Arcade Learning Environment Technical Manual (v.0.5.0)

June 13, 2015

Contents

1 Overview

This document describes how to install the Arcade Learning Environment as well as how to use it. We will present the different ways to implement an agent:

1. **Shared Library interface** (C++): Loads ALE as a shared library (Section **??**).
2. **CTypes interface** (Python): Uses the ALE as an external Python library, allowing one to write ALE code by just importing the correspondent library. This is a very fast interface (Section 4).
3. **RL-Glue interface** (C/C++, Java, Python, Matlab or Lisp): Communicates with ALE via RL-Glue (Section 6).
4. **FIFO interface** (any language): Communicates with ALE through a text interface using `stdin/stdout` (Section 5)..

We also discuss some sensible features of this environment, such as its stochasticity.

2 Installing

2.1 Requirements

The basic requirements to build and run the ALE are:

- `CMake / make / g++`

Notice that in the past (previous to version 0.5.0) the recommended method to compile the ALE was through Makefiles written by the ALE developers. This has changed and now we suggest everyone to use CMake to generate a Makefile and then to compile the code using such generated Makefile. The old Makefiles written for Linux and OS X (files `makefile.mac` and `makefile.unix`) are still available, but we do not support them anymore.

The ALE provides some functionalities that are not activated by default, but if chosen the user is required to use additional packages:

- `SDL / RL-Glue`

2.2 Installation/Compilation

One first has to install the ALE requirements before compiling the ALE itself. Assuming that `g++` and `make` are already available to the user, to install CMake and SDL is straightforward, and it can be done through package managers in both OS X and Linux:

OS X:

```
> brew install cmake
> brew install sdl
```

Linux (e.g.: Ubuntu):

```
> sudo apt-get install cmake
> sudo apt-get install libsdl1.2-dev
```

To install RL-Glue, in both systems, we recommend the user to go to the RL-Glue webpage[1] and to follow the presented instructions.

We are going to assume the ALE was extracted to `ale_0_5`. Then, compiling it with CMake is very simple (in both systems):

```
> cd ale_0_5
ale_0_5> cmake .
ale_0_5> make -j4
```

This compiles the code without SDL and RL-Glue. If one wants to compile ALE with such libraries it is enough to change such flags in the file `ale_0_5/CMakeLists.txt`. More specifically, lines 3 and/or 4 have to be changed (by replacing the word `OFF` by `ON` in the correspondent line):

```
option(USE_SDL "Use SDL" OFF)
option(USE_RLGLUE "Use RL-Glue" OFF)
```

Due to different operational systems and installed libraries, some errors mainly related to the SDL paths being wrong have been reported. If you face such issues please look at Section 11.

3 Implementing an agent in C++ (shared libraries)

If one wants to implement an agent in C++, the best approach is to use the ALE shared library. The shared library interface allows agents to directly access ALE via a class called `ALEInterface`, defined in `ale_interface.hpp`. We are going to discuss here, step-by-step how to implement an agent that plays randomly. A code containing an example is available at `doc/examples/sharedLibraryInterfaceExample.cpp`.

This example is automatically compiled with the ALE when following the steps presented in Section 2. If one wants to disable such an option it is enough to replace `ON` by `OFF` in the line below, present in file `ale_0_5/CMakeLists.txt`:

```
option(BUILD_EXAMPLES "Build Example Agents" ON)
```

To implement an agent, the first step is to include the library `ale_interface.hpp`, either via the relative path `#include ‘‘path/from/your/code/ale_interface.hpp’’`, or as a standard header: `#include <ale_interface.hpp>`. If the later is chosen, remember to add the proper path using the flag `-I` when compiling the code.

[1] `http://glue.rl-community.org/wiki/Main_Page`

To instantiate the Arcade Learning Environment it is enough to do:

```
ALEInterface ale;
```

Once the environment is initialized, it is now possible to set its arguments. This is done with the functions `setBool(), setInt(), setFloat()`. The complete list of flags is available in Section 9. Just as an example, if one wants to set the environment's seed he must do:

```
ale.setInt("random_seed", 123);
```

After setting all variables we can now load the game ROM:

```
ale.loadROM(asterix.bin /*rom name*/);
```

There are two different sets of actions in the ALE: the "legal" action set and the "minimal" action set. The legal action set consists of 18 actions regardless of the game, therefore, some actions may not have an effect on that game. On the other hand, the minimal action set contains only the actions that do have some effect on that game. To obtain such list of actions one has to call, for example:

```
ActionVect legal_actions = ale.getLegalActionSet();
```

Then, to act one can call the function `act()` passing an object of `Action` as parameter:

```
Action a = legal_actions[rand() % legal_actions.size()];
float reward = ale.act(a);
```

Finally, one can check if the game is over using the function `ale.game_over()`. With these functions one can already implement a very simple agent that plays randomly once:

```
#include <iostream>
#include <ale_interface.hpp>

using namespace std;

int main(int argc, char** argv) {
    if (argc < 2) {
        std::cerr << "Usage: " << argv[0] << " rom_file" << std::endl;
        return 1;
    }

    ALEInterface ale;
```

```
    ale.setInt("random_seed", 123);
    ale.loadROM(argv[1]);

    ActionVect legal_actions = ale.getLegalActionSet();

    float totalReward = 0;
    while (!ale.game_over()) {
        Action a = legal_actions[rand() % legal_actions.size()];
        float reward = ale.act(a);
            totalReward += reward;
        }
        cout << "The episode ended with score: " << totalReward << endl;
    }
    return 0;
}
```

Recall this is a very simple example using the ALE. It is also important to remember to use the flag `-L` to link the ALE library. Also, to run the agent, you may need to add ALE to your library path:

```
export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/path/to/ale_0_5
```

or under Mac OS X,

```
export DYLD_LIBRARY_PATH=$DYLD_LIBRARY_PATH:/path/to/ale_0_5
```

As a final remark, a complete list of functions available in the class `ALEInterface` is presented in Section 8.

4 Implementing an agent in Python (shared libraries)

To use the Python interface it is necessary to install it after the ALE was compiled. If one has root access to the machine it is enough to run:

```
pip install .
```

otherwise, if he does not have root access, the following should work:

```
pip install --user .
```

Then, to use it in a Python code it is enough to import it properly, for example, as in the example below (`from ale_python_interface import ALEInterface`). A code containing an example is available at `doc/examples/python_example.py`.

All the functions are the same as in the C++ interface. Therefore, it is pointless to discuss all functions again. A complete example in Python is below:

```
import sys
from random import randrange
from ale_python_interface import ALEInterface

if len(sys.argv) < 2:
  print 'Usage:', sys.argv[0], 'rom_file'
  sys.exit()

ale = ALEInterface()
ale.setInt('random_seed', 123)
ale.loadROM(sys.argv[1])

# Get the list of legal actions
legal_actions = ale.getLegalActionSet()

total_reward = 0
while not ale.game_over():
  a = legal_actions[randrange(len(legal_actions))]
  reward = ale.act(a);
  total_reward += reward
print 'Episode ended with score:', total_reward
```

5 FIFO Interface

The FIFO interface is text-based and allows the possibility of run-length encoding the screen. This section documents the actual protocol used; sample code implementing this protocol in Java is also included in this release.

After preliminary handshaking, the FIFO interface enters a loop in which ALE sends information about the current time step and the agent responds with both players' actions (in general agents will only control the first player). The loop is exited when one of a number of termination conditions occurs.

5.1 Handshaking

ALE first sends the width and height of its screen matrix as a hyphen-separated string:

`www-hhh\n`

where `www` and `hhh` are both integers.

The agent then responds with a comma-separated string:

`s,r,k,R\n`

where `s`, `r`, `R` are 1 or 0 to indicate that ALE should or should not send, at every time step, screen, RAM and episode-related information (see below for details). The third argument, `k`, is deprecated and currently ignored.

5.2 Main Loop – ALE

After handshaking, ALE will then loop until one of the termination conditions occurs; these conditions are described below in Section 5.4. If terminating, ALE sends

```
DIE\n
```

Otherwise, ALE sends

```
<RAM_string><screen_string><episode_string>\n
```

Where each of the three strings is either the empty string (if the agent did not request this particular piece of information), or the relevant data terminated by a colon.

5.2.1 `RAM_string`

The RAM string is 128 2-digit hexadecimal numbers, with the i^{th} pair denoting the i^{th} byte of RAM; in total this string is 256 characters long, not including the terminating ':'.

5.2.2 `screen_string`

In "full" mode, the screen string is `www` $\times$ `hhh` 2-digit hexadecimal numbers, each representing a pixel. Pixels are sent row by row, with `www` characters for each row. In total this string is $2 \times$ `www` $\times$ `hhh` characters long.

In run-length encoding mode, the screen string consists of a variable number of (colour,length) pairs denoting a run-length encoding of the screen, also row by row. Both colour and length are described using 2-digit hexadecimal numbers. Each pair indicates that the next 'length' pixels take on the given colour; run length is thus limited to 255. Runs may wrap around onto the next row. The encoding terminates when the last pixel (i.e. the bottom-right pixel) is encoded. The length of this string is 4 characters per (colour,length) pair, and varies depending on the screen.

In either case, the screen string is terminated by a colon.

5.2.3 `episode_string`

The episode string contains two comma-separated integers indicating episode termination (1 for termination, 0 otherwise) and the most recent reward. It is also colon-terminated.

5.2.4 Example

Assuming that the agent requested screen, RAM and episode-related information, a string sent by ALE might look like:

```
000100...A401B2:3C3C3C3C00003C3C3C...4F4F0000:0,1:\n
^ 2x128 characters   ^ 2x160x210 characters    ^ongoing episode, reward of 1
```

5.3 Main Loop – Agent

After receiving a string from ALE, the agent should now send the actions of player A and player B. These are sent as a pair of comma-separated integers on a single line, e.g.:

```
2,18\n
```

where the first integer is player A's action (here, FIRE) and the second integer, player B's action (here, NOOP). Emulator control (reset, save/load state) is also handled by sending a special action value as player A's action. See Section 7.1 for the list of available actions.

5.4 Termination

ALE will terminate (and potentially send a `DIE` message to the agent) whe one of the following conditions occur:

- `stdin` is closed, indicating that the agent is no longer sending data, or
- The maximum number of frames (user-specified, with no maximum by default) has been reached.

ALE will send an end-of-episode signal when one the following is true:

- The maximum number of frames for this episode (user-specified, with no maximum by default) has been reached, or
- The game has ended, usually when player A loses their last life.

6 RL-Glue Interface

The RL-Glue interface implements the RL-Glue 3.0 protocol. It requires the user to first install the RL-Glue core. Additionally, the example agent and environment require the RL-Glue C/C++ codec. Both of these can be found on the RL-Glue web site[2].

[2] http://glue.rl-community.org

In order to use the RL-Glue interface, ALE must be compiled with RL-Glue support. This is achieved by setting `option(USE_RLGLUE "Use RL-Glue" ON)` in the CMakeListst.txt file.

Specifying the command-line argument `-game_controller rlglue` is sufficient to put ALE in RL-Glue mode. It will then communicate with the RL-Glue core like a regular RL-Glue environment.

6.1 Sample Agent and Experiment

Example source code can be found under

`ale_0_4/doc/examples`

Assuming you installed ALE under `/path/to/ale_0_5`, the RL-Glue agent and experiment can be compiled with the following command:

```
make rlglueAgent
```

As with any RL-Glue application, you will need to start the following processes to run the sample RL-Glue agent in ALE:

- `rl_glue`
- `RLGlueAgent`
- `RLGlueExperiment`
- `ale` (with command-line argument `-game_controller rlglue`)

Please refer to the RL-Glue documentation for more details.

6.2 Actions and Observations

The action space consists of both Player A and Player B's actions (see Section 7.1 for details). In general, Player B's action may safely be set to noop (18) but it should be left out altogether if the `-restricted_action_set` command–line argument was set to true.

The observation space depends on whether the `-send_rgb` argument was passed to ALE. If it was not passed, or it was set to false, the observation space consists of 33,728 integers: first the 128 bytes of RAM (taking values in 0– 255), then the 33,600 screen pixels (taking value in 0–127). The screen is provided in row-order, i.e. beginning with the 160 pixels that compose the first row.

If `-send_rgb` was set to true on the command-line, the observation space consists of 100,928 integers: first the same 128 bytes of RAM, followed by 100,800 bytes describing the screen. Each pixel is described by three bytes, taking values from 0–255, specifying the pixel's red, green and blue components in that order. The pixel order is the same as in the default case.

7 Environment Specifications

This section provides additional information regarding the environment implemented in ALE.

7.1 Available Actions

The following regular actions are defined in `common/Constants.h` and interpreted by ALE:

noop (0)	fire (1)	up (2)	right (3)	left (4)
down (5)	up-right (6)	up-left (7)	down-right (8)	down-left (9)
up-fire (10)	right-fire (11)	left-fire (12)	down-fire (13)	up-right-fire (14)
up-left-fire (15)	down-right-fire (16)	down-left-fire (17)	reset* (40)	

Note that the `reset` (40) action toggles the Atari 2600 switch, rather than reset the environment, and as such is ignored by most interfaces. In general it should be replaced by either a call to `StellaEnvironment::reset` or by sending the `system_reset` command, depending on the interface.

Player B's actions are the same as those of Player A, but with 18 added. For example, Player B's up action corresponds to the integer 20.

In addition to the regular ALE actions, the following actions are also processed by the FIFO interfaces:

save-state (43)	load-state (44)	system-reset (45)

7.2 Terminal States

Once the end of episode is reached (a terminal state in RL terminology), no further emulation takes place until the appropriate reset command is sent. This command is distinct from the Atari 2600 reset. This "system reset" avoids odd situations where the player can reset the game through button presses, or where the game normally resets itself after a number of frames. This makes for a cleaner environment interface. With the exception of the RL-Glue interface, which automatically resets the environment, the interfaces described here all provide a system reset command or method.

7.3 Saving and Loading States

State saving and loading operates in a stack-based manner: each call to save stores the current environment state onto a stack, and each call to load restores the last saved copy and removes it from the stack. The ALE 0.2 save/load mechanism, provided for backward compatibility, instead overwrites its saved copy when a save is requested. When loading a state, the currently saved copy is preserved.

7.4 Colour Averaging

Many Atari 2600 games display objects on alternating frames (sometimes even less frequently). This can be an issue for agents that do not consider the whole screen history. By default, *colour averaging* is not enabled. If enabled, the environment output (as observed by agents) is going to be a weighted blend of the last two frames. This behaviour can be turned on using the command-line argument `-color_averaging` (or the `setBool` function).

7.5 Randomness

The Atari 2600 games, as emulated by Stella, are deterministic. However, previous works have shown that it is of interest to add stochasticity to this domain, to avoid the exploration of open loop planning (trajectory optimization), *i.e.* to prevent agents from simply learning a trajectory by rote.

Such stochasticity is implemented in the ALE through the performed actions. With a probability p the previous executed action is going to be executed in the next frame, regardless of the current requested action. p has a default value of 0.25. The motivation of this implementation is to resemble how humans play games. Humans are not capable of being extremely precise regarding which frame they will take an action. Because of that we thought it would be a reasonable approach to implement stochasticity in such a way.

Also, it is important to stress that we have added TinyMT as the random number generator. This was made to avoid any sort of problem one could have by overriding the environment's seed with later calls (this can happen with `rand()/srand()`). In practice this has no effect to those using the ALE, but it is important to stress such feature.

7.6 Minimal Action Set

It may sometimes be convenient to restrict the agent to a smaller action set. This can be accomplished by querying the `RomSettings` class using the method `getMinimalActionSet`. This then returns a set of actions judged "minimal" to play a given game. Of course, algorithm designers are encouraged to devise algorithms that don't depend on this minimal action set for success.

8 ALE Interface Specifications

Below are listed the functions available in the ALE interface with the description of their behavior. The functions were divided in different sections to make the presentation more clear.

8.1 Initialization

`ALEInterface(bool display_screen)`: ALE constructor. If the `display_screen` parameter is set to `true`, and the ALE was compiled with SDL, the game display will be

presented. If set to `false` one will not see the game being played.

`void loadROM(string rom_file)`: Resets the ALE and then loads a game. After this call the game should be ready to play. If one changes (or sets) a setting (Section 8.2), it is necessary to call this function after the change so it can take effect.

8.2 Parameters setting and retrieval

`string getString(const string& key)`: Get the value of any flag passed as parameter that has a string value; *e.g.*: `getString(''record_sound_filename'')`.

`int getInt(const string& key)`: Get the value of any flag passed as parameter that has an integer value; *e.g.*: `getInt(''frame_skip'')`.

`bool getBool(const string& key)`: Get the value of any flag passed as parameter that has a boolean value; *e.g.*: `getBool(''restricted_action_set'')`.

`float getFloat(const string& key)`: Get the value of any flag passed as parameter that has a float value; *e.g.*: `getBool(''repeat_action_probability'')`.

`void setString(const string& key, const string& value)`: Set the value of any flag that has a string type; *e.g.*: `setString("random_seed", "time")`. `loadRom()` must be called before the setting will take effect.

`void setInt(const std::string& key, const int value)`: Set the value of any flag that has an integer type; *e.g.*: `setInt("frame_skip", 1)`. `loadRom()` must be called before the setting will take effect.

`void setBool(const std::string& key, const bool value)`: Set the value of any flag that has a boolean type; *e.g.*: `setBool("restricted_action_set", false)`. `loadRom()` must be called before the setting will take effect.

`void setFloat(const std::string& key, const float value)`: Set the value of any flag that has a float type; *e.g.*: `setFloat("repeat_action_probability", 0.25)`. `loadRom()` must be called before the setting will take effect.

8.3 Acting and Perceiving

`reward_t act(Action action)`: Applies an action to the game and returns the reward. It is the user's responsibility to check if the game has ended and reset when necessary (this method will keep pressing buttons on the game over screen).

`bool game_over()`: Indicates if the game has ended.

`void reset_game()`: Resets the game, but not the full system (it is not "equivalent" to unplug the console from electricity).

`ActionVect getLegalActionSet()`: Returns the vector of legal actions (all the 18 actions). This should be called only after the ROM is loaded.

`ActionVect getMinimalActionSet()`: Returns the vector of the minimal set of actions needed to play the game (all actions that have some effect on the game). This should be called only after the ROM is loaded.

`int getFrameNumber()`: Returns the current frame number since the loading of the ROM.

`const int lives()`: Returns the agent's remaining number of lives. If the game does not have the concept of lives (*e.g.* FREEWAY), this function returns 0.

`int getEpisodeFrameNumber()`: Returns the current frame number since the start of the current episode.

`const ALEScreen &getScreen()`: Returns a matrix containing the current game screen.

`const ALERAM &getRAM()`: Returns a vector containing current RAM content (byte-level).

`void saveState()`: Saves the current state of the system if one wants to be able to recover a state in the future; *e.g.* in search algorithms.

`void loadState()`: Loads a previous saved state of the system once we have a state saved.

8.4 Recording trajectories

`void saveScreenPNG(const string& filename)`: Saves the current screen as a `png` file.

`ScreenExporter *createScreenExporter(const string &path) const`: Creates a ScreenExporter object which can be used to save a sequence of frames. Frames are saved in the directory 'path', which needs to exists. This is used to generate movies depicting the behavior of agents.

9 Command-line Arguments

Command-line arguments are passed to ALE before the ROM filename. The current version removed all the internal agents in the game, therefore, the command-line arguments are mostly useful to start the communication when using FIFO pipes.

However, these parameters can also be set when using the C++ or Python interface, for example. In this case, they are set using those functons discussed in Section 8.2. The configuration file `ale_0_5/stellarc` can also be used to set frequently used command-line arguments.

The parameters that can be set are all listed below.

9.1 Main Arguments

```
-help -- prints out help information

-game_controller <fifo|fifo_named|rlglue> -- selects an ALE interface
  default: unset

-random_seed <###|time> -- picks the ALE random seed, or sets it to current time
  default: time
```

```
-display_screen <true|false> -- if true and SDL is enabled, displays ALE screen
  default: false

-sound  <true|false> -- if true and SDL is enabled, the game will have game
  sounds  enabled
  default: false
```

9.2 Environment Arguments

```
-max_num_frames ### -- max. total number of frames, or 0 for no maximum
  (it is not available in the shared library interface, i.e. to be set
  by C++  or Python code directly linking to the shared library)
  default: 0

-max_num_frames_per_episode ### -- max. number of frames per episode
  default: 0

-frame_skip ### -- frame skipping rate; 1 indicates no frame skip
  default: 1

-color_averaging <true|false> -- if true, enables colour averaging
  default: false

-record_screen_dir [save_directory] -- saves game screen images to
  save_directory

-repeat_action_probability -- stochasticity in the environment. It is the
  probability the previous action will repeated without executing the new
  one
  default: 0.25
```

9.3 FIFO Interface Arguments

```
-run_length_encoding <true|false> -- if true, encodes data using run-length
  encoding
  default: true
```

9.4 RL-Glue Interface Arguments

```
-send_rgb <true|false> -- if true, RGB values are sent for each pixel
  instead of the pallette index values
  default: false
```

```
-restricted_action_set <true|false> -- if true, agents use a smaller set of
  actions (RL-Glue interfaces only)
  default: false
```

10 Miscellaneous

This section provides additional relevant ALE information.

10.1 Displaying the Screen

ALE offers screen display and audio capabilities via the Simple DirectMedia Layer (SDL). This requires the following libraries:

- **libsdl**
- **libsdl-gfx**
- **libsdl-image**

To compile with SDL support, you should change the file `CMakeLists.txt`, replacing the word `OFF` by `ON` in the line `option(USE_SDL "Use SDL" OFF)`. Then, screen display can be enabled with the `display_screen` argument (command line or function `setBool`).

SDL support has been tested under Linux and Mac OS X.

10.2 Recording Movies

ALE now contains support for recording frames and sound via SDL (Section 10.1). An example C++ program is provided which will record a single episode of play. This program is located at

`doc/examples/videoRecordingExample.cpp`

Compiling and running this program will create a directory `record`[3] in which frames will be saved sequentially and named according to their frame numbers. Thus, if the episode lasts 683 frames then the files `record/000000.png` to `record/000682.png` are created. Furthermore, sound output is also recorded as `record/sound.wav`. The following flags control recording behaviour:

```
-display_screen <true|false> -- should be set to true for recording
  default: false
-sound <true|false> -- whether to enable sound output
  default: false
-record_screen_dir -- path to record screens; if empty, no recording occurs
  default: ""
-record_sound_filename -- path to single wav file to be recorded;
          if empty, no recording occurs
  default: ""
```

[3]The example program creates this directory, using a system call to `mkdir`. If this fails on your machine, you will need to manually create this directory.

Once frames and/or sound have been recorded, they may be joined into a movie file using the external program `ffmpeg` (installable on Mac OS X and most *nix OSes through a package manager). For your convenience, two example scripts are provided:

- `doc/scripts/videoRecordingExampleJoinMacOSX.sh`
- `doc/scripts/videoRecordingExampleJoinUnix.sh`

These should be run from the same directory that the C++ example was run from. Unfortunately `ffmpeg` is a complicated beast and taming it may require tweaking specific to your system configuration. Please contact us if you would like to provide an example script for a different configuration.

Here is a full, step-by-step example on Mac OS X (after building the project):

```
> cd ale_0_4
ale_0_4> doc/examples/videoRecordingExample space_invaders.bin

A.L.E: Arcade Learning Environment (version 0.4.4)
[Powered by Stella]

[ usual ALE output ]

Recording screens to directory: record

Recording complete. See manual for instructions on creating a video.

ale_0_4> doc/scripts/videoRecordingExampleJoinMacOSX.sh

ffmpeg version 2.6.1 Copyright (c) 2000-2015 the FFmpeg developers
  built with Apple clang version 4.1 (tags/Apple/clang-421.11.65) ...

[ loads of output ]

ale_0_4> open agent.mov
```

11 Troubleshooting / FAQ

- Where are the ROMS?

  We do not provide them due to copyrights.

- unsuported ROM file

  the rom file should be specified as the last argument on the command line. also it has to be lower case. Maybe it is too late but it is probably because you are not entering the right name for thebinary file. Each game has to be entered with a specific name which is specified at the header file (hpp) of the game. You can find these header files at games/supported if I am not wrong. Hope this solves the problem.

- sdl not found.

  add the other path.

- extract lives signal.

  now we support it.

- pixel do not have value greater than 127.

  I hope have already solved your problem, but I am going to answer you anyway. When you get the pixel value, it is a byte, but you want 7 bits. You assumed the most significative bit was always zero, but in fact, what is always zero is the least significative bit. Therefore, you have to shift the byte ¿¿ 1 to obtain the colour value.

- usr/bin/ld: cannot find -lrlagent (I manually search inside the libs and includes of rlglue and lrlagent it not even there and the directory ld doesnt exist in /usr/bin)

  install rl-glue!