PowerLoom™ Manual

Powerful knowledge representation and reasoning with
delivery in Common-Lisp, Java, and C++

This manual describes
PowerLoom 3.0 or later.

30 September 2003

The PowerLoom development team
Hans Chalupsky
Robert M. MacGregor
Thomas Russ
{hans,tar}@isi.edu

1 Introduction

This document describes the PowerLoom knowledge representation and reasoning system. PowerLoom is the successor to the Loom knowledge representation system. It provides a language and environment for constructing intelligent applications. PowerLoom uses a fully expressive, logic-based representation language (a variant of KIF). It uses a Prolog-technology backward chainer as its deductive component. The backward chainer is not (yet) a complete theorem prover, but it can handle Horn rules, negation, and simple equality reasoning. Contrary to Prolog, it also handles recursive rules without the risk of infinite recursion. An alternative reasoner, PowerLoom's description classifier, uses technology derived from the Loom classifier to classify descriptions expressed in full first order predicate calculus. PowerLoom uses modules as a structuring device for knowledge bases, and ultra-lightweight worlds to support hypothetical reasoning.

To implement PowerLoom we developed a new programming language called STELLA, which is a Strongly Typed, Lisp-like LAnguage that can be translated into Lisp, Java, and C++. STELLA tries to preserve those features of Lisp that facilitate symbolic programming and rapid prototyping, while still allowing translation into readable as well as efficient Java and C++ code. Because PowerLoom in written STELLA, we are able to deliver it in all three languages.

2 Powerloom History

<to be written>

3 Installation

3.1 System Requirements

To install and use PowerLoom you'll approximately need the following amounts of disk space:

- 15 MB for the tar-red or zip-ped archive file
- 60 MB for the untarred sources, tanslations, compiled Java files and documentation
- 14 MB to compile a Lisp version
- 15 MB to compile the C++ version (without -g)
- 4 MB to compile the Java version (already included)

This means that you will need approximately 90 MB to work with one Lisp, one C++ and one Java version of PowerLoom in parallel. If you also want to experiment with the Lisp translation variant that uses structures instead of CLOS instances to implement STELLA objects, then you will need an extra 15 MB to compile that.

The full PowerLoom development tree is quite large, since for every STELLA source file there are three to four translated versions and as many or more compiled versions thereof. The actual PowerLoom libraries that you have to ship with an application, however, are quite small. For example, the Java jar file '`powerloom.jar`' is only 2.2 MB (4 MB including Java sources). The dynamic C++ libraries '`libstella.so`' and '`liblogic.so`' compiled on a Linux platform are about 7 MB total. Additionally, if you don't need all the different translations of PowerLoom, you can delete some of the versions to keep your development tree smaller See Section 3.6 [Removing Unneeded Files], page 6.

To run the Lisp version of PowerLoom you need an ANSI Common-Lisp (or at least one that supports CLOS and logical pathnames). We have successfully tested PowerLoom with Allegro-CL 4.2, 4.3, 5.0, 6.0 and 6.1, Macintosh CL 3.0 and 4.0, Lucid CL 4.1 (plus the necessary ANSI extensions and Mark Kantrowitz's logical pathnames implementation) and the freely available CMUCL 18c. Our main development platform is Allegro CL running under Sun Solaris and Linux RedHat, so, the closer your environment is to ours, the higher are the chances that everything will work right out of the box. Lisp development under Windows is also not a problem.

To run the C++ version of PowerLoom you need a C++ compiler such as g++ that supports templates and exception handling. We have successfully compiled and run PowerLoom with g++ 3.2 under Linux Redhat 8.0, and with CygWin 5.0 under Windows 2000 (CygWin provides a very Unix-like environment). We have not yet tried to run the C++ version fully natively under Windows. The main portability issue is the garbage collector. It is supposed to be very portable and run natively on Windows platforms, but we have never verified that.

For the Java version you will need Java JDK 1.2 or later. To get reasonable performance, you should use JDK 1.3 or later. We've run the Java version of PowerLoom on a variety of platforms without any problems.

Any one of the Lisp, C++ or Java implementations of PowerLoom can be used to develop your own PowerLoom-based applications. Which one you choose is primarily a matter of your application and programming environment. The Lisp and Java versions are comparable in speed, the C++ version is usually a factor of 2-3 faster than Lisp or Java.

3.2 Unpacking the Sources

Uncompress and untar the file '`powerloom-X.Y.Z.tar.gz`' (or unzip the file '`powerloom-X.Y.Z.zip`') in the parent directory of where you want to install PowerLoom ('X.Y.Z' are place holders for the actual version numbers). This will create the PowerLoom tree in the directory '`powerloom-X.Y.Z/`' (we will use Unix syntax for pathnames). All pathnames mentioned below will be relative to that directory which we will usually refer to as the "PowerLoom directory".

3.3 Lisp Installation

To install the Lisp version of PowerLoom startup Lisp and load the file '`load-powerloom.lisp`' with:

```
(CL:load "load-powerloom.lisp")
```

The first time around this will compile all Lisp-translated STELLA files before they are loaded. During subsequent sessions the compiled files will be loaded right away.

If you want to use the version that uses Lisp structs instead of CLOS objects to implement STELLA objects do the following:

```
(CL:setq cl-user::*load-cl-struct-stella?* CL:t)
(CL:load "load-powerloom.lisp")
```

Alternatively, you can edit the initial value of the variable `*load-cl-struct-stella?*` in the file '`load-stella.lisp`'. Using structs instead of CLOS objects greatly improves slot access speed, however, it may cause problems with incremental re-definition of STELLA classes (this is only an issue if you are developing your application code in the STELLA language. In that case it is recommended to only use the struct option for systems that are in or near the production stage).

Once all the files are loaded, you should see a message similar to this:

```
PowerLoom 3.0.0 loaded.
Type `(powerloom)' to get started.
Type `(in-package "STELLA")' to run PowerLoom commands directly
   from the Lisp top level.
USER(2):
```

To reduce startup time, you might want to create a Lisp image that has all of PowerLoom preloaded.

Now type

```
(in-package "STELLA")
```

to enter the `STELLA` Lisp package where all the PowerLoom code resides. Alternatively, you can type

```
(powerloom)
```

which will bring up a PowerLoom listener that will allow you to execute PowerLoom commands.

IMPORTANT: All unqualified Lisp symbols in this document are assumed to be in the `STELLA` Lisp package. Moreover, the `STELLA` package does **NOT** inherit anything from the `COMMON-LISP` package (see the file '`sources/stella/cl-lib/cl-setup.lisp`' for the few exceptions), hence, you have to explicitly qualify every Lisp symbol you want to use with `CL:`. For example, to get the result of the previous evaluation you have to type `CL:*` instead of `*`.

3.4 C++ Installation

To compile the C++ version of PowerLoom change to the native C++ directory and run `make`:

```
% cd native/cpp/logic
% make
```

This will compile all PowerLoom and STELLA files, the C++ garbage collector and generate static or dynamic '`libstella`' and '`liblogic`' library files in the directory '`native/cpp/lib`' which can later be linked with your own C++-translated PowerLoom (or other) code. To test whether the compilation was successful you can run PowerLoom from the top-level PowerLoom directory like this:

```
% ./native/cpp/logic/logic
Initializing STELLA...
Initializing PowerLoom...

    Welcome to PowerLoom 3.0.0

Copyright (C) USC Information Sciences Institute, 1997-2003.
PowerLoom comes with ABSOLUTELY NO WARRANTY!
Type '(copyright)' for detailed copyright information.
Type '(help)' for a list of available commands.
Type '(demo)' for a list of example applications.
Type 'bye', 'exit', 'halt', 'quit', or 'stop', to exit.


|=
```

This will run various PowerLoom startup code and then bring up a PowerLoom command loop where you can execute commands. Type

```
(demo)
```

to bring up a menu of available demos, type

```
(run-powerloom-tests)
```

to run the PowerLoom test suite, or type

```
exit
```

to exit PowerLoom.

3.5 Java Installation

Nothing needs to be done to install the Java version. Since Java class files are platform independent, they are already shipped with the PowerLoom distribution and can be found in the directory 'native/java' and its subdirectories. Additionally, they have been collected into the file 'powerloom.jar' in the PowerLoom directory. To try out the Java version of PowerLoom run the following in the PowerLoom directory:

```
% java -jar powerloom.jar
Initializing STELLA...
Initializing PowerLoom...

    Welcome to PowerLoom 3.0.0

Copyright (C) USC Information Sciences Institute, 1997-2003.
PowerLoom comes with ABSOLUTELY NO WARRANTY!
Type '(copyright)' for detailed copyright information.
Type '(help)' for a list of available commands.
Type '(demo)' for a list of example applications.
Type 'bye', 'exit', 'halt', 'quit', or 'stop', to exit.


|=
```

Similar to the C++ executable, this will run various PowerLoom startup code and then bring up a PowerLoom command loop where you can execute commands. Type

```
(demo)
```

to bring up a menu of available demos, type

```
(run-powerloom-tests)
```

to run the PowerLoom test suite, or type

```
exit
```

to exit PowerLoom.

3.6 Removing Unneeded Files

To save disk space you can remove files that you don't need. For example, if you are not interested in the C++ version of PowerLoom, you can delete the directory 'native/cpp'. Similarly, you can remove 'native/java' to eliminate all Java-related files. You could do the same thing for the Lisp directory 'native/lisp', but (in our opinion) that would make it less convenient for you to develop new PowerLoom code that is written in STELLA. Finally, if you don't need any of the STELLA sources, you can delete the directory 'sources/stella'. If you don't need local copies of the STELLA and PowerLoom documentation, you can delete parts or all of the directories 'sources/stella/doc' and 'sources/logic/doc'.

4 Conceptual Framework

This chapter presents the fundamental conceptual building blocks that are used to construct PowerLoom knowledge bases. The PowerLoom language is based on KIF, which provides a syntax and a declarative semantics for first-order predicate calculus expressions. KIF is a proposed ANSII standard language used by a variety of knowledge representation systems. Practical knowledge representation systems necessarily add a procedural semantics that defines the interpretation of knowledge structures when definitions and facts are retracted or modified. This chapter assumes that the reader has some familiarity with the semantics of the predicate calculus, and instead focuses on aspects of the semantics that go beyond the traditional (KIF) semantics.

A PowerLoom knowledge base is constructed by first defining the terminology (concepts and relations) for a domain, and then asserting additional rules and facts about that domain. Facts can be asserted and later retracted, so the answers returned by queries may change over time. The knowledge structures are organized into logical containers called "modules". The division into modules means that in general, facts are not asserted globally, but instead hold only within a specific context. For example, a logical proposition may evaluate as true within one module, and evaluate as false within a different one.

The discussion below uses some examples of actual PowerLoom syntax to illustrate certain points. However, we gloss over the fine points of syntax, and instead focus on semantic issues. The next chapter reverses that emphasis, and presents a series of examples that illustrate the breadth of syntactic constructs implemented for the PowerLoom language.

4.1 Terms and Propositions

A knowledge base attempts to capture in abstract (machine interpretable) form a useful representation of a physical or virtual world. The entities in that world are modeled in the knowledge base by objects we call *terms*. Examples of terms are "Georgia" (denoting the U.S., state), "BenjaminFranklin" (denoting the historical person by that name), the number three, the string `"abc"`, and the concept "Person". Unlike objects in an object-oriented programming language, the terms in a PowerLoom knowledge base usually have distinct names (unless there are sufficiently many that naming them all becomes impractical).

Terms are categorized or related to one another by objects called *relations*. Examples of relations are "has age", "greater than", "is married to", "plus". Concepts such as "Person", "State", "Company", and "Number" are considered a subcategory of relations.

A *proposition* is a logical sentence that has an associated truth value. Examples are "Ben Franklin is a person", "Bill is married to Hillary", "Two plus three equals six". PowerLoom follows KIF in adopting a prefix notation for the predicate calculus to represent propositions. Possible representations of the three propositions just mentioned are `(person ben-franklin)`, `(married-to Bill Hillary)`, and `(= (+ 2 3) 6)`. These three propositions make reference to relations named `person`, `married-to`, `plus`, and `=`.

The predicate calculus constructs complex sentences out of simpler ones using the logical connectives `and`, `or`, `not`, `<=`, `=>`, and `<=>`, and the quantifiers `exists` and `forall`. Some examples are `(not (crook richard))` "Richard is not a crook", and `(forall ?p (person ?p) (exists ?m (has-mother ?p ?m)))` "every person has a mother".

4.2 Definitions

PowerLoom requires that relations are defined before they are used within assertions and queries. The commands `defconcept`, `defrelation`, and `deffunction` are used to define concepts, relations, and functions, respectively. The definitions

```
(defconcept person)
(defrelation married-to ((?p1 person) (?p2 person))
(deffunction + ((?n1 number) (?n2 number)) :-> (?sum number))
```

declare that `person` is a concept, that `married-to` is a binary relation that takes arguments of type `person`, and that `+` is a function that takes arguments of type `number`[1]. The requirement that relations be defined before they are referenced can be inconvenient at times. For example, suppose we wish to define `parent` as "a person who is the parent of another person" and we also wish to state that the first argument to the `parent-of` relation has type `parent`:

```
(defconcept parent (?p)
  :<=> (and (person ?p) (exists ?c (parent-of ?p ?c))))
(defrelation parent-of ((?p parent) (?c person)))
```

In this example, the first reference to `parent-of` occurs before it is defined. PowerLoom permits circular references such as these as long as they occur within definitions. It does so by deferring evaluation of rules that occur within definitions. Here is a specification that is logically equivalent, but is not legal because the `parent-of` relation appears in an assertion before it is defined:

```
(defconcept parent (?p))
(assert (forall (?p) (<=> (parent ?p)
                          (and (person ?p) (exists ?c (parent-of ?p ?c))))))
(defrelation parent-of ((?p parent) (?c person)))
```

So when does the rule inside of the first `parent` definition get defined? All axioms (facts and rules) that appear within the boundaries of a definition are evaluated just prior to the next occurrence of a PowerLoom query. Hence, in the example above where the rule occurred *within* the definition, there was no error because evaluation of that rule occured sometime after the second definition (which defines the otherwise problematic reference to `parent-of`).

One will sometimes see the command `(process-definitions)` appearing at intervals within a file containing PowerLoom commands. Each such appearance forces the definitions that preceed it to be fully-evaluated. This is done so that the interval between a definition and its evaluation not be too great; it can get confusing if PowerLoom reports a semantic violation long after the origin of the conflict.

PowerLoom definitions commands (those prefixed by "def") have one other semantic property that distinguishes them from ordinary assertions. Any axioms that appear within a definition are tied to that definition. If a definition is modified and then reevaluated, axioms that don't survive the modification are retracted. For example, suppose we evaluate the following two commands.

---

[1] The function `+` and the concept `number` are predefined in PowerLoom.

```
(defrelation parent-of ((?p1 person) (?p2 person))
  :=> (relative-of ?p1 ?p2))
(defrelation parent-of ((?p1 person) (?p2 person)))
```

The first definition defines `person` as a binary relation, and also states a rule that "`parent-of` implies `relative-of`". The second definitions erases that rule, i.e., the cumulative effect is as if the first definition did not appear. In contrast, consider the following commands:

```
(defrelation parent-of ((?p1 person) (?p2 person)))
(assert (=> (parent-of ?p1 ?p2) (relative-of ?p1 ?p2)))
(defrelation parent-of ((?p1 person) (?p2 person)))
```

The assertion in this latter sequence is logically equivalent to the axiom introduced by the `:=>` keyword in the former sequence. However, at the end of this sequence, the "`parent-of` implies `relative-of`" rule is still in effect, since it appeared on its own, outside of a definition.

4.3 Truth Values

A PowerLoom proposition is tagged with a truth value that has one of five different settings—`true`, `false`, `default-true`, `default-false`, or `unknown`. The most common setting is `true`; when we make an assertion as in `(assert (Person Bill))`, the proposition `(Person Bill)` is assigned the truth value `true`. To assign the value `false` to a proposition, one asserts that it is not true, e.g., `(assert (not (crook Richard)))`. The command `presume` is used to assign a proposition the value `default-true`, as in `(presume (weather-in Los-Angeles Sunny))`. Presuming a negated proposition assigns it the value `default-false`.

The assignment of a truth value to a proposition via `assert` or `presume` can upgrade the "strength" of a proposition, but it cannot downgrade it. Hence, if a proposition currently has the value `unknown`, then it may be assigned any of the other four values. If the value is `default-true` or `default-false`, an assertion that assigns the value `true` or `false` will overwrite the existing value. However, if the truth value of a proposition is either `true` or `false`, assigning it the value `default-true` or `default-false` will have no effect.

If a proposition is asserted to be `true` and subsequently is asserted to be `false` (or vice-versa), a *clash* (or contradiction) results. When a clash is detected by PowerLoom, a `clash-exception` is thrown. The system's default behavior is for the exception to be caught and ignored, with the result that an assertion that would otherwise cause a clash never takes effect. Applications that execute commands slightly below the top-level (i.e., below the clash exception catcher) can catch the exception themselves and perform a specialized response. PowerLoom's proof-by-contradiction specialist catches clashes to determine that a contradiction has occurred.

If a user or application wants to assign a proposition a truth value that isn't stronger than the current value, it must first `retract` the current value. The PowerLoom `retract` operator has the effect of undoing a prior assertion. For example, if we assert that Mary is a parent of Fred, and then retract that assertion, the value of the proposition `(parent-of Mary Fred)` becomes `unknown`. The proposition can then be assigned any other truth value.

We should note that executing a retraction does not necessarily cause a proposition to cease being true. Consider the following sequence:

```
(defconcept Person)
(defconcept Employee (?e)
  :=> (Person ?e))
(assert (Person Mary))
(assert (Employee Mary))
(retract (Person Mary))
```

If we now ask PowerLoom whether or not Mary is a person, the answer will be yes (TRUE) because Mary is asserted to be an employee, and membership in `employee` implies membership in `person`. In other words, although the direct assertion that Mary is a person is not present in the knowledge base, a logical proof exists that the proposition "Mary is a person" is true.

4.4 Modules

The knowledge loaded into an executing PowerLoom system is divided into logical partitions called "modules". The modules are arranged into a hierarchy; knowledge inherits down the hierarchy from parents to children. A convenient way to organize knowledge is to put definitional knowledge higher up in the module hierarchy, and factual knowledge lower down. For example, suppose we want to build a knowledge base that defines a business domain, and include a substantial number of facts about individual companies. We might use one or a few modules to define terminology that relates to the business domain, and then places the set of facts about each company in its own module. If we were querying the knowledge base about one or a few companies, it would not be necessary to load the modules for the remaining companies into the system.

Facts asserted within a module are not visible in sibling modules, or in ancestor modules. Thus, if we enter into PowerLoom an assertion that"Georgia is a state", we are not asserting that Georgia is a state in all possible worlds, but that, from the vantage point of the current module and those modules below, it is the case that Georgia is a state. If we want the fact that Georgia is a state to be recognized as true in many or most other modules, then we should make our assertion in a module that is relatively high up in the hierarchy, so that is visible to (inherited by) the other modules.

The inheritance of facts is *not monotonic*—a child module can retract or override facts inherited from its ancestors. For example, suppose we have two modules, called `above` and `below` such that the `below` module is below (inherits from) the `above` module. Next, suppose we make an assertion within the `above` module that "Joel is a duck", and then we shift to the `below` module and retract the proposition that "Joel is a duck". From the vantage point of the `below` module, if we now ask if Joel is a duck, we will get back the value `unknown`. However, if we switch to the `above` module and ask the same question, we get back the answer `true` This occurs because the effect of the retraction operation that was applied to the `below` module is not "visible" to modules above it (or to any sibling modules). Hence, when module hierarchies are involved, it is oversimplifying to state that a retraction has the effect of erasing a prior assertion.

The PowerLoom execution process maintains a pointer to the current module, and all asserions, queries, etc. are made relative to that module. Hence, when we talk about "switching" from one module to another, we are speaking literally—a `change-module` command (or one of its equivalents) is invoked to switch from one module to another.[2]

PowerLoom comes with some modules already built-in. The module named `PL-KERNEL` contains a set of general-purpose concept and relation definitions that collectively form the foundation for constructing application-specific knowledge bases. PowerLoom attaches specialized reasoners to many of the relations in `PL-KERNEL`. The command interpreter starts up in a module named `PL-USER`. That module is initially empty, and is intended as a convenient place to experiment with PowerLoom.

[2] Many of the Powerloom API procedures take a module argument that causes a temporary switch to a different module within the scope of that procedure.

5 Annotated Example

The section presents a small example of a PowerLoom knowledge base. It introduces the fundamental PowerLoom modelling concepts and illustrates the syntax of basic PowerLoom declarations, assertions, and commands. This section can be read stand-alone, but readers who intend to use PowerLoom to create their own models are encouraged to load the demo file **???**, and run the examples "live".

The conceptual terms introduced in this section include modules, concepts, relations, functions, instances, propositions, assertions, queries, retraction, positive and negative facts, clipping, rules, and contexts.

5.1 Using Modules

We begin by creating a PowerLoom "module", which is a logical container that holds the term definitions, rules, facts, etc. that make up all or a portion of a domain model. We will call our module `business`. The `defmodule` command defines a new module. The `:includes` option within the `defmodule` tells PowerLoom that the `business` module inherits all definitions and assertions present in the `PL-USER` module, or in ancestor modules inherited by the `PL-USER` module. In particular, by inheriting `pl-user`, we indirectly inherit the `pl-kernel` module that contains all of the built-in concepts and relations. The `in-module` command tells the PowerLoom system to make `business` the current module. Until the current module is changed again, all new introductions of terms and facts will be placed in the `business` module.

```
(defmodule "business"
  :includes ("PL-USER"))
(in-module "business")
```

The basic building blocks of a model are its concepts, relations, and instances.[1] A concept defines classes/categories of entities that populate the domain model. A relation defines attributes and relationships that allow the declaration of facts about an entity. Instances are members of concepts. They appear as arguments to propositional assertions.

5.2 Concepts

Concepts are defined using the `defconcept` command. Here we define the concepts `company` and `corporation`:

```
(defconcept company)
(defconcept corporation (?c company))
```

The first definition tells the system that `company` is a concept (in the `business` module). The second definition defines a concept `corporation`. The type declaration `(?c company)` indicates that `corporation` is a subconcept of `company`, i.e., all instances of `corporation` are also instances of `company`. Let us now create a couple of companies:

---

[1] PowerLoom modules are case-insensitive by default. This means, for example, that a logical constant named "Foo" may be referenced by any of the symbols 'FOO', 'foo', 'foO' etc.

```
(assert (company ACME-cleaners))
(assert (corporation megasoft))
```

These two assertions create two new entities denoted by the terms `ACME-cleaners` and `megasoft`. Both of these entities are members of the concept `company`. `megasoft` is also a member of the concept `corporation`. We can test this by executing some PowerLoom queries:

```
(retrieve all ?x (company ?x))
⇒
There are 2 solutions:
  #1: ?X=ACME-CLEANERS
  #2: ?X=MEGASOFT

(retrieve all ?x (corporation ?x))
⇒
There is 1 solution:
  #1: ?X=MEGASOFT
```

5.3 Relations

So far, our two companies aren't very interesting. In order to say more about them, we can define some relations and functions using the declarations `defrelation` and `deffunction`:

```
(defrelation company-name ((?c company) (?name STRING)))
```

This declaration defines a binary relation `company-name`. The first value in a `company-name` tuple must be an instance of type `company`, while the second value must be a string. We can now give our companies names, using the command `assert`:

```
(assert (company-name ACME-cleaners "ACME Cleaners, LTD"))
(assert (company-name megasoft "MegaSoft, Inc."))
```

We can retrieve pairs of companies and their names with the following query:

```
(retrieve all (?x ?y) (company-name ?x ?y))
⇒
There are 2 solutions:
  #1: ?X=MEGASOFT, ?Y="MegaSoft, Inc."
  #2: ?X=ACME-CLEANERS, ?Y="ACME Cleaners, LTD"
```

5.4 Relation Hierarchies

PowerLoom permits the specification of hierarchies both for concepts and relations. Previously , we defined a small concept hierarchy with `company` on top and `corporation` below it. We now define a subrelation of the relation `company-name` called `fictitious-business-name`:

```
(defrelation fictitious-business-name ((?c company) (?name STRING))
  :=> (company-name ?c ?name))
```

PowerLoom defines a subconcept/subrelation relationship between a pair of concepts or a pair of relations by asserting an "implication" relation between them. The above implication expands into the assertion "for all values of ?c and ?name, if the `fictitious-business-name` relation holds for ?c and?name, then the `company-name` relation also holds for ?c and ?name". This is equivalent to the assertion

```
(forall (?c ?name) (=> (fictitious-business-name ?c ?name)
                       (company-name ?c ?name)))
```

Since implication relationships occur very commonly, PowerLoom provides several syntactic shortcuts for defining them. We have seen one such shortcut earlier; our definition of `corporation` included the clause "`(company ?c)`", which specified that `corporation` is a subconcept of `company`. In our definition of `fictitious-business-name`, the keyword `:=>` introduces a similar shortcut, which tells us that `fictitious-business-name` is a subrelation of `company-name`. Let us assert a fictious business name for MegaSoft:

```
(assert (fictitious-business-name megasoft "MegaSoft"))
```

If we query for the company names of MegaSoft, we get two names, one of them asserted directly, and one of them infered by the subrelation rule:

```
(retrieve all ?x (company-name megasoft ?x))
⇒
There are 2 solutions:
  #1: ?X="MegaSoft"
  #2: ?X="MegaSoft, Inc."
```

5.5 Functions

This illustrates another point: A PowerLoom relation is by default "multi-valued", which in the case of a binary relation means that a single first value can be mapped by the relation to more than one second value. In the present case, our model permits a `company` entity to have more than one `company-name`. If a (binary) relation always maps its first argument to exactly one value (i.e., if it it "single-valued") we can specify it as a `function` instead of a `relation`. For example, we can use a function to indicate the number of employees for a company:

```
(deffunction number-of-employees ((?c company)) :-> (?n INTEGER))
```

When defining a function, all arguments but the last appear just as they do for a relation. The last argument (and its type) appears by itself following the keyword `:->`. Defining a single-valued relation as a function allows us to refer to it using a functional syntax within a logical sentence, as in the following:

```
(assert (= (number-of-employees acme-cleaners) 8))
(assert (= (number-of-employees megasoft) 10000))
```

The functional syntax often results in shorter expressions than equivalents that use relational syntax. For example to retrieve all companies with fewer than 50 employees, we can simply write:

```
(retrieve all ?x (and (company ?x) (< (number-of-employees ?x) 50)))
⇒
There is 1 solution:
  #1: ?X=ACME-CLEANERS
```

Using the syntax for relations, the same query would require the introduction of an existential quantifier, as in:

```
(retrieve ?x (and (company ?x)
                  (exists ?n (and (number-of-employees ?x ?n)
                                  (< ?n 50)))))
⇒
There is 1 solution so far:
  #1: ?X=ACME-CLEANERS
```

To repeat ourselves slightly, Powerloom allows users the choice of using either relational or functional syntax when using a function in predicate position. For example, if `f` is a function, then the expressions `(f ?x ?y)` and `(= (f ?x) ?y)` are equivalent.

5.6 Defined Concepts

If we find ourselves writing the same query (or subexpression) repeatedly, we may wish to define a name for the concept embodying that expression. For example, below we define the term `small-company` to represent the class of all companies with fewer than 50 employees:

```
(defconcept small-company ((?c company))
  :<=> (and (Company ?c)
            (< (number-of-employees ?c) 50)))
```

Notice that we have used a new keyword, `:<=>`. This keyword defines a bidirectional implication called "if-and-only-if". Formally it is equivalent to the following pair of assertions:

```
(assert (forall ?c (=> (and (Company ?c)
                            (< (number-of-employees ?c) 50))
                       (small-company ?c))))
(assert (forall ?c (=> (small-company ?c)
                       (and (Company ?c)
                            (< (number-of-employees ?c) 50)))))
```

In other words, the `:<=>` keyword is a shortcut for an assertion that uses the `<=>` relation, which itself is a shortcut representing the conjunction of two single arrow implications. For example, `(<=> P Q)` is equivalent to `(and (<= P Q) (=> P Q))`, where the `<=` relation is defined to be the inverse of the relation `=>`.

Its not necessary that we exactly specify the number of employees in a company. Below, all we know about ZZ Productions is that they have fewer than 20 employees:

```
(assert (and (company zz-productions)
             (< (number-of-employees zz-productions) 20)))
```

These facts are sufficient to classify ZZ Productions as a small business:

```
(retrieve all ?x (small-company ?x))
⇒
There are 2 solutions:
  #1: ?X=ZZ-PRODUCTIONS
  #2: ?X=ACME-CLEANERS
```

5.7 Negation and Open and Closed World Semantics

PowerLoom implements a three-valued logic—the truth value of each proposition entered into a PowerLoom knowledge base is recorded as being either true, false, or unknown.[2] Many other systems (e.g., relational DBMSs) implement a two-valued logic, wherein if a fact is not asserted to be true, it is assumed to be false. The PowerLoom command `ask` returns one of three (five) values: `true` if it can prove the truth of a proposition, `false` if it can *easily* prove the falsity of a proposition[3] and otherwise it returns `unknown`. (The values `default-true` and `default-false` are also possible if defaults are used).

Below, PowerLoom knows nothing about a newly-introduced concept `s-corporation`, so `ask` returns `unknown` to both a positive query and its negation:

```
(defconcept s-corporation (?c corporation))
(ask (s-corporation zz-productions))
⇒
UNKNOWN
(ask (not (s-corporation zz-productions)))
⇒
UNKNOWN
```

If we assert that ZZ Productions is not an S-corporation, then PowerLoom knows that the proposition in question is false:

```
(assert (not (s-corporation zz-productions)))
(ask (s-corporation zz-productions))
⇒
FALSE
(ask (not (s-corporation zz-productions)))
⇒
TRUE
```

After asserting that ZZ Productions is not an S-corporation, a repeat of the query asking if it *is* one will now return `false`, because the explicit assertion of the negation allows a quick disproof of the positive query.

Note: PowerLoom uses all its effort to prove that the proposition in question is true, and only uses some effort to prove that it is false. Therefore, only falsities that are discovered "on the way" or with shallow inference strategies will be found (which was the case above). If you want to check whether a proposition is false with maximum effort, simply ask the negated proposition by wrapping an explicit `not` arount it. The reason for this asymmetry is that checking for truth and falsity really amounts to asking two separate and possibly expensive queries, and the user or programmer should decide whether the effort should be expended to ask both queries instead of just one.

PowerLoom can sometimes infer a negative fact without the necessity of a direct assertion. For example:

---

[2] Actually, PowerLoom implements a *five-valued* logic — the remaining two values are "default true" and "default false". However, the present discussion defers the subject of default truth values.

[3] Because proving negations can be very difficult, PowerLoom will only conduct a very quick and shallow search for a disproof. More extensive reasoning is used if a negation is asked about explicitly, thus it may be the case that PowerLoom will return `unknown` if asked about `P`, but true if asked about `(not P)`.

```
(ask (= (number-of-employees acme-cleaners) 8))
⇒
TRUE
(ask (= (number-of-employees acme-cleaners) 10))
⇒
FALSE
(ask (not (= (number-of-employees acme-cleaners) 10)))
⇒
TRUE
```

PowerLoom can infer the second and third answers because it knows that the function `number-of-employees` can return only one value, and if that value is the number eight, it cannot also be something else (in this case, ten).

Many systems, in particular, database systems and Prolog, make the assumptions that if a proposition cannot be proved true, then it must be false. This is called the “closed world assumption”. By default, PowerLoom makes an open-world assumption, but for specific relations it can be instructed to assume a closed world if a user wants closed world semantics. For example, suppose we introduce a relation `works-for`, and we assume that all `works-for` facts have been entered in our knowledge base:

```
(defrelation works-for (?p (?c Company)))
(assert (works-for shirly ACME-cleaners))
(assert (works-for jerome zz-productions))
```

If we ask PowerLoom whether Jerome does NOT work for MegaSoft, it will return `unknown`. But if we assert that the relation `works-for` is `closed`, then PowerLoom will assume that Jerome only works for ZZ Productions:

```
(ask (not (works-for jerome megasoft)))
⇒
UNKNOWN

(assert (closed works-for))
(ask (not (works-for jerome megasoft)))
⇒
TRUE
```

The reasoning employed to achieve the above result (that Jerome does not work for MegaSoft) is called “negation as failure”, which means that if a proof of a proposition fails, then one may assume that the proposition is false. We can achieve a negation-as-failure result a second way (i.e., other than by using a closed world assumption) by employing the query operator `fail`. Here we retract the closure assumption for `works-for` and achieve the desired result using `fail`:

```
(retract (closed works-for))
(ask (not (works-for jerome megasoft)))
⇒
UNKNOWN

(ask (fail (works-for jerome megasoft)))
⇒
```

```
TRUE
```

When you see the operator "not" in an SQL query or a Prolog program, it really stands for "fail".

5.8 Retraction

Below, we introduce a few new terms for defining geographic information. We define a relation called `contains` to assert that one geographic location (the second argument to `contains`) is located within another:

```
(defconcept geographic-location)
(defconcept country (?l geographic-location))
(defconcept state (?l geographic-location))
(defconcept city (?l geographic-location))
(defrelation contains ((?l1 geographic-location) (?l2 geographic-location)))
```

Now, we can assert some facts about U.S. geography (including one deliberate mistake):

```
(assert (and
         (country united-states)
         (geographic-location eastern-us) (contains united-states eastern-us)
         (state georgia) (contains eastern-us georgia)
         (city atlanta) (contains georgia atlanta)
         (geographic-location southern-us) (contains united-states southern-us)
         (state texas) (contains eastern-us texas)
         (city dallas) (contains texas dallas)
         (city austin) (contains texas austin)
         ))
```

We would like to repair the incorrect assertion `(contains eastern-us texas)`. The PowerLoom command `retract` allows us to erase assertions that should not be true:

```
(ask (contains eastern-us texas))
⇒
TRUE

(retract (contains eastern-us texas))
(assert (contains southern-us texas))

(ask (contains eastern-us texas))
⇒
UNKNOWN
```

Retraction should not be confused with assertion of negative propositions. For example, asserting that Texas is not a state would not retract the assertion that it is (a state). Instead, an evident logical contradiction is detected as a "clash", and the clashing proposition is disallowed:

```
(assert (not (state texas)))
⇒
Derived both TRUE and FALSE for the proposition '|P|(STATE TEXAS)'.
```

```
   Clash occurred in module ``|MDL|/PL-KERNEL-KB/business'.

(ask (not (state texas)))
⇒
FALSE
```

5.9 Clipping of Values

Programmers are accustomed to changing the values of attributes for program objects just by overwriting previous values. PowerLoom implements a similar semantics for the special case of functions and single-valued relations. When a second value is asserted for one of these relations the previous value is automatically retracted. We call this *clipping*.

To illustrate this behavior for both kinds of relations (a function is considered a kind of relation), we will define a mapping from a company to a city that contains its headquarters in two different ways:

```
(deffunction headquarters ((?c company)) :-> (?city city))
(defrelation headquartered-in ((?c company) (?city city))
  :axioms (single-valued headquartered-in))
```

The clause `":axioms (single-valued headquartered-in)"` tells PowerLoom that the `headquartered-in` relation is single-valued, i.e., that it can map a company to at most one city. This makes its behavior similar to that of the function `headquarters`. Here is an example of clipping for the function `headquarters`:

```
(assert (= (headquarters zz-productions) atlanta))
(retrieve all ?x (= ?x (headquarters zz-productions)))
⇒
There is 1 solution:
  #1: ?X=ATLANTA

(assert (= (headquarters zz-productions) dallas))
(retrieve all ?x (= ?x (headquarters zz-productions)))
⇒
There is 1 solution:
  #1: ?X=DALLAS
```

Here is the same kind of clipping using a single-valued relation:

```
(assert (headquartered-in megasoft atlanta))
(retrieve all ?x (headquartered-in megasoft ?x))
⇒
There is 1 solution:
  #1: ?X=ATLANTA

(assert (headquartered-in megasoft dallas))
(retrieve all ?x (headquartered-in megasoft ?x))
⇒
There is 1 solution:
  #1: ?X=DALLAS
```

5.10 Rule-based Inference

Suppose that we want to retrieve all geographic locations that are contained in the Southern US, based on the set of assertions about geography that we entered in earlier. The following query returns only one of such location:

```
(retrieve all ?x (contains southern-us ?x))
⇒
There is 1 solution:
  #1: ?X=TEXAS
```

We would like the cities Austin and Dallas to be retrieved as well. To do this, we can assert a `rule` that states that `contains` is a transitive relation:

```
(defrule transitive-contains
  (<= (contains ?l1 ?l3)
      (and (contains ?l1 ?l2)
           (contains ?l2 ?l3))))
```

The `defrule` declaration does two things—it asserts a proposition, and it associates a name with that proposition (in the above case, the name is `transitive-contains`). This name is used by the system in displaying traces of its inferencing. It also makes redefinition of the proposition easier. If we wish to retract an unnamed proposition, it is necessary to explicitly retract that proposition using a syntax identical to the assertion[4] If on the other hand, a proposition has a name, then a new `defrule` declaration that uses the same name will automatically retract any existing proposition having the same name.

Our transitive closure rule failed to include any logical quantifiers for the variables `?l1`, `?l2`, and `?l3`. When PowerLoom parses a top-level proposition, it automatically supplies universal quantifiers for any unquantified variables. So, the above rule is equivalent to the rule:

```
(defrule transitive-contains
  (forall (?l1 ?l2 ?l3)
          (<= (contains ?l1 ?l3)
              (and (contains ?l1 ?l2)
                   (contains ?l2 ?l3)))))
```

Note: Instead of defining a `transitive-contains` rule, we could have achieved the same effect by asserting that the `contains` relation is transitive, e.g., by stating `(assert (transitive contains))`.

Now that we have told the system that our `contains` relation is transitive, let us rerun our query:

```
(retrieve all ?x (contains southern-us ?x))
⇒
There are 3 solutions:
  #1: ?X=TEXAS
  #2: ?X=AUSTIN
  #3: ?X=DALLAS
```

[4] Actually, PowerLoom isn't quite as strict as just stated–its search for an identical proposition can accomodate changes in the names of variables.

5.11 Explanation

PowerLoom provides a command called `why` that you can use to get an explanation of the logic behind one of its answers. The `why` command explains the last query entered into the system, i.e., it should invoked after one has submitted a `retrieve` or an `ask` command. Before asking a `why` command, you must enable the justifications feature:

```
(set-feature justifications)
```

Queries execute a bit more slowly with jusifications enabled, which is why it is disabled by default. Having enabled justifications, we must (re)run a query. Here is how we can ask why Dallas is contained in the Southern US:

```
(ask (contains southern-us dallas))
⇒
TRUE
(why)
⇒
1 (CONTAINS SOUTHERN-US DALLAS)
    follows by Modus Ponens
    and substitution {?l3/DALLAS, ?l2/TEXAS, ?l1/SOUTHERN-US}
    since 1.1 ! (forall (?l1 ?l3)
                   (<= (CONTAINS ?l1 ?l3)
                       (exists (?l2)
                          (and (CONTAINS ?l1 ?l2)
                               (CONTAINS ?l2 ?l3)))))
    and   1.2 ! (CONTAINS SOUTHERN-US TEXAS)
    and   1.3 ! (CONTAINS TEXAS DALLAS)
```

The above explanation tells us that a rule (our transitivity rule) was invoked during the proof, and that two ground assertions `(CONTAINS SOUTHERN-US TEXAS)` and `(CONTAINS TEXAS DALLAS)` were accessed to supply preconditions for the rule. These combined assertions lead to the conclusion `(CONTAINS SOUTHERN-US DALLAS)`. Within an explanation, directly asserted propositions are indicated with the prefix '!'.

We can also ask `why` after a `retrieve` query. However, if the query has multiple solutions, each one has a separate explanation. In order to ask `why`, we need to ask for one solution at a time. This can be done by omitting the word `all` from the `retrieve` query, and subsequently calling `(retrieve)` to obtain results one-at-a-time.[5]

```
(retrieve ?x (contains southern-us ?x))
⇒
  #1: ?X=DALLAS
(retrieve)
⇒
There are 2 solutions so far:
  #1: ?X=DALLAS
  #2: ?X=TEXAS
(retrieve)
```

[5] Note: The order of solutions will not necessarily be the same as shown here.

```
⇒
There are 3 solutions so far:
  #1: ?X=DALLAS
  #2: ?X=TEXAS
  #3: ?X=AUSTIN
(why)
⇒
1 (CONTAINS SOUTHERN-US AUSTIN)
    follows by Modus Ponens
    with substitution {?l1/SOUTHERN-US, ?l3/AUSTIN, ?l2/TEXAS}
    since 1.1 ! (FORALL (?l1 ?l3)
                   (<= (CONTAINS ?l1 ?l3)
                       (EXISTS (?l2)
                          (AND (CONTAINS ?l1 ?l2)
                               (CONTAINS ?l2 ?l3)))))
    and   1.2 ! (CONTAINS SOUTHERN-US TEXAS)
    and   1.3 ! (CONTAINS TEXAS AUSTIN)
```

The following query combines a variety of relations that have been entered into the business modules. It retrieves names of companies whose headquarters are in the southern US. Note that query variables that do not appear in the output (i.e., variables not listed after the `all`

```
(retrieve ?name (exists (?city ?company)
                            (and (contains southern-us ?city)
                                 (headquartered-in ?company ?city)
                                 (company-name ?company ?name))))
⇒
There is 1 solution so far:
  #1: ?NAME="MegaSoft, Inc."

(why)
⇒
1 (and (COMPANY-NAME MEGASOFT MegaSoft, Inc.)
       (HEADQUARTERED-IN MEGASOFT DALLAS)
       (CONTAINS SOUTHERN-US DALLAS))
    follows by And-Introduction
    since 1.1 ! (COMPANY-NAME MEGASOFT MegaSoft, Inc.)
    and   1.2 ! (HEADQUARTERED-IN MEGASOFT DALLAS)
    and   1.3   (CONTAINS SOUTHERN-US DALLAS)

1.3 (CONTAINS SOUTHERN-US DALLAS)
    follows by Modus Ponens
    and substitution {?l3/DALLAS, ?l2/TEXAS, ?l1/SOUTHERN-US}
    since 1.3.1 ! (forall (?l1 ?l3)
                     (<= (CONTAINS ?l1 ?l3)
                         (exists (?l2)
                            (and (CONTAINS ?l1 ?l2)
```

```
                                   (CONTAINS ?l2 ?l3)))))
      and   1.3.2 ! (CONTAINS SOUTHERN-US TEXAS)
      and   1.3.3 ! (CONTAINS TEXAS DALLAS)
```

5.12 Contexts and Modules

The final feature that we will illustrate in this section is the PowerLoom context mechanism. PowerLoom organizes its knowledge into a hierarchy of logical containers called "contexts". A PowerLoom context is either a "module", a somewhat heavyweight object that includes its own symbol table, or a "world", a very lightweight object designed for fast switching from one world to another. All contexts inherit from a single root context. The most important feature of a context is that a fact asserted into it is inherited by all contexts below it. However, a "parent" context is unaware of any knowledge entered into one of its descendants.

Here we concern ourselves only with modules. We first define a second module, called `alternate-business`, to be a subcontext of our `business` module, and then we switch into the new module:

```
(defmodule "alternate-business"
  :includes ("business"))
(in-module "alternate-business")
```

Next, within the scope of the `alternate-business` module, we will create a new company. And just for good measure, we will change the name of MegaSoft while we are at it:

```
(assert (and (Company web-phantoms)
             (company-name web-phantoms "Web Phantoms, Inc.")))
(retract (company-name megasoft "MegaSoft, Inc."))
(assert (company-name megasoft "MegaZorch, Inc."))
```

First, here are pairs of companies and company names from the vantage point of the `Business` module:

```
(in-module "business")
(retrieve all (?x ?y) (company-name ?x ?y))
⇒
There are 3 solutions:
  #1: ?X=ACME-CLEANERS, ?Y="ACME Cleaners, LTD"
  #2: ?X=MEGASOFT, ?Y="MegaSoft, Inc."
  #3: ?X=MEGASOFT, ?Y="MegaSoft"
```

Now observe the same query executed from within the alternate `Business` module:

```
(in-module "alternate-business")
(retrieve all (?x ?y) (company-name ?x ?y))
⇒
There are 4 solutions:
  #1: ?X=ACME-CLEANERS, ?Y="ACME Cleaners, LTD"
  #2: ?X=MEGASOFT, ?Y="MegaZorch, Inc."
  #3: ?X=WEB-PHANTOMS, ?Y="Web Phantoms, Inc."
```

```
#4: ?X=MEGASOFT, ?Y="MegaSoft"
```

We see that all facts pertaining to company names have inherited down from the Business to the Alternate Business module, except for the name for MegaSoft that we explicitly retracted. Also, the new facts asserted within the Alternate Business module appear mixed in with the inherited facts.

5.13 Classification, Subsumption

5.14 Truth Maintenance

5.15 Inference Control

5.16 Keyword Axioms

5.17 Cardinality/Type Reasoning with Frame Predicates

5.18 Loom-to-PowerLoom

5.19 Deviations from KIF

5.20 Differences from Loom

5.21 Defaults

5.22 Sets, Lists, SETOFALL, KAPPA

6 Communicating with PowerLoom

There are basically three modes that users can choose from for interacting with the PowerLoom system. The simplest is to use the PowerLoom command interpreter. The interpreter supports a type-in window that allows line-at-a-time entry of commands. You can use the interpreter to load files of PowerLoom declarations, to create and edit knowledge base objects, to ask queries, and to modify settings in the execution environment.

The second mode of interaction involves writing an application that makes calls to the PowerLoom API (see Chapter 8 [PowerLoom API], page 38). PowerLoom implements an extensive list of procedures that can be called to control the logic system. These procedures range from very specific procedures to assert or query a single fact, to general procedures that interpret arbitrary queries. The STELLA translator offers users a choice of Common Lisp, Java, or C++ -based versions of the PowerLoom system; users can choose whichever is the best match for their language of choice for their applications.

Finally, the Ontosaurus Web Browser offers an ideal way to view the contents of PowerLoom knowledge bases. The Ontosaurus Web server allows one to surf across a knowledge base, offering several different kinds of views of the knowledge.

6.1 Command Interpreter

Currently, the primary means for interacting with PowerLoom is its command interpreter. The command interpreter can be used either interactively, or it can be invoked directly from a program to evaluate individual commands. All PowerLoom commands (see Chapter 7 [Commands], page 27) can be evaluated using the command interpreter.

The interactive command interpreter is invoked by calling the function `powerloom` without any arguments. In the Java versions of PowerLoom, the interpreter in called by the `main` routine in the class `PowerLoom` within the `logic` package. In the C++ versions of PowerLoom, `powerloom` is also called within the `main` routine. In the Lisp version, `(STELLA::powerloom)` has to be called explicitly. However, in Lisp it is not really necessary to use the command interpreter, since all commands can also be executed directly at the Lisp top level[1].

The interactive command interpreter functions as a simple read/eval/print loop that prompts for input with a `|=` prompt, reads a user command from standard input, evaluates it, and prints the result to standard output. To exit the command interpreter, type `quit` or `stop`.

To evaluate commands directly from a program, the PowerLoom API provides the following evaluator functions:

evaluate ((*command* OBJECT) (*module* MODULE) (*environment* OBJECT)) : OBJECT [Function]

> Evaluate the command *command* within *module* and return the result. Currently, only the evaluation of (possibly nested) commands and global variables is supported. Commands are simple to program in Common Lisp, since they are built into the language, and relatively awkward in Java and C++. Users of either of those languages are more likely to want to call `s-evaluate`.

[1] If you are executing within a case sensitive module, then you may see some differences in behavior between commands evaluated by the command interpreter and commands invoked from the Lisp Listener.

evaluate-string ((*expression* STRING)) : OBJECT [Function]

Evaluate the expression represented by *expression* and return the result. This is equivalent to `(evaluate (unstringify expression))`.

6.2 Persistent Knowledge Bases

Serious users of PowerLoom will want to construct knowledge bases that persist between sessions. PowerLoom's primary medium of persistence is file-based; users construct their knowledge bases by entering PowerLoom statements into ASCII-formatted files, and then using the `load` command to load them into PowerLoom. There is also a `save-module` command that saves the current assertions of a module to a file. Large-scale persistence via a backend database is currently under development and will become available in one of the next releases.

7 Commands

This chapter lists all available PowerLoom commands alphabetically. Each command is documented with its name, a (possibly empty) list of parameters specified as `(<name> <type>)` pairs, its return type, and its category (*Command*). Almost all of the commands implicitly quote their arguments, meaning that when calling them, you don't need to add any quotation yourself. For example, the command `all-facts-of` is defined as follows:

> **all-facts-of** ((*instanceRef* NAME)) *:* (CONS OF PROPOSITION) [Command]
> Return a cons list of all definite (TRUE or FALSE) propositions that reference the instance *instanceRef*.

The `all-facts-of` command has one parameter called *instanceRef* of type NAME, and returns a STELLA LIST containing zero or more objects of type PROPOSITION as its result. The type NAME subsumes the types SYMBOL, SURROGATE, STRING, and KEYWORD. Unless you are in a case-sensitive module, the following four commands are equivalent:

```
(all-facts-of Merryweather)
(all-facts-of :MERRYWEATHER)
(all-facts-of "merryweather")
(all-facts-of @MerryWeather)
```

Commands can also have `&rest` parameters (similar to Lisp functions). These are either used to allow a variable number of arguments, or to handle optional arguments, since STELLA does not directly support optional arguments.

Here is a list of important parameter types used in the command specifications below:

- GENERALIZED-SYMBOL: A generalized symbol is either a plain symbol (similar to a Lisp symbol) such as `Merryweather`, a keyword (similar to a Lisp keyword) such as `:KIF`, or a STELLA surrogate which is a symbol starting with an at-sign, e.g., `@CONS`. STELLA surrogates are used as names for objects of arbitrary types.
- NAME: Names can be either a string, or a GENERALIZED-SYMBOL (i.e., a symbol, a keyword, or a surrogate). If a symbol is supplied, only its symbol-name is used. Commands that take names as arguments usually coerce whatever argument is entered into a string, but by allowing a NAME they make it a little bit more convenient to type a name in an interactive invocation.[1]
- PARSE-TREE: A parse tree is similar to a Lisp s-expression, i.e., it can either be an atom such as a symbol, number, or a string, or a list of zero or more parse trees. For example, the expression `(happy Fred)` is a parse tree, and so are its components `happy` and `Fred`.

Here is the list of all available PowerLoom commands:

all-facts-of ((*instanceRef* NAME)) *:* (CONS OF PROPOSITION) [Command]
Return a cons list of all definite (TRUE or FALSE) propositions that reference the instance *instanceRef*. This includes propositions asserted to be true by default, but it does not include propositions that are found to be TRUE only by running the query

[1] Lisp programmers are typically spoiled, and find it inconvenient to wrap double-quotes around their arguments.

engine. Facts inferred to be TRUE by the forward chainer will be included. Hence, the returned list of facts may be longer in a context where the forward chainer has been run then in one where it has not (see `run-forward-rules`).

ask (&rest (*proposition&options* PARSE-TREE)) : TRUTH-VALUE [Command]

Perform inference to determine whether the proposition specified in *proposition&options* is true. Return the truth-value found. `ask` will spend most of its effort to determine whether the proposition is true and only a little effort via shallow inference strategies to determine whether it is false. To find out whether a proposition is false with full inference effort `ask` its negation.

KIF example: `(ask (happy Fred))` will return TRUE if Fred was indeed found to be happy. Note, that for this query to run, the logic constant `Fred` and the relation `happy` must already be defined (see `assert`). Use `(set/unset-feature goal-trace)` to en/disable goal tracing of the inference engine.

The `ask` command supports two options, declared with the keywords `:TIMEOUT` and `:DONT-OPTIMIZE`. The argument to `:TIMEOUT` is an integer or floating point time limit, specified in seconds. For example, the command `(ask (nervous Fred) :timeout 2.0)` will cease inference after two seconds if a proof has not been found by then. The argument to `:DONT-OPTIMIZE?` is the constant TRUE, which tells PowerLoom not to optimize the query before evaluating it.

assert ((*proposition* PARSE-TREE)) : OBJECT [Command]

Assert the truth of *proposition*. Return the asserted proposition object. KIF example: "(assert (happy Fred))" asserts that Fred is indeed happy. Note that for this assertion to succeed, the relation `happy` must already be defined. If the constant `Fred` has not yet been created, it is automatically created as a side-effect of calling `assert`.

assert-from-query ((*query* CONS) &rest (*options* OBJECT)) : (CONS OF PROPOSITION) [Command]

Evaluate *query*, instantiate the query proposition for each generated solution and assert the resulting propositions. The accepted syntax is as follows:

```
(assert-from-query <query-command>
                   [:relation <relation-name>]
                   [:pattern <description-term>]
                   [:module <module-name>])
```

<query-command> has to be a strict or partial retrieval command. If a :relation option is supplied, <relation-name> is used as the relation of the resulting propositions. In this case the bindings of each solution will become arguments to the specified relation in the order of *query*s output variables (the arities have to match). The :pattern option is a generalization of this mechanism that specifies an arbitrary proposition pattern to be instantiated by the query's solution. In this case <description-term> has to be a SETOFALL or KAPPA expression whose IO-variables will be bound in sequence to the bindings of a query solution to generate the resulting proposition. Finally, if a :module option is specified, the assertions will be generated in that module. Note that for this to work the relations referenced in the query proposition or pattern have to be visible in the module. Also, instances will not be copied to the target

module, therefore, the resulting propositions might reference external out-of-module objects in case they are not visible there. Here are some examples:

```
(assert-from-query (retrieve all (foo ?x ?y)))
(assert-from-query (retrieve all (?y ?x)
                             (exists ?z (and (foo ?x ?z) (foo ?z ?y))))
                   :relation bar :module other)
(assert-from-query
  (retrieve all (and (relation ?x) (symmetric ?x)))
  :pattern (kappa (?pred)
             (forall (?x ?y)
               (=> (holds ?pred ?x ?y)
                   (holds ?pred ?y ?x))))))
```

assert-rule ((*ruleName* NAME)) *:* PROPOSITION [Command]

Set the truth value of the rule named *ruleName* to TRUE. The proposition having the name *ruleName* may be any arbitrary proposition, although we expect that it is probably a material implication. (See `retract-rule`).

cc (&rest (*name* NAME)) *:* CONTEXT [Command]

Change the current context to the one named *name*. Return the value of the new current context. If no *name* is supplied, return the pre-existing value of the current context. `cc` is a no-op if the context reference cannot be successfully evaluated.

classify-relations ((*module* NAME) (*local?* BOOLEAN)) *:* [Command]

Classify named relations visible in *module*. If *local?*, only classify descriptions defined within *module*, i.e., don't classify descriptions inherited from ancestor modules. If *module* is NULL, classify relations in all modules.

Conceptually, the classifier operates by comparing each concept or relation with all other concepts/relations, searching for a proof that a subsumption relation exists between each pair. Whenever a new subsumption relation is discovered, the classifier adds an `implication` link between members of the pair, thereby augmenting the structure of the concept or relation hierarchy. The implemented classification algorithm is relatively efficient – it works hard at limiting the number of concepts or relations that need to be checked for possible subsumption relationships.

classify-instances ((*module* NAME) (*local?* BOOLEAN)) *:* [Command]

Classify instances visible in *module*. If *local?*, only classify instances that belong to *module*, i.e., don't classify instances inherited from ancestor modules. If *module* is NULL, classify instances in all modules.

Conceptually, the classifier operates by comparing each instance with all concepts in the hierarchy, searching for a proof for each pairing indicating that the instance belongs to the concept. Whenever a new `is-a` relation is discovered, the classifier adds an `is-a` link between the instance and the concept, thereby recording an additional fact about the instance. The implemented classification algorithm is relatively efficient – it works hard at limiting the number of concepts or relations that need to be checked for possible is-a relationships.

clear-instances (&rest (*name* NAME)) : [Command]

Destroy all instances belonging to module *name* or any of its children. Leave meta-objects, e.g., concepts and relations, alone. If no *name* is supplied, the current module will be cleared after confirming with the user.

clear-module (&rest (*name* NAME)) : [Command]

Destroy all objects belonging to module *name* or any of its children. If no *name* is supplied, the current module will be cleared after confirming with the user. Important modules such as STELLA are protected against accidental clearing.

conceive ((*formula* PARSE-TREE)) : OBJECT [Command]

Guess whether *formula* represents a term or a sentence/proposition. If we are not sure, assume its a proposition. If its, a term, return its internal representation. If a proposition, construct a proposition for *formula* without asserting its truth value. Return the conceived proposition object. KIF example: "(conceive (happy Fred))" builds the proposition expressing that Fred is happy without explictly asserting or denying it. Note, that for this to succeed, the relation happy must already be defined (see assert). If the logic constant Fred has not yet been created, it is automatically created as a side-effect of calling conceive.

copyright () : [Command]

Print detailed PowerLoom copyright information.

defconcept (&rest (*args* PARSE-TREE)) : NAMED-DESCRIPTION [Command]

Define (or redefine) a concept. The accepted syntax is:

```
(defconcept <conceptconst> [(<var> <parent>*)]
   [:documentation <string>]
   [:<= <sentence>] | [:=> <sentence>] |
   [:<<= <sentence>] | [:=>> <sentence>] |
   [:<=> <sentence>] | [:<=>> <sentence>] | [:<<=> <sentence>] |
   [:<<=>> <sentence>] |
   [:axioms {<sentence> | (<sentence>+)}] |
   <keyword-option>*)
```

Declaration of a concept variable <var> is optional, unless any implication (arrow) options are supplied that need to reference it. A possibly empty list of concept names following <var> is taken as the list of parents of <conceptconst>. Alternatively, parents can be specified via the :=> option. If no parents are specified, the parent of <conceptconst> is taken to be THING. <keyword-option> represents a keyword followed by a value that states an assertion about <conceptconst>. See defrelation for a description of <keyword-option>s.

deffunction (&rest (*args* PARSE-TREE)) : NAMED-DESCRIPTION [Command]

Define (or redefine) a logic function. The accepted syntax is:

```
(deffunction <funconst> (<vardecl>+) [:-> <vardecl>]
   [:documentation <string>]
   [:<= <sentence>] | [:=> <sentence>] |
   [:<<= <sentence>] | [:=>> <sentence>] |
   [:<=> <sentence>] | [:<=>> <sentence>] |
```

```
[:<<=> <sentence>] | [:<<=>> <sentence>] |
[:axioms {<sentence> | (<sentence>+)}]
[<keyword-option>*])
```

Function parameters can be typed or untyped. If the `:->` option is supplied, it specifies the output variable of the function. Otherwise, the last variable in the parameter list is used as the output variable. See `defrelation` for a description of `<keyword-option>`s.

definstance (&rest (*args* PARSE-TREE)) *:* LOGIC-OBJECT [Command]

Define (or redefine) a logic instance (`definstance` is an alias for `defobject` which see).

defmodule ((*name* NAME) &rest (*options* OBJECT)) *:* [Command]

Define (or redefine) a module named *name*. The accepted syntax is:

```
(defmodule <module-name>
   [:documentation <docstring>]
   [:includes {<module-name> | (<module-name>*)}]
   [:uses {<module-name> | (<module-name>*)}]
   [:lisp-package <package-name-string>]
   [:java-package <package-specification-string>]
   [:cpp-namespace <namespace-name-string>]
   [:java-catchall-class
   [:api? {TRUE | FALSE}]
   [:case-sensitive? {TRUE | FALSE}]
   [:shadow (<symbol>*)]
   [:java-catchall-class <class-name-string>]
   [<other-options>*])
```

name can be a string or a symbol.

Modules include objects from other modules via two separate mechanisms: (1) they inherit from their parents specified via the `:includes` option and/or a fully qualified module name, and (2) they inherit from used modules specified via the `:uses` option. The main difference between the two mechanisms is that inheritance from parents is transitive, while uses-links are only followed one level deep. I.e., a module A that uses B will see all objects of B (and any of B's parents) but not see anything from modules used by B. Another difference is that only objects declared as public can be inherited via uses-links (this is not yet enforced). Note that - contrary to Lisp - there are separate name spaces for classes, functions, and variables. For example, a module could inherit the class `CONS` from the `STELLA` module, but shadow the function of the same name.

The above discussion of `:includes` and `:uses` semantics keyed on the inheritance/visibility of symbols. The PowerLoom system makes another very important distinction: If a module `A` is inherited directly or indirectly via `:includes` specification(s) by a submodule `B`, then all definitions and facts asserted in `A` are visible in `B`. This is not the cases for `:uses`; the `:uses` options does not impact inheritance of propositions at all.

The list of modules specified in the `:includes` option plus (if supplied) the parent in the path used for *name* become the new module's parents. If no `:uses` option was supplied, the new module will use the `STELLA` module by default, otherwise, it will use the set of specified modules. If `:case-sensitive?` is supplied as TRUE, symbols in the module will be interned case-sensitively, otherwise (the default), they will be converted to uppercase before they get interned. Modules can shadow definitions of functions and classes inherited from parents or used modules. Shadowing is done automatically, but generates a warning unless the shadowed type or function name is listed in the `:shadow` option of the module definition .

Examples:

```
(defmodule "PL-KERNEL/PL-USER"
  :uses ("LOGIC" "STELLA")
  :package "PL-USER")

(defmodule PL-USER/GENEALOGY)
```

The remaining options are relevant only for modules that contain STELLA code. Modules used only to contain knowledge base definitions and assertions have no use for them:

The keywords `:lisp-package`, `:java-package`, and `:cpp-package` specify the name of a native package or name space in which symbols of the module should be allocated when they get translated into one of Lisp, Java, or C++. By default, Lisp symbols are allocated in the `STELLA` package, and C++ names are translated without any prefixes. The rules that the STELLA translator uses to attach translated Java objects to classes and packages are somewhat complex. Use :java-package option to specify a list of package names (separated by periods) that prefix the Java object in this module. Use :java-catchall-class to specify the name of the Java class to contain all global & special variables, parameter-less functions and functions defined on arguments that are not classes in the current module. The default value will be the name of the module.

When set to TRUE, the :api? option tells the PowerLoom User Manual generator that all functions defined in this module should be included in the API section. Additionally, the Java translator makes all API functions `synchronized`.

defobject (&rest (*args* PARSE-TREE)) *:* LOGIC-OBJECT [Command]

Define (or redefine) a logic instance. The accepted syntax is:

```
(defobject <constant>
   [:documentation <string>]
   [<keyword-option>*])
```

`<keyword-option>` represents a keyword followed by a value that states an assertion about <constant>. See `defrelation` for a description of `<keyword-option>`s.

`defobject` provides a sugar-coated way to assert a collection of facts about a logic constant, but otherwise adds nothing in terms of functionality.

defproposition (&rest (*args* PARSE-TREE)) *:* PROPOSITION [Command]

Define (or redefine) a named proposition. The accepted syntax is:

```
(defproposition <name> <sentence>
   [:documentation <string>]
   [:forward-only? {true | false}]
```

```
        [:backward-only? {true | false}]
        [:dont-optimize? {true | false}]
        [:confidence-level {:strict | :default}]
        [<keyword-option>*])
```

`<sentence>` can be any sentence that is legal as a top-level assertion. `<name>` can be a string or symbol and will be bound to the asserted proposition represented by `<sentence>`. After this definition every occurrence of `<name>` will be replaced by the associated proposition.

The options :forward-only? and :backward-only? can be used to tell the inference engine to only use the rule in forward or backward direction (this can also be achieved by using the `<<=` or `=>>` implication arrows). :dont-optimize? tells the inference engine to not rearrange the order of clauses in the antecedent of a rule and instead evaluate them in their original order. :confidence-level can be used to mark a proposition as default only.

`<keyword-option>` represents a keyword followed by a value that states an assertion about the proposition `<name>`. See **`defrelation`** for a description of `<keyword-option>`s.

defrelation (&rest (*args* PARSE-TREE)) *:* NAMED-DESCRIPTION [Command]

Define (or redefine) a logic relation. The accepted syntax is:

```
(defrelation <relconst> (<vardecl>+)
   [:documentation <string>]
   [:<= <sentence>] | [:=> <sentence>] |
   [:<<= <sentence>] | [:=>> <sentence>] |
   [:<=> <sentence>] | [:<=>> <sentence>] |
   [:<<=> <sentence>] | [:<<=>> <sentence>] |
   [:axioms {<sentence> | (<sentence>+)}]
   [<keyword-option>*])
```

Relation parameters can be typed or untyped. `<keyword-option>` represents a keyword followed by a value that states an assertion about `<relconst>`. For example, including the option `:foo bar` states that the proposition `(foo <relconst> bar)` is true. `:foo (bar fum)` states that both `(foo <relconst> bar)` and `(foo <relconst> fum)` are true. `:foo true` states that `(foo <relconst>)` is true, `:foo false` states that `(not (foo <relconst>))` is true.

defrule (&rest (*args* PARSE-TREE)) *:* PROPOSITION [Command]

Define (or redefine) a named rule (`defrule` is an alias for `defproposition` which see).

delete-rules ((*relation* NAME)) *:* [Command]

Delete the list of rules associated with *relation*. This function is included mainly for debugging purposes, when a user wants to verify the behavior of different sets of rules.

demo (&rest (*fileandpause* OBJECT)) *:* [Command]

Read logic commands from a file, echo them verbatimly to standard output, and evaluate them just as if they had been typed in interactively. When called with no arguments, present a menu of example demos, otherwise, use the first argument as the name of the file to demo. Pause for user confirmation after each expression has been read but before it is evaluated. Pausing can be turned off by suppling FALSE as the optional second argument, or by typing `c` at the pause prompt. Typing `?` at the pause prompt prints a list of available commands.

deny ((*proposition* PARSE-TREE)) *:* OBJECT [Command]
Assert the falsity of *proposition*. Return the asserted proposition object. KIF example: "(deny (happy Fred))" asserts that Fred is not happy, which could have been done equivalently by "(assert (not (happy Fred)))". Note, that for this to succeed, the relation `happy` must already be defined (see `assert`).

describe ((*name* OBJECT) &rest (*mode* OBJECT)) *:* [Command]
Print a description of an object in :verbose, :terse, or :source modes.

destroy ((*objectSpec* PARSE-TREE)) *:* OBJECT [Command]
Find an object or proposition as specified by *objectSpec*, and destroy all propositions and indices that reference it. *objectSpec* must be a name or a parse tree that evaluates to a proposition. Return the deleted object, or NULL if no matching object was found.

get-rules ((*relation* NAME)) *:* (CONS OF PROPOSITION) [Command]
Return the list of rules associated with *relation*.

help (&rest (*commands* SYMBOL)) *:* [Command]
Describe specific commands, or print a list of available commands.

in-module ((*name* NAME)) *:* MODULE [Command]
Change the current module to the module named *name*.

load ((*file* STRING)) *:* [Command]
Read logic commands from *file* and evaluate them.

load-file ((*file* STRING)) *:* [Command]
Read STELLA commands from *file* and evaluate them. The file should begin with an `in-module` declaration that specifies the module within which all remaining commands are to be evaluated The remaining commands are evaluated one-by-one, applying the function `evaluate` to each of them.

presume ((*proposition* PARSE-TREE)) *:* OBJECT [Command]
Presume the default truth of *proposition*. Return the presumed proposition object. KIF example: "(presume (happy Fred))" states that Fred is most probably happy. Note, that for this to succeed, the relation `happy` must already be defined (see `assert`).

print-features () *:* [Command]
Print the currently enabled and available PowerLoom environment features.

print-rules ((*relation* OBJECT)) *:* [Command]
Print the list of rules associated with *relation*.

process-definitions () *:* [Command]
Finish processing all definitions and assertions that have been evaluated/loaded since that last call to `process-definitions`. PowerLoom defers complete processing of

definitions to make it easier to support cyclic definitions. Following finalization of definitions, this call performs semantic validation of any assertions evaluated since the last call to `process-definitions`. PowerLoom calls this function internally before each query; the primary reason to call it explicitly is to force the production of any diagnostic information that results from the processing and validation.

propagate-constraints (&rest (*name* NAME)) : [Command]

Trigger constraint propagation over all propositions of module *name*. If no *name* is supplied, the current module will be used. This also enables incremental constraint propagation for future monotonic updates to the module. Once a non-monotonic update is performed, i.e., a retraction or clipping of a function value, all cached inferences will be discarded and constraint propagation will be turned off until this function is called again.

repropagate-constraints (&rest (*name* NAME)) : [Command]

Force non-incremental constraint propagation over all propositions of module *name*. If no *name* is supplied, the current module will be used. This also enables incremental constraint propagation for future monotonic updates to the module similar to `propagate-constraints`.

reset-features () : (LIST OF KEYWORD) [Command]

Reset the PowerLoom environment features to their default settings.

retract ((*proposition* PARSE-TREE)) : OBJECT [Command]

Retract the truth of *proposition*. Return the retracted proposition object. KIF example: "(retract (happy Fred))" retracts that Fred is happy. Note that for this assertion to succeed, the relation `happy` must already be defined. If the constant `Fred` has not yet been created, it is automatically created as a side-effect of calling `retract`.

retract-facts-of ((*instanceRef* OBJECT)) : [Command]

Retract all definite (TRUE or FALSE) propositions that reference the instance *instanceRef*.

retract-from-query ((*query* CONS) &rest (*options* OBJECT)) : (CONS OF PROPOSITION) [Command]

Evaluate *query* which has to be a strict or partial retrieval command, instantiate the query proposition for each generated solution and retract the resulting propositions. See `assert-from-query` for available command options.

retract-rule ((*ruleName* NAME)) : PROPOSITION [Command]

If it is currently TRUE, set the truth value of the rule named *ruleName* to UNKNOWN This command may be used alternately with `assert-rule` to observe the effects of querying with or without a particular (named) rule being asserted within the current context. The proposition having the name *ruleName* may be any arbitrary proposition, although we expect that it is probably a material implication.

retrieve (&rest (*query* PARSE-TREE)) : QUERY-ITERATOR [Command]

Retrieve elements of a relation (tuples) that satisfy a proposition. The accepted syntax is:

```
(retrieve [<integer> | all]
          [[{<vardecl> | (<vardecl>+)}]
          <proposition>])
```

The variables and proposition are similar to an `exists` sentence or `kappa` term without the explicit quantifier. If variables are declared, they must match the free variables referenced by <proposition>. Otherwise, the free variables referenced in <proposition> will be used as the query variables. If <proposition> is omitted, the most recently asked query will be continued.

A solution is a set of bindings for the listed variables for which <proposition> is true. The optional first argument controls how many solutions should be generated before control is returned. The keyword `all` indicates that all solutions should be generated. By default, `retrieve` returns after it has found one new solution or if it cannot find any more solutions.

`retrieve` returns an iterator which saves all the necessary state of a query and stores all generated solutions. When used interactively, the returned iterator will print out with the set of solutions collected so far. Calling `retrieve` without any arguments (or only with the first argument) will generate one (or more) solutions to the most recently asked query.

KIF examples:

```
(retrieve (?x Person) (happy ?x))
```

will try to find the next happy person and store it in the returned query iterator.

```
(retrieve 10 (?x Person) (happy ?x))
```

will try to find 10 happy people.

```
(retrieve 10)
```

will try to find the next 10 happy people.

```
(retrieve all (?x Person) (happy ?x))
```

will find all happy people.

```
(retrieve all (?x Person))
```

will find all people.

```
(retrieve (?x Person) (or (happy ?x) (parent-of Fred ?x)))
```

will try to find a person that is happy or has Fred as a parent.

```
(retrieve (or (happy ?x) (parent-of Fred ?x)))
```

will do the same in a more concise syntax by omitting the query variable declaration.

```
(retrieve ((?x Person) (?y Person)) (parent-of ?x ?y))
```

will try to find the next pair of parent/child.

```
(retrieve all (?x Person)
          (exists (?y Person) (parent-of ?x ?y)))
```

will generate the set of all parents. Note, that for these queries to run, the class `Person`, the relations `happy` and `parent-of`, and the logic constant `Fred` must already be defined (see `assert`).

Use `(set/unset-feature trace-subgoals)` to en/disable goal tracing of the inference engine.

save-module ((*name* NAME) (*file* STRING)) : [Command]

Save all definitions and assertions of module *name* to *file*.

set-feature (&rest (*features* NAME)) : (LIST OF KEYWORD) [Command]

Enable the PowerLoom environment feature(s) named by *features*. Return the list of enabled features. Calling `set-feature` without any arguments can be used to display the currently enabled features. The following features are supported:

`just-in-time-inference`: Enables interleaving of forward chaining inference within backward chaining queries.

`iterative-deepening`: Tells the query processor to use iterative deepening instead of a depth-first search to find answers. This is less efficient but necessary for some kinds of highly recursive queries.

`trace-subgoals`: Enables the generation of subgoaling trace information during backchaining inference.

`trace-solutions`: Prints newly found solutions during retrieval right when they are generated as opposed to when the query terminates.

`trace-classifier`: Tells the classifier to describe the inferences it draws.

`justifications`: Enables the generation of justifications during inference, which is a prerequiste for the generation of explanations with `(why)`.

`emit-thinking-dots`: Tells PowerLoom to annotate its inference progress by outputting characters indicating the completion of individual reasoning steps.

By default, the features `emit-thinking-dots` and `just-in-time-inference` are enabled, and the others are disabled.

unset-feature (&rest (*features* NAME)) : (LIST OF KEYWORD) [Command]

Disable the PowerLoom environment feature(s) named by *features*. Return the list of enabled features. Calling `unset-feature` without any arguments can be used to display the currently enabled features. See `set-feature` for a description of supported features.

why (&rest (*args* OBJECT)) : [Command]

Print an explanation for the result of the most recent query. Without any arguments, `why` prints an explanation of the top level query proposition down to a maximum depth of 3. `(why all)` prints an explanation to unlimited depth. Alternatively, a particular depth can be specified, for example, `(why 5)` explains down to a depth of 5. A proof step that was not explained explicitly (e.g., due to a depth cutoff) can be explained by supplying the label of the step as the first argument to `why`, for example, `(why 1.2.3 5)` prints an explanation starting at 1.2.3 down to a depth of 5 (which is counted relative to the depth of the starting point). The keywords `brief` and `verbose` can be used to select a particular explanation style. In brief mode, explicitly asserted propositions are not further explained and indicated with a `!` assertion marker. Additionally, relatively uninteresting proof steps such as AND-introductions are skipped. This explanation style option is sticky and will affect future calls to `why` until it gets changed again. The various options can be combined in any way, for example, `(why 1.2.3 brief 3)` explains starting from step 1.2.3 down to a depth of 3 in brief explanation mode.

8 PowerLoom API

This chapter lists functions that collectively define the PowerLoom API. The first section describes the API functions themselves. The signature is the basic Stella signature. Information on how to translate the names of the functions and their arguments into the programming languages Common Lisp, C++ or Java is given in the Language Specific Interface section.

8.1 API Functions

Many of the functions take a 'module' argument that causes the function to be evaluated in the context of that module. Passing in a NULL value for the module argument means that evaluation takes place in the current module. The module argument is frequently followed by an 'environment' argument that specifies which inference environment should be assumed during evaluation. Values for 'environment' are 'ASSERTION-ENV', 'TAXONOMIC-ENV', and 'INFERENCE-ENV'. 'ASSERTION-ENV' specifies that a knowledge base query or lookup should take into account only explicitly asserted propositions. 'TAXONOMIC-ENV' specifies that a knowledge base query should take into account explicitly-asserted propositions plus any rules that specify subsumption relationships. 'INFERENCE-ENV' specifies that a knowledge base query should take all relevant propositions into account, including those generated during forward inferencing. A NULL value for the the 'environment' argument defaults to 'TAXONOMIC-ENV'.

Many of the functions that take PowerLoom or Stella objects as inputs also have an analog version whose name starts with the prefix "s-" that take strings as inputs. This is provided as a convenience so that programmers will not necessarily need to manipulate PowerLoom objects directly.

ask ((*query* CONS) (*module* MODULE) (*environment* OBJECT)) : TRUTH-VALUE [Function]

Returns a truth value for the *query* in *module* and *environment*. The truth value represents the degree of belief in the answer. See also the helping functions `is-true`, `is-false` , `is-unknown`.

assert-binary-proposition ((*relation* LOGIC-OBJECT) (*arg* OBJECT) (*value* OBJECT) (*module* MODULE) (*environment* OBJECT)) : PROPOSITION [Function]

Assert that the proposition (*relation arg value*) is TRUE in *module*. Return the asserted proposition.

assert-nary-proposition ((*relation-and-arguments* OBJECT) (*module* MODULE) (*environment* OBJECT)) : PROPOSITION [Function]

Assert that the proposition represented by the list `arguments` satisfies the relation `relation`.

assert-proposition ((*proposition* PROPOSITION) (*module* MODULE) (*environment* OBJECT)) : PROPOSITION [Function]

Assert that the proposition *proposition* is true in *module*. Return the asserted proposition.

assert-unary-proposition ((*relation* LOGIC-OBJECT) (*arg* OBJECT) (*module* MODULE) (*environment* OBJECT)) : PROPOSITION [Function]

Assert that the proposition (*relation arg*) is TRUE in *module*. Return the asserted proposition.

change-module ((*module* MODULE)) : OBJECT [Function]

Set the current module to *module*. The return value is *module* unless the context switch cannot be performed, in which case the current module is returned.

clear-module ((*module* MODULE)) : MODULE [Function]

Destroy the contents of the module *module* as well as the contents of all of its children, recursively.

conceive ((*sentence* OBJECT) (*module* MODULE) (*environment* OBJECT)) : (PL-ITERATOR OF PROPOSITION) [Function]

Create one or more proposition objects from the sentence *sentence* in the module *module*. Return an iterator of the propositions. If any of the new propositions has the same structure as an already existing proposition, an automatic check for duplicates will return the pre-existing proposition. Multiple propositions may be returned for a single sentence because of normalization of equivalences, conjunctions, etc.

Signals a `Proposition-Error` if PowerLoom could not conceive *sentence*.

cons-to-pl-iterator ((*self* CONS)) : PL-ITERATOR [Function]

Convert a Stella cons list into an API iterator.

create-concept ((*name* STRING) (*parent* LOGIC-OBJECT) (*module* MODULE) (*environment* OBJECT)) : LOGIC-OBJECT [Function]

Create a concept named *name* in the designated *module*, with the designated *parent* superconcept (which can be left undefined). Additional superconcepts can be added via assertions of the `subset-of` relation. Note that a specified *parent* concept needs to be created separately.

create-enumerated-list ((*members* CONS)) : LOGIC-OBJECT [Function]

Create a logical term that denotes a list containing *members*. Useful for passing lists as arguments to parameterized queries.

create-enumerated-set ((*members* CONS)) : LOGIC-OBJECT [Function]

Create a logical term that denotes the enumerated set containing *members*.

create-function ((*name* STRING) (*arity* INTEGER) (*module* MODULE) (*environment* OBJECT)) : LOGIC-OBJECT [Function]

Create a function named *name* with arity *arity* in the designated *module*. Domain and range information can be added via assertions of `nth-domain` (or `domain` and `range`) relations.

create-object ((*name* STRING) (*concept* LOGIC-OBJECT) (*module* MODULE) (*environment* OBJECT)) : LOGIC-OBJECT [Function]

Create an object named *name* of type *concept* in the designated module. Both *name* and *concept* can be `null`. If *name* is `null` then an object will be created with a new, non-conflicting name based on the name of *concept*, or system-generated if no concept is specified. If *concept* is `null`, then the object will be of type THING.

Return the object.

create-relation ((*name* STRING) (*arity* INTEGER) (*module* MODULE) (*environment* OBJECT)) : LOGIC-OBJECT [Function]

Create a relation named *name* with arity *arity* in the designated module. Domain and range information can be added via assertions of `nth-domain` (or `domain` and `range`) relations.

destroy-object ((*object* LOGIC-OBJECT)) : [Function]

Delete the object *object*, retracting all facts attached to it.

empty? ((*self* PL-ITERATOR)) : BOOLEAN [Method]

Return TRUE if the iterator *self* has no more elements.

evaluate ((*command* OBJECT) (*module* MODULE) (*environment* OBJECT)) : OBJECT [Function]

Evaluate the command *command* within *module* and return the result. Currently, only the evaluation of (possibly nested) commands and global variables is supported. Commands are simple to program in Common Lisp, since they are built into the language, and relatively awkward in Java and C++. Users of either of those languages are more likely to want to call `s-evaluate`.

generate-unique-name ((*prefix* STRING) (*module* MODULE) (*environment* OBJECT)) : STRING [Function]

Generates a name based on *prefix* with a number appended that is not currently in use in the knowledge base.

get-arity ((*relation* LOGIC-OBJECT)) : INTEGER [Function]

Return the arity of the relation *relation*.

get-binary-proposition ((*relation* LOGIC-OBJECT) (*arg1* OBJECT) (*arg2* OBJECT) (*module* MODULE) (*environment* OBJECT)) : PROPOSITION [Function]

Return a proposition such that (*relation arg1 arg2*) is true. The *relation* argument must be bound to a relation. One or both of the *arg1* and *arg2* arguments may be set to NULL, which is interpreted as a wildcard. If more than one proposition matches the input criteria, the selection is arbitrary. This procedure is normally applied to single-valued relations or functions.

get-binary-propositions ((*relation* LOGIC-OBJECT) (*arg1* OBJECT) (*arg2* OBJECT) (*module* MODULE) (*environment* OBJECT)) : (PL-ITERATOR OF PROPOSITION) [Function]

Return propositions such that (*relation arg1 arg2*) is true. The *relation* argument must be bound to a relation. One or both of the *arg1* and *arg2* arguments may be set to NULL, which is interpreted as a wildcard.

get-child-modules ((*module* MODULE)) : (PL-ITERATOR OF MODULE) [Function]

Return the modules that are immediate children of *module*.

get-column-count ((*obj* OBJECT)) : INTEGER [Function]

Return the number of columns in *obj*, which must be of type proposition, cons, vector or PL-iterator. For a proposition, the number includes both the predidate and arguments. For the PL-iterator case,the number of columns is for the current value of the iterator.

get-concept ((*name* STRING) (*module* MODULE) (*environment* OBJECT)) : LOGIC-OBJECT [Function]

Return a class/concept named *name* that is local to or visible from the module *module*.

get-concept-instance-matching-value ((*concept* LOGIC-OBJECT) (*relation* LOGIC-OBJECT) (*value* OBJECT) (*module* MODULE) (*environment* OBJECT)) : OBJECT [Function]

Return a member of concept *concept* that has an attribute matching *value* for the binary relation *relation*, i.e., `(relation <result> value)` holds.

get-concept-instances ((*concept* LOGIC-OBJECT) (*module* MODULE) (*environment* OBJECT)) : PL-ITERATOR [Function]

Return instances of the concept *concept*. Include instances of subconcepts of *concept*. Depending on *concept*, the return values could be (wrapped) literals.

get-concept-instances-matching-value ((*concept* LOGIC-OBJECT) (*relation* LOGIC-OBJECT) (*value* OBJECT) (*module* MODULE) (*environment* OBJECT)) : PL-ITERATOR [Function]

Return members of concept *concept* that have an attribute matching *value* for the binary relation *relation*, i.e., `(relation <result> value)` holds.

get-direct-concept-instances ((*concept* LOGIC-OBJECT) (*module* MODULE) (*environment* OBJECT)) : PL-ITERATOR [Function]

Return instances of concept *concept*. Exclude instances of subconcepts of *concept*. Depending on *concept*, the return values could be (wrapped) literals.

get-direct-subrelations ((*relation* LOGIC-OBJECT) (*module* MODULE) (*environment* OBJECT)) : (PL-ITERATOR OF LOGIC-OBJECT) [Function]

Return relations that directly specialize *relation*. Non-reflexive.

get-direct-superrelations ((*relation* LOGIC-OBJECT) (*module* MODULE) (*environment* OBJECT)) : (PL-ITERATOR OF LOGIC-OBJECT) [Function]

Return relations that directly generalize *relation*. Non-reflexive.

get-direct-types ((*object* LOGIC-OBJECT) (*module* MODULE) (*environment* OBJECT)) : (PL-ITERATOR OF LOGIC-OBJECT) [Function]

Return most specific concepts that *object* belongs to.

get-domain ((*relation* LOGIC-OBJECT)) : LOGIC-OBJECT [Function]

Return the type (a concept) for the first argument to the binary relation *relation*.

get-enumerated-collection-members ((*collection* OBJECT)) : CONS [Function]

Returns the members of an enumerated collection. This works on all types of collection, i.e., sets and lists

get-home-module ((*object* LOGIC-OBJECT)) *:* MODULE [Function]
Return the module in which *object* was created.

get-inferred-binary-proposition-values ((*relation* LOGIC-OBJECT) (*arg* OBJECT) (*module* MODULE) (*environment* OBJECT)) *:* PL-ITERATOR [Function]
Return all values `v` such that (*relation arg* `v`) has been asserted or can be inferred.

get-module ((*name* STRING) (*environment* OBJECT)) *:* MODULE [Function]
Return a module named *name*.

get-modules () *:* (PL-ITERATOR OF MODULE) [Function]
Return all modules currently loaded into PowerLoom.

get-name ((*obj* OBJECT)) *:* STRING [Function]
Return the name of *obj*, if it has one. Otherwise return `null`.

get-nth-domain ((*relation* LOGIC-OBJECT) (*n* INTEGER)) *:* LOGIC-OBJECT [Function]
Return the type (a concept) for the the nth argument of the relation *relation*. Counting starts at zero.

get-nth-float ((*sequence* OBJECT) (*n* INTEGER)) *:* FLOAT [Function]
Return the floating point value in the `nth` column of *sequence*. Counting starts at zero. *sequence* must be of type proposition, cons, vector or PL-iterator. A zero column number returns a proposition's relational predicate. For the PL-iterator case, the the current value pointed to by the iterator is used. If this is not a floating point value, then NULL-FLOAT will be returned.

get-nth-integer ((*sequence* OBJECT) (*n* INTEGER)) *:* INTEGER [Function]
Return an integer representation of the value in the `nth` column of *sequence*. Counting starts at zero. *sequence* must be of type proposition, cons, vector or PL-iterator. A zero column number returns a proposition's relational predicate. For the PL-iterator case, the the current value pointed to by the iterator is used. If this is not an integer value, then NULL-INTEGER will be returned.

get-nth-logic-object ((*sequence* OBJECT) (*n* INTEGER)) *:* LOGIC-OBJECT [Function]
Return a logic object representation of the value in the `nth` column of *sequence*. Counting starts at zero. *sequence* must be of type proposition, cons, vector or PL-iterator. A zero column number returns a proposition's relational predicate. For the PL-iterator case, the the current value pointed to by the iterator is used. A zero column number returns the proposition's relational predicate.

get-nth-string ((*sequence* OBJECT) (*n* INTEGER)) *:* STRING [Function]
Return a string representation of the value in the `nth` column of *sequence*. Counting starts at zero. *sequence* must be of type proposition, cons, vector or PL-iterator. A zero column number returns a proposition's relational predicate. For the PL-iterator case, the the current value pointed to by the iterator is used. This will always succeed, even if the `nth` value is not a string object. In that case, a string reprensentation will be returned.

get-nth-value ((*sequence* OBJECT) (*n* INTEGER)) : OBJECT [Function]

Return the value in the `nth` column of *sequence*. Counting starts at zero. *sequence* must be of type proposition, cons, vector or PL-iterator. A zero column number returns a proposition's relational predicate. For the PL-iterator case, the number of columns is for the current value of the iterator.

get-object ((*name* STRING) (*module* MODULE) (*environment* OBJECT)) : OBJECT [Function]

Look for an object named *name* that is local to or visible from the module *module*.

get-parent-modules ((*module* MODULE)) : (PL-ITERATOR OF MODULE) [Function]

Return the modules that are immediate parents of *module*.

get-predicate ((*prop* PROPOSITION)) : LOGIC-OBJECT [Function]

Return the concept or relation predicate for the proposition *prop*.

get-proper-subrelations ((*relation* LOGIC-OBJECT) (*module* MODULE) (*environment* OBJECT)) : (PL-ITERATOR OF LOGIC-OBJECT) [Function]

Return relations that specialize *relation*. Non-reflexive.

get-proper-superrelations ((*relation* LOGIC-OBJECT) (*module* MODULE) (*environment* OBJECT)) : (PL-ITERATOR OF LOGIC-OBJECT) [Function]

Return relations that generalize *relation*. Non-reflexive.

get-proposition ((*relation-and-arguments* OBJECT) (*module* MODULE) (*environment* OBJECT)) : PROPOSITION [Function]

Return a proposition matching *relation-and-arguments* that has been asserted (or inferred by forward chaining). *relation-and-arguments* is a sequence containing objects and nulls. The first argument must be the name of a relation. A null value acts like a wild card. If more than one proposition matches the input criteria, the selection among satisficing propositions is arbitrary. This procedure is normally applied to single-valued relations or functions.

get-propositions ((*relation-and-arguments* OBJECT) (*module* MODULE) (*environment* OBJECT)) : (PL-ITERATOR OF PROPOSITION) [Function]

Return propositions matching *relation-and-arguments* that have been asserted (or inferred by forward chaining). *relation-and-arguments* is a sequence containing objects and nulls. The first argument must be the name of a relation. A null value acts like a wild card.

get-propositions-in-module ((*module* MODULE) (*environment* OBJECT)) : (PL-ITERATOR OF PROPOSITION) [Function]

Return propositions that have been conceived in the module *module*.

get-propositions-of ((*object* LOGIC-OBJECT) (*module* MODULE) (*environment* OBJECT)) : (PL-ITERATOR OF PROPOSITION) [Function]

Return all propositions that have *object* among their arguments, and that are TRUE in the scope of the module *module*.

get-range ((*relation* LOGIC-OBJECT)) : LOGIC-OBJECT [Function]

Return the type (a concept) for fillers of the binary relation *relation*.

get-relation ((*name* STRING) (*module* MODULE) (*environment* OBJECT)) *:* LOGIC-OBJECT [Function]

Return a concept or relation named *name* that is local to or visible from the module *module*.

get-relation-extension ((*relation* LOGIC-OBJECT) (*module* MODULE) (*environment* OBJECT)) *:* (PL-ITERATOR OF PROPOSITION) [Function]

Return propositions that satisfy *relation*. Include propositions that satisfy subrelations of *relation*.

get-rules ((*relation* LOGIC-OBJECT) (*module* MODULE) (*environment* OBJECT)) *:* (PL-ITERATOR OF PROPOSITION) [Function]

Return rules attached to the concept/relation *relation* in either antecedent or consequent position.

get-types ((*object* LOGIC-OBJECT) (*module* MODULE) (*environment* OBJECT)) *:* (PL-ITERATOR OF LOGIC-OBJECT) [Function]

Return all named concepts that *object* belongs to.

initialize () *:* [Function]

Initialize the PowerLoom logic system. This needs to be called by all applications before using PowerLoom.

is-a ((*object* OBJECT) (*concept* LOGIC-OBJECT) (*module* MODULE) (*environment* OBJECT)) *:* BOOLEAN [Function]

Return TRUE if *object* is a member of the concept *concept*.

is-default ((*tv* TRUTH-VALUE)) *:* BOOLEAN [Function]

Tests whether *tv* is a default truth value.

is-enumerated-collection ((*obj* OBJECT)) *:* BOOLEAN [Function]

Test whether *obj* is an enumerated collection. This subsumes both sets and lists.

is-enumerated-list ((*obj* OBJECT)) *:* BOOLEAN [Function]

Test whether *obj* is an enumerated list

is-enumerated-set ((*obj* OBJECT)) *:* BOOLEAN [Function]

Test whether *obj* is an enumerated set.

is-false ((*tv* TRUTH-VALUE)) *:* BOOLEAN [Function]

Tests whether *tv* is a false truth value. It can be false either absolutely or by default.

is-float ((*obj* OBJECT)) *:* BOOLEAN [Function]

Test whether *obj* is of type FLOAT (double)

is-integer ((*obj* OBJECT)) *:* BOOLEAN [Function]

Test whether *obj* is of type INTEGER

is-logic-object ((*obj* OBJECT)) *:* BOOLEAN [Function]

Test whether *obj* is of type LOGIC-OBJECT

is-number ((*obj* OBJECT)) : BOOLEAN [Function]
Test whether *obj* is of type NUMBER. This can be either an integer or a floating point number. One key characteristic is that `object-to-integer` and `object-to-float` will both work on it.

is-strict ((*tv* TRUTH-VALUE)) : BOOLEAN [Function]
Tests whether *tv* is a strict (non-default) truth value.

is-string ((*obj* OBJECT)) : BOOLEAN [Function]
Test whether *obj* is of type STRING

is-subrelation ((*sub* LOGIC-OBJECT) (*super* LOGIC-OBJECT) (*module* MODULE) (*environment* OBJECT)) : BOOLEAN [Function]
Return TRUE if *sub* is a subconcept/subrelation of *super*.

is-true ((*tv* TRUTH-VALUE)) : BOOLEAN [Function]
Tests whether *tv* is a true truth value. It can be true either absolutely or by default.

is-true-binary-proposition ((*relation* LOGIC-OBJECT) (*arg* OBJECT) (*value* OBJECT) (*module* MODULE) (*environment* OBJECT)) : BOOLEAN [Function]
Return TRUE if the proposition (*relation arg value*) has been asserted (or inferred by forward chaining).

is-true-proposition ((*proposition* PROPOSITION) (*module* MODULE) (*environment* OBJECT)) : BOOLEAN [Function]
Return TRUE if *proposition* is TRUE in the module *module*.

is-true-unary-proposition ((*relation* LOGIC-OBJECT) (*arg* OBJECT) (*module* MODULE) (*environment* OBJECT)) : BOOLEAN [Function]
Return TRUE if the proposition (*relation arg*) has been asserted (or inferred by forward chaining).

is-unknown ((*tv* TRUTH-VALUE)) : BOOLEAN [Function]
Tests whether *tv* is an unknown truth value.

iterator-to-pl-iterator ((*self* ITERATOR)) : PL-ITERATOR [Function]
Convert an arbitrary Stella iterator into an API iterator.

length ((*self* PL-ITERATOR)) : INTEGER [Method]
Number of items in *self*.

list-to-pl-iterator ((*self* LIST)) : PL-ITERATOR [Function]
Convert a Stella list into an API iterator.

load ((*filename* STRING)) : [Function]
Read logic commands from the file named *filename* and evaluate them. The file should begin with an `in-module` declaration that specifies the module within which all remaining commands are to be evaluated The remaining commands are evaluated one-by-one, applying the function `evaluate` to each of them.

next? ((*self* PL-ITERATOR)) : BOOLEAN [Method]

Advance the PL-Iterator *self* and return `true` if more elements are available, `false` otherwise.

object-to-float ((*self* OBJECT)) : FLOAT [Function]

Coerce *self* to a float, or throw a Stella Exception if the coersion is not feasible.

object-to-integer ((*self* OBJECT)) : INTEGER [Function]

Coerce *self* to an integer, or throw a Stella Exception if the coersion is not feasible.

object-to-parsable-string ((*self* OBJECT)) : STRING [Function]

Return a string representing a printed representation of the object *self*. Like `object-to-string`, but puts escaped double quotes around strings.

object-to-string ((*self* OBJECT)) : STRING [Function]

Return a printed representation of the term *self* as a string.

print-rules ((*relation* OBJECT)) : [Command]

Print the list of rules associated with *relation*.

retract ((*proposition* PARSE-TREE)) : OBJECT [Command]

Retract the truth of *proposition*. Return the retracted proposition object. KIF example: "(retract (happy Fred))" retracts that Fred is happy. Note that for this assertion to succeed, the relation `happy` must already be defined. If the constant `Fred` has not yet been created, it is automatically created as a side-effect of calling `retract`.

retract-binary-proposition ((*relation* LOGIC-OBJECT) (*arg* OBJECT) (*value* OBJECT) (*module* MODULE) (*environment* OBJECT)) : PROPOSITION [Function]

Retract that the proposition (*relation arg value*) is TRUE in *module*. Return the asserted proposition.

retract-nary-proposition ((*relation-and-arguments* OBJECT) (*module* MODULE) (*environment* OBJECT)) : PROPOSITION [Function]

Retract the proposition that `arguments` satisfies the relation `relation`.

retract-proposition ((*proposition* PROPOSITION) (*module* MODULE) (*environment* OBJECT)) : PROPOSITION [Function]

Retract the truth of the proposition *proposition* in *module*. Return the retracted proposition.

retract-unary-proposition ((*relation* LOGIC-OBJECT) (*arg* OBJECT) (*module* MODULE) (*environment* OBJECT)) : PROPOSITION [Function]

Retract that the proposition (*relation arg*) is TRUE in *module*. Return the asserted proposition.

retrieve ((*query* CONS) (*module* MODULE) (*environment* OBJECT)) : PL-ITERATOR [Function]

Returns an iterator for variables that satisfy *query* in `module-name` and *environment*. This uses the normal PowerLoom query syntax:

`[n-values] output-variables query-form [options]`

The `output-variables` should either be a single variable name – preceded by the `?` character – or a list of one or more such names. If a single variable name is provided, then each element in the returned iterator will be a value binding. If a list (even of one variable name) is provided, then each element in the returned iterator can be accessed using the `get-nth-`... functions.

run-forward-rules ((*module* OBJECT) (*force?* BOOLEAN)) : [Function]

Run forward inference rules in module *module*. If *module* is NULL, the current module will be used. If forward inferencing is already up-to-date in the designated module, no additional inferencing will occur, unless `force` is set to TRUE, in which case all forward rules are run or rerun.

Calling `run-forward-rules` temporarily puts the module into a mode where future assertional (monotonic) updates will trigger additional forward inference. Once a non-monotonic update is performed, i.e., a retraction or clipping of relation value, all cached forward inferences will be discarded and forward inferencing will be disabled until this function is called again.

s-ask ((*query* STRING) (*module-name* STRING) (*environment* OBJECT)) : TRUTH-VALUE [Function]

Returns a truth value for the *query* in *module-name* and *environment*. The truth value represents the degree of belief in the answer. See also the helping functions `is-true`, `is-false` , `is-unknown`.

s-assert-proposition ((*sentence* STRING) (*module-name* STRING) (*environment* OBJECT)) : (PL-ITERATOR OF PROPOSITION) [Function]

Assert that the logical sentence *sentence* is true in the module named *module-name*. Return an iterator of the propositions resulting from sentence.

s-change-module ((*name* STRING) (*environment* OBJECT)) : OBJECT [Function]

Set the current module to the module named *name*. The return value is the module named *name* unless the context switch cannot be performed, in which case the current module is returned.

s-clear-module ((*name* STRING) (*environment* OBJECT)) : MODULE [Function]

Destroy the contents of the module named *name*, as well as the contents of all of its children, recursively.

s-conceive ((*sentence* STRING) (*module-name* STRING) (*environment* OBJECT)) : (PL-ITERATOR OF PROPOSITION) [Function]

Create one or more proposition objects from the sentence *sentence* in the module named *module-name*. Return an iterator of the propositions. If any of the new propositions has the same structure as an already existing proposition, an automatic check for duplicates will return the pre-existing proposition. Multiple propositions may be returned for a single sentence because of normalization of equivalences, conjunctions, etc.

Signals a `Proposition-Error` if PowerLoom could not conceive *sentence*.

s-create-concept ((*name* STRING) (*parent-name* STRING) (*module-name* STRING) (*environment* OBJECT)) *:* LOGIC-OBJECT [Function]

Create a concept named *name* in the designated module, with with the concept named *parent-name* as superconcept (which can be left undefined). Additional superconcepts can be added via assertions of the `subset-of` relation. Note that a specified parent concept needs to be created separately.

s-create-function ((*name* STRING) (*arity* INTEGER) (*module-name* STRING) (*environment* OBJECT)) *:* LOGIC-OBJECT [Function]

Create a function named *name* with arity *arity* in the designated module. Domain and range information can be added via assertions of `domain`, `nth-domain` and `range` relations.

s-create-object ((*name* STRING) (*concept-name* STRING) (*module-name* STRING) (*environment* OBJECT)) *:* LOGIC-OBJECT [Function]

Create an object named *name* of type *concept-name* in the designated module. Both *name* and *concept-name* can be null strings. If *name* is a null string then an object will be created with a new, non-conflicting name based on *concept-name*, or system-generated if no concept nameis specified. If *concept-name* is the null string, then the object will be of type THING.

Return the object.

s-create-relation ((*name* STRING) (*arity* INTEGER) (*module-name* STRING) (*environment* OBJECT)) *:* LOGIC-OBJECT [Function]

Create a relation named *name* with arity *arity* in the designated module. Domain and range information can be added via assertions of `nth-domain` (or `domain` and `range`) relations.

s-destroy-object ((*object-name* STRING) (*module-name* STRING) (*environment* OBJECT)) *:* [Function]

Delete the object named *object-name*, retracting all facts attached to it.

s-evaluate ((*command* STRING) (*module-name* STRING) (*environment* OBJECT)) *:* OBJECT [Function]

Evaluate the command represented by the string *command* within `module` and return the result. Currently, only the evaluation of (possibly nested) commands and global variables is supported.

s-get-arity ((*relation-name* STRING) (*module-name* STRING) (*environment* OBJECT)) *:* INTEGER [Function]

Return the arity of the relation named *relation-name*.

s-get-child-modules ((*name* STRING) (*environment* OBJECT)) *:* (PL-ITERATOR OF MODULE) [Function]

Return the modules that are immediate children of module *name*.

s-get-concept ((*name* STRING) (*module-name* STRING) (*environment* OBJECT)) *:* LOGIC-OBJECT [Function]

Return a class/concept named *name* that is local to or visible from the module *module-name*.

s-get-concept-instances ((*concept-name* STRING) (*module-name* STRING) (*environment* OBJECT)) : PL-ITERATOR [Function]

Return instances of concept *concept-name*. Include instances of subconcepts of *concept-name*. Depending on *concept-name*, the return values could be (wrapped) literals.

s-get-direct-concept-instances ((*concept-name* STRING) (*module-name* STRING) (*environment* OBJECT)) : PL-ITERATOR [Function]

Return instances of concept *concept-name*. Exclude instances of subconcepts of *concept-name*. Depending on *concept-name*, the return values could be (wrapped) literals.

s-get-domain ((*relation-name* STRING) (*module-name* STRING) (*environment* OBJECT)) : LOGIC-OBJECT [Function]

Return the type (concept) for the first argument to the binary relation *relation-name*.

s-get-inferred-binary-proposition-values ((*relation-name* STRING) (*arg-name* STRING) (*module-name* STRING) (*environment* OBJECT)) : PL-ITERATOR [Function]

Return all values `v` such that (*relation-name arg-name* `v`) has been asserted or can be inferred.

s-get-nth-domain ((*relation-name* STRING) (*n* INTEGER) (*module-name* STRING) (*environment* OBJECT)) : LOGIC-OBJECT [Function]

Return the type (a concept) for the nth argument of the relation named *relation-name*. Counting starts at zero.

s-get-object ((*name* STRING) (*module-name* STRING) (*environment* OBJECT)) : OBJECT [Function]

Look for an object named *name* that is local to or visible from the module *module-name*.

s-get-parent-modules ((*name* STRING) (*environment* OBJECT)) : (PL-ITERATOR OF MODULE) [Function]

Return the modules that are immediate parents of module *name*.

s-get-parent-modules ((*name* STRING) (*environment* OBJECT)) : (PL-ITERATOR OF MODULE) [Function]

Return the modules that are immediate parents of module *name*.

s-get-proposition ((*relation-and-arguments* STRING) (*module-name* STRING) (*environment* OBJECT)) : PROPOSITION [Function]

Return a proposition matching *relation-and-arguments* that has been asserted (or inferred by forward chaining). *relation-and-arguments* is a string that begins with a left parenthesis, followed by a relation name, one or more argument identifiers, and terminated by a right parenthesis. Each argument identifier can be the name of a logical constant, a literal reference (e.g., a number), the null identifier, or a variable (an identifier that begins with a question mark). Each occurrence of a null or a variable acts like a wild card. If more than one proposition matches the input criteria, the selection among satisficing propositions is arbitrary. This procedure is normally applied to single-valued relations or functions.

s-get-propositions ((*relation-and-arguments* STRING) (*module-name* STRING) (*environment* OBJECT)) : (PL-ITERATOR OF PROPOSITION) [Function]

Return propositions matching *relation-and-arguments* that have been asserted (or inferred by forward chaining). *relation-and-arguments* is a string that begins with a left parenthesis, followed by a relation name, one or more argument identifiers, and terminated by a right parenthesis. Each argument identifier can be the name of a logical constant, a literal reference (e.g., a number), the null identifier, or a variable (an identifier that begins with a question mark). Each occurrence of a null or a variable acts like a wild card.

s-get-propositions-of ((*object-name* STRING) (*module-name* STRING) (*environment* OBJECT)) : (PL-ITERATOR OF PROPOSITION) [Function]

Return all propositions that have the object named *object-name* among their arguments, and that are TRUE in the scope of the module `module`.

s-get-range ((*relation-name* STRING) (*module-name* STRING) (*environment* OBJECT)) : LOGIC-OBJECT [Function]

Return the type (a concept) for fillers of the binary relation *relation-name*.

s-get-relation ((*name* STRING) (*module-name* STRING) (*environment* OBJECT)) : LOGIC-OBJECT [Function]

Return a concept or relation named *name* that is local to or visible from the module *module-name*.

s-get-relation-extension ((*relation-name* STRING) (*module* MODULE) (*environment* OBJECT)) : (PL-ITERATOR OF PROPOSITION) [Function]

Return propositions that satisfy the relation named *relation-name*. Include propositions that satisfy subrelations of the relation.

s-get-rules ((*relation-name* STRING) (*module-name* STRING) (*environment* OBJECT)) : (PL-ITERATOR OF PROPOSITION) [Function]

Return rules attached to the concept/relation named *relation-name* found in the module named *module-name*.

s-is-true-proposition ((*relation-and-arguments* STRING) (*module-name* STRING) (*environment* OBJECT)) : BOOLEAN [Function]

Return TRUE if a proposition that prints as the string *relation-and-arguments* is true in the module named *module-name*.

s-print-rules ((*name* STRING) (*stream* OUTPUT-STREAM) (*module-name* STRING) (*environment* OBJECT)) : [Function]

Print rules attached to the concept/relation named *name*.

s-retract-proposition ((*sentence* STRING) (*module-name* STRING) (*environment* OBJECT)) : (PL-ITERATOR OF PROPOSITION) [Function]

Retract the truth of the logical sentence *sentence* in the module named *module-name*. Return an iterator of the retracted propositions resulting from sentence.

s-retrieve ((*query* STRING) (*module-name* STRING) (*environment* OBJECT)) : PL-ITERATOR [Function]

Returns an iterator for variables that satisfy *query* in *module-name* and *environment*. This uses the normal PowerLoom query syntax:

`"[n-values] output-variables query-form [options]"`

The `output-variables` should either be a single variable name – preceded by the `?` character – or a list of one or more such names. If a single variable name is provided, then each element in the returned iterator will be a value binding. If a list (even of one variable name) is provided, then each element in the returned iterator can be accessed using the `get-nth-`... functions.

s-save-module ((*module-name* STRING) (*filename* STRING) (*ifexists* STRING) (*environment* OBJECT)) : [Function]

Save the contents of the module *module-name* into a file named *filename*. If a file named *filename* already exists, then the action taken depends on the value of *ifexists*. Possible values are "ASK", "REPLACE", "WARN" and "ERROR":

REPLACE => Means overwrite without warning. WARN => Means overwrite with a warning. ERROR => Means don't overwrite, signal an error instead. ASK => Ask the user whether to overwrite or not. If not overwritten, an exception is thrown.

save-module ((*module* MODULE) (*filename* STRING) (*ifexists* STRING) (*environment* OBJECT)) : [Function]

Save the contents of the module `mod` into a file named *filename*. If a file named *filename* already exists, then the action taken depends on the value of *ifexists*. Possible values are "ASK", "REPLACE", "WARN" and "ERROR":

REPLACE => Means overwrite without warning. WARN => Means overwrite with a warning. ERROR => Means don't overwrite, signal an error instead. ASK => Ask the user whether to overwrite or not. If not overwritten, an exception is thrown.

string-to-object ((*string* STRING) (*type* LOGIC-OBJECT) (*module* MODULE) (*environment* OBJECT)) : OBJECT [Function]

Evaluate *string* with respect to *module* and *environment* and return the corresponding logical term. *type* is a concept used to assist the correct interpretation of *string*.

Currently *type* only has an effect on the interpretation of literal types.

8.2 Language Specific Interface

This section contains the description of the programming language specific aspects of using the PowerLoom API. Each section describes the naming conventions and namespace issues related to calling the API functions from that programming language.

8.2.1 Lisp API

This section tells how to call the API functions in PowerLoom's Common Lisp implementation from a Lisp program. The function names are identical to the Stella names in the PowerLoom API description See Chapter 8 [PowerLoom API], page 38. They are exported from the `PLI` package. Other Stella symbols and names are in the `STELLA` package, but **currently none of the Stella symbols are exported!**.

PowerLoom can be used from Allegro Common Lisp, CMU Common Lisp, LispWorks Common Lisp and Macintosh Common Lisp. It may be possible to use the system from other Common Lisp systems, but they have not been tested.

8.2.1.1 Common Lisp Initialization

Loading the Common Lisp version of PowerLoom will normally initialize the system as part of the loading process. The Common Lisp version can be loaded by loading the file '`load-powerloom.lisp`' from the top-level '`powerloom`' directory. This will make the system available for use.

8.2.1.2 Type Declarations

Stella is a typed language, and the Common Lisp translation uses the type information for Common Lisp type declarations. That means that values specified as being of type INTEGER, STRING and FLOAT must have the correct type. In particular, integer values will not be coerced to floating point values by the code. The following native type assignments are made:

```
Stella      Common Lisp
=======     ===========
INTEGER     FIXNUM
FLOAT       DOUBLE-FLOAT
STRING      SIMPLE-STRING
```

For convenience, loading PowerLoom will set the default format for reading floating point numbers in Common Lisp to be `double-float`.

Stella `CONS` objects are implmented as native Lisp conses. Boolean values can take on the values `stella::true` or `stella::false`.

8.2.1.3 NULL values

One additional consequence of the strong typing of the language is that there are specialized `NULL` values for numeric and string parameters.

```
Stella Type   Null Value
===========   ===========
  INTEGER     stella::null-integer
  FLOAT       stella::null-float
  STRING      stella::null-string
```

8.2.1.4 Wrapped Literal Values

Literal values (integers, floats, strings, etc.) that are used in PowerLoom appear as wrapped values. The PowerLoom API functions `object-to-...` can be used to coerce the values into the appropriate return type.

<to be written: wrapping values>

8.2.1.5 Special Variables

All Stella special variables are implemented as Common Lisp special variables. Binding of the values can be used normally.

8.2.1.6 CLOS Objects versus Structs

PowerLoom can be translated in one of two ways for Common Lisp. One method uses CLOS objects as the basis for all Stella and PowerLoom objects. For faster execution, it is also possible to use a version in which Stella and PowerLoom objects are implemented using Common Lisp structs instead. This is controlled by the special variable `cl-user::*load-cl-struct-stella?*`. If this is set to `cl:t`, then the struct version will be loaded. This needs to be set before loading the '`load-powerloom.lisp`' file.

8.2.2 C++ API

<to be written>

8.2.3 Java API

This section tells how to call the API functions in PowerLoom's Java implementation from a Java program. The Java translation is written for Java version 1.2. All of the PowerLoom Interface functions appear as static methods of the class `edu.isi.powerloom.PLI`.

8.2.3.1 Initializing PowerLoom

PowerLoom needs to run initialization functions to set up its environment for proper operation when it starts up. The simplest method for initializing PowerLoom is to use the static method call:

```
PLI.initialize()
```

This must be called before using any PowerLoom features and before loading any PowerLoom knowledge bases. It may be called more than once without ill effect.

8.2.3.2 PowerLoom Java Conventions

PowerLoom's Java code is automatically generated by a translator from underlying Stella code. The character set for legal Stella names is larger than the character set for legal Java identifiers, so there is some mapping involved.

PowerLoom names are words separated by hyphen (-) characters. For Java, we have attempted to closely follow the Java conventions:

- Class names begin with a capital letter and each word is capitalized. The hyphens from the PowerLoom names are removed. Example:

```
string-wrapper   =>  StringWrapper
```

Exceptions are made for class names that would otherwise conflict with normal Java Classes. In that case, the prefix "Stella_" is added to each class name. At the moment this applies only to the following exceptions:

```
object    =>  Stella_Object
class     =>  Stella_Class
```

- Method and Function names begin with a lower case letter but each subsequent word is capitalized. The hyphens from PowerLoom names are removed. Example:

```
wrapper-value    =>  wrapperValue
```

- Storage slots are implemented as Java fields. The names begin with a lower case letter but each subsequent word is capitalized. The hyphens from PowerLoom names are removed. Example:

```
dynamic-slots    =>  dynamicSlots
```

- Global and Special variable names are written in all uppercase. The hyphens from PowerLoom are replaced by underscore (_) characters. By convention, special variables are written in PowerLoom with surrounding asterisk (*) characters. The asterisks are replaced by dollar signs ($). Example:

```
*html-quoted-characters*  =>  $HTML_QUOTED_CHARACTERS$
```

The most common non-alphanumeric characters are mapped as follows. A full set of mappings is in section See Section 8.2.3.7 [Java Character Mapping], page 58.

```
?  =>  P      (for Predicate)
!  =>  X      (eXclamation)
$  =>  B      (Buck)
%  =>  R      (peRcent)
&  =>  A      (Ampersand)
*  =>  $      Special variable marker.
```

The character mappings use uppercase characters if the basic identifier uses mixed or lower case. The mappings use lowercase characters if the basic identifier uses upper case.

Stella modules are mapped to Java packages. The basic system distribution includes the following package hierarchy:

```
edu
   isi
      stella
          javalib
      powerloom
          logic
          pl_kernel_kb
              loom_api
```

Basic system functionality and data structures such as Cons and List objects are defined in stella. PowerLoom's logic (concepts, relations, rules, etc.) are defined in the logic package. There is a set of interface functions in the PLI package. They are described in their own section below.

We recommend the following import statements in Java files that use PowerLoom:

```
import edu.isi.stella.*;
import edu.isi.stella.javalib.*;
import edu.isi.powerloom.PLI;
import edu.isi.powerloom.logic.*;
```

Functions (in Java terms, static Methods) are translated as static methods on the class of their first argument (as long as that argument is not a primitive type and is in the same Stella module). Functions which take no arguments, those whose first argument is a primitive type, and those whose first argument is a class not defined in the same module are all placed into a class with the same name as the Stella module in which it appers. It will be in the package corresponding to that Stella module. Java constructors should not be called directly. Instead, there will be a static method `new<ClassName>` (with the class name in mixed case!) that should be used instead.

Most of the functions of interest will be in the edu.isi.stella.Stella, edu.isi.powerloom.PLI or edu.isi.powerloom.logic.Logic classes.

Methods typically refer to their first argument as "self".

Methods which return more than one return value will take a final argument which is an array of `Stella_Object`, which will be used to return the additional arguments.

Primitive types in Stella have the following mapping in Java:

```
     Stella          Java
     ======          ====
     INTEGER         int
     FLOAT           double
     NUMBER          double
     CHARACTER       char
     BOOLEAN         boolean
     STRING          String
     MUTABLE-STRING  StringBuffer

NATIVE-OUTPUT-STREAM  java.io.PrintStream
NATIVE-INPUT-STREAM   java.io.PushbackInputStream  (May change!!!)
```

Inside Stella/PowerLoom objects and collections, primitive types are wrapped using Stella wrappers instead of Java's primitive classes. So integers will be wrapped as edu.isi.stella.IntegerWrapper rather than java.lang.Integer. Wrappers have a field called wrapperValue which accesses the internal value. Example of use:

```
import edu.isi.stella.*;
import edu.isi.stella.javalib.*;
...
IntegerWrapper iWrap = IntegerWrapper.newIntegerWrapper(42);
...
int answer = iWrap.wrapperValue;
...
```

8.2.3.3 Using the PLI Class

To make interoperability between PowerLoom and Java a little simpler, we are providing a (PowerLoom Interface class named `PLI` which handles synchronization issues, setting and restoring the reasoning context, and the a more convenient use of some Java-native objects rather than Stella objects. Generally that means that strings are used for PowerLoom expressions and return values rather than Stella `Cons` objects.

Details about the methods can be found in the section See Chapter 8 [PowerLoom API], page 38. The names of functions in that section will need to be converted to their Java equivalents using the conventions described in See Section 8.2.3.2 [PowerLoom Java Conventions], page 53. We also provide javadoc documentation for the `edu.isi.powerloom.PLI` class. We recommend using this method for accessing PowerLoom functionality. We expect to expand the range of PowerLoom interface functions that have an analog in the `PLI` class over time.

8.2.3.4 Using Stella Objects

Stella objects can also be used directly. The most common ones used by PowerLoom users are `Module` and `LogicObject`. Other potentially useful Stella objects are `Cons`, `Symbol`, `Keyword` and `List`. Except for `LogicObject`, these are in the `edu.isi.stella` package. `LogicObject` is in the `edu.isi.powerloom.logic` package.

If one wishes to construct `Cons` objects (for example to create objects to pass to interface functions, one would begin by building items up using `Stella_Object.cons` static method, which takes a stella object and a cons. The empty cons is kept in the `edu.isi.stella.Stella.NIL` static variable. Another way to create stella objects is to use `edu.isi.stella.Stella.unstringify` static method. This method takes a string representation of a stella object and returns the object. If passed a list, an object of type `Cons` will be returned.

As an alternative, one can also convert one and two dimensional arrays of `Stella_Object` into `Cons` objects using the overloaded function `edu.isi.stella.javalib.arrayToCons`. These functions will return `Cons` objects constructed from the input arrays.

Keywords and symbols are objects that are stored in global static variables. The variable names are all in upper case and are constructed by concatenating the tag SYM with the module name and the name of the symbol or concatenating the tag KWD with the name of the keyword. For example, the symbol BACKWARD in the logic module would be stored in

```
edu.isi.powerloom.logic.Logic.SYM_LOGIC_BACKWARD
```

whereas the keyword :ERROR in the stella module would be in

```
edu.isi.stella.Stella.KWD_ERROR
```

8.2.3.5 PowerLoom and Threads

The most important consideration when using PowerLoom in a threaded environment is that the core of PowerLoom must not execute in concurrently running threads. The PLI class takes care of this for interface functions that run through that class. Other PowerLoom functions that are called need to synchronize on a lock object

```
edu.isi.powerloom.logic.Logic.$POWERLOOM_LOCK$
```

for proper operation. This is not needed for setting Special Variables, since they are implemented on a per-thread basis. The most important special variable is the reasoning context. See See Section 8.2.3.6 [Setting and Restoring Global Variable Values], page 57.

8.2.3.6 Setting and Restoring Global Variable Values

As noted above, special variables in Stella are implemented as static fields in a catchall class named the same as the Stella module. It will be in the java package corresponding to that Stella module. The values of Special variables are stored in Java objects of the type `StellaSpecialVariable`, a subclass of Java's `InheritableThreadLocal`. Any changes made to the values will not affect any other running threads. This means that the changes don't need to be synchronized. Note that global (as opposed to special) variables don't use these objects.

Numbers and boolean values are stored in special variables using the corresponding Java classes `Integer`, `Double`, `Boolean`, etc. The naming convention is to have all upper case letters with a dollar sign ($) at the beginning and end of the name.

To temporarily change the value of one of these variables, users will need to be responsible for saving and restoring the old values. Use of the `"try ... finally ..."` construct is very useful for this, since it guarantees that the restore of values will be done. An example follows of how to safely change modules. Contexts should be changed using the functions, although other global variables can be changed by using the set method. Note that we use variables of type Object to hold the values, since that avoids the need to cast when extracting the current value, since the only operation we do with the current value is save it to restore it later.

```
import edu.isi.stella.*;

// CONTEXT CHANGE.
Object savedModule = Stella.$MODULE$.get();
Module newModule
   = Stella.getStellaModule(contextName, true);
if (newModule == null) { // Handle missing module
}
try {
  Module.changeCurrentModule(newModule)
  // Code that uses the newModule

} finally {
  Module.changeCurrentModule(savedModule);
}


// INTEGER VALUE CHANGE:
```

```
Object savedValue = Stella.$SAFETY$.get();
try {
  Stella.$SAFETY$.set(new Integer(3));
  // Code that uses the newModule

} finally {
  Stella.$SAFETY$.set(savedValue);
}


// BOOLEAN VALUE CHANGE:
Object savedValue = Stella.$PRINTREADABLY$.get();
try {
  Stella.$PRINTREADABLY$.set(Boolean.TRUE);
  // Code that uses the newModule

} finally {
  Stella.$PRINTREADABLY$.set(savedValue);
}
```

The need to change the module using this type of code can be largely avoided by using the functions in the PLI interface package. They take a module argument and can handle the binding and restoration of the module value themselves.

8.2.3.7 Java Character Mapping

The full Stella to Java character mapping is the following. The character mappings use uppercase characters if the basic identifier uses mixed or lower case. The mappings use lowercase characters if the basic identifier uses upper case.

```
Stella  Java    Mnemonic
======  ====    ========
  !   =>  X     (eXclamation)
  "   =>  _
  #   =>  H     (Hash)
  $   =>  B     (Buck)
  %   =>  R     (peRcent)
  &   =>  A     (Ampersand)
  '   =>  Q     (Quote)
  (   =>  _
  )   =>  _
  *   =>  $
  +   =>  I     (Increase)
  ,   =>  _
  -   =>  _
  .   =>  D     (Dot)
  /   =>  S     (Slash)
  :   =>  C     (Colon)
```

```
      ;   =>  _
      <   =>  L      (Less than)
      =   =>  E      (Equal)
      >   =>  G      (Greater than)
      ?   =>  P      (Predicate)
      @   =>  M      (Monkey tail)
      [   =>  J      (Arbitrary (array index?))
      \   =>  _
      ]   =>  K      (Arbitrary (array index?)
      ^   =>  U      (Up arrow)
      `   =>  _
      {   =>  Y      (Arbitrary (adjacent free letter))
      |   =>  V      (Vertical bar)
      }   =>  Z      (Arbitrary (adjacent free letter))
      ~   =>  T      (Tilde)
 <space>  => _
```

8.2.3.8 Stella Exceptions in Java

Stella exceptions are implemented as a subtype of `java.lang.Exception` (actually `RunTimeException`) and may be caught normally. All Stella Exceptions belong to the `edu.isi.stella.StellaException` class or one of its subclasses. The more specfic PowerLoom exceptions belong to the `edu.isi.powerloom.logic.LogicException` class or one of its subclasses.

8.2.3.9 Iteration in Java

Iteration in Stella (and by extension) PowerLoom is organized a little bit differently than in Java. You can either use the Stella iterators directly, or else use one of the wrapper classes described in the section See Section 8.2.3.10 [Utility Classes for Java], page 60. It will present a more familiar Java interface. Since the iteration models are a bit different, it would be unwise to mix accesses between the iteration models.

Stella iterators do not compute any values until the next? method (in Java: `nextP`) is called. This method will try to compute the next value of the iterator and it will return a boolean value which is true if more values are present. Each time it is called, the iteration advances. Values can be read out of the value field of the iterator, which will have type `Stella_Object`. Some iterators will also have a key field which can be read.

The way one would normally use a Stella iterator is as follows, with possible casting of the value field:

```
Iterator iter = ...;

while (iter.nextP()) {
    processValue(iter.value);
}
```

8.2.3.10 Utility Classes for Java

To make interoperation of Stella and Java easier, there are several convenience classes for wrapping Stella iterators and having them behave like Java enumerations or iterators. These convenience classes are in the `edu.isi.stella.javalib` package:

`ConsEnumeration.java`	`Enumeration class for Cons objects`
`ConsIterator.java`	`Iterator class for Cons objects`
`StellaEnumeration.java`	`Enumeration interface to Stella's Iterator`
`StellaIterator.java`	`Iterator interface to Stella's Iterator`

All of the iterators and enumerators return objects that are actually of type `Stella_Object`, but the signature specifies `java.lang.Object` as required for compatibility with the standard Java signature. The `Cons...` classes take a `Cons` in their constructor. The `Stella...` classes take a `edu.isi.stella.Iterator` object in their constructor.

9 Built-In Relations

This chapter lists all of the relations that come predefined in PowerLoom. They are defined in the module `PL-KERNEL`; users can access them by including or using the `PL-KERNEL` module within the declarations of their own modules.

***** ((*?x* NUMBER) (*?y* NUMBER)) :-> (*?z* NUMBER) [Function]

Function that multiplies two numbers.

+ ((*?x* NUMBER) (*?y* NUMBER)) :-> (*?z* NUMBER) [Function]

Function that adds two numbers.

- ((*?x* NUMBER) (*?y* NUMBER)) :-> (*?z* NUMBER) [Function]

Function that subtracts two numbers.

/ ((*?x* NUMBER) (*?y* NUMBER)) :-> (*?z* NUMBER) [Function]

Function that divides two numbers.

< ((*?x* NUMBER) (*?y* NUMBER)) [Relation]

True if ?x < ?y.

=< ((*?x* NUMBER) (*?y* NUMBER)) [Relation]

True if ?x <= ?y.

> ((*?x* NUMBER) (*?y* NUMBER)) [Relation]

True if ?x > ?y.

>= ((*?x* NUMBER) (*?y* NUMBER)) [Relation]

True if ?x >= ?y.

ABSTRACT ((*?r* RELATION)) [Relation]

True if there are no direct assertions made to the relation ?r.

AGGREGATE ((*?a* AGGREGATE)) [Concept]

?a is an aggregate

ANTISYMMETRIC ((*?r* RELATION)) [Relation]

A binary relation ?r is antisymmetric if whenever (?r ?x ?y) is true (?r ?y ?x) is false unless ?x equals ?y.

ARITY ((*?r* RELATION)) :-> (*?arity* INTEGER) [Function]

The number of arguments/domains of the relation *?r*.

BINARY-RELATION ((*?r* RELATION)) [Concept]

The class of binary relations.

BOUND-VARIABLES ((*?arguments* THING)) [Relation]

True if all arguments are bound. The `bound-variables` predicate is used as a performance enhancer, to prevent other predicates from backchaining excessively while searching for bindings of certain of their arguments. Purists will shun the use of this predicate, but some rules are inherently inefficient without the addition of some kind of control logic. Because evaluation of the `bound-variables` predicate evaluation of predicates being `guarded`, using this predicate has the side-effect of locally disabling query optimization. (See `collect-into-set` for an example that uses `bound-variables`.

CARDINALITY ((*?c* SET)) :-> (*?card* INTEGER) [Function]

Function that returns the cardinality of a set.

CLOSED ((*?c* COLLECTION)) [Relation]

The collection ?c is closed if all of its members are known. Asserting that a relation is closed makes certain computations easier. For example, suppose that the relation `happy` is closed, implying that all things that are happy will be asserted as such. To prove `(not (happy Fred))`, PowerLoom can use a negation-as-failure proof strategy which returns TRUE if `Fred` cannot be proved to be happy. Also, if the relation `children` is closed, then a value for the expression `(range-max-cardinality children Fred)` can be inferred merely by counting the number of fillers of the `children` role on `Fred`.

COLLECT-INTO-SET ((*?c* COLLECTION)) :-> (*?l* SET) [Function]

Infer as many members of ?c as possible and collect them into a set ?l. For example, here is a rule used to compute bindings for the `fillers` predicate:

```
(<= (fillers ?r ?i ?v)
    (and (bound-variables ?r ?i)
         (collect-into-set (setofall ?v (holds ?r ?i ?v)) ?members)))
```

When `?r` and `?i` are bound, the term `(setofall ?v (holds ?r ?i ?v))` evaluates to a unary relation satisfied for each filler of the relation in `?r` applied to the instance in `?i`. `collect-into-set` causes the extension of this (dynamically-defined) unary relation to be computed. Note the use of `bound-variables` to screen out unbound variables before they are passed to the `setofall` predicate.

COLLECTION ((*?c* AGGREGATE)) [Concept]

The class of all collections. This includes all sets, lists, concepts, and relations.

COLLECTIONOF ((*?m* THING)) :-> (*?c* COLLECTION) [Function]

Abstract function existing to subsume `SETOF` and `LISTOF`.

COMMENT ((*?x* THING) (*?s* STRING)) [Relation]

?s is a comment attached to ?x. Comments are a generalization of other annotations such as `documentation` and `issue` strings.

COMMUTATIVE ((*?r* RELATION)) [Relation]

A relation ?r is commutative if its truth value is invariant with any permutation of its arguments.

CONCEPT ((*?x* RELATION)) [Concept]

The class of reified unary relations. The Powerloom notion of `concept` corresponds to the object-oriented notion of `class`. From a logic standpoint, the notion of a concept is hard to distinguish from the notion of `unary relation`. The conceptual distinction is best illustrated in the domain of linguistics, where concepts are identified with collective nouns while unary relations are identified with adjectives. For example, `Rock` is a concept, while `rocky` is a unary relation.

CONCEPT-PROTOTYPE ((*?c* CONCEPT)) :-> (*?i* THING) [Function]

Function that, given a concept, returns a prototypical instance that inherits all constraints that apply to any concept member, and has no additional constraints.

COVERING ((*?c* COLLECTION) (*?cover* SET)) [Relation]

True if ?c is a subset of the union of all collections in the set ?cover (see `disjoint-covering`).

CUT ((*?arguments* THING)) [Relation]

Prolog-like CUT. Succeeds the first time and then fails. Side-effect: Locally disables query optimization.

DIRECT-SUBRELATION ((*?r* RELATION) (*?sub* RELATION)) [Relation]

True iff ?sub is a direct subrelation of ?r; written in set notation, ?sub `<` ?r, and there is no ?s such that ?sub `<` ?s `<` ?r. This relation will generate bindings for at most one unbound argument.

DIRECT-SUPERRELATION ((*?r* RELATION) (*?super* RELATION)) [Relation]

True iff ?super is a direct superrelation of ?r; in set notation, ?super `>` ?r, and there is no ?s such that ?super `>` ?s `>` ?r. This relation will generate bindings for at most one unbound argument.

DISJOINT ((*?c1* COLLECTION) (*?c2* COLLECTION)) [Relation]

True if the intersection of ?c1 and ?c2 is empty.

DISJOINT-COVERING ((*?c* COLLECTION) (*?disjointcover* SET)) [Relation]

True if ?c is covered by the collections in ?disjointCover and if the member sets in ?disjointCover are mutually-disjoint. For example the concepts `Igneous-Rock`, `Metamorphic-Rock`, and `Sedimentary-Rock` together form a disjoint covering of the concept `Rock`.

DOCUMENTATION ((*?x* THING) (*?s* STRING)) [Relation]

?s is a documentation string attached to ?x. Some of the PowerLoom text processing tools look for documentation strings and import them into documents.

DOMAIN ((*?r* RELATION) (*?d* CONCEPT)) [Relation]

True if for any tuple `T` that satifies ?r, the first argument of `T` necessarily belongs to the concept ?d. `domain` exists for convenience only and is defined in terms of `nth-domain`. `domain` assertions should be avoided, since they create redundant `nth-domain` propositions (use `nth-domain` directly).

DUPLICATE-FREE ((*?c* COLLECTION)) [Relation]

?c is duplicate-free if no two members denote the same object.

DUPLICATE-FREE-COLLECTION ((*?c* DUPLICATE-FREE)) [Concept]

?c is free of duplicates

EMPTY ((*?c* COLLECTION)) [Relation]

The collection ?c is empty if it has no members. Note that for collections possessing open-world semantics, (e.g., most concepts) the fact that the collection has no known members does not necessarily imply that it is empty.

EQUIVALENT-RELATION ((*?r* RELATION) (*?equiv* RELATION)) [Relation]

True if ?r is equivalent to ?equiv; written in set notation, ?r = ?equiv. This relation will generate bindings for at most one unbound argument.

EXAMPLE ((*?r* RELATION) (*?e* THING)) [Relation]

?e is an example of (the use of) ?r.

FILLERS ((*?r* RELATION) (*?i* THING)) *:->* (*?members* SET) [Function]

Given a relation ?r and instance ?i, returns a set of known fillers of ?r applied to ?i. IMPORTANT: this also collects intensional fillers such as skolems that might be identical extensionally.

FRAME-PREDICATE ((*?c* RELATION)) [Concept]

A frame predicate is a second-order relation that is used to describe constraints on the set of fillers for a binary relation applied to an instance. Examples of frame predicates are `range-cardinality`, `range-type`, and `numeric-minimum`. Frame predicates are typically used to capture the kinds of relations manipulated by description logic systems such as USC/ISI's Loom system.

FUNCTION ((*?r* RELATION)) [Concept]

A relation is a function if its last argument is a function of its first n-1 arguments, i.e., if it is a single-valued relation. Functions explicitly declared as such differ from relations in that they may appear syntactically as a term applied to n-1 arguments. For example, to express the sentence "two plus two equals four", because `+` is a function we can write `(= (+ 2 2) 4)`. The same sentence written in relational syntax would look like `(+ 2 2 4)`. If a relation is introduced using the `defrelation` syntax and also declared to be single-valued, the functional syntax does not apply; only the explicit use of `deffunction` sanctions the use of that syntax.

GOES-FALSE-DEMON ((*?r* RELATION) (*?computation* COMPUTED-PROCEDURE)) [Relation]

Names a `computation` (a function) that is attached (logically) to ?r Each time a proposition with predicate ?r becomes false, the function is applied to that proposition.

GOES-TRUE-DEMON ((*?r* RELATION) (*?computation* COMPUTED-PROCEDURE)) [Relation]

Names a `computation` (a function) that is attached (logically) to ?r Each time a proposition with predicate ?r becomes true, the function is applied to that proposition.

GOES-UNKNOWN-DEMON ((*?r* RELATION) (*?computation* COMPUTED-PROCEDURE)) [Relation]

Names a `computation` (a function) that is attached (logically) to ?r Each time a proposition with predicate ?r becomes unknown, the function is applied to that proposition.

HOLDS ((*?relation* RELATION) (*?arguments* THING)) [Relation]

True if the tuple *?arguments* is a member of the relation *?relation*. `holds` is a variable arity predicate that takes a relation as its first argument, and zero or more additional arguments. It returns values equivalent to a subgoal that has the first argument as a predicate and the remaining arguments shifted one place to the left. For `holds` to succeed, the (first) relation argument must be bound – PowerLoom will NOT cycle through all relations searching for ones that permit the proof to succeed. However, users can obtain the same effect if they choose by using other second-order predicates to generate relation bindings. For example, the query

```
(retrieve all ?x (and (Relation ?r)
                      (holds ?r Fred ?x)))
```

retrieves all constants for which there is some binary relation that relates `Fred` to that relation.

IMAGE-URL ((*?x* THING) (*?url* STRING)) [Relation]

?url is a URL pointing to an image illustrating ?x. The Ontosaurus browser looks for `image-url` values attached to objects it is presenting, and displays them prominently, thereby spiffing up its displays.

INEQUALITY ((*?x* NUMBER) (*?y* NUMBER)) [Relation]

Abstract superrelation of inequality relations.

INSTANCE-OF ((*?x* THING) (*?c* CONCEPT)) [Relation]

True if ?x is an instance of ?c. Can be used to generate concept values of ?c, given an instance ?x.

INVERSE ((*?r* BINARY-RELATION)) :-> (*?inverserelation* THING) [Function]

Function that returns the inverse relation for ?r. PERFORMANCE NOTE: for best results there should be only one `(inverse R I)` assertion per relation pair `R` and `I`. In that case `R` is viewed as the canonical relation and `I` simply provides a different access mechanism to the canonical relation. In a logic-based KR paradigm inverse relations are redundant and do not add anything that couldn't be represented or queried without them, however, sometimes they can provide some extra convenience for users. Asserting `(inverse I R)` also will not cause an error but can degrade backward inference performance due to the extra redundant rule that gets generated. If domain rules will be written in terms of both `R` and `I` (as opposed to only `R`), `(inverse I R)` should be asserted also to get full inferential connectivity between the two relations.

IRREFLEXIVE ((*?r* RELATION)) [Relation]

A binary relation ?r is irreflexive if it is false when both of its arguments are identical.

ISSUE ((*?x* THING) (*?s* STRING)) [Relation]

?s is an issue attached to ?x. An issue string normally comments on a topic that has not been resolved to everyone's satisfaction.

IST ((*?context* CONTEXT) (*?p* PROPOSITION)) [Relation]

True if proposition *?p* is true in context *?context*. The `IST` (is true) relation allows one to evaluate a query or rule in more than one context. A common use of `IST` is in defining `lifting axioms` that import knowledge from one context to another. For example, below is a rule that accesses a `patient-record` relation in a module called `Medical-Kb`, `lifts-out` the `age` column, and imports it into a `has-age` relation in the current context.

```
(<= (has-age ?person ?age)
    (and (has-ssn ?person ?ssn)
         (exists (?1 ?2 ?3 ?4)
           (ist Medical-Kb (patient-record ?ssn ?1 ?2 ?age ?3 ?4))))))
```

LENGTH ((*?x* THING)) :-> (*?z* INTEGER) [Function]

Function that returns the length of a string or a logical list. NOT YET IMPLEMENTED FOR LISTS.

LENGTH-OF-LIST ((*?l* COLLECTION) (*?length* INTEGER)) [Relation]

Computes the length of the list or set ?l.

LEXEME ((*?r* THING) (*?s* STRING)) [Relation]

?s is a lexeme for the relation or individual ?r. A relation or individual ?r can have zero or more lexemes, words that are natural langage equivalents of a logical constant. The same lexeme may be attached to more than one constant.

LIST ((*?l* ORDERED)) [Concept]

A list is an ordered collection of elements. The range of the function `listof` consists of elements of type `List`.

LISTOF ((*?m* THING)) :-> (*?c* LIST) [Function]

Term-forming function that defines an ordered list consisting of all function arguments. Within logical expressions `listof` is most commonly used in conjunction with the `member-of` predicate. For example the query

```
(retrieve ?x (member-of ?x (listof a b c)))
```

returns the constants `a`, `b`, and `c` on successive iterations.

MAXIMUM-VALUE ((*?l* COLLECTION) (*?max* NUMBER)) [Relation]

Binds ?max to the maximum of the numbers in the list ?l.

MEAN-VALUE ((*?l* COLLECTION) (*?mean* NUMBER)) [Relation]

Binds ?mean to the mean of the numbers in ?l.

MEDIAN-VALUE ((*?l* COLLECTION) (*?median* NUMBER)) [Relation]

Binds ?median to the median of the numbers in ?l.

MEMBER-OF ((*?x* THING) (*?c* COLLECTION)) [Relation]

TRUE if ?x is a member of collection ?c. A common use of `member-of` is for binding a variable to successive members in a list or set (see `listof` and `setof`).

MINIMUM-VALUE ((*?l* COLLECTION) (*?min* NUMBER)) [Relation]

Binds ?min to the minimum of the numbers in the list ?l.

MUTUALLY-DISJOINT-COLLECTION ((*?s* SET)) [Relation]

True if the members of ?s are pair-wise disjoint. Used most often to expresse disjointness constraints between concepts. For example

```
(mutually-disjoint-collection (setof MAN WOMAN))
```

states that the concepts MAN and WOMAN are disjoint.

NTH-DOMAIN ((*?r* RELATION) (*?i* INTEGER) (*?d* CONCEPT)) [Relation]
True if the nth value for a tuple T satisfying ?r must belong to the concept ?d. Argument counting starts at zero.

NUMERIC-MAXIMUM ((*?r* RELATION) (*?i* THING) (*?n* NUMBER)) [Relation]
Relation that specifies an upper bound *?n* on any numeric value that can belong to the set of fillers of the relation ?r applied to ?i.

NUMERIC-MINIMUM ((*?r* RELATION) (*?i* THING) (*?n* NUMBER)) [Relation]
Relation that specifies a lower bound *?n* on any numeric value that can belong to the set of fillers of the relation ?r applied to ?i.

NUMERIC-SET ((*?s* COLLECTION)) [Concept]
?s is a set of numbers

ORDERED ((*?c* COLLECTION)) [Relation]
?c is ordered if the ordering of its members is significant. Lists are ordered, while sets are not.

PHRASE ((*?r* THING) (*?s* STRING)) [Relation]
A phrase is a variablized sentence, a template, that is used to express individual axiomatic facts as natural language sentences. By convention, a phrase contains one or more occurrences of each variable in a relation or concept definition, it does not begin with a capital letter, and it has no concluding period. Systematic attachment of phrases to relations can be leveraged by tools that generate natural language paraphrases of logic sentences.

PROJECT-COLUMN ((*?i* INTEGER) (*?c* COLLECTION)) :-> (*?l* LIST) [Function]
Project elements in column ?i (zero-based) of the tuples of ?c and collect them into a list ?l.

PROPER-SUBRELATION ((*?r* RELATION) (*?sub* RELATION)) [Relation]
True iff ?sub is a proper subrelation of ?r; written in set notation, ?sub `<` ?r. This relation will generate bindings for at most one unbound argument.

PROPER-SUPERRELATION ((*?r* RELATION) (*?super* RELATION)) [Relation]
True iff ?super is a proper superrelation of ?r; written in set notation, ?super `>` ?r. This relation will generate bindings for at most one unbound argument.

RANGE ((*?r* RELATION) (*?rng* CONCEPT)) [Relation]
True if for any tuple `T` that satifies ?r, the last argument of `T` necessarily belongs to the concept ?rng. `range` exists for convenience only and is defined in terms of `nth-domain`. `range` assertions should be avoided, since they create redundant `nth-domain` propositions (use `nth-domain` directly).

RANGE-CARDINALITY ((*?r* RELATION) (*?i* THING)) :-> (*?card* INTEGER) [Function]
Function that returns the cardinality of the set of fillers of the relation ?r applied to ?i. The cardinality function returns a value only when the relations `range-min-cardinality` and `range-max-cardinality` compute identical values, i.e., when the best lower and upper bounds on the cardinality are equal. Each of these bounding functions employs a variety of rules to try and compute a tight bound.

RANGE-CARDINALITY-LOWER-BOUND ((*?r* RELATION) (*?i* THING) (*?lb* INTEGER)) [Relation]

Relation that specifies a lower bound on the cardinality of the set of fillers of the relation ?r applied to ?i. The difference between `range-cardinality-lower-bound` and `range-min-cardinality` is subtle but significant. Suppose we state that nine is a lower bound on the number of planets in the solar system, and then ask if eight is (also) a lower bound:

```
(assert (range-cardinality-lower-bound hasPlanets SolarSystem 9))
(ask (range-cardinality-lower-bound hasPlanets SolarSystem 8))  ==> TRUE
```

PowerLoom will return TRUE. However if we ask if the minimum cardinality of the solar system's planets is eight, we get back UNKNOWN

```
(ask (range-min-cardinality hasPlanets SolarSystem 8)) ==> UNKNOWN
```

because eight is not the tightest lower bound.

RANGE-CARDINALITY-UPPER-BOUND ((*?r* RELATION) (*?i* THING) (*?ub* INTEGER)) [Relation]

Relation that specifies an upper bound on the cardinality of the set of fillers of the relation ?r applied to ?i. (see the discussion for `range-cardinality-lower-bound`).

RANGE-MAX-CARDINALITY ((*?r* RELATION) (*?i* THING)) :-> (*?maxcard* INTEGER) [Function]

Returns the strictest computable upper bound on the cardinality of the set of fillers of the relation ?r applied to ?i. (see the discussion for `range-cardinality-lower-bound`).

RANGE-MIN-CARDINALITY ((*?r* RELATION) (*?i* THING)) :-> (*?mincard* INTEGER) [Function]

Returns the strictest computable lower bound on the cardinality of the set of fillers of the relation ?r applied to ?i. (see the discussion for `range-cardinality-lower-bound`).

RANGE-TYPE ((*?r* RELATION) (*?i* THING) (*?type* CONCEPT)) [Relation]

Relation that specifies a type/range of the relation ?r applied to ?i. Multiple range types may be asserted for a single pair <?r,?i>. Technically, a retrieval of types for a given pair should include all supertypes (superconcepts) of any type that is produced, but for utility's sake, only asserted or directly inferrable types are returned.

REFLEXIVE ((*?r* RELATION)) [Relation]

A binary relation ?r is reflexive if it is always true when both of its arguments are identical.

RELATION ((*?x* SET)) [Concept]

The class of relations. This includes all concepts and all functions.

RELATION-COMPUTATION ((*?r* RELATION) (*?computation* COMPUTED-PROCEDURE)) [Relation]

Names a `computation` (a function) that evaluates an (atomic) relation proposition during query processing. The function is passed a proposition for evaluation for which all arguments are bound. The function returns a BOOLEAN if it represents a predicate, or some sort of value if it is a function.

RELATION-CONSTRAINT ((*?r* RELATION) (*?computation* COMPUTED-PROCEDURE)) [Relation]

Names a `computation` (a function) that evaluates an (atomic) relation proposition during query processing. The function is passed a proposition for evaluation for which at most one argument is unbound. The function returns a BOOLEAN if it represents a predicate, or some sort of value if it is a function. If all arguments are bound the function computes whether the constraint holds. If all but one argument is bound and the unbound argument is a pattern variable then the missing value is computed.

RELATION-EVALUATOR ((*?r* RELATION) (*?ev* COMPUTED-PROCEDURE)) [Relation]

Names an `evaluator` (a function) that evaluates an (atomic) relation proposition during constraint propagation. This defines an extensible means for computing using auxiliary data structures. The function is passed a proposition for evaluation. All side-effects are made by updates to the proposition, or by generating other assertions.

RELATION-SPECIALIST ((*?r* RELATION) (*?sp* COMPUTED-PROCEDURE)) [Relation]

Names a `specialist` (a function) that evaluates an (atomic) relation proposition during query processing. This defines an extensible means for computing with the control stack. The function is passed a CONTROL-FRAME that contains the proposition, and returns a keyword :FINAL-SUCCESS, :CONTINUING-SUCCESS, :FAILURE, or :TERMINAL-FAILURE that controls the result of the computation.

SCALAR ((*?x* SCALAR)) [Concept]

The class of scalar quantities.

SCALAR-INTERVAL ((*?x* SCALAR)) [Concept]

An interval of scalar quantities.

SET ((*?s* DUPLICATE-FREE-COLLECTION)) [Concept]

This class denotes the mathematical notion of a `set`; a collection that has no duplicates.

SETOF ((*?m* THING)) :-> (*?c* SET) [Function]

Term-forming function that defines an enumerated set consisting of all function arguments. `setof` is like `listof` except that it removes duplicate values.

SINGLE-VALUED ((*?c* RELATION)) [Relation]

The relation ?c is single-valued if the value of its last argument is a function of all other arguments. All functions are single-valued (see `function`).

SQUARE-ROOT ((*?x* NUMBER) (*?y* NUMBER)) [Relation]

Relation that returns the positive and negative square roots: ?y = sqrt(?x). For positive roots only see function SQRT.

STANDARD-DEVIATION ((*?l* COLLECTION) (*?sd* NUMBER)) [Relation]

Binds ?sd to the standard deviation of the numbers in ?l.

STRING-CONCATENATE ((*?x1* STRING) (*?x2* STRING)) *:->* (*?x3* STRING) [Function]

Concatenate strings ?x1 and ?x2 and bind ?x3 to the result.

SUBRELATION ((*?r* RELATION) (*?sub* RELATION)) [Relation]

True iff ?sub is a subrelation of ?r; written in set notation, ?sub =< ?r. This relation will generate bindings for at most one unbound argument.

SUBSET-OF ((*?sub* COLLECTION) (*?super* COLLECTION)) [Relation]

True if ?sub is a subset of ?super. For performance reasons, the `subset-of` predicate refuses to search for bindings if both of its variables are unbound. Implementation note: `subset-of` is treated specially internally to PowerLoom, and hence Powerloom does not permit the augmentation of `subset-of` with additional inference rules. In otherwords, `subset-of` behaves semantically like an operator instead of a relation.

SUBSTRING ((*?s* STRING) (*?start* INTEGER) (*?end* INTEGER)) *:->* (*?sub* STRING) [Function]

Generate the substring of ?s starting at position ?start (zero-based), ending just before position ?end and bind ?sub to the result. This is the PowerLoom equivalent to the STELLA method `subsequence`. In addition, this function can be used to locate substrings in strings by supplying values for ?s and ?sub and allowing ?start and ?end to be bound by the function specialist. In other words, (retrieve all (?start ?end) (substring "foo" ?start ?end "o")) ==> ?start = 1, ?end = 2, ?start = 2, ?end = 3.

SUM ((*?l* COLLECTION) (*?sum* NUMBER)) [Relation]

Binds ?sum to the sum of the numbers in the list ?l.

SUPERRELATION ((*?r* RELATION) (*?super* RELATION)) [Relation]

True iff ?super is a superrelation of ?r; written in set notation, ?super >= ?r. This relation will generate bindings for at most one unbound argument.

SYMMETRIC ((*?r* RELATION)) [Relation]

A binary relation ?r is symmetric if it is commutative.

SYNONYM ((*?term* THING) (*?synonym* THING)) [Relation]

Assert that *?synonym* is a synonym of *?term*. This causes all references to *?synonym* to be interpreted as references to *?term*. Retraction eliminates a synonym relation.

TOTAL ((*?r* FUNCTION)) [Relation]

True if the function *?r* is defined for all combinations of inputs. By default, functions are not assumed to be total (unlike Prolog, which `does` make such an assumption. For example, if we define a two-argument function `foo` and then retrieve its value applied to some random instances `a` and `b`, we get nothing back:

```
(deffunction foo (?x ?y) :-> ?z)
(retrieve ?x (= ?x (foo a b)))
```

However, if we assert that `foo` is total, then we get a skolem back when we execute the same retrieve:

```
(assert (total foo))
(retrieve ?x (= ?x (foo a b)))
```

TRANSITIVE ((*?r* RELATION)) [Relation]

A binary relation ?r is transitive if (?r ?x ?y) and (?r ?y ?z) implies that (?r ?x ?z). Note that functions cannot be transitive, since their single-valuedness would not allow multiple different values such as (?r ?x ?y) and (?r ?x ?z) due to the Unique Names Assumption made by PowerLoom.

TYPE-OF ((*?c* CONCEPT) (*?x* THING)) [Relation]

True if ?x is a member of the concept ?c.

VALUE ((*?function* FUNCTION) (*?arguments* THING)) :-> (*?value* THING) [Function]

True if applying *?function* to *?arguments* yields the value `?value`. The `value` predicate is the analog of `holds`, except that it applies to functions instead of relations.

VARIABLE-ARITY ((*?r* RELATION)) [Relation]

Asserts that the relation ?r can take a variable number of arguments.

VARIANCE ((*?l* COLLECTION) (*?variance* NUMBER)) [Relation]

Binds ?variance to the variance of the numbers in ?l.

10 Miscellaneous

This is a catch-all section for documented functions, methods and relations that haven't been categorized yet into any of the previous sections. They are in random order and many of them will never be part of the official PowerLoom interface. So beware!

2-d-element ((*array* 2-D-ARRAY) (*row* INTEGER) (*column* INTEGER)) *:* (LIKE (ANY-VALUE SELF)) [Method]
Return the element of *array* at position [*row*, *column*].

2-d-element ((*array* 2-D-FLOAT-ARRAY) (*row* INTEGER) (*column* INTEGER)) *:* FLOAT [Method]
Return the element of *array* at position [*row*, *column*].

2-d-element-setter ((*array* 2-D-ARRAY) (*value* OBJECT) (*row* INTEGER) (*column* INTEGER)) *:* (LIKE (ANY-VALUE SELF)) [Method]
Set the element of *array* at position [*row*, *column*] to *value* and return the result.

2-d-element-setter ((*array* 2-D-FLOAT-ARRAY) (*value* FLOAT) (*row* INTEGER) (*column* INTEGER)) *:* (LIKE (ANY-VALUE SELF)) [Method]
Set the element of *array* at position [*row*, *column*] to *value* and return the result.

add-testing-example ((*form* CONS) (*score* PARTIAL-MATCH-SCORE)) *:* [Command]
Add a query and score pair to the master list of testing examples

add-training-example ((*form* CONS) (*score* PARTIAL-MATCH-SCORE)) *:* [Command]
Add a query and score pair to the master list of training examples

all-asserted-types ((*self* OBJECT)) *:* (CONS OF NAMED-DESCRIPTION) [Function]
Return a set of all of the types that are asserted to be satisfied by *self*.

all-class-instances ((*type* SURROGATE)) *:* CONS [Function]
Return a set of instances that belong to the class *type*.

all-cycles ((*module* MODULE) (*local?* BOOLEAN)) *:* (CONS OF CONS) [Function]
Return a list of lists of descriptions that are provably co-extensional.

all-direct-subrelations ((*relation* NAMED-DESCRIPTION) (*removeEquivalents?* BOOLEAN)) *:* (CONS OF NAMED-DESCRIPTION) [Function]
Return a set of relations that immediately specialize *relation*. If *removeEquivalents?* (recommended), don't include any relations equivalent to *relation*.

all-direct-superrelations ((*relation* NAMED-DESCRIPTION) (*removeEquivalents?* BOOLEAN)) *:* (CONS OF NAMED-DESCRIPTION) [Function]
Return a set of relations that immediately subsume *relation*. If *removeEquivalents?* (recommended), don't include any relations equivalent to *relation*.

all-direct-types ((*self* OBJECT)) *:* (CONS OF LOGIC-OBJECT) [Function]
Return a set of most specific types that are satisfied by *self*.

all-equivalent-relations ((*relation* NAMED-DESCRIPTION) (*reflexive?* BOOLEAN)) : (CONS OF NAMED-DESCRIPTION) [Function]

Return a list of all relations equivalent to *relation*. If *reflexive?*, include *relation* in the list.

all-facts-of-instance ((*self* OBJECT) (*includeunknownfacts?* BOOLEAN) (*elaborate?* BOOLEAN)) : (LIST OF PROPOSITION) [Function]

Return a list of all definite (TRUE or FALSE) propositions attached to *self*.

all-facts-of-n ((*n* INTEGER) &rest (*instanceRefs* NAME)) : (CONS OF PROPOSITION) [Command]

This is a generalization of `all-facts-of` (which see). With $n = 0$ and only one instance this command behaves just like `all-facts-of`. Otherwise, returns a cons list of all definite (TRUE or FALSE) propositions that reference any of the instances listed in *instanceRefs*, plus if *n* >= 1 all propositions that reference any instances that are arguments of propositions collected in the previous step, plus if *n* >= 2... and so on. That is, if we only consider binary propositions, this can be viewed as growing a graph with instances as its nodes and predicates as its arcs starting from the set of seed *instanceRefs* to depth *n*-1. Caution: with a fully connected KB and large enough *n* this could return the whole knowledge base.

The returned propositions include those asserted to be true or false by default, but it does not include propositions that are found to be true only by running the query engine. Facts inferred to be true by the forward chainer will be included. Hence, the returned list of facts may be longer in a context where the forward chainer has been run then in one where it has not (see `run-forward-rules`).

all-instances ((*module* MODULE) (*local?* BOOLEAN)) : (ITERATOR OF LOGIC-OBJECT) [Function]

Iterate over all instances (or individuals) visible from *module*. Only instances that haven't been deleted will be considered. If *local?*, only return instances created locally in *module*.

all-named-descriptions ((*module* MODULE) (*local?* BOOLEAN)) : (ITERATOR OF NAMED-DESCRIPTION) [Function]

Iterate over all named descriptions visible from *module*. If *local?*, return only named descriptions interned in *module*. If *module* is null, return all named descriptions interned everywhere.

all-named-instances ((*module* MODULE) (*local?* BOOLEAN)) : (ITERATOR OF LOGIC-OBJECT) [Function]

Iterate over all named instances (or individuals) visible from *module*. Only instances that haven't been deleted will be considered. If *local?*, only return instances created locally in *module*.

all-named-terms ((*module* MODULE) (*local?* BOOLEAN)) : (ITERATOR OF OBJECT) [Function]

Iterate over all named terms visible from *module*. A term can be an instance (or individual) as well as a description. Only terms that haven't been deleted will be considered. If *local?*, only return terms created locally in *module*.

all-propositions ((*module* MODULE) (*local?* BOOLEAN)) *:* (ITERATOR OF PROPOSITION) [Function]

Iterate over all conceived propositions visible from *module*. Only propositions that haven't been deleted will be considered. If *local?*, only return propositions conceived locally in *module*.

all-relation-values ((*relation* SURROGATE) (*nMinusOneArguments* CONS)) *:* CONS [Function]

Return a set of values that satisfy the relation *relation* (a surrogate) applied to *nMinusOneArguments* plus that last value.

all-sentences-of ((*instanceRef* OBJECT)) *:* (CONS OF STRING-WRAPPER) [Command]

Return a list of sentences describing facts about *instanceRef*.

all-slot-value-types ((*self* LOGIC-OBJECT) (*relation* SURROGATE)) *:* (CONS OF NAMED-DESCRIPTION) [Function]

Return a set of the most specific types for fillers of the slot *relation* applied to *self*.

all-slot-values ((*self* LOGIC-OBJECT) (*relation* SURROGATE)) *:* CONS [Function]

Return a set of values for the slot *relation* (a surrogate) applied to *self* (an object).

all-subrelations ((*relation* NAMED-DESCRIPTION) (*removeequivalents?* BOOLEAN)) *:* (CONS OF NAMED-DESCRIPTION) [Function]

Return a set of all (named) relations that specialize relation.

all-superrelations ((*relation* NAMED-DESCRIPTION) (*removeequivalents?* BOOLEAN)) *:* (CONS OF NAMED-DESCRIPTION) [Function]

Return a set of all relations that subsume relation.

all-terms ((*module* MODULE) (*local?* BOOLEAN)) *:* (ITERATOR OF OBJECT) [Function]

Return a list of all terms visible from *module*. A term can be an instance (or individual) as well as a description. Only terms that haven't been deleted will be considered. If *local?*, only return terms created locally in *module*.

all-types ((*self* OBJECT)) *:* (CONS OF NAMED-DESCRIPTION) [Function]

Return a set of all of the types that are satisfied by *self*.

all-unnamed-terms ((*module* MODULE) (*local?* BOOLEAN)) *:* ITERATOR [Function]

Iterate over all unnamed terms visible from *module*. A term can be an instance (or individual) as well as a description. Only terms that haven't been deleted will be considered. If *local?*, only return terms created locally in *module*.

allocate-supported-closure-iterator ((*startnode* CONS) (*allocateadjacencyiterator* FUNCTION-CODE) (*filterfunction* FUNCTION-CODE)) *:* SUPPORTED-CLOSURE-ITERATOR [Function]

Similar to `allocate-transitive-closure-iterator` (which see), but return a SUPPORTED-CLOSURE-ITERATOR instead.

allocate-transitive-closure-iterator ((*startNode* OBJECT) (*allocateAdjacencyIterator* FUNCTION-CODE) (*filterFunction* FUNCTION-CODE)) : ITERATOR [Function]

Return an iterator that generates the transitive closure of applying iterators generated by *allocateAdjacencyIterator* to *startNode*. If *filterFunction* is non-null, that function is applied as a filter to each node generated (nodes filtered out still generate descendants, but they don't get returned).

apply-ask (&body (*body* CONS)) : OBJECT [Macro]

Execute a yes/no query composed of input-variables `inputVariables` and body `queryBody`. Before executing, bind variables to `inputBindings` (in sequence). `(apply-ask inputVariables queryBody inputBindings)`

apply-kappa? ((*description* DESCRIPTION) (*vector* VECTOR)) : BOOLEAN [Function]

Apply (inherit) the description *description* to members of the vector *vector*. Return TRUE if no clash was detected. Constraint propagation happens only if it is enabled prior to calling `apply-kappa?`.

apply-retrieve (&body (*body* CONS)) : OBJECT [Macro]

Execute a query composed of io-variables `variables` and body `queryBody`. Before executing, bind variables to `inputBindings` (in sequence). If one variable is left unbound, returns a cons list of bindings of that variable. If two or more are unbound, returns a cons list of cons lists of bindings: `(apply-retrieve variables queryBody inputBindings)`

ask-partial (&rest (*proposition*&options PARSE-TREE)) : FLOAT [Command]

Similar to `ask` (which see), but return the highest partial match score for the supplied proposition instead of a truth value. If the option :MAXIMIZE-SCORE? is set to FALSE, return after the first partial match score has been generated.

bottom? ((*self* OBJECT)) : BOOLEAN [Function]

Return TRUE if *self* is the undefined individual BOTTOM.

call-all-facts-of ((*instanceRef* OBJECT)) : (LIST OF PROPOSITION) [Command]

Return a list of all definite (TRUE or FALSE) propositions that reference the instance *instanceRef*.

call-ask ((*query* OBJECT)) : TRUTH-VALUE [Function]

Callable version of `ask` (which see). Accepts queries specified by a query iterator, or specified as a CONS-list of arguments as they would be supplied to `ask`. Raises LOGIC-EXCEPTIONs in case of illegal queries and logical expressions.

call-defconcept ((*arguments* CONS)) : NAMED-DESCRIPTION [Function]

Callable version of the `defconcept` command (which see). Expects the same arguments as `defconcept` but supplied as a list.

call-deffunction ((*arguments* CONS)) : NAMED-DESCRIPTION [Function]

Callable version of the `deffunction` command (which see). Expects the same arguments as `deffunction` but supplied as a list.

call-defobject ((*arguments* CONS)) *:* LOGIC-OBJECT [Function]
Callable version of the `defobject` command (which see). Expects the same arguments as `defobject` but supplied as a list.

call-defproposition ((*arguments* CONS)) *:* PROPOSITION [Function]
Callable version of the `defproposition` command (which see). Expects the same arguments as `defproposition` but supplied as a list.

call-defrelation ((*arguments* CONS)) *:* NAMED-DESCRIPTION [Function]
Callable version of the `defrelation` command (which see). Expects the same arguments as `defrelation` but supplied as a list.

call-list-undefined-relations ((*module* MODULE) (*local?* BOOLEAN)) *:* CONS [Function]
Callable version of `list-undefined-relations` (which see).

call-propagate-constraints ((*context* CONTEXT)) *:* [Function]
Trigger constraint propagation over all propositions in the module or world *context*.

call-retrieve ((*query* OBJECT)) *:* QUERY-ITERATOR [Function]
Callable version of `retrieve` (which see). Accepts queries specified by a query iterator, or specified as a CONS-list of arguments as they would be supplied to `retrieve`. Raises LOGIC-EXCEPTIONs in case of illegal queries and logical expressions.

call-retrieve-partial ((*query* OBJECT)) *:* QUERY-ITERATOR [Function]
Callable version of `retrieve-partial` (which see). Accepts queries specified by a query iterator, or specified as a CONS-list of arguments as they would be supplied to `retrieve-partial`. Raises LOGIC-EXCEPTIONs in case of illegal queries and logical expressions.

call-run-forward-rules ((*module* MODULE) (*force?* BOOLEAN)) *:* [Function]
Run forward inference rules in module *module*. If *module* is NULL, the current module will be used. If forward inferencing is already up-to-date in the designated module, no additional inferencing will occur, unless *force?* is set to TRUE, in which case all forward rules are run or rerun.

call-set-inference-level ((*levelKeyword* KEYWORD) (*module* MODULE)) *:* KEYWORD [Function]
Set the inference level of *module* to the level specified by *levelKeyword*. If *module* is NULL and we are inside a query, set the level of the current query iterator. Otherwise, set the level globally.

class? ((*objectRef* OBJECT)) *:* BOOLEAN [Function]
Return TRUE if *objectRef* denotes a class.

coerce-to-instance ((*self* OBJECT) (*original* OBJECT)) *:* LOGIC-OBJECT [Function]
Return the logic instance referred to by *self*.

coerce-to-instance-or-literal ((*self* OBJECT) (*original* OBJECT)) *:* OBJECT [Function]
Return the logic instance referred to by *self*, or *self* if it is a literal (e.g., string or number) that can't be coerced.

coerce-to-vector ((*self* OBJECT)) *:* VECTOR [Function]
Return a vector containing the elements in *self*. Coerce each element of *self* to be a logic object or literal.

collection? ((*objectRef* OBJECT)) *:* BOOLEAN [Function]
Return TRUE if *objectRef* denotes a relation or a class.

conceive-term ((*tree* OBJECT)) *:* OBJECT [Command]
tree is a term expression (a string or an s-expression), or is a class reference (a symbol or surrogate). Return a (possibly newly-conceived) term representing the internalized representation of that term.

conjoin-truth-values ((*tv1* TRUTH-VALUE) (*tv2* TRUTH-VALUE)) *:* TRUTH-VALUE [Function]
Return the logical conjunction of truth values *tv1* and *tv2*.

consify ((*self* QUERY-ITERATOR)) *:* CONS [Method]
Generate all solutions for the query self, and collect them into a cons list of result tuples. If `:SINGLETONS? TRUE`, collect a list of atoms rather than a list of lists for tuples of arity=1.

consify ((*self* QUERY-SOLUTION-TABLE)) *:* CONS [Method]
Collect all solutions of *self* into a cons list and return the result.

consify ((*self* JUSTIFICATION)) *:* CONS [Method]
Return a CONS tree representation of the proof *self*. Each proof step is represented as a CONS tree of the form (<proposition> (<key> <value>...) <antecedent>...) where each <antecedent> is a CONS tree representing a subproof. The consification follows the original proof structure literally, i.e., no uninteresting nodes such as patterns or AND-introductions are suppressed.

consify-current-solutions ((*self* QUERY-ITERATOR)) *:* CONS [Method]
Collect the current solutions of *self* into a cons list of result tuples. If `:SINGLETONS? TRUE`, collect a list of atoms rather than a list of lists for tuples of arity=1.

consify-justification ((*self* JUSTIFICATION) (*style* KEYWORD)) *:* CONS [Function]
Return a CONS tree representation of the proof *self*. Each proof step is represented as a CONS tree of the form (<proposition> (<key> <value>...) <antecedent>...) where each <antecedent> is a CONS tree representing a subproof. *style* indicates what nodes in the proof tree should be suppressed. :RAW preserves the original structure literally, :VERBOSE keeps AND- introductions but suppresses all auxiliary (non-logical) nodes such as pattern nodes, and :BRIEF additionally suppresses AND-introduction nodes.

constant? ((*objectRef* OBJECT)) *:* BOOLEAN [Function]
Return TRUE if *objectRef* denotes a literal or scalar.

copy ((*self* JUSTIFICATION)) *:* (LIKE SELF) [Method]
Return a copy of the proof starting at *self*. Allocates all new justification objects, but structure-shares other information such as propositions and substitutions.

create ((*name* GENERALIZED-SYMBOL) &rest (*type* GENERALIZED-SYMBOL)) : OBJECT [Command]

Create a logic object with name *name* and return it. If *type* is also supplied, assert that the object belongs to that type.

create-2-d-array ((*nof-rows* INTEGER) (*nof-columns* INTEGER) &rest (*values* OBJECT)) : 2-D-ARRAY [Function]

Create a two-dimensional array with *nof-rows* rows and *nof-columns* columns, and initialize it in row-major-order from *values*. Missing values will be padded with NULL, extraneous values will be ignored.

create-2-d-float-array ((*nof-rows* INTEGER) (*nof-columns* INTEGER) &rest (*values* FLOAT)) : 2-D-FLOAT-ARRAY [Function]

Create a two-dimensional array with *nof-rows* rows and *nof-columns* columns, and initialize it in row-major-order from *values*. Missing values will be padded with NULL, extraneous values will be ignored.

create-float-vector (&rest (*values* FLOAT)) : FLOAT-VECTOR [Function]

Return a vector containing *values*, in order.

create-marker-storage ((*supportRecall?* BOOLEAN)) : MARKER-TABLE [Function]

Return a new marker storage object, used to remember with objects have been `marked`. If *supportRecall?* is set, then the iterator `recall-marked-objects` can be invoked on the new marker storage object.

create-vector (&rest (*values* OBJECT)) : VECTOR [Function]

Return a vector containing *values*, in order.

current-inference-level () : NORMAL-INFERENCE-LEVEL [Command]

Return the current inference level that is active in the current query, the current module, or, otherwise, globally.

default-truth-value? ((*self* TRUTH-VALUE)) : BOOLEAN [Function]

Return TRUE if *self* is a default truth value.

define-arithmetic-operation-on-wrappers ((*name* SYMBOL) (*operation-name* SYMBOL)) : OBJECT [Macro]

Defines *name* as an arithmetic comparision operation using the test `test-name`. It will take two wrapped number parameters and return a wrapped number. The code will use the appropriate test for the specific subtype of wrapped number actually passed in, and return the appropriate subtype of wrapped number based on the normal arithmetic contagion rules.

For example, if both input parameters are wrapped integers then the output will be a wrapped integer. If the inputs are a wrapped integer and a wrapped float then the output will be a wrapped float, etc.

define-arithmetic-test-on-wrappers ((*name* SYMBOL) (*test-name* SYMBOL)) : OBJECT [Macro]

Defines *name* as an arithmetic comparision operation using the test *test-name*. It will take two wrapped number parameters and return a `boolean`. The code will use the appropriate test for the specific subtype of wrapped number actually passed in.

define-computed-constraint ((*name* SYMBOL) (*var-list* CONS) (*constraint-test* CONS) &body (*position-computations* CONS)) *:* OBJECT [Macro]

Defines *name* to be a constraint computation which uses *constraint-test* to determine if a fully bound set of variables satisfies the constraint. The forms in *position-computations* are used to compute the value for each of the positions. All such computations must set the variable `value` to be the result computed for the missing position. Setting `value` to `null` for any such computation means that that particular argument cannot be computed from the others. The input variables in *var-list* will be bound to the N arguments to the constraint. The generated function will return a Stella Object and take as inputs the values of the N arguments to the constraint. A value of `null` means that the value is not available. If all arguments are not `null`, then the return value will be a Stella wrapped boolean indicating whether the constraint is satisified or not. If more than one input value is `null`, then this constraint code will not be called.

deobjectify-tree ((*self* OBJECT)) *:* OBJECT [Function]

Return a copy of *self* where all logic objects are replaced by their `generated` parse-tree version. This is useful to convert the result of a retrieval query into a regular parse tree.

describe-object ((*self* NAMED-DESCRIPTION) (*stream* OUTPUT-STREAM) (*mode* KEYWORD)) *:* [Method]

Prints a description of *self* to stream *stream*. *mode* can be :terse, :verbose, or :source. Used by `describe`.

description-name ((*self* NAMED-DESCRIPTION)) *:* SYMBOL [Method]

Return the name of the description *self*.

description-name ((*self* DESCRIPTION)) *:* SYMBOL [Method]

Return the name of the description *self*, if it has one.

destroy-instance ((*self* OBJECT)) *:* [Function]

Destroy all propositions that reference *self*, and mark it as `deleted?`, thereby making it invisible within class extensions.

destroy-object ((*self* OBJECT)) *:* [Function]

Destroy *self* which can be a term or a proposition. Destroy all propositions that reference *self* and mark it as `deleted?` (thereby making it invisible within class extensions).

destroy-proposition ((*proposition* PROPOSITION)) *:* PROPOSITION [Function]

Retract and destroy the proposition *proposition*. Recursively destroy all propositions that reference *proposition*. Also, destroy all satellite propositions of *proposition*.

destroy-term ((*self* LOGIC-OBJECT)) *:* [Function]

Destroy all propositions that reference *self*, and mark it as `deleted?`, thereby making it invisible within class extensions. Unlink descriptions from native relations.

direct-superrelations ((*self* RELATION)) *:* (ITERATOR OF (LIKE SELF)) [Function]

Return direct super classes/slots of *self*.

disabled-powerloom-feature? ((*feature* KEYWORD)) *:* BOOLEAN [Function]
Return true if the STELLA *feature* is currently disabled.

disjoin-truth-values ((*tv1* TRUTH-VALUE) (*tv2* TRUTH-VALUE)) *:* TRUTH-VALUE [Function]
Return the logical disjunction of truth values *tv1* and *tv2*.

disjoint-terms? ((*d1* DESCRIPTION) (*d2* DESCRIPTION)) *:* BOOLEAN [Function]
Return TRUE if *d1* and *d2* belong to disjoint partitions.

do-clear-instances ((*module* MODULE)) *:* [Function]
Function version of `clear-instances` that evaluates its argument.

do-save-module ((*module* MODULE) (*store* OBJECT)) *:* [Function]
Save *module* to the persistent store *store* which can either be an output stream or a persistent OBJECT-STORE.

empty? ((*self* QUERY-SOLUTION-TABLE)) *:* BOOLEAN [Method]
Return TRUE if *self* has zero entries.

empty? ((*self* FLOAT-VECTOR)) *:* BOOLEAN [Method]
Return TRUE if *self* has length 0.

enabled-powerloom-feature? ((*feature* KEYWORD)) *:* BOOLEAN [Function]
Return true if the STELLA *feature* is currently enabled.

estimated-length ((*self* PAGING-INDEX)) *:* INTEGER [Method]
Return the estimated length of the sequences in *self*, which could be too large if some of the members have been deleted.

explain-why ((*label* STRING) (*style* KEYWORD) (*maxdepth* INTEGER) (*stream* OUTPUT-STREAM)) *:* [Function]
Programmer's interface to WHY function.

explain-whynot ((*label* STRING) (*style* KEYWORD) (*maxdepth* INTEGER) (*summary?* BOOLEAN) (*stream* OUTPUT-STREAM)) *:* [Function]
Programmer's interface to the WHYNOT function.

false-truth-value? ((*self* TRUTH-VALUE)) *:* BOOLEAN [Function]
Return TRUE if *self* represents some form of falsehood.

fetch-instance ((*store* OBJECT-STORE) (*name* OBJECT)) *:* OBJECT [Method]
Fetch the instance identified by *name* (a string or symbol) from *store* and return it as an appropriate logic object. This needs to be appropriately specialized on actual OBJECT-STORE implementations.

fetch-relation ((*store* OBJECT-STORE) (*name* OBJECT)) *:* NAMED-DESCRIPTION [Method]
Fetch the relation identified by *name* (a string or symbol) from *store* and return it as a named description. This needs to be appropriately specialized on actual OBJECT-STORE implementations.

fill-array ((*self* 2-D-ARRAY) &rest (*values* OBJECT)) *:* [Method]

Fill the two-dimensional array *self* in row-major-order from *values*. Missing values will retain their old values, extraneous values will be ignored.

fill-array ((*self* 2-D-FLOAT-ARRAY) &rest (*values* FLOAT)) *:* [Method]

Fill the two-dimensional array *self* in row-major-order from *values*. Missing values will retain their old values, extraneous values will be ignored.

finalize-objects () *:* [Function]

Finalize all currently unfinalized objects. The user-level entry point for this is `(process-definitions)`.

find-direct-supers-and-subs ((*self* DESCRIPTION) (*onlysupers?* BOOLEAN)) *:* (CONS OF DESCRIPTION) (CONS OF DESCRIPTION) (CONS OF DESCRIPTION) [Function]

Classify *self* and return three values, its direct supers, direct subs, and a list of equivalent descriptions. Setting `supersOnly?` may speed up the computation (perhaps by a lot). If `description` is nameless and has no dependent propositions, then it is automatically removed from the hierarchy after classification.

find-direct-supers-of-instance ((*self* OBJECT)) *:* (CONS OF LOGIC-OBJECT) [Function]

Classify *self* and return a list of most specific named descriptions among all descriptions that it satisfies.

find-instance ((*instanceRef* OBJECT)) *:* OBJECT [Command]

Return the nearest instance with name *instanceRef* visible from the current module. *instanceRef* can be a string, symbol, or surrogate. If *instanceRef* is a surrogate, the search originates in the module the surrogate was interned in.

find-rule ((*ruleName* NAME)) *:* PROPOSITION [Command]

Search for a rule named *ruleName*. Like `get-rule`, but `find-rule` implicity quotes its input argument.

function? ((*relationRef* OBJECT)) *:* BOOLEAN [Function]

Return TRUE if *relationRef* references a function.

generate-expression ((*self* LOGIC-OBJECT) (*canonicalizevariablenames?* BOOLEAN)) *:* OBJECT [Function]

Return an s-expression representing the source expression for *self*.

generate-specialized-term ((*self* LOGIC-THING)) *:* OBJECT [Method]

Method to generate a specialized term for *self*. This is designed to allow for extension of the term generation code to cover other types of objects for the logic. This particular method will signal an error unless there is a surrogate-value-inverse link set.

get-class ((*instanceRef* OBJECT)) *:* LOGIC-OBJECT [Function]

Return the nearest class with name *instanceRef* visible from the current module. *instanceRef* can be a string, symbol, or surrogate. If *instanceRef* is a surrogate, the search originates in the module the surrogate was interned in.

get-instance ((*instanceRef* OBJECT)) *:* OBJECT [Function]
Return the nearest instance with name *instanceRef* visible from the current module. *instanceRef* can be a string, symbol, or surrogate. If *instanceRef* is a surrogate, the search originates in the module the surrogate was interned in.

get-module ((*moduleRef* OBJECT)) *:* MODULE [Function]
Return a module named *moduleRef*.

get-relation ((*instanceRef* OBJECT)) *:* LOGIC-OBJECT [Function]
Return the nearest relation with name *instanceRef* visible from the current module. *instanceRef* can be a string, symbol, or surrogate. If *instanceRef* is a surrogate, the search originates in the module the surrogate was interned in.

get-self-or-prototype ((*instanceRef* OBJECT)) *:* LOGIC-OBJECT [Function]
Used to convert a computation to reference so-called `template` slots rather than `own` slots: If *instanceRef* denotes a class, return a prototype of that class. Otherwise, return *instanceRef*.

get-slot-maximum-cardinality ((*self* LOGIC-OBJECT) (*relation* SURROGATE)) *:* INTEGER [Function]
Return a maximum value for the number of fillers of relation *relation* (a surrogate) applied to the instance *self* (an object).

get-slot-minimum-cardinality ((*self* LOGIC-OBJECT) (*relation* SURROGATE)) *:* INTEGER [Function]
Return a minimum value for the number of fillers of relation *relation* (a surrogate) applied to the instance *self* (an object).

get-slot-value ((*self* LOGIC-OBJECT) (*relation* SURROGATE)) *:* OBJECT [Function]
Return a single value for the slot *relation* (a surrogate) applied to *self* (an object).

get-slot-value-type ((*self* LOGIC-OBJECT) (*relation* SURROGATE)) *:* NAMED-DESCRIPTION [Function]
Return a most specific type for fillers of the slot *relation* (a surrogate) applied to *self*. If there is more than one, pick one.

get-why-justification ((*label* STRING)) *:* JUSTIFICATION [Function]
Returns the current WHY justification. May also throw one of the following subtypes of EXPLAIN-EXCEPTION: EXPLAIN-NO-QUERY-EXCEPTION EXPLAIN-NO-SOLUTION-EXCEPTION EXPLAIN-NO-MORE-SOLUTIONS-EXCEPTION EXPLAIN-NOT-ENABLED-EXCEPTION EXPLAIN-NO-SUCH-LABEL-EXCEPTION EXPLAIN-QUERY-TRUE-EXCEPTION

get-whynot-justifications ((*query* QUERY-ITERATOR) (*label* STRING) (*mapping* EXPLANATION-MAPPING)) *:* (LIST OF JUSTIFICATION) [Function]
Programmer's interface to WHYNOT function. Derive justifications why *query* failed, or, if *label* was supplied as non-NULL, lookup its justification relative to *mapping* and return the result.

in-dialect ((*dialect* NAME)) *:* KEYWORD [Command]

Change the current logic dialect to *dialect*. Currently supported dialects are `KIF`, `STELLA`, and `PREFIX-STELLA`. The STELLA dialects are not recommended for the construction of knowledge bases, they are mainly used internally by PowerLoom.

insert-at ((*self* QUERY-SOLUTION-TABLE) (*key* (LIKE (ANY-KEY SELF))) (*value* (LIKE (ANY-VALUE SELF)))) *:* [Method]

Insert *value* identified by *key* into *self*. If a solution with that key already exists, destructively modify it with the slot values of *value*. This is necessary to preserve the order of solutions in *self*.

invert-truth-value ((*self* TRUTH-VALUE)) *:* TRUTH-VALUE [Function]

Return the logical negation of *self*.

length ((*self* QUERY-SOLUTION-TABLE)) *:* INTEGER [Method]

Return the number of entries in *self*.

list-features () *:* LIST [Command]

Return a list containing two lists, a list of currently enabled PowerLoom features, and a list of all available PowerLoom features.

list-unclassified-instances ((*module* NAME) (*local?* BOOLEAN)) *:* (CONS OF LOGIC-OBJECT) [Command]

Collect all instances in *module* (or in any module if *module* is NULL) that were not (or will not be) classified due to their lack of non-inferable/primitive type assertions.

list-unclassified-relations ((*module* NAME) (*local?* BOOLEAN)) *:* (CONS OF NAMED-DESCRIPTION) [Command]

Collect all named description in *module* (or in any module if *module* is NULL) that were not (or will not be) classified due to their lack of non-inferable/primitive ancestor relations.

list-undefined-relations ((*module* NAME) (*local?* BOOLEAN)) *:* (CONS OF NAMED-DESCRIPTION) [Command]

Return a list of as yet undefined concepts and relations in *module*. These relations were defined by the system, since they were referenced but have not yet been defined by the user. If *module* is NULL look in the current module. If *local?* only look in *module* but not in any modules it inherits.

listify ((*self* QUERY-ITERATOR)) *:* LIST [Method]

Just like `QUERY-ITERATOR.consify` but return a LIST instead.

load-stream ((*stream* INPUT-STREAM)) *:* [Function]

Read logic commands from *stream* and evaluate them.

logic-class? ((*self* CLASS)) *:* BOOLEAN [Function]

Return TRUE if the class *self* or one of its supers supports indices that record extensions referenced by the logic system. Also return true for literal classes.

logic-form-less? ((*o1* OBJECT) (*o2* OBJECT)) *:* BOOLEAN [Function]

A sorting predicate for objects *o1* and *o2* that can appear in logical forms. Performs a combined numeric and lexocographic sort that accounts for lists, collections and propositions. Numbers precede all other values, `null` follows all other values.

logic-module? ((*self* MODULE)) *:* BOOLEAN [Function]

Return TRUE if *self* is a logic module, implying that relations defined within it define a knowledge base. A module is a logic module iff it inherits the module `PL-KERNEL`.

lookup ((*self* QUERY-SOLUTION-TABLE) (*key* (LIKE (ANY-KEY SELF)))) *:* (LIKE (ANY-VALUE SELF)) [Method]

Lookup the solution identified by *key* in *self* and return its value, or NULL if no such solution exists.

natural-deduction-mode? () *:* BOOLEAN [Function]

True if normalization is governed by natural deduction semantics.

non-empty? ((*self* QUERY-SOLUTION-TABLE)) *:* BOOLEAN [Method]

Return TRUE if *self* has at least 1 entry.

non-empty? ((*self* FLOAT-VECTOR)) *:* BOOLEAN [Method]

Return TRUE if *self* has length `>` 0.

nth ((*self* QUERY-SOLUTION-TABLE) (*position* INTEGER)) *:* (LIKE (ANY-VALUE SELF)) [Method]

Return the nth solution in *self*, or NULL if it is empty.

object-name ((*self* OBJECT)) *:* SYMBOL [Function]

Return the name symbol for the logic object *self*.

object-name-string ((*self* OBJECT)) *:* STRING [Function]

Return the name string for the logic object *self*.

object-surrogate ((*self* OBJECT)) *:* SURROGATE [Function]

Return the surrogate naming the logic object *self*.

object-surrogate-setter ((*self* OBJECT) (*name* SURROGATE)) *:* SURROGATE [Function]

Return the name of the logic object *self* to *name*.

pop ((*self* QUERY-SOLUTION-TABLE)) *:* (LIKE (ANY-VALUE SELF)) [Method]

Remove and return the first solution of *self* or NULL if the table is empty.

powerloom () *:* [Function]

Run the PowerLoom listener. Read logic commands from the standard input, evaluate them, and print their results. Exit if the user entered `bye`, `exit`, `halt`, `quit`, or `stop`.

pretty-print-logical-form ((*form* OBJECT) (*stream* OUTPUT-STREAM)) *:* [Function]

Pretty-print the logical form *form* to *stream* according to the current setting of `*logic-dialect*`.

print-array ((*self* 2-D-ARRAY) (*stream* NATIVE-OUTPUT-STREAM)) *:* [Method]
Print the array *self* to *stream*.

print-array ((*self* 2-D-FLOAT-ARRAY) (*stream* NATIVE-OUTPUT-STREAM)) *:* [Method]
Print the array *self* to *stream*.

print-extension-sizes ((*module* MODULE) (*sizeCutoff* INTEGER)) *:* [Function]
Print the extension sizes of concepts visible in *module*. If *module* is NULL the current module is used. Do not report extensions with size less than *sizeCutoff* (default is 10).

print-goal-stack ((*frame* CONTROL-FRAME) (*verbose?* BOOLEAN)) *:* [Function]
Print stack of goals. Assumes that query has been interrupted with a full stack of control frames.

print-logical-form ((*form* OBJECT) (*stream* OUTPUT-STREAM)) *:* [Function]
Print the logical form *form* to *stream* according to the current setting of `*logic-dialect*`. Pretty-printing is controlled by the current setting of `*prettyPrintLogicalForms?*`.

print-logical-form-in-dialect ((*self* OBJECT) (*dialect* KEYWORD) (*stream* OUTPUT-STREAM)) *:* [Function]
Produce a stringified version of a logical representation of *self* and write it to the stream *stream*. Use the dialect *dialect*, or use the current dialect if *dialect* is NULL.

print-unformatted-logical-form ((*form* OBJECT) (*stream* OUTPUT-STREAM)) *:* [Function]
Print the logical form *form* to *stream* according to the current setting of `*logic-dialect*`. Pretty-printing is explicitly forced to be turned off.

print-whynot-justification ((*justification* JUSTIFICATION) (*stream* OUTPUT-STREAM) (*maxDepth* INTEGER) (*style* KEYWORD) (*summary?* BOOLEAN)) *:* [Function]
Print a WHYNOT *justification* to *stream* according to *maxDepth* and *style*. Print a summary only if *summary?* is TRUE.

random-float ((*n* FLOAT)) *:* FLOAT [Function]
Generate a random integer in the interval [0..n-1]. *n* must be <= 2^15.

recall-marked-objects ((*self* MARKER-TABLE)) *:* LIST-ITERATOR [Method]
Return an iterator that generates all marked objects recorded in *self*.

record-justifications? () *:* BOOLEAN [Function]
Return TRUE if every query records justifications to enable the explanation of concluded results.

register-logic-dialect-print-function ((*dialect* KEYWORD) (*fn* FUNCTION-CODE-WRAPPER)) *:* [Function]
Register *fn* as a logic-object print function for *dialect*. Each function should have the signature `((self OBJECT) (stream OUTPUT-STREAM))`. Any return values will be ignored.

relation-name ((*self* NAMED-DESCRIPTION)) *:* STRING [Function]

Given a relation object, return it's name.

relation? ((*objectRef* OBJECT)) *:* BOOLEAN [Function]

Return TRUE if *objectRef* denotes a relation or a class.

remove-at ((*self* QUERY-SOLUTION-TABLE) (*key* (LIKE (ANY-KEY SELF)))) *:* [Method]

Remove the solution identified by *key* from *self*. To preserve the solution ordering chain, the solution is marked as deleted and will be completely removed upon the next iteration through *self*.

remove-deleted-members ((*self* PAGING-INDEX)) *:* (LIKE SELF) [Method]

Destructively remove all deleted members of *self*.

reset-query-caches () *:* [Function]

Zero out all caches managed by the query optimizer, so that it will reoptimize subgoal queries upon next invocation.

retract-facts-of-instance ((*self* LOGIC-OBJECT)) *:* [Function]

Retract all definite (TRUE or FALSE) propositions attached to *self*.

retrieve-partial (&rest (*tree* PARSE-TREE)) *:* QUERY-ITERATOR [Command]

Partial-match version of `retrieve` (which see) that generates scored partial solutions based on the current partial match strategy. By supplying `BEST` instead of `ALL`, or by adding the option `:SORT-BY :SCORE`, the generated solutions will be sorted so that solutions with higher scores come first. Use the `:MATCH-MODE` option to override the global default setting established by `set-partial-match-mode`, e.g., use `:MATCH-MODE :NN` to use the neural net partial match mode. The `:MINIMUM-SCORE` option can be used to only retrieve solutions that have at least the specified minimum match score. By default, `retrieve-partial` does not maximize the match scores of its returned bindings. To only get maximal scores use `:MAXIMIZE-SCORE? TRUE` (this is not yet implemented - you can use `ask-partial` to maximize scores for individual solutions by hand).

run-forward-rules ((*moduleRef* NAME) &rest (*force* KEYWORD)) *:* [Command]

Run forward inference rules in module *moduleRef*. If *moduleRef* is NULL, the current module will be used. If forward inferencing is already up-to-date in the designated module, no additional inferencing will occur, unless the optional keyword `:force` is included, in which case all forward rules are run or rerun.

Calling `run-forward-rules` temporarily puts the module into a mode where future assertional (monotonic) updates will trigger additional forward inference. Once a non-monotonic update is performed, i.e., a retraction or clipping of relation value, all cached forward inferences will be discarded and forward inferencing will be disabled until this function is called again.

run-powerloom-tests () *:* [Command]

Run the PowerLoom test suite. Currently this simply runs all demos and echos commands and their results to standard output. The output can then be diffed with previously validated runs to find deviations.

satisfies? ((*instanceOrTuple* OBJECT) (*relationRef* OBJECT)) : TRUTH-VALUE [Function]

Try to prove whether *instanceOrTuple* satisfies the definition of the relation *relationRef* and return the result truth value of the query. *instanceOrTuple* can be a single object, the name or surrogate of an object, or a collection (a list or vector) of objects. *relationRef* can be a relation, description, surrogate or relation name.

save-all-neural-networks ((*file* STRING)) : [Command]

Save all neural networks to *file* (if *file* is non-NULL). If networks are saved periodically (see `set-save-network-cycle`) this file name will be used to perform periodic saves.

set-error-print-cycle ((*i* INTEGER)) : [Command]

Set number of cycles between which error rates are saved to the file established by the last call to `save-all-neural-networks` appended with extension `.err`. A number <= 0 (or NULL) turns off periodic saving.

set-inference-level ((*level* NAME) (*module* NAME)) : KEYWORD [Command]

Set the inference level of *module* to the level specified by `levelKeyword`. If *module* is NULL, set the level globally.

set-marker ((*self* MARKER-TABLE) (*object* OBJECT)) : [Method]

Record membership of *object* in the marker storage object *self*.

set-num-neighbors ((*d* INTEGER)) : [Command]

Sets the number of nearest neighbors to predict from.

set-num-training-per-case ((*d* INTEGER)) : [Command]

Sets the number of training examples for each case in the training set.

set-powerloom-feature ((*feature* KEYWORD)) : [Function]

Enable the PowerLoom environment feature *feature*.

set-save-network-cycle ((*i* INTEGER)) : [Command]

Set number of cycles between which networks are saved to the file established by the last call to `save-all-neural-networks`. A number <= 0 or a NULL number turns off periodic saving.

sort ((*self* QUERY-SOLUTION-TABLE) (*predicate* FUNCTION-CODE)) : (LIKE SELF) [Method]

Perform a stable, destructive sort of *self* according to *predicate*, and return the result. If *predicate* has a `<` semantics, the result will be in ascending order.

specializes? ((*subObject* OBJECT) (*superObject* OBJECT)) : TRUTH-VALUE [Function]

Try to prove if the description associated with *subObject* specializes the description for *superObject* and return the result truth value of the query.

strengthen-truth-value ((*tv1* TRUTH-VALUE) (*tv2* TRUTH-VALUE)) : TRUTH-VALUE [Function]

If *tv2* has greater strength than *tv1*, adapt the strength of *tv1* (not its value!) and return the result. Otherwise, return *tv1* unmodified.

strict-truth-value? ((*self* TRUTH-VALUE)) : BOOLEAN [Function]
Return TRUE if *self* is a strict truth value.

termify ((*self* OBJECT)) : OBJECT [Function]
Convert *self* into an equivalent PowerLoom object that can be passed as an argument wherever an instance is expected.

test-closed-slot? ((*relation* SURROGATE)) : BOOLEAN [Function]
Return TRUE if *relation* (a surrogate) is asserted to be closed or if the current module closes all relations.

test-function-slot? ((*relation* SURROGATE)) : BOOLEAN [Function]
Return TRUE if *relation* (a surrogate) is a function.

test-marker? ((*self* MARKER-TABLE) (*object* OBJECT)) : BOOLEAN [Method]
Return TRUE if *object* is stored (marked) in *self*.

test-relation-on-arguments? ((*relation* SURROGATE) (*arguments* CONS)) : BOOLEAN [Function]
Return TRUE if *relation* (a surrogate) is TRUE when applied to *arguments*.

test-slot-value? ((*self* LOGIC-OBJECT) (*relation* SURROGATE) (*filler* OBJECT)) : BOOLEAN [Function]
Return TRUE if the proposition `(<relation> <self> <filler>)` is true.

test-special-marker-table? ((*self* OBJECT)) : BOOLEAN [Function]
Return TRUE if the object *self* is stored (marked) in the table pointed at by the special variable *specialMarkerTable*. Designed for use by `remove-if`.

test-subrelation? ((*subrelation* SURROGATE) (*superrelation* SURROGATE)) : BOOLEAN [Function]
Return TRUE if *subrelation* specializes *superrelation*.

test-type-on-instance? ((*self* OBJECT) (*type* SURROGATE)) : BOOLEAN [Function]
Return TRUE if *self* satisfies *type*.

true-truth-value? ((*self* TRUTH-VALUE)) : BOOLEAN [Function]
Return TRUE if *self* represents some form of truth.

unassert ((*proposition* PARSE-TREE)) : [Command]
Retract the truth or falsity of *proposition*. This is a more general version of `retract` that also handles falsity. For example, if we assert the proposition "(not (sad Fred))", and then execute the statement "(unassert (sad Fred))", the truth value of the proposition "(sad Fred)" will be set to UNKNOWN. If we had called `retract` in place of `unassert`, the proposition "(sad Fred)" would remain set to FALSE. Note that for this unassertion to succeed, the logic constant `Fred` and the relation `sad` must already be defined.

unassert-fact ((*self* PROPOSITION)) : [Function]
Retract the truth or falsity of the proposition *self*

unknown-truth-value? ((*self* TRUTH-VALUE)) *:* BOOLEAN [Function]
Return TRUE if *self* represents UNKNOWN.

unset-powerloom-feature ((*feature* KEYWORD)) *:* [Function]
Disable the PowerLoom environment feature *feature*.

upclassify-all-descriptions () *:* [Function]
Classify all named descriptions.

upclassify-all-instances () *:* [Function]
Classify all named instances.

upclassify-instances ((*module* MODULE) (*local?* BOOLEAN)) *:* [Function]
Classify instances local to *module* and inherited by *module*. If *local?*, don't classify inherited descriptions. If *module* is NULL, classify descriptions in all modules.

upclassify-named-descriptions ((*module* MODULE) (*local?* BOOLEAN)) *:* [Function]
Classify named descriptions local to *module* and inherited by *module*. If *local?*, don't classify inherited descriptions. If *module* is NULL, classify descriptions in all modules.

weaken-truth-value ((*tv1* TRUTH-VALUE) (*tv2* TRUTH-VALUE)) *:* TRUTH-VALUE [Function]
If *tv2* has lesser strength than *tv1*, adapt the strength of *tv1* (not its value!) and return the result. Otherwise, return *tv1* unmodified.

with-logic-environment ((*moduleForm* OBJECT) (*environment* OBJECT) &body (*body* CONS)) *:* OBJECT [Macro]
Execute *body* within the module resulting from *moduleForm*. `*module*` is an acceptable *moduleForm*. It will locally rebind `*module*` and `*context*` and shield the outer bindings from changes.

within-classification-session ((*descriptionorinstance* KEYWORD) &body (*body* CONS)) *:* OBJECT [Macro]
Used during classification. Execute *body* within the indicated classification session and inference world.

within-meta-cache (&body (*body* CONS)) *:* OBJECT [Macro]
Execute *body* within the meta cache of the current module. Set appropriate special variables.

is-true-proposition1 ((*relation-and-arguments* OBJECT) (*module* MODULE) (*environment* OBJECT)) *:* BOOLEAN [Function]
Return TRUE if a proposition (`relation args`) has been asserted (or inferred by forward chaining).

main () *:* [Function]
Main PowerLoom entry point for your code in C++ and Java.

initialize-kernel-kb () *:* [Command]
Bootstrap the PowerLoom built-in kernel KB.

11 Glossary

This glossary contains brief definitions for terms used in the PowerLoom User's Manual and/or used by the knowledge representation community. It is impractical to give a logically precise definition for many of these terms, because their interpretation varies quite a bit. In this case, the glossary attempts to indicate a range of interpretations consistent with their use in PowerLoom.

Assertion: An assertion states that a particular proposition is `True` or `False`.

Backward and Forward Inference: ???

BACKWARD RULE: **???**

Binary Relation: A relation having two arguments (arity equals two), often as a mapping from one concept domain to another. This is by far the most common form of relation.

Classifier: A classifier is a type of an inference engine that implements efficient strategies for computing subsumption relations between pairs of concepts, or for computing instance-of relations between a concept an a set of instances. PowerLoom implements a classifier that can be explicitly invoked by an application program.

Clipping: If a function or single-valued binary relation maps an instance to two or more other instances, a logical contradiction (a clash) exists. If clipping is enabled, PowerLoom will automatically retract all assertions but the last that lead to a clash. Clipping can be toggled on or off; it is enabled by default.

Closed-World Semantics: Under closed-world semantics it is assumed that "if proposition `P` cannot be proved `True`, then assume that `P` is `False`." PowerLoom gives programmers the option to explicitly declare that concept or a relation operates under the assumption of closed-world semantics (See also Open-World Semantics).

Concept: A concept defines a category or class of individuals. PowerLoom categorizes a concept as a special kind of relation. The distinction between a concept and a unary relation is subtle (some logicians do not believe that there is any distinction[1]). In linguistics, the distinction is that between a noun and an adjective. In logic, the test we favor is whether or not the relation has a domain — a unary relation has a domain, while a concept does not. For example, the relation 'married' has domain 'person', while the concept 'married-person' does not have a domain (or is its own domain).

Constraint: "Constraint" at its most general is a synonym for "rule". Often a constraint is conceptualized as a rule that restricts the types of the arguments that can appear within a tuple.

Context: ???

Default Rule: A default rule expresses an conditional implication that applies only when its consequent is consistent with current state of the knowledge base. In other words, the rule applies only when it will not lead to a contradition.

Definition: A definition binds a name to a logical expression. PowerLoom syntax defines several operators with names of the form `defxxx` (e.g., `defconcept` and `defrule`) that declare definitions for various types of entities.

[1] but they are mistaken :).

Description: A "description" is an expression that defines a particular logical relation (e.g., the class of all three-legged black cats). In PowerLoom, the terms "concept" and "relation" generally refer to `named` relations, while a description may or may not have a name. The KIF operators `kappa` and `setofall` are used to define unnamed descriptions.

Description Logic: The term "description logic" refers to a logic that focuses on descriptions as its principal means for expressing logical expressions. A description logic system emphasises the use of classification and subsumption reasoning as its primary mode of inference. Loom and Classic were two early examples of knowledge representation systems that implement description logics.

Domain Model: A collection of definitions, rules, and facts that characterizes the possible states of some real or imagined world. The domain model specifies a terminology (of concepts and relations) that is useful for describing objects in that world. Often "domain model" refers to that portion of a world's representation that does not change over time.

Extension: Given a relation `R` with arity `N`, the extension of `R` is the set of ground propositions of the form `(R x1 ... xN)` whose truth value is true. If R is a concept, then its extension of often considered to be, not a set of unary tuples, but the set of argument fillers of those tuples, i.e., the set of instances that belong to the concept.

Fact: A fact is a proposition that has been asserted to be either `True` or `False`. The term "fact" usually refers to a "ground proposition", i.e., a proposition that can be represented as a predicate applied to a sequence of instances or literals.

Filler: The second argument to a binary tuple is often refered to as its "filler". When a multiple-valued binary relation maps an instance to a set of values, these values are also called "fillers".

Forward Rule: ???

Function: Formally, a function is a relation such that the value of the last (nth) argument of a relational tuple is a function of the values of the first n-1 arguments. This definition coincides with the notion of a "single-valued relation". PowerLoom (and KIF) support specialized syntax that allows functions that have been defined using the operator `deffunction` to appear in term expressions (e.g., `(= (f ?x) 42))`).

Instance: An instance denotes an entity within a domain model, a member of the concept *Thing*. Depending on ones interpretation, this could include almost everything. Often the term "instance" is used more narrowly, to exclude literals and other objects whose properties do not change over time. PowerLoom assumes that concepts and relations are instances.

KIF: Short for "Knowledge Interchange Format", KIF is a language that defines a Lisp-like syntax for the predicate calculus. There is an ANSII-standard that defines the KIF syntax and semantics. PowerLoom adopts KIF as its representation language, and adds a few extensions.

Knowledge Base: A knowledge base attempts to capture in abstract (machine interpretable) form a useful representation of a physical or virtual world. The entities in that world are modeled in the knowledge base by objects we call *terms*. Examples of terms are "Georgia" (denoting the U.S., state), "BenjaminFranklin" (denoting the historical person by that name), the number three, the string `"abc"`, and the concept "Person".

Literal: A logically static constant. Examples are numbers, strings, quantities, and truth values.

Module: ???

Open-World Semantics: PowerLoom assumes an open-world semantics, unless a user explicitly specifies that it use closed-world semantics. Under this assumption, if PowerLoom cannot prove or disprove a proposition, then it assigns that proposition the value Unknown (See also Closed-World Semantics).

Predicate:. The term *predicate* is a syntactic notion that refers to the zeroth arguments of a proposition. Predicates denote relations and properties, i.e., sets.

Proposition:. A logical sentence whose truth value can be evaluated with respect to some context. Each PowerLoom assertion assigns the value True or False to some proposition.

Primitive Relation: P is a primitive concept or relation if and only if a proof that (P x1 ... xn) is true exists only for the case that there exists an explicit assertion of a proposition (Q x1 ... xn) and either Q equals P or Q is a proper subrelation of P. In otherwords, the only rules that imply membership in P are those that relate P to one of its (proper) subconcepts or subrelations.

Query: A query probes the informational state of a knowledge base. An ask query test the truth of its propositional argument. A retrieve asks for sets of constants (bindings) that make its propositional argument true when the constants are substituted in place of its variables. The propositional argument to ask and retrieve arbitrary expression in the first-order predicate calculus. Because of constraints imposed either by resource limitations or inherent undecidability, PowerLoom cannot guarantee the completeness of its inferences.

Relation: ???

Retraction: A retraction changes the truth value of a proposition from either True or False to the value Unknown. Retraction is a procedural (non-declarative) operation.

Rule: A "rule" is any universally-quantified proposition, i.e., a proposition of the form (forall (?x1 ... ?xn) <logical sentence with free variables ?x1 ... ?xn). PowerLoom supports several different syntactic constructs for defining rules. (See also Forward Rule and Backward Rule).

Subsumption: A subsumption relation specifies the relative generality of two concepts. A concept A subsumes a concept B if the definitions of A and B logically imply that members of B must also be members of A.

Truth-Maintenance: ???

Type: Often used a synonym for the term *concept*. The phrase "a type of an instance" generally refers to (one of) the concepts that the instance belongs to. The phrase "nth domain type" refers to a concept that contains all instances of the nth column of a relation.

World: ???

12 PowerLoom Grammar

The syntax of PowerLoom is described below using a modified BNF notation adapted from the KIF specification.

12.1 Alphabet

We distinguish between terminals, which are part of the language, and nonterminals. All nonterminals are bracketed as follows `<nonterminal>`. Squared brackets means zero or one instances of the enclosed expression; `<nonterminal>*` means zero or more occurrences and `<nonterminal>+` means one or more occurrences of `<nonterminal>`. The notation `<nonterminal1> - <nonterminal2>` refers to all of the members of `<nonterminal1>` except for those in `<nonterminal2>`.

A word is a contiguous sequence of characters, which include all upper case letters, lower case letters, digits and alpha characters (ASCII character set from 93 to 128) excluding some special characters like white spaces, single and double quotes and brackets.

`<word> ::=` *a primitive syntactic object*

Special words are those who refer to a variable. All variables are preceded by a question mark.

`<indvar> ::=` *a word beginning with the character* `?`

A string `<string>` is a character sequence including words plus all special charcters (except double quotes) enclosed in double quotes. A double quote can be included in a string if it is preceeded by the escape character '\'.

12.2 Grammar

Legal expressions in PowerLoom are forms, which are either a statement or a definition, described in more detail below.

```
<form> ::= <statement> | <definition>
```

12.2.1 Constants and Typed Variables

The language consists of several groups of operators, defined as follows:

```
<termop> ::= listof | setof | the | setofall | kappa

<sentop> ::= = | /= | not | and | or | forall | exists
            | <= | => | <=> |<<= | =>> | <~ | ~> | <<~ | ~>>

<defop> ::= defconcept | deffunction | defrelation | defrule |
            :documentation | :-> |
            :<= | :=> | :<<= | :=>> |
            :<=> | :<=>> :<<=> | :<<=>> | := |
            :axioms

<operator> ::= <termop> | <sentop> | <defop>
```

All other words are constants (words which are not operators or variables):

```
<constant> ::= <word> - <indvar> - <operator>
```

Semantically, there are different categories of constants — *Concept* constants `<conceptconst>`, *Function* constants `<funconst>`, *Relation* constants `<relconst>`, *Rule* constants `<ruleconst>` and *Logical* constants `<logconst>`. The differences between these categories are entirely semantic. However, some operators will only accept specific constants.

In contrast to the specification of KIF3.0, PowerLoom supports a typed syntax. Therefore, variables in quantified terms and sentences can appear either typed or untyped, as follows:

```
<vardecl> ::= (<indvar> <constant>) | <indvar>
```

12.2.2 Terms

Terms are used to denote objects in the world being described:

```
<term> ::= <indvar> | <constant> | <funterm> | <listterm> | <setterm> |
           <quanterm>
```

```
<listterm> ::= (listof <term>*)
```

```
<setterm> ::= (setof <term>*)
```

```
<funterm> ::= (<funconst> <term>+)
```

Note: Zero arguments are allowed for `<funterm>` in KIF3.0: `<term>*`

```
<quanterm> ::= (the <vardecl> <sentence>) |
               (setofall <vardecl> <sentence>) |
               (kappa {<vardecl> | (<vardecl>+)} <sentence>) |
               (lambda {<vardecl> | (<vardecl>+)} <term>)
```

Note: KIF3.0 allows `<term>` instead of `<vardecl>` for `setofall`. No `<quanterm>` as well as no `<setterm>` in core of KIF as a result of descision 95-3 (March 1995).

12.2.3 Sentences

Sentences are used to express propositions about the world:

```
<sentence> ::= <constant> | <equation> | <inequality> |
               <relsent> | <logsent> | <quantsent>
```

```
<equation> ::= (= <term> <term>)
```

```
<inequality> ::= (/= <term> <term>)
```

```
<relsent> ::= (<constant> <term>+)
```

Note: Zero arguments allowed in KIF3.0 for `<relsent>` (`<term>*`). `<funconst>` is currently not allowed in PowerLoom (use `(= <funterm> <term>)` instead).

```
<logsent> ::= (not <sentence>) |
              (and <sentence>*) |
              (or <sentence>*) |
              (=> <sentence>* <sentence>) | (=>> <sentence>* <sentence>) |
              (<= <sentence> <sentence>*) | (<<= <sentence> <sentence>*) |
```

```
                (~> <sentence>* <sentence>) | (~>> <sentence>* <sentence>) |
                (<~ <sentence> <sentence>*) | (<<~ <sentence> <sentence>*)
   <quantsent> ::= (forall {<vardecl> | (<vardecl>+)} <sentence>) |
                   (forall {<vardecl> | (<vardecl>+)} <sentence> <sentence>)
|
                   (exists {<vardecl> | (<vardecl>+)} <sentence>)
```

12.2.4 Definitions

PowerLoom supports two distinct categories of definitions — relation definitions (including concept and function definitions) and rule definitions. A relation definition introduces a new logical constant, and states some facts about that constant (e.g., who its parents are in a subsumption taxonomy). A rule definitions binds a new constant to a proposition (so that the constant *denotes* the proposition) and asserts the truth of that proposition. Usually, the proposition asserted by a `defrule` is an implication. The assertional truth of a proposition defined by a rule can be altered by asserting or retracting the constant that denotes the proposition.

```
<keyword-option> ::= <keyword> <word>
<definition> ::= <reldefinition> | <objdefinition> | <ruledefinition>

<reldefinition> ::=
 (defconcept <conceptconst> <vardecl>
    [:documentation <string>]
    [:<= <sentence>] | [:=> <sentence>] |
    [:<<= <sentence>] | [:=>> <sentence>] |
    [:<=> <sentence>] | [:<=>> <sentence>] | [:<<=> <sentence>] |
    [:<<=>> <sentence>] |
    [:axioms {<sentence> | (<sentence>+)}] |
    [<keyword-option>*])
 |
 (deffunction <funconst> (<vardecl>+)
    [:documentation <string>]
    [:-> <vardecl>]
    [:<= <sentence>] | [:=> <sentence>] |
    [:<<= <sentence>] | [:=>> <sentence>] |
    [:<=> <sentence>] | [:<=>> <sentence>] | [:<<=> <sentence>] |
    [:<<=>> <sentence>] |
    [:axioms {<sentence> | (<sentence>+)}]
    [<keyword-option>*])
 |
 (defrelation <relconst> (<vardecl>+)
    [:documentation <string>]
    [:<= <sentence>] | [:=> <sentence>] |
    [:<<= <sentence>] | [:=>> <sentence>] |
    [:<=> <sentence>] | [:<=>> <sentence>] | [:<<=> <sentence>] |
    [:<<=>> <sentence>] |
```

```
        [:axioms {<sentence> | (<sentence>+)}]
        [<keyword-option>*])

    <objdefinition> ::= (defobject <constant>
                            [:documentation <string>]
                            [<keyword-option>*])

    <ruledefinition> ::= (defrule <constant> <sentence>
                            [:documentation <string>]
                            [<keyword-option>*])
```

```
<ruledefinition> ::= (defrule <ruleconst> <sentence>)
```

Function Index

D

E

F

G

H

I

L

M

N

O

P

R

S

T

U

V

W

Variable Index

Concept Index

Table of Contents