RL-Glue Lisp Codec 1.0 Manual

Gabor Balazs
gabalz@rl-community.org

Contents

1 Introduction

RL-Glue codecs provide TCP/IP connectivity to the RL-Glue reinforcement learning software library. This codec makes possible to create agent, environment and experiment programs in the Common Lisp programming language.

For general information and motivation about the *RL-Glue library*[1], please refer to the documentation provided with that project.

This software is licensed under the *Apache 2.0 license*[2]. We are not lawyers, but our intention is that this codec should be used however it is useful. We would appreciate to hear what you are using it for, and to get credit if appropriate.

1.1 Software requirements

Required libraries for the codec.

split-sequence (http://www.cliki.net/SPLIT-SEQUENCE)

usocket (http://common-lisp.net/project/usocket/)

1.2 Supported Lisp implementations

SBCL (http://www.sbcl.org/)

CMUCL (http://www.cons.org/cmucl/)

Lispworks (http://www.lispworks.com/)

Allegro CL (http://www.franz.com/products/allegrocl/)

CLISP (http://www.gnu.org/software/clisp/)

Scieneer (http://www.scieneer.com/scl/)

CCL / OpenMCL (http://www.clozure.com/clozurecl.html)

1.3 Getting the codec

The codec can be downloaded either as a tarball or can be checked out of the subversion repository.

The tarball distributions can be found here:
http://code.google.com/p/rl-glue-ext/wiki/Lisp

The subversion trunk can be checked out this way:

```
$ svn co http://rl-glue-ext.googlecode.com/svn/trunk/projects/codecs/Lisp lisp-codec
```

[1] http://glue.rl-community.org/
[2] http://www.apache.org/licenses/LICENSE-2.0.html

1.4 Installation

The codec provides ASDF packages, so the *ASDF library*[3] has to be set up for your Lisp implementation. If you want to know more about it, take a look at our quick ASDF introduction.

The main ASDF packages are `rl-glue-codec` and `rl-glue-utils`. The former provides the connectivity with the RL-Glue component, the latter contains utilities which can be helpful during using the codec. The `rl-glue-examples` package contains the agent, environment and experiment examples described later in this document.

The codec is distributed in three different ways described in the following sections. Choose according to your needs and knowledge, but if you remain unsure at the end, we recommend the full distribution.

1.4.1 Full distribution

Recommended for users without ASDF experience.

This distribution contains a pre-configured ASDF environment with all the libraries required to use the codec. It also provides the manual and the API reference documentation. You don't need any ASDF installation and configuration knowledge to use this distribution.

The installation is very simple.

Choose the install destination.
```
$ cd /path/to
```

Unpack the files.
```
$ tar -zxf lisp-codec-full-<version>.tar.gz
```

Then you are able to use the codec.

Start your Lisp.
```
$ lisp
*
```

Initialize the ASDF sustem.
```
* (load #p"/path/to/lisp-codec-full/setup")
```

Now you are ready to load the rl-glue ASDF packages. The ASDF system will compile them automatically on the first time to reduce the loading time of the later occasions. Continue by reading the usage instructions.

[3] http://common-lisp.net/project/asdf/

1.4.2 ASDF distribution

Recommended for users with ASDF experience.

If you are an experienced ASDF user, you probably have an installed and configured ASDF system. Then this distribution is for you, it contains the `rl-glue-codec`, `rl-glue-utils` and `rl-glue-examples` ASDF packages which can be installed as any other ASDF package. It also provides the manual and the API reference documentation.

If you choose this distribution, you have to install the library dependencies on your own. Of course, you can use a sophisticated installer (like *ASDF-Install*[4] or *clbuild*[5]) which can handle this problem.

1.4.3 Developer distribution

Recommended for Lisp codec developers.

This distribution is a snapshot of the Lisp codec development directory. It contains all the source code, tests, documentation and development tools. If you choose this, you have to do the installation as in case of the ASDF distribution.

1.5 Uninstallation

Just delete what you don't need anymore.

If you used a sophisticated tool for the installation (like ASDF-Install or clbuild), you should use its uninstall method.

If you have used ASDF-Binary-Locations, don't forget to remove the `fasl` files.

1.6 Credits and acknowledgment

Gábor Balázs wrote the Lisp codec, that's me!

I would like thank the lispers on the `#lisp` *IRC channel*[6] on *irc.freenode.net* of their great help in many Lisp specific questions.

1.6.1 Contributing

If you would like to become a member of this project and contribute updates/changes to the code, please send a message to rl-glue@googlegroups.com.

[4] http://www.cliki.net/ASDF-Install
[5] http://common-lisp.net/project/clbuild/
[6] http://www.cliki.net/IRC

2 Using the codec

This section describes how the codec can be used to create agents, environments and experiments in Lisp, and how they can be glued into a running system.

2.1 Packages

The codec has an own (ASDF) package named `rl-glue-codec`.
```
* (asdf:oos 'asdf:load-op :rl-glue-codec)
```

You can use the proper package qualified symbol names, e.g.
```
* (defclass my-agent (rl-glue-codec:agent) ...)
* (defmethod rl-glue-codec:env-init ((env my-env)) ...)
```

Or you can import all the symbols into your package by the `:use` directive, and use the codec symbols without any package qualification (the further examples will assume this case).
```
* (defpackage :my-package (:use :rl-glue ...) ...)
```

There are a few utilities for the codec which can be useful. These can be accessed in the `rl-glue-utils` (ASDF) package. More information about them can be found in their section.

2.2 Types

There is an abstract data type, `rl-abstract-type`, which can contain integers, floating point numbers and a character string. Its slots can be accessed by the `int-array`, `float-array` and `char-string` functions. There are two macros for number array creation, `make-int-array` and `make-float-array`, which automatically set the type of the contained elements according to the codec requirements. The usage of them is strongly suggested.

observation	rl-abstract-type
action	rl-abstract-type
reward	double-float
terminal	boolean
task specification	string
state key	rl-abstract-type
random seed key	rl-abstract-type

2.3 Agents

On writing an agent first, you have to create an own agent class, e.g.
```
* (defclass my-agent (agent) ...)
```

Second, implement the following methods for it.
```
* (defmethod agent-init ((agent my-agent) task-spec) ...)
* (defmethod agent-start ((agent my-agent) first-observation) ...)
* (defmethod agent-step ((agent my-agent) reward observation) ...)
```

```
* (defmethod agent-end ((agent my-agent) reward) ...)
* (defmethod agent-cleanup ((agent my-agent)) ...)
* (defmethod agent-message ((agent my-agent) input-message) ...)
```

A detailed description of the methods can be obtained this way.

```
* (documentation #'<method-name> 'function)
```

When your agent is ready, you can run it.

```
* (run-agent (make-instance 'my-agent)
             :host "192.168.1.1"
             :port 4096
             :retry-timeout 10)
```

It will try to connect your agent to an RL-Glue component listening on `192.168.1.1` and port `4096`, waiting 10 seconds between the trials. Its detailed description can be checked this way.

```
* (documentation #'run-agent 'function)
```

It will prompt something like this, where each dot denotes a connection trial.

```
RL-Glue Lisp Agent Codec Version 1.0, Build 414
        Connecting to 192.168.1.1:4096 .... ok
```

2.4 Environments

On writing an environment, first you have to create an own environment class, e.g.

```
* (defclass my-env (environment) ...)
```

Second, implement the following methods for it.

```
* (defmethod env-init ((env my-env)) ...)
* (defmethod env-start ((env my-env)) ...)
* (defmethod env-step ((env my-env) action) ...)
* (defmethod env-cleanup ((env my-env)) ...)
* (defmethod env-message ((env my-env) input-message) ...)
```

A detailed description of the methods can be obtained this way.

```
* (documentation #'<method-name> 'function)
```

When your environment is ready, you can run it.

```
* (run-env (make-instance 'my-env)
           :host "192.168.1.1"
           :port 4096
           :retry-timeout 10)
```

It will try to connect your environment to an RL-Glue component listening on `192.168.1.1` and port `4096`, waiting 10 seconds between the trials. Its detailed description is here.

```
* (documentation #'run-env 'function)
```

It will prompt something like this, where each dot denotes a connection trial.

```
RL-Glue Lisp Environment Codec Version 1.0, Build 414
        Connecting to 192.168.1.1:4096 .... ok
```

2.5 Experiments

On writing an experiment, first you have to create an own experiment class, e.g.
`* (defclass my-exp (experiment) ...)`

Second, implement your experiment. For this the codec provides client functions which hides the necessary buffer handling and network operation. These are the following.

`rl-init`, `rl-start`, `rl-step`, `rl-cleanup`, `rl-close`, `rl-return`, `rl-num-steps`, `rl-num-episodes`, `rl-episode`, `rl-agent-message`, `rl-env-message`.

A detailed description of these functions can be obtained this way.
`* (documentation #'<function name> 'function)`

Do not forget to call the `rl-close` function at the end of your experiment, because it closes the network connection and so terminates the RL-Glue session.

2.6 Running

After you have an agent, an environment and an experiment, which could be written on any of the supported languages, you can connect them by RL-Glue.

First start the server.
`$ rl_glue`

You should see this kind of output on the server side.
```
RL-Glue Version 3.0, Build 909
RL-Glue is listening for connections on port=4096
        RL-Glue :: Agent connected.
        RL-Glue :: Environment connected.
        RL-Glue :: Experiment connected.
```

Then start the agent, the environment and the experiment.

The output of the Lisp agent.
```
RL-Glue Lisp Agent Codec Version 1.0, Build 414
        Connecting to 127.0.0.1:4096 . ok
```

The output of the Lisp environment.
```
RL-Glue Lisp Environment Codec Version 1.0, Build 414
        Connecting to 127.0.0.1:4096 . ok
```

The output of the Lisp experiment.
```
RL-Glue Lisp Experiment Codec Version 1.0, Build 414
        Connecting to 127.0.0.1:4096 . ok
```

2.7 Utilities

The utilities has an own (ASDF) package named `rl-glue-utils`.

```
* (asdf:oos 'asdf:load-op :rl-glue-utils)
```

2.7.1 Task specification parser

This is a parser for the *task specification language*[7]. Only the RL-Glue version 3.0 specification type is supported.

The parser can return a `task-spec` object from a specification string, e.g.

```
* (parse-task-spec
    "VERSION RL-Glue-3.0 PROBLEMTYPE episodic DISCOUNTFACTOR 1
     OBSERVATIONS INTS (3 0 1) ACTIONS DOUBLES (3.2 6.5) CHARCOUNT 50
     REWARDS (-1.0 1.0) EXTRA extra specification")
#<TASK-SPEC>
```

The `task-spec` object has the following slots.

`version`, `problem-type`, `discount-factor`,
`int-observations`, `float-observations`, `char-observations`,
`int-actions`, `float-actions`, `char-actions`,
`rewards`, `extra-spec`

Their names are appropriately show their functionalities according to the task specification documentation. The `int-` and `float-` observation and action slot values contain `int-range` and `float-range` objects appropriately. These have the `repeat-count`, `min-value` and `max-value` slots. For the latter two there are three special symbols which are `'-inf`, `'+inf` and `'unspec`. They are used to represent the NEGINF, POSINF and UNSPEC specification keywords.

The `task-spec` type supports the `to-string` generic function with which it can be converted to a string. There are also two other helper functions, namely `across-ranges` and `ranges-dimension`, which can be useful for range vector handling. Their documentation can be checked by the `documentation` function.

```
* (documentation 'rl-glue-utils:across-ranges 'function)
* (documentation 'rl-glue-utils:ranges-dimension 'function)
```

2.8 Examples

There is an (ASDF) package named `rl-glue-examples` located in the `example` directory. You have to load it before you run the examples.

```
* (pushnew #p"/path/to/lisp-codec/examples/" asdf:*central-registry*)
* (asdf:oos 'asdf:load-op :rl-glue-examples)
```

[7] http://glue.rl-community.org/Home/rl-glue/task-spec-language

2.8.1 The skeleton example

The `skeleton` example has a very simple agent, environment and experiment of which duty is only an introduction about the usage of the codec for the user. The environment is a "discrete line" with 21 states and the agent starts from the middle and can go left and right. It randomly selects the actions and rewarded differently on the ends of the line. The experiment tries to cover the presentation of the used features.

To run the example you have to do the following.
Running the RL-Glue core.
```
$ rl_glue
```
Starting the agent (in the first Lisp thread).
```
* (rl-glue-skeleton:start-agent)
```
Starting the environment (in the second Lisp thread).
```
* (rl-glue-skeleton:start-environment)
```
Starting the experiment (in the third Lisp thread).
```
* (rl-glue-skeleton:start-experiment)
```

2.8.2 The mines-sarsa example

The `mines-sarsa` example has an episodic environment. At the beginning of each episode the agents is put somewhere onto the mine field. Its goal is to find the exit point without stepping to a mine. The rewards are -1 for each intermediate step, -100 for stepping to a mine and $+10$ for reaching the exit. The example contains a simple sarsa agent with epsilon greedy action selection policy and a simple lookup table to store the action values. A few basic functionalities like turning learning or exploration on/off, saving the value function and setting the starting position are implemented by the message system of RL-Glue.

To run the example you have to do the following.
Running the RL-Glue core.
```
$ rl_glue
```
Starting the agent (in the first Lisp thread).
```
* (rl-glue-mines-sarsa:start-agent)
```
Starting the environment (in the second Lisp thread).
```
* (rl-glue-mines-sarsa:start-environment)
```
Starting the experiment (in the third Lisp thread).
```
* (rl-glue-mines-sarsa:start-experiment)
```

3 Lisp related topics

We provide here example solutions for a few Lisp related problems.

3.1 Quick introduction to ASDF

ASDF (Another System Definition Facility) is a library with which another Lisp libraries can be distributed. The home page of ASDF is here http://common-lisp.net/project/asdf/, and many ASDF libraries can be found here http://www.cliki.net/Library.

Sometimes ASDF is included by the Lisp implementation (e.g. in case of SBCL or CCL), in other cases you can download it from its home page. It is only a single Lisp file which does not depend on anything, so it can be simply loaded.

The ASDF facility usually has two directories. The source directory in which the packages are stored, and the system directory from which symlinks are pointed to the .asd system definition files. In this manual we refer to the former by `asdf-source-dir` and to the latter by `asdf-system-dir`. If your file system does not support symbolic links (e.g. in case of *FAT*[8] or *NTFS*[9]) or you just don't want to use symbolic links, take a look at the Using ASDF without symbolic links section. Be aware of that *Cygwin*[10] supports symbolik links, but it probably works only with Lisps compiled under it, and might not function well with the pre-compiled Windows Lisp binaries.

To set up ASDF create these directories, e.g.:

```
$ mkdir /.../asdf
$ mkdir /.../asdf/source
$ mkdir /.../asdf/system
```

Put the ASDF Lisp file, `asdf.lisp` into `/.../asdf` and create or extend your Lisp init file (check the manual of your Lisp about it) by the followings to prevent typing it all the time.

```
(load #p"/.../asdf/asdf.lisp")
(pushnew #p"/.../asdf/system/" asdf:*central-registry*)
```

The ending slash is important in the path, don't forget it!

Then you are ready to install ASDF libraries by downloading and unpacking them into the `asdf-source-dir` directory and creating a symlink to the appropriate (there can be more, e.g. for testing) .asd file(s) from the `asdf-system-dir` directory.

You can compile and load the installed libraries by this command.

```
* (asdf:oos 'asdf:load-op :library-name)
```

By default the compiled Lisp file is placed next to the source which is usually unwanted because of two reasons. First it messes up the source directory, but more importantly if the source directory was created by an other user (e.g. in case of system-wide installations), the current user may not have write access for it and on such an attempt an error is signalled.

The cure for this problem is the ASDF-Binary-Locations library. Its home page is located here http://common-lisp.net/project/asdf-binary-locations/. It stores the compiled files under a specifiable directory in the same structure as the sources are organised.

[8] http://en.wikipedia.org/wiki/File_Allocation_Table
[9] http://en.wikipedia.org/wiki/NTFS
[10] http://www.cygwin.com/

You can install this library using the above described method. Download and unpack it to `asdf-source-dir`, then create the appropriate symbolic link.

```
$ cd asdf-system-dir
$ ln -s /asdf-source-dir/asdf-binary-locations/asdf-binary-locations.asd
        asdf-binary-location.asd
```

After the installation, you might want to create a directory for the compilation result files, e.g.
`$ mkdir /.../fasls`

And add the followings to your Lisp init file.

```
(setf asdf:*centralize-lisp-binaries* t)
(setf asdf:*default-toplevel-directory* #p"/.../fasls/")
```

3.2 Using ASDF without symbolic links

This manual always considers that you are using symlinks, but if you apply this method, just ignore those parts, everything should work without them.

When symbolic links are not available or not welcomed to use, its system definition search functionality can be redefined to find the `.asd` files recursively under the `asdf:*central-registry*` directories. It will let you simply "drop-in" the new packages into these directories, and they will be available for loading without any further steps. For this you might want to add the following to the Lisp init file just after loading the `asdf.lisp` file.

```
(in-package :asdf)
(defvar *subdir-search-registry* '(#p"/my/lisp/libraries/")
  "List of directories to search subdirectories within.")
(defvar *subdir-search-wildcard* :wild
  "Value of :wild means search only one level of subdirectories;
   value of :wild-inferiors means search all levels of subdirectories
   (I don't advise using this in big directories!)")
(defun sysdef-subdir-search (system)
  (let ((latter-path (make-pathname :name (coerce-name system)
                                    :directory (list :relative
                                                     *subdir-search-wildcard*)
                                    :type "asd"
                                    :version :newest
                                    :case :local)))
    (dolist (d *subdir-search-registry*)
      (let* ((wild-path (merge-pathnames latter-path d))
             (files (directory wild-path)))
        (when files
          (return (first files)))))))
(pushnew 'sysdef-subdir-search *system-definition-search-functions*)
(in-package :cl-user)
```

This method is published on the http://www.cliki.net/asdf page in the *Alternative Sysdef Search functionality* section.