Introduction to Artificial Intelligence
CISC-481/681

Program 0: Intro to Lisp

Reading:

- Class Notes
- Handouts Summarizing 100'ds of Standard Common Lisp Functions
- Handouts on Harlequin Liquid Lisp
- "Welcome to Lisp" [Chapter 2 of Graham's ANSI Common Lisp]

The purpose of the exercises below is to familiarize you with the basics of the Lisp language and interpreter. Spending a little time on simple mechanics now will save you a *great* deal of time over the rest of the semester.

1. Starting Harlequin Liquid Common Lisp (LCL)

Grad students: on the Solaris EECIS machines, the LCL binary is in `/usr/local/lang/lisp/lisp`. Of course you can use any machine as your X display, but our license for LCL 5.0 is for Solaris only so you'll need to run the image itself from a Solaris machine. Since we have brand-new desktops, please use them and DON'T run things on REN! Better yet, open a window on Strauss, set your DISPLAY to the local machine, and xhost your local machine to Strauss (and follow the undergrad directions given below)!!! To run lisp locally, You should copy the file `~decker/lisp-init.lisp` to your top-level login directory. Using an X display with the DISPLAY environment var set correctly, start the text interface from unix with "lisp". After everything's loaded, start the X-windows interface via the lisp form `(env:start-environment)`. When I refer to `$COURSEHOME`, I am referring to the EECIS directory `~decker/courses/681`.

Undergrads: On the Composers (strauss and mahler), you should read/print the file `~decker/Class/CISC681/student-readme`. You should follow the instructions there to set up your account, and "newgrp 2092" when you work on the course. As far as I can tell, you already have /opt/bin in your path, as the lisp image is located at /opt/bin/lisp on the Composers. You should also copy `~decker/lisp-init.lisp` you your top-level login directory. *Note that the init files for the EECIS and Composers are different.* Now, using an X display with the DISPLAY environment var set correctly, start the text interface from unix with "lisp". After everything's loaded, start the X-windows interface via the lisp form `(env:start-environment)`. When I refer to `$COURSEHOME`, I am referring to the composer directory `~decker/Class/CISC681`.

Emacs users: You can also run lisp inside XEmacs and GnuEmacs. See the appendix to this assignment.

2. Getting started with LCL

After dismissing the LCL "Common Lispworks" environment dialog box, the GC (Garbage Collection) monitor and Lispworks Podium windows appear. The GC-mon shows you the GC status and allows asyncronous garbage collection (not something to worry about in this course). The Podium allows you to start any other Lispworks Tool (like an Editor or Lisp Listener). The GC-monitor has four buttons, the bottom one will say "Idle" when there is nothing happening, "Running" when Lisp is working, and "GC" when the garbage collector has suspended Lisp. If nothing is happening when you type/click, make sure that the GC monitor is not currently displaying "GC" before you start worrying :-).

In the Lispworks environment, you will use a text-editing system which incorporates an editor closely resembling the widely used editor Emacs. You may want to check out the "Guide to the Editor" manual, which is on the web. You can get to it by clicking on the Help menu, and selecting Manuals..., which will bring up a dialog box from which you can mouse on a manual. The manuals are displayed via Netscape. If you already have a Netscape window running, then the manual will appear there.

The Listener is a window that has an interpreter to which you can type Lisp forms and see the values back. Other important tools are a window-based debugger that can show you the stack visually, and an Inspector that allows you to view data objects and point and click to follow links, record fields, and object instance variables.

Evaluating expressions

Start up an Editor and a Listener from the Podium. Type some Lisp expressions in the Listener.

Because of the simple syntax of Lisp, it is extremely important to always properly indent expressions and to display long expressions over several lines for readability. An expression may be typed on a single line or on several lines; the Lisp interpreter ignores redundant spaces and carriage returns. It is to your advantage to format your work so that you (and others) can read it easily. It is also helpful in detecting errors introduced by incorrectly placed parentheses. For example the two expressions

```
(* 5 (- 2 (/ 4 2) (/ 8 3)))

(* 5 (- 2 (/ 4 2)) (/ 8 3))
```

look deceptively similar but have different values. Properly indented, however, the difference is obvious.

```
(* 5
   (- 2
      (/ 4 2)
      (/ 8 3)))

(* 5
   (- 2
      (/ 4 2))
   (/ 8 3))
```

LCL provides several commands that "pretty-print" your code, e.g., indents lines to reflect the inherent structure of the Lisp expressions. Typing TAB, or C-i, will fix the indentation of the current line in either the Editor or the Listener.

Creating a file

Since the Lisp Listener will chronologically list all the expressions you evaluate, and since you will generally have to try more than one version of the same procedure as part of the coding and debugging process, it is usually better to keep your procedure definitions in a separate editing buffer, rather than to work only in the Listener. You can save this other buffer in a file on your disk so you can split your work over more than one session.

The basic idea is that you type your programs into the editor buffer, and then *compile* or *evaluate* them. When running *evaluated* (or *interpreted*) code, the code is run semi-symbolically, reduced line-by-line (lisp-form-by-lisp-form). This can be great for debugging, since if there is an error it will occur while executing recognizable lisp forms. On the other hand, interpreted code is slow. Compiling code produces machine instructions (i.e. RISC instructions for the Sun Sparcs) and is very fast. Lisp is dynamically linked and incrementally compiled, so you can compile some functions, and interpret others. Often I compile everything, and then evalute/interpret the few functions I am currently debugging. For beginners, you might just start by evaluating everything until you're a bit sure of yourself.

You can compile or evaluate Files, Buffers, and individual Definitions, using these menus at the top of the editor. There are also corresponding keystrokes. The only tricky one is compile file, which compiles a whole file on disk, producing a binary ".sbin" file. When compiling a *file* it is *not* automatically loaded into the lisp environment—you'll see that the editor provides a "compile and load" command to do both. When compiling an Editor Buffer or a single Definition, the compiled function is automatically loaded into the lisp environment.

You can then start running your program by calling a top-level function in the the Listener.

The lisp environment remembers everything you do, every function defined, for the entire length of your session! This can mess up beginners who forget that they once defined a function "foo", and then later use that name, thinking that it's a new function, and forget to redefine it, upon which time they get the old, remembered definition instead.

To practice these ideas, go to the empty editing buffer. First, type `M-x Lisp Mode` (where "M" is the Emacs Meta key). Normally you'd visit a file with the extension ".lisp" so this would be done automatically. In this buffer, create a definition for a simple procedure, by typing in the following (verbatim):

```
(defun square (x) (*x x))
```

Now, click on Definitions—Evaluate.

Type "(square 4)" into a Lisp Listener and hit return. The listener prints

```
CL-USER 10 > (square 4)
```

```
>>Error: The function *X is undefined

SQUARE
Original code: (NAMED-LAMBDA SQUARE (X) (BLOCK SQUARE (*X X)))
   Required arg 0 (X): 4
:C  0: Try evaluating #'*X again
:A  1: Return to level 0.
    2: Return to top loop level 0.
    3: Kill process "Listener 1"
    4: Dangerously Kill process "Listener 1" (without doing Unwind-Protect c

CL-USER 11 : 1 >
```

This is because we left out the space between the `*` and `x`. This is the textual lisp debugger. Each line, numbered 0 through 4, is called a *continuation*. Two of the lines, numbers 0 and 1, have a second reference, `:C` for "Continue" and `:A` for "Abort". These are the "typical" things you might want to do. The debugger is waiting for you to type a number (meaning, "do this number continuation") or the strings ":C" or ":A". You may also use many other debugger commands here. Most debugger commands begin with ":". Type ":?" to see a short list of common debugger commands.

Exit the debugger with ":A" (meaning "Abort", returning to interpreter level 0). Entering the debugger from the top level (level 0) will put you at level 1, entering the debugger at level 1 puts you at level 2, etc. etc. You can see the level printed at the end of the prompt; level 0 prints nothing (except your current package and the command line number).

Edit the definition to insert a space between `*` and `x`. Re-evaluate the definition and try evaluating `(square 4)` in the Listener again.

As a second method of evaluating expressions, try the following. Go to the Lisp Listener, and again type in:

```
(defun square (x) (*x x))
```

Place the cursor at the end of the line, and type return to evaluate this expression. Again try `(square 4)`. When it fails this time, exit the dubugger (remember how?). Then you should type `M-p` several times, until the definition of `square` that you typed in appears on the screen. Edit this definition to insert a space between `*` and `x` (Emacs commands work in the Listener, too), type return to evaluate the new expression, and use `M-p` to get back the expression `(square 4)`. Make a habit of using `M-p`, rather than going back and editing previous expressions in the Lisp Listener in place. That way, the buffer will contain an intact record of your work, line by line.

3. Another Debugging example

While you work, you will often need to debug programs. This section contains an exercise to acquaint you with some of the features of lisp to aid in debugging. Learning to use the debugging features will

save you much grief on later problem sets. Remember that `:?` gives a short command list; additional information about the debugger can be found by typing `:??` in the debugger.

Copy the code for this problem set from `$CLASSHOME/hw/prog0.lisp` into your own directory. Visit the file. This file contains definitions of the following three procedures `p1`, `p2` and `p3`:

```
(defun p1 (x y)
  (+ (p2 x y)
     (p3 x y)))

(defun p2 (z w)
  (* z w))

(defun p3 (a b)
  (+ (p2 a)
     (p2 b)))
```

Creating a TAGS file. Lispworks uses the standard GNU Emacs TAGS file format to store the location of function definitions. Type "M-x Create Tags Buffer" to create a TAGS buffer for all the files in the current directory. You can save the buffer to a file named TAGS, if you want. This will come in handy later, when we are working on large programs with many functions, divided among many different files.

Now, evaluate the buffer with the definitions of p1, p2, and p3. In the Lisp Listener, evaluate the expression `(p1 1 2)`. This should signal an error, with the message:

```
>>Error: Wrong number of arguments to P2

P2
Original code: (NAMED-LAMBDA P2 (Z W) (BLOCK P2 (* Z W)))
:A  0: Return to level 0.
    1: Return to top loop level 0.
    2: Kill process "Listener 1"
    3: Dangerously Kill process "Listener 1" (without doing Unwind-Protect cleanups

CL-USER 25 : 1 >
```

Don't panic. Beginners have a tendency, when they hit an error, to quickly type `:A`, often without even reading the error message. Then they stare at their code in the editor trying to see what the bug is. Indeed, the example here is simple enough so that you probably can find the bug by just reading the code. Instead, however, let's see how lisp can be coaxed into producing some helpful information about the error.

First of all, there is the error message itself. It tells you that the error was caused by a procedure being called with the wrong number of arguments. Unfortunately, the error message alone doesn't say where in the code the error occurred. In order to find out more, you need to use the debugger. Start the window debugger by selecting Debug—Debugger from the menus on the Lisp Listener. A new window, the Debugger Window, will pop up.

Using the Lisp window debugger

The debugger allows you to grovel around examining pieces of the execution in progress, in order to learn more about what may have caused the error. When you start the debugger, it will create a new window showing: a)The current error condition/exception, b) a backtrace of the stack, and c) variables at this point on the stack.

```
Condition:
 Wrong number of arguments to P2
Backtrace:
 P2
 P3
 P1
 EVAL
 LISPWORKS-TOOLS:LISTENER-TOP-LEVEL-FUNCTION
 ...
Variables:
 Required arg 0 NIL
```

You can select a "frame" in the BACKTRACE section by clicking on its line with the mouse or by using the ordinary cursor line-motion commands to move from line to line and typing a space. Notice that the bottom, information, buffer changes as the selected line changes.

The frames in the list in the backtrace buffer represent the steps in the evaluation of the expression. The functions below EVAL are part of the Lispworks Environment—the part that was running before you called your code (yes—most of the Common Lispworks lisp environment is itself written in common lisp) So "LISTENER-TOP-LEVEL-FUNCTION" is the main read-eval-print loop in the listener, and it called "EVAL" to evaluate what you typed in, `(p1 1 2)`. Click left once on "EVAL" and notice that the first argument is what you typed into the Listener.

So, starting at eval and working upwards, we see that P1 was called. If you click on P1, you see that it had two required args, X and Y, which were bound to 1 and 2 respectively. If we move up the stack, P1 must have called P3, and called it with two required args A and B, bound to 1 and 2. Finally, we see that it is function P3 that called P2 with the wrong number of arguments. If you previously did a "M-x Create Tags Buffer" in the Editor, then if you double click on the name P3, it will pop up the editor with the function P3 already highlighted.

You can now:

- fix the error in the editor (adding the missing arguments to the calls to p2 inside of the definition of p3)
- Re-evaluate the new p3 Definitions—Evaluate

- Bring up the debugger window, click on the P3 frame and then click on the button Restart Frame. This will reinvoke the NEW definition of p3 on the old arguments. This is often great if you find an error in the middle of an expensive computation and don't wish to start over from the beginning! Otherwise, you could also ABORT and try the `(p1 1 2)` form again from the Listener.

4. Exploring the system

The following exercises are meant to help you practice editing and debugging, and using on-line documentation.

Exercise 1: More debugging Change the definition of p3 back so that it has the argument bug in it. THIS TIME, try COMPILING the BUFFER (Buffers—Compile). Note that the compiler catches the wrong-number-of-args error. (There is nothing to turn in for this!)

Exercise 2: Still more debugging The example file also contains a buggy definition of a procedure meant to compute the factorials of positive integers: $n! = n \cdot (n-1) \cdot (n-2) \cdots 3 \cdot 2 \cdot 1$. Evaluate the expression `(fact 5)` (which is supposed to return 120). Find the bug and correct the definition. Compute the factorial of 243.

Exercises 3–11: Simple Lisp Practice Read Chapter 2, "Welcome to Lisp", from the Graham book (Class handout). Do all of the end-of-chapter questions (1-9).

What to hand in Turn in your answers and your fixed prog0.lisp file, either on printouts or hand-written.

APPENDIX. Using XEmacs and LEP

If you can't wean yourself from regular emacs, you can use LEP (the Lisp-Emacs-Protocol). The philosophy of the interface is to make the editor seem as though it were implemented in Lisp. If it were, then the editor would be able to manipulate objects not as text, but as first-class Lisp objects. Emacs would then be able to know more about programs and be able to extract information from them easily.

Because the Emacs-Lisp interface uses Emacs, which runs as a separate Unix process from Lisp, a Lisp-Editor protocol has been designed and implemented to make communicating information between Emacs and Lisp easier and more natural. A strong requirement of this communication and information exchange is that the user not be aware of it – it must happen in the background. This hidden communication is accomplished by using multiprocessing in Lisp and process filters in Emacs. The

latter is necessary because Emacs does not have multiprocessing. I've ony used it with XEmacs, but it should work with vanilla GNU Emacs too. You need the lines

```
(load "lep-user-init")
(add-hook 'lisp-mode-hook 'turn-on-font-lock)
(add-hook 'l-lisp-mode-hook 'turn-on-font-lock)
```

in your .emacs file, and you need to copy the file `~decker/emacs/lep-user-init.el` to your directory. This will give you a LEP menu on the XEmacs menubar.

Start/Find Common Lisp will start up or find the current running lisp process. This will create a buffer that acts exactly like a Lisp Listener.

M-C-x compiles the current defun.

Complete documentation is in (for EE/CIS)
`/usr/local/lang/lisp/common/lcl/5-0/manual/lep/lep-users-guide.ps`

If all else fails. You can run lisp inside of a simple XEmacs shell window. This gives you about 50% of the functionality of LEP for absolutely free.