25	list-proper.....	43
get-set-node	31	list-rel	45
get-type-node.....	31	list-rel-inverse	46
has-link?	40	list-roles.....	45
has-no-link?	40	list-subtypes	42
incoming-a-elements.....	14	list-superiors.....	43
incoming-b-elements	14	list-supertypes.....	42
incoming-c-elements.....	14	list-synonyms	21
incoming-context-elements	14	list-uppermost.....	43
incoming-parent-elements.....	14	load-kb	8
		lookup-definitions	20
		lookup-element.....	18
		lookup-element-or-defer	18

lookup-element-test.....	18	new-indv-role	37
lowermost?.....	48	new-integer	30
map-node?	40	new-intersection-type.....	32
mark	23	new-is-a.....	32
mark-all-x-inverse-of-y	45	new-is-not-a	33
mark-all-x-of-y	44	new-map	32
mark-boolean	24	new-measure	51
mark-children	42	new-members	37
mark-equals.....	43	new-not-statement	35
marker-bit.....	23	new-number.....	30
marker-count.....	23	new-quantity.....	51
marker-off?.....	24	new-ratio.....	30
marker-on?.....	24	new-relation.....	35
mark-inferiors.....	43	new-split.....	36
mark-instances	42	new-split-subtypes	37
mark-intersection	43	new-statement.....	35
mark-lowermost	43	new-string	30
mark-most-specific.....	44	new-struct	30
mark-named-elements	21	new-type	31
mark-parents	42	new-type-role.....	38
mark-proper	43	new-union-type.....	32
mark-rel	45	next-element.....	14
mark-rel-inverse	46	n-marker-pairs.....	23
mark-role	44	n-markers	23
mark-role-inverse	44	node?	40
mark-roles	45	node-value	30
mark-subtypes.....	42	not-eq-link?.....	40
mark-superiors	43	not-statement?	40
mark-supertypes	42	number-node?	40
mark-the-x-inverse-of-y.....	44	parent-element	14
mark-the-x-of-y	44	previous-element	14
mark-uppermost	43	primitive-node?.....	40
new-cancel	34	process-deferred-connections	10
new-complete-members	37	proper-indv-node?	40
new-complete-split	36	push-element-property.....	15
new-complete-split-subtypes.....	37	ratio-node?.....	40
new-context	26	refresh-context.....	26
new-defined-type	31	relation?	40
new-eq.....	33	remove-context.....	39
new-float	30	remove-currently-loading-file	39
new-function	30	remove-element.....	39
new-indv.....	29	remove-elements-after	39

remove-last-element.....	39	split?.....	40
remove-last-loaded-file	39	split-elements.....	14
role-node?	40	splits-ok?	48
set-element-property	15	statement?	40
show-all-x-inverse-of-y	45	statement-true?	45
show-all-x-of-y.....	44	string-node?	40
show-children.....	42	struct-node?	40
show-element-counts.....	46	symmetric?.....	40
show-elements	42	the-x-inverse-of-y.....	45
show-equals	43	the-x-of-y.....	44
show-inferiors	43	the-x-of-y-is-a-z	38
show-instances	42	the-x-of-y-is-not-a-z	38
show-intersection.....	43	the-x-role-of-y	45
show-lowermost.....	43	transitive?.....	41
show-marked.....	42	type-map-node?.....	40
show-marker-counts	46	type-node?	40
show-most-specific.....	44	type-role-node?	40
show-not-marked	42	unmark	23
show-ontology.....	46	unmark-boolean.....	24
show-parents.....	42	uppermost?.....	48
show-proper	43	upscan	47
show-rel.....	45	which-split-member?	48
show-rel-inverse.....	46	with-markers	25
show-roles	45	x-is-a-y-of-z.....	38
show-subtypes	42	x-is-not-a-y-of-z	38
show-superiors	43	x-is-not-the-y-of-z.....	38
show-supertypes	42	x-is-the-y-of-z	38
show-synonyms.....	21		
show-uppermost	43		
simple-is-x-a-y?.....	41		
simple-is-x-eq-y?	41		