Using LISP

We only have one reliable LISP implementation available: Liquid Common LISP on the leland workstations. This handout describes how to use lcl on the . Since Common LISP is quite portable, you should be able to write your code pretty much wherever you like, so if you have another environment you'd like to use, feel free. No matter which platform you develop on, you will need to upload your code to the `leland` workstations for final testing and e-submission when you're done.

This handout gives the briefest of introductions to the interpreters, but in the past, students have found it relatively straightforward to get around in the new environment. As always, you're welcome to stop by office hours or e-mail `cs107@cs` if you need more assistance.

Liquid Common LISP

For those of you who love emacs and have grown really accustomed to hanging out in Sweet Hall or logging in remotely, you'll be delighted to hear of a LISP interpreter for workstations called `lcl` (for Liquid Common Lisp). All machines I logged into, save the amys and the raptors, have lcl, so amy and/raptor fans will need to peel themselves away and use the other machines this time. The amys and the raptors support another LISP environment called `acl` (Allegro Common Lisp), but it's flaky in comparison and I'm suggesting that you avoid it.

To get started:

- Make sure that `/usr/local/bin` is in your path.
- Type `lcl` to enter the interpreter. Typing lcl at the command prompt often stalls the processor for 15 to 20 seconds, but the interpreter will indeed fire up—we promise.
- The interpreter prompt ends with '>'. Completing an S-expression and hitting return will cause the interpreter to evaluate the expression and print the result.
- `(load "mycode.lisp")` will load and evaluate the file "`mycode.lisp`" from within the current directory. It tends to complain about your code not beginning with in-package, but you can safely ignore this.
- `(quit)` will exit the interpreter when you're ready to end your session.

You will edit your code in emacs and then load the code into lcl again to replace the previous function definitions as you make changes.

Liquid debugger

When something goes wrong during evaluation, the system gives an error message such as the following and put you into their debugger:

```
>>Error: The value of NUMBER1, "hello", should be a NUMBER

 (:ADVICE SYSTEM:ENTER-DEBUGGER LIQUID::LEP)
   Rest arg 0 (ARGS): (#<Condition TYPE-ERROR FDDCFAE> "" NIL)
:C  0: Use a new value
:A  1: Return to level 0.
```

You can type `?` to get a list of the most commonly used debugger commands or `:??` for the full list of options. Some of the more useful ones to know are `:a` to abort processing of the current expression and terminate the evaluation and `:b` get a backtrace that shows where you are in evaluation at the point the error occurred.

You can also use the "step" command in Liquid as in MCL although it is a little more difficult to follow in a text-based mode. Try evaluating `(step (+ (* 3 4) (/ 5 2)))`. Type `?` to see the options at each stage, `:n` will allow you to repeatedly single-step through the stages of evaluation.

Liquid graphical interface

There is an optional XWindows interface to Liquid that you may find more pleasant to use that the command-line version. The X version is only available if you have an X client or work on the console directly. Once you start `lcl` normally, evaluate the expression `(env:start-environment)` to start up the windowing interface. Experimentation seems to show that the X environment on the amy hosts is not compatible with the version needed for Liquid's interface, but it seems to work fine from the raptor and firebird hosts. From there, a window appears with some menus that allow you to access the windowing features. Under the **Tools** menu, you can start up a new **Listener** window. You can access and edit files using commands from the **File** menu. The editor is a little primitive relative to other nice GUI tools you might have used, but it gets the job done. There is access to all the help system and online manuals under the **Help** menu.

Liquid online manual

There is a full HTML manual of all the Liquid features available on the leland file system as `/usr/local/apps/liquid5.0/common/lcl/5-0/manual/online`.

Other interpreters

In the past, various resourceful students have found free and shareware LISP interpreters to use on their own computers. Since Common LISP is quite portable, it should be possible to develop in one environment and move to another without hassle. However, be safe, not sorry— be sure to reserve time to handle any glitches that might

arise before submitting your final version on leland, where we will test your code using lcl.

One PC version has been successfully used in the past is Harlequin's LispWorks personal edition (see `http://www.harlequin.com/products/ads/lisp/win.shtml`). Harlequin is the same company that creates Liquid Common Lisp, so I would assume their PC product is similar in nature. Not being a PC user myself, I can't really comment on how nice or reliable it is, but it seems folks in the past have found it acceptable.

On the Mac side of things, you might want to check out the shareware PowerLisp at `http://www.corman.net/PowerLisp.html`. It is PowerPC-native, compliant with the ANSI standard, and has a nice integrated editor.

If you know of others worth considering, feel free to post your information to the newsgroup for the benefit of other students.

Uploading and converting files

One thing to be aware of that PC, Mac, and UNIX all have different end-of-line conventions for text files. (sigh, standards…) Thus if you edit your files on a Mac or PC, when you upload them to the UNIX system, you may have to fuss with the formatting to recover your original intentions. (It is not recommended that you submit a garbled file with no proper newlines. LISP code is unreadable enough as it is!)

If you use Fetch to move your files from Mac to UNIX, it does the newline translation automatically. If you use MacLeland, it won't be done and you'll need to do it yourself. The `tr` command (see `man` page) can help you convert the Macintosh return character to the UNIX newline character. To take the contents of `macFile.l` and write a new version with translated newlines into `unixFile.l`, do this:

```
tr '\r' '\n' < macFile.l > unixFile.l
```

General Lisp debugging advice

In general, it can be real trouble to debug more than a few lines of LISP at once, so your development strategy should be to write many small functions and test each thoroughly in isolation before integrating them into larger, more complicated expressions. Here are a few of the more simple pitfalls you might stumble across to give you a little warning about some of the common mistakes you might have trouble with and how the symptom would be reported:

- Don't forget the single quote in front of those expressions that you don't intend to be evaluated, such as lists of data or symbols. For example:

```
(cons 1 (a b))
```

  will give the result that it cannot apply function `a` to argument `b`. You probably wanted a single quote in front of the list `'(a b)`.

- A quote in front of an S-expression suppresses evaluation for the entire expression, start to finish. Putting extra quotes inside won't help and can cause problems:

```
(cons 1 '(a '(b c)))
```

  will give the result `(1 a '(b c))` (note the inner quote was retained).

- The single quote can also be used to avoid evaluation of symbol names:

```
(cons apple '(1 2))
```

  will report error `apple` is unbound. Adding a single quote in front of `apple` instructs the interpreter to take the symbol as a literal and not try to look up its value. With that change, the above expression evaluates to `(APPLE 1 2)`.

- Don’t mistakenly drop back into C syntax for function calls and put the arguments inside a set of parentheses:

```
(cons (1 '(3 4))))
```

  will give an error about trying to apply the function `1` to the argument `'(3 4)`.

- Getting the #' backwards when extracting the functional object:

```
(sort '(4 10 3) '#<)
```

  will produce a "Reader Error" when reading the input expression.

- Forgetting the else clause in an if expression:

```
(if (> 4 5) "hello")
```

  This evaluates to NIL. When there is no explicit else expression, nil is assumed. This is probably not good to depend on since it is more likely to be an oversight than intentional. You can explicitly give nil if that is what you want for the else.

- Passing an atom, not a list, as the second argument to cons:

```
(cons 3 5)
```

  The second argument to cons should be a list and the first is the new member to push on the front of the list. However, if you pass an atom as the second arg, it still works and creates something called a "dotted pair" which is printed as

`(3.5)`. We won't really talk about dotted pairs and there will be no need to use them in CS107. If you find that your output has lists with dots in the middle, it is a sign that at some point you made this mistake and need to hunt it down.