`runlisp` – run scripts written in Common Lisp

Mark Wooding

3 March 2024

Contents

`runlisp` is a small C program intended to be run from a script `#!` line. It selects and invokes a Common Lisp implementation, so as to run the script. In this sense, `runlisp` is a partial replacement for `cl-launch`.

Currently, the following Lisp implementations are supported:

- Armed Bear Common Lisp (`abcl`),
- Clozure Common Lisp (`ccl`),
- GNU CLisp (`clisp`),
- Carnegie–Mellon Univerity Common Lisp (`cmucl`),
- Embeddable Common Lisp (`ecl`), and

- Steel Bank Common Lisp (sbcl).

Adding more Lisps is simply a matter of writing the necessary runes in a configuration file. Of course, there's a benefit to having a collection of high-quality configuration runes curated centrally, so I'm happy to accept submissions in support of any free[1] Lisp implementations.

1 Writing scripts in Common Lisp

1.1 Basic use

The obvious way to use runlisp is in a shebang (#!) line at the top of a script. For example:

```
#! /usr/local/bin/runlisp
(format t "Hello from Lisp!~%")
```

Script interpreters must be named with absolute pathnames in shebang lines; if your runlisp is installed somewhere other than /usr/local/bin/ then you'll need to write something different. Alternatively, a common hack involves abusing the env program as a script interpreter, because it will do a path search for the program it's supposed to run:

```
#! /usr/bin/env runlisp
(format t "Hello from Lisp!~%")
```

1.2 Specific Lisps

Lisp implementations are not created equal – for good reason. If your script depends on the features of some particular Lisp implementation, then you can tell runlisp that it must use that implementation to run your script using the -L option; for example:

```
#! /usr/local/bin/runlisp -Lsbcl
(format t "Hello from Steel Bank Common Lisp!~%")
```

If your script supports several Lisps, but not all, then list them all in the -L option, separated by commas:

```
#! /usr/local/bin/runlisp -Lsbcl,ccl
(format t #.(concatenate 'string
                         "Hello from "
                         #+sbcl "Steel Bank"
                         #+ccl "Clozure"
                         #-(or sbcl ccl) "an unexpected"
                         " Common Lisp!~%"))
```

[1] Here I mean free as in freedom.

1.3 Embedded options

If your script requires features of particular Lisp implementations *and* you don't want to hardcode an absolute path to `runlisp`, then you have a problem. Most Unix-like operating systems will parse a shebang line into the initial `#!`, the pathname to the interpreter program, and a *single* optional argument: any further spaces don't separate further arguments: they just get included in the first argument, all the way up to the end of the line. So

```
#! /usr/bin/env runlisp -Lsbcl
(format t "Hello from Steel Bank Common Lisp!~%")
```

won't work: it'll just try to run a program named `runlisp -Lsbcl`, with a space in the middle of its name, and that's quite unlikely to exist.

To help with this situation, `runlisp` reads *embedded options* from your script. Specifically, if the script's second line contains the token `@RUNLISP:` then `runlisp` will parse additional options from this line. So the following will work properly.

```
#! /usr/bin/env runlisp
;;; @RUNLISP: -Lsbcl
(format t "Hello from Steel Bank Common Lisp!~%")
```

Embedded options are split at spaces properly. Spaces can be escaped or quoted in (an approximation to) the usual shell manner, should that be necessary. See the manpage for the gory details.

1.4 Common environment

`runlisp` puts some effort into making sure that Lisp scripts get the same view of the world regardless of which implementation is running them.

For example:

- The `asdf` and `uiop` systems are loaded and ready for use.
- The script's command-line arguments are available in `uiop:*command-line-arguments*`. Its name can be found by calling `(uiop:argv0)` – though it's probably also in `*load-pathname*`.
- The prevailing Unix standard input, output, and error files are available through the Lisp `*standard-input*`, `*standard-output*`, and `*error-ouptut*` streams, respectively. (This is, alas, not a foregone conclusion.)
- The keyword `:runlisp-script` is added to the `*features*` list. This means that your script can tell whether it's being run from the command line, and should therefore do its thing and then quit; or merely being loaded into a Lisp system, e.g., for debugging or development, and should sit still and not do anything until it's asked.

See the manual for the complete list of guarantees.

2 Invoking Lisp implementations

2.1 Basic use

A secondary use of `runlisp` is in build scripts for Lisp programs. If the entire project is just a Lisp library, then it's possibly acceptable to just provide an ASDF system definition and expect users to type `(asdf:load-system "mumble")` to use it. If it's a program, or there are things other than Lisp which ASDF can't or shouldn't handle – significant pieces in other languages, or a Lisp executable image to make and install – then it seems sensible to make the project's main build system be something language-agnostic, say Unix `make`, and arrange for that to invoke ASDF at the appropriate time.

But how should that be arranged? It's relatively easy for a project' Lisp code to support multiple Lisp implementation; but each implementation wants different runes for evaluating Lisp forms from the command line, and some of them don't provide an ideal environment for integrating into a build system. So `runlisp` provides a simple common command-line interface for evaluating Lisp forms. For example:

```
$ runlisp -e '(format t "~A~%" (+ 1 2))'
3
```

If your build script needs to get information out of Lisp, then wrapping `format`, or even `princ`, around forms is annoying; so `runlisp` has a `-p` option which prints the values of the forms it evaluates.

```
$ runlisp -e '(+ 1 2)'
3
```

If a form produces multiple values, then `-p` will print all of them, as if by `princ`, separated by spaces, on a single line:

```
$ runlisp -p '(floor 5 2)'
2 1
```

There's also a `-d` option, which does the same thing as `-p`, only it prints values as if by `prin1`. For example,

```
$ runlisp -p '"Hello, world!"'
Hello, world!
runlisp -d '"Hello, world!"'
"Hello, world!"
```

In addition to evaluating forms with `-e`, and printing their values with `-d` and `-p`, you can also load a file of Lisp code using `-l`.

When `runlisp` is acting on `-e`, `-p`, and/or `-l` options, it's said to be running in *eval* mode, rather than its usual *script* mode. In eval mode, it *doesn't* set `:runlisp-script` in `*features*`.

You can still insist that runlisp use a particular Lisp implementation, or one of a subset of implementations, using the -L option mentioned above.

```
$ runlisp -Lsbcl -p "(lisp-implementation-type)"
"SBCL"
```

2.2 Command-line processing

When scripting a Lisp – as opposed to running a Lisp script – it's not necessarily the case that your script knows in advance exactly what it needs to ask Lisp to do. For example, it might need to tell Lisp to install a program in a particular directory, determined by Autoconf. While it's certainly *possible* to quote such data and splice them into Lisp forms, it's more convenient to pass them in separately. So runlisp ensures that the command-line options are available to Lisp forms via uiop:*command-line-arguments*, as they are to a Lisp script.

```
$ runlisp -p "uiop:*command-line-arguments*" one two three
("one" "two" "three")
```

When running Lisp forms like this, (uiop:argv0) isn't very meaningful. (Currently, it reveals the name of the script which runlisp uses to implement this feature.)

3 Configuring runlisp

3.1 Where runlisp looks for configuration

You can influence which Lisp implementations are chosen by runlisp by writing configuration files, and/or setting environment variables.

The runlisp program looks for configuration in a number of places.

- There's a system-global directory SYSCONFDIR/runlisp/runlisp.d/. All of the files in this directory named SOMETHING.conf are read, in increasing lexicographical order by name. The package comes with a file 0base.conf intended to be read first, so that it can be overridden if necessar. This sets up basic definitions, and defines the necessary runes for those Lisp implementations which are supported 'out of the box'. New Lisp packages might come with additional files to drop into this directory.
- There's a system-global file SYSCONFDIR/runlisp/runlisp.conf which is intended to be edited by the system administrator to account for any local quirks. This is read *after* the directory, which is intended to be used by distribution packages, so that the system administrator can override them.
- Users can create files $HOME/.runlisp.conf and/or $HOME/.config/runlisp.conf[2] in their home directories to add support for privately installed Lisp systems, or to override settings made by earlier configuration files.

The configuration syntax is complicated, and explained in detail in the **runlisp.conf** manpage.

[2] More properly, in $XDG_CONFIG_HOME/runlisp.conf, if you set that.

Configuration options can also be set on the command line, though the effects are subtly different. Again, see the manual pages for details.

3.2 Deciding which Lisp implementation to use

The `prefer` option specifies a *preference list* of Lisp implementations. The value is a list of Lisp implementation names, as you'd give to `-L`, separated by commas and/or spaces. If the environment variable `RUNLISP_PREFER` is set, then this overrides any value found in the configuration files.

When deciding which Lisp implementation to use, `runlisp` works as follows. It builds a list of *acceptable* Lisp implementations from the `-L` command-line option, and a list of *preferred* Lisp implementations from the `prefer` configuration option (or environment variable). If there aren't any `-L` options, then it assumes that *all* Lisp implementations are acceptable; if no `prefer` option is set then it assumes that *no* Lisp implementations are preferred. It then works through the preferred list in order: if it finds an implementation which is installed and acceptable, then it uses that one. If that doesn't work, then it works through the acceptable implementations that it hasn't tried yet, in order, and if it finds one of those that's installed, then it runs that one. Otherwise it reports an error and gives up.

3.3 Supporting new Lisp implementations

`runlisp` tries hard to make adding support for a new Lisp as painless as possible. An awkward Lisp will of course cause trouble, but `runlisp` itself is easy.

As a simple example, let's add support for the 32-bit version of Clozure CL. The source code for Clozure CL easily builds both 32- and 64-bit binaries in either 32- or 64-bit userlands, and one might reasonably want to use the 32-bit CCL for some reason. The following configuration stanza is sufficient

```
[ccl32]
@PARENTS = ccl
command = ${@ENV:CCL32?ccl32}
```

- The first line heads a configuration section, providing the name which will be used for this Lisp implementation, e.g., in `-L` options or `prefer` lists.
- The second line tells `runlisp` that configuration settings not found in this section should be looked up in the `ccl` section instead.
- The third line defines the command to be used to invoke the Lisp system. It tries to find an environment variable named `CCL32`, falling back to looking up `ccl32` in the path otherwise.

And, err…, that's it. The `@PARENTS` setting uses the detailed command-line runes for `ccl`, so they don't need to be written out again.

That was rather anticlimactic, because all of the work got done somewhere else. So let's look at a complete example: Steel Bank Common Lisp. (SBCL's command-line interface is well thought-out, so this is an ideal opportunity to explain how `runlisp` configuration works, without getting bogged down in the details of fighting less amenable Lisps.)

The provided 0base.conf file defines SBCL as follows.

```
[sbcl]

command = ${@ENV:SBCL?sbcl}
image-file = ${@NAME}+asdf.core

run-script =
        ${command} --noinform
                $?@IMAGE{--core "${image-path}" --eval "${image-restore}" |
                         --eval "${run-script-prelude}"}
                --script "${@SCRIPT}"

dump-image =
        ${command} --noinform --no-userinit --no-sysinit --disable-debugger
                --eval "${dump-image-prelude}"
                --eval "(sb-ext:save-lisp-and-die \"${@IMAGENEW|q}\")"
```

Let's take this in slightly larger pieces.

- We see the [sbcl] section heading, and the command setting again. These should now be unsurprising.
- There's no @PARENTS setting, so by default the sbcl section inherits settings from the @COMMON section, defined in 0base.conf. We shall use a number of definitions from this section.
- The image-file gives the name of the custom image file to look for when trying to start SBCL, but not the directory. (The directory is named by the image-dir configuration setting.) The image file will be named sbcl+asdf.core, but this isn't what's written. Instead, it uses ${@NAME}, which is replaced by the name of the section being processed. When we're running SBCL, this does the same thing; but if someone wants to configure a new foo Lisp and set @PARENTS to sbcl, then the image file for foo will be named foo+asdf.core by default. You needn't take such care when configuring Lisp implementations for your own purposes, but it's important for configurations which will be widely used.
- The run-script setting explains how to get SBCL to run a script. This string is broken into words at (unquoted) spaces.

  The syntax $?VAR{CONSEQ|ALT} means: if a configuration setting VAR is defined, then expand to CONSEQ; otherwise, expand to ALT. In this case, if the magic setting @IMAGE is defined, then we add the tokens --core "${image-path}" --eval "${image-restore}" to the SBCL command line; otherwise, we add --eval "${run-script-prelude}". The @IMAGE setting is defined by runlisp only if (a) a custom image was found in the correct place, and (b) use of custom images isn't disabled on its command line.

  The ${image-path} token expands to the full pathname to the custom image file; image-restore is a predefined Lisp expression to be run when starting from a dumped image (e.g., to get ASDF to refresh its idea of which systems are available).

  The run-script-prelude is another (somewhat involved) Lisp expression which sets up

a Lisp environment suitable for running scripts – e.g., by arranging to ignore `#!` lines, and pushing `:runlisp-script` onto `*features*`.

Finally, regardless of whether we're using a custom or vanilla image, we add the tokens `--script "${@SCRIPT}"` to the command line. The `${@SCRIPT}` token is replaced by the actual script pathname. `runlisp` then appends further arguments from its own command line and runs the command. (For most Lisps, `uiop` needs a `--` marker before the user arguments, but not for SBCL.)

- Finally, `dump-image` defines a command line for dumping a custom images. The `dump-image-prelude` setting is a Lisp expression for setting up a Lisp so that it will be in a useful state when dumped: it's very similar to `run-script-prelude`, and is built out of many of the same pieces.

  The thing we haven't seen before is `${@IMAGENEW|q}`. The `@IMAGENEW` setting is defined by the `dump-runlisp-image` program to name the file in which the new image should be saved.[3] The `|q` 'filter' is new: it means that the filename should be escaped suitable for inclusion in a Lisp quoted string, by prefixing each `\` or `"` with a `\`.

That's more or less all there is. SBCL is a particularly simple example, but mostly because other Lisp implementations require fancier stunts *at the Lisp level*. The `runlisp`-level configuration isn't any more complicated than SBCL.

4 What's wrong with `cl-launch`?

The short version is that `cl-launch` is slow and inconvenient. `cl-launch` is a big, complicated Common Lisp/Bourne shell polyglot which tries to do everything but doesn't quite succeed.

4.1 It's slow.

I took a trivial Lisp script:

```
(format t "Hello from ~A!~%~
           Script = '~A'~%~
           Arguments = (~{'~A'~^, ~})~%"
        (lisp-implementation-type)
        (uiop:argv0)
        uiop:*command-line-arguments*)
```

I timed how long it took to run on all of `runlisp`'s supported Lisp implementations, and compared them to how long `cl-launch` took: the results are shown in table 1. `runlisp` is *at least* two and half times faster at running this script than `cl-launch` on all implementations except Clozure CL[4], and approaching four and a half times faster on SBCL.

But this is using the 'vanilla' Lisp images installed with the implementations. `runlisp` by default builds custom images for most Lisp implementations, which improves startup

[3] `dump-runlisp-image` wants to avoid clobbering an existing image with a half-finished one, so it tries to arrange for the new image to be written to a different file, and then renames it once it's been created successfully.)

[4] I don't know why Clozure CL shows such a small difference here.

Implementation	`cl-launch` (s)	`runlisp` (s)	`runlisp` (factor)
ABCL	7.3378	2.6474	2.772
Clozure CL	1.2888	0.9742	1.323
GNU CLisp	1.2405	0.2703	4.589
CMU CL	0.9521	0.3097	3.074
ECL	0.8020	0.3236	2.478
SBCL	0.3205	0.0874	3.667

Table 1: `cl-launch` vs `runlisp` (with vanilla images)

Implementation	`cl-launch` (s)	`runlisp` (s)	`runlisp` (factor)
ABCL	7.3378	2.7023	2.715
Clozure CL	1.2888	0.0371	34.739
GNU CLisp	1.2405	0.0191	64.948
CMU CL	0.9521	0.0060	158.683
ECL	0.8020	0.3275	2.449
SBCL	0.3205	0.0064	50.078

Table 2: `cl-launch` vs `runlisp` (with custom images)

performance significantly; see table 2. (I don't currently know how to build a useful custom image for ABCL. `runlisp` does build a custom image for ECL, but it doesn't help significantly.) These results are summarized in figure 1.

Unlike `cl-launch`, with some Lisp implementations at least, `runlisp` startup performance is usefully comparable to other popular scripting language implementations. I wrote similarly trivial scripts in a number of other languages, and timed them; the results are tabulated in table 3 and graphed in figure 2.

(All the timings in this section were performed on the same 2020 Dell XPS13 laptop running Debian 'buster'. The tools used to make the measurements are included in the source distribution, in the `bench/` subdirectory.)

Implementation	Time (ms)
Clozure CL	37.1
GNU CLisp	19.1
CMU CL	6.0
SBCL	6.4
Perl	1.1
Python	6.8
Debian Almquist shell (dash)	1.2
GNU Bash	1.5
Z Shell	3.1
Tiny C (compile & run)	1.6
GCC (precompiled)	0.6

Table 3: `runlisp` vs other interpreters

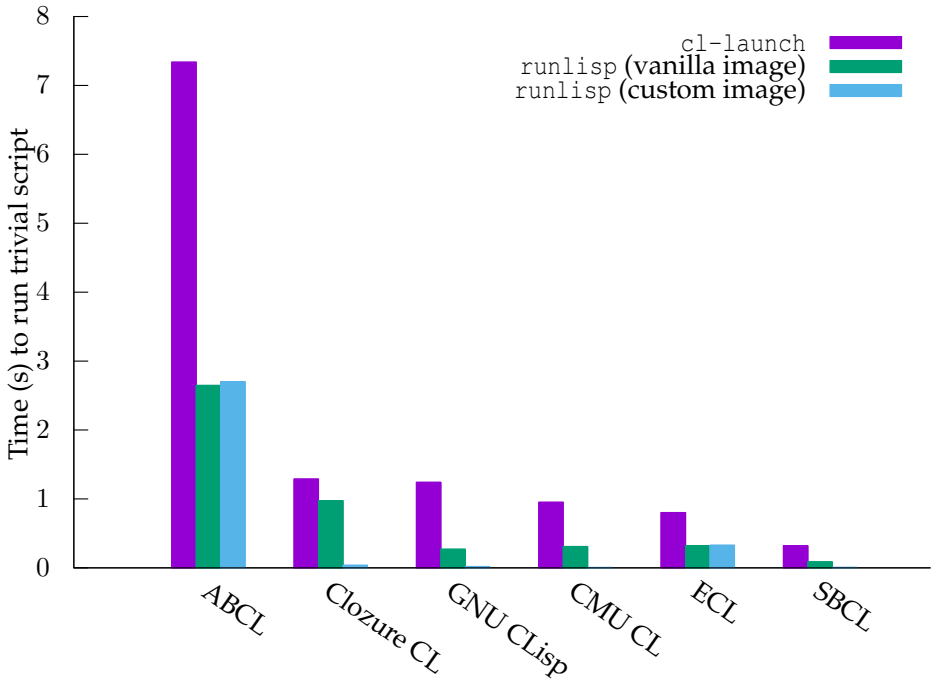

Figure 1: Comparison of `runlisp` and `cl-launch` times

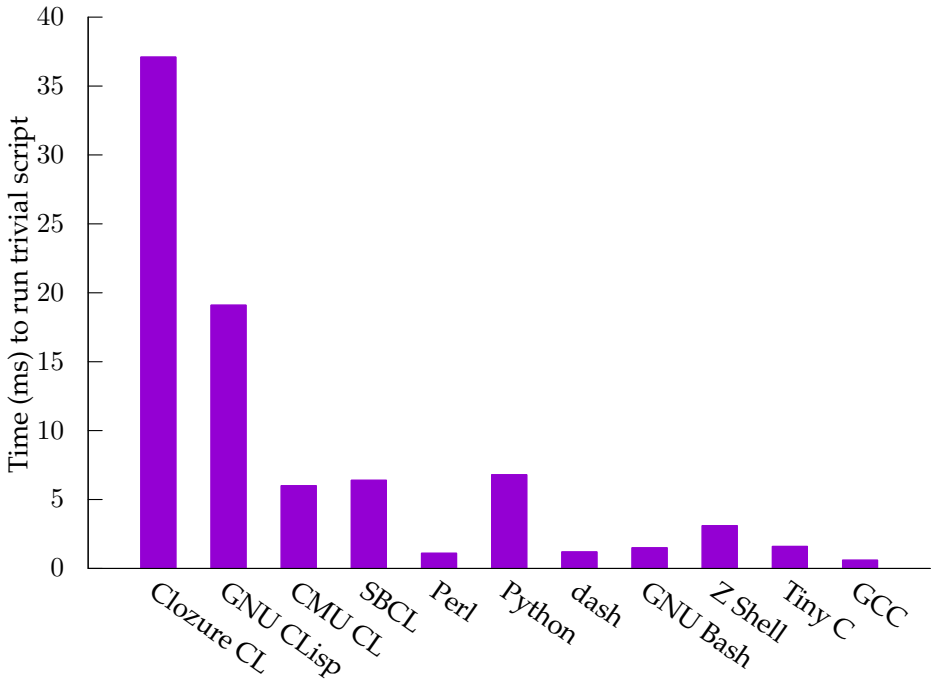

Figure 2: Comparison of `runlisp` and other script interpreters

4.2 It's inconvenient

cl-launch has this elaborate machinery which reads shell script fragments from various places and sets variables like $LISPS, but it doesn't quite work.

Unlike other scripting languages such as Perl or Python, Common Lisp has lots of implementations, and they all have various unique features (and bugs) which a script might rely on (or need to avoid). Also, a user might have preferences about which Lisps to use. cl-launch's approach to this problem is a system_preferred_lisps shell function which can be used in ~/.cl-launchrc to select a Lisp system for a particular 'software system', though this notion doesn't appear to be well-defined, but this all works by editing a single $LISPS shell variable. By contrast, runlisp has a -L option with which scripts can specify the Lisp systems they support (in a preference order), and a prefer configuration setting with which users can express their own preferences: runlisp will never choose a Lisp system which the script can't deal with, but it will respect the user's relative preferences.

Also, cl-launch is a monolith. Adding a new Lisp implementation to it, or changing how a particular implementation is invoked, is rather involved. By contrast, runlisp makes this remarkably easy, as described in 3.3.

4.3 It doesn't establish a (useful) common environment

A number of Lisp systems are annoyingly deficient in their handling of scripts.

For example, when GNU CLisp's -x option is used, it rebinds *standard-input* to an internal string stream holding the expression passed in on the command line, leaving the process's actual stdin nearly impossible to access.

```
$ date | cl-launch -l sbcl -i "(princ (read-line nil nil))" # expected
Sun  9 Aug 14:39:10 BST 2020
$ date | cl-launch -l clisp -i "(princ (read-line nil nil))" # bug!
NIL
```

As another example, Armed Bear Common Lisp doesn't seem to believe in the stderr stream: when it starts up, *error-ouptut* is bound to the standard output, just like *standard-output*. Also, cl-launch loading ASDF causes a huge number of style-warning messages to be written to stdout, making ABCL pretty much useless for writing filter scripts.

```
$ cl-launch -l sbcl -i '(progn
                          (format *standard-output* "output~%")
                          (format *error-output* "error~%"))' \
  > >(sed 's/^/stdout: /') 2> >(sed 's/^/stderr: /')
stdout: output
stderr: error
$ cl-launch -l abcl -i '(progn
                          (format *standard-output* "output~%")
                          (format *error-output* "error~%"))' \
  > >(sed 's/^/stdout: /') 2> >(sed 's/^/stderr: /')
[1813 lines of compiler warnings tagged 'stdout:']
stdout: output
```

```
stdout: error
```

runlisp takes care of all of this, providing a basic but useful common level of shell integration for all its supported Lisp implementations. In particular:

- It ensures that the standard Unix 'stdin', 'stdout', and 'stderr' file descriptors are hooked up to the Lisp *standard-input*, *standard-output*, and *error-output* streams.
- It ensures that starting a script doesn't write a deluge of diagnostic drivel.

The complete details are given in runlisp's manpage.

4.4 Why might one prefer cl-launch anyway?

On the other hand, cl-launch is well established and full-featured.

cl-launch compiles scripts before trying to run them, so they'll run faster on Lisps which use an interpreter by default. It has a caching feature so running a script a second time doesn't need to recompile it. If your scripts are compute-intensive and benefit from ahead-of-time compilation then maybe cl-launch is preferable.

cl-launch supports more Lisp systems. I only have six installed on my development machine at the moment, so those are the ones that runlisp supports. If you want your scripts to be able to run on other Lisps, then cl-launch is the way to do that. Of course, I welcome patches to help runlisp support other free Lisp implementations. cl-launch also supports proprietary Lisps: I have very little interest in these, so if you want to run scripts using Allegro or LispWorks then cl-launch is your only choice.