1 Turning in Programming Assignments

When turning in a programming assignment to your instructor, you need to provide a copy of the procedures and a sample of their execution. This should be done using the program **script**, which is described in the following.

- First design and test your procedures and store them in a file. Suppose that this file is named **program-1.lisp**.
- At the Unix prompt, execute the command

  script <**script-file**>

  where <**script-file**> is the name of a new file that will contain the results to be handed in to your instructor. This starts a session during which everything that appears on your terminal screen will be copied into the file <**script-file**>. (**Note:** <**script-file**> **is NOT program-1.lisp. They must be two entirely different file names.**)
- If you have aliases established in your .login or .cshrc files, you will need to execute the command **source .login** or **source .cshrc** as appropriate, so that the aliases will be recognized during the script session.
- Print out the file containing your procedures, by executing the command

  cat program-1.lisp
- Enter lisp and load the file **program-1.lisp**. Then execute each procedure with appropriate data to show that your procedures work. If you are told to run a test program in file **test-1.lisp**, then load the file **test-1.lisp** into lisp and then execute its main function.
- Leave lisp. At the unix level, execute the command

  exit

  This terminates the script session. The file <**script-file**> now contains everything that appeared on your screen during the script session.
- Print the file <**script-file**> on the line printer and turn in the printout to your instructor.

2 Getting Started with Lucid Common Lisp

2.1 Setting Up and Invoking Lisp

On the Composers, you should have **/opt/bin** in your path, since the lisp image is located at **/opt/bin/lisp** on the Composers. **You should also copy** ~carberry/lisp-init.lisp to your top-level login directory. Your lisp-init.lisp file works like a .login file in that it is automatically loaded into lisp as you start the system up. It initially contains some function definitions you will need. You may add additional lines in order to customize lisp to your needs. Now start the Lisp interface from unix with **lisp** or **/opt/bin/lisp**. You will get some initialization messages and finally the lisp prompt: >. You are now in lisp and ready to go!

2.2 Loading Your Lisp Files

To load a file into lisp, type: **(load "<file-name>")**. If you don't have any syntax errors like unmatched parenthesis, you will get a message such as the following:

```
;;; Loading source file ‘‘fns.lisp’’
;;; Warning: File ‘‘fns.lisp’’ does not begin with IN-PACKAGE.
    Loading into package ‘‘USER’’
#P"/home/usra/01/user-number/fns.lisp"
>
```

and all functions defined in the file loaded will now be defined for Lisp. Don't worry about the warnings concerning the packages.

3 Editing and Debugging your Lisp Files

There are several ways of editing and debugging your Lisp Files. The first method, described in Section 3.1 entitled *Simple Editing and Debugging*, is easy to learn but does not provide the wealth of debugging aids offered by the Lisp Listener/Editor/Debugger described in Section 3.2 entitled *Lisp Listener/Editor/Debugger*. The first method will suffice for the purposes of this course, but if you are going to do extensive Lisp programming, then you may want to become an expert with the Lisp Listener/Editor/Debugger.

3.1 Simple Editing and Debugging

3.1.1 Emacs and Lisp

There are several ways to use emacs on your lisp files. The preferred option involves running lisp from the emacs editor. emacs has a "lisp mode" which is automatically invoked when a file with a .lisp extension is edited.

Some useful emacs commands:

Hold $\Longrightarrow$	Ctrl	Esc	Ctrl-Esc	Press $\Downarrow$
Move to beginning of	line	sentence	definition	a
Move to end of	line	sentence	definition	e
Delete to end of	line	sentence	expression	k
Move back one	character	word	expression	b
Move forward one	character	word	expression	f
Delete next	character	word	expression	d

To run lisp from emacs: This option is preferred. It allows you to have both lisp and your file on the screen at the same time (often helpful in debugging!), and to have the ability to do many emacs commands in the lisp window (such as editing a line, copying a line previously typed, etc.). In addition, emacs will highlight matching parenthesis on your typed input. Get into emacs with the file that your lisp functions are in. Split the screen (<Ctrl>-x 2). In one half of the screen get

a shell running (<Esc>-x shell). The shell is a regular unix window (except you can do things like scroll in it and execute a number of emacs commands). Invoke lisp from this window (by typing **/opt/bin/lisp**). Now you are able to go back and forth between editing your file and running lisp by simply going from one window to the other. Don't forget that you must re-load the file into lisp each time you change it (that is, the two windows are really independent of each other).

3.1.2 Vi and Lisp

Functions are also provided which allow you to use the vi editor from lisp. These are invoked by typing `(vi ``<file-name>'')` and `(vil ``<file-name>'')`. Both will put you into the vi editor editing the specified file. As above, **vil** causes the edited file to be loaded into lisp when vi is exited.

3.1.3 Some Helpful Functions

Several functions are available for finding out about functions in lisp:

`(grindef fn-name)` – prints the definition of the function.

`(pp fn-name)` – a macro that has been defined for you to do the same as grindef.

If you find that the symbol # replaces pieces of the printed function, try evaluating the lisp form **(setf *print-level* nil)** and then printing the function again. Evaluating the lisp form **(setf *print-length* nil)** will also cause lisp to print lists of any length. Note that both *print-level* and *print-length* are by default set differently for debugger levels.

`(describe 'obj-name)` – prints a description of the object.

Several functions are useful in debugging:

`(break "message")` – When the break is evaluated, the argument string giving a message is output, execution is temporarily suspended, and a message is printed about resuming execution. (Typically, typing **:C** resumes execution and typing **:A** gets you back to the top level in Lisp.) At this point, objects and expressions can be evaluated before execution is resumed.

`(trace fn-list)` **and** `(untrace fn-list)` – Invoking trace with one or more function names (symbols) causes the functions named to be traced. Henceforth, whenever such a function is invoked, information about the call, the arguments passed, and the eventually returned values, if any, will be printed. Calling trace with no arguments will return a list of functions currently being traced.

Invoking untrace with one or more function names will cause those functions not to be traced any more. Calling untrace with no arguments will cause all currently traced functions to be no longer traced.

Consider the following recursive function defined in lisp:

```
(defun fact (n)
;computes n*(n-1)*...*1
  (cond ((equal n 1) n)
        ((* n (fact (1- n))))))
```

Once the file is loaded into lisp, we can trace it and watch its execution. Each recursive call will be shown with the arguments, embedded calls will be indented, and the results of the calls will be shown indented evenly with the initial call.

```
> (trace fact)
(FACT)
> (fact 5)
1 Enter FACT 5
| 2 Enter FACT 4
|   3 Enter FACT 3
|   | 4 Enter FACT 2
|   |   5 Enter FACT 1
|   |   5 Exit FACT 1
|   | 4 Exit FACT 2
|   3 Exit FACT 6
| 2 Exit FACT 24
1 Exit FACT 120
120
```

`(step <form>)` – This evaluates form and returns what form returns. However, the user is allowed to interactively "single-step" through the evaluation of form, at least through those evaluation steps that are performed interpretively. Once you get into "step", system help is available by typing **?**

(dribble '<myfile>) – Sends a record of the input/output interaction to the named file. The primary purpose of this is to create a readable record of an interactive session. `(dribble)` terminates the recording of input and output and closes the dribble file.

(shell "<any-function-call>") – this function allows you to call any unix command from lisp. An example is **(shell "cat debug.lisp")**. This is very useful for things like directory listings etc..

3.1.4 Debugging Programs

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 that aid in debugging. Learning to use the debugging features will save you much grief on later problem sets.

Copy the code for this problem set from `~carberry/debug.lisp` into your own directory. Visit (open) the file in the emacs editor. Load the file into lisp via the command **(load "debug.lisp")**. (Remember that your window is split in half, with one-half containing an emacs editor and the other half running lisp.) This file contains definitions of the following procedures `p1`, `p2`, `p3` and `multply`.:

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

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

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

(defun multply (list1)
   (break "entering list1")
   (cond ((null list1) 1)
         (t (* (car list1) (multply (cdr list1))))))
```

Now evalute 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.

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. So one thing that we can try is *stepping* through the evaluation of `(p1 1 2)`. To do this, evaluate the list expression `(step (p1 1 2))`. You will see that the response is

```
  (P1 1 2) ->
```

indicating that lisp has entered the evaluation of the expression `(p1 1 2)`. Now type `?` to get a list of selections that you can make while in the *stepper*. We'll repeatedly type `:n` to move one step further along in evaluation of `(p1 1 2)`. The results are shown below. (Note the repeated typing of `:n` which I've moved out to the right so that you can see it).

```
  (P1 1 2) -> :n
    (FUNCTION P1) ->                                :n
    #<Interpreted-Function P1 FE141BE>
    1 = 1
    2 = 2
    (BLOCK P1 (+ (P2 X Y) (P3 X Y))) ->             :n
```

```
      (+ (P2 X Y) (P3 X Y)) ->                          :n
        (FUNCTION +) ->                                 :n
        #<Compiled-Function + 3F1936>
        (P2 X Y) ->                                     :n
          (FUNCTION P2) ->                              :n
          #<Interpreted-Function P2 FE14776>
          X = 1
          Y = 2
          (BLOCK P2 (* Z W)) ->                         :n
            (* Z W) ->                                  :n
              (FUNCTION *) ->                           :n
              #<Compiled-Function * 3F195E>
              Z = 1
              W = 2
            2
          2
        2
        (P3 X Y) ->                                      :n
          (FUNCTION P3) ->                               :n
          #<Interpreted-Function P3 FE14D4E>
          X = 1
          Y = 2
          (BLOCK P3 (+ (P2 A) (P2 B))) ->                :n
            (+ (P2 A) (P2 B)) ->                         :n
              (FUNCTION +) ->                            :n
              #<Compiled-Function + 3F1936>
              (P2 A) ->                                  :n
                (FUNCTION P2) ->                         :n
                #<Interpreted-Function P2 FE14776>
                A = 1
>>Error: Wrong number of arguments to P2

P2
Original code: (NAMED-LAMBDA P2 (Z W) (BLOCK P2 (* Z W)))
:A  0: Return to level 1.
    1: Return to level 0.
```

Note that each time you move a step further along in evaluation of the function, the stepper shows you any new functions that are being called, bindings for parameters, and the result returned by the function call. So in the above, we see that the problem occurred when `p2` was being evaluated and was given the wrong number of arguments. Looking further back, we see that this occurred during evaluation of `p3` via a form in `p1`. So we look at the definition of `p3` and immediately find the incorrect specification `(p2 a)` — and if we are smart, we note that `p2` is also invoked a second time with the wrong number of arguments in `p3`. So we move back to the editor, correct the definition of `p3`, save the corrected version, load the corrected file back into lisp, and again execute `(p1 1 2)`. Whoops — we forgot to turn off the stepper. To terminate stepping, type `:x` which finishes evaluating the form but turns the stepper off.

Let's examine the options that you are given when an error occurs. Typically, it will look something like the following:

```
P1
Original code: (NAMED-LAMBDA P1 (X Y) (BLOCK P1 (+ # #)))
```

```
:A  0: Return to level 3.
    1: Return to level 2.
    2: Return to level 1.
    3: Return to level 0.
```

At this point, you can type `:A` to abort or return to one level up, or we could type one of the numbers. For example, if we type `3` at this point, we will return to the lisp top level. Generally, once you have found your error, you will want to return to the level where you called the function that produced the error, or you will want to return to the top level and start the evaluation all over again.

You can also insert **breaks** in your code. Note that a break has already been entered in the definition of the function `multply` in file `debug.lisp`. `Multply` takes a non-empty list of integers as its argument and returns the product of them. Evaluate the form `(Multply '(4 8 2))`. You will get a response that shows you have encountered a break; it also tells you the binding of the parameter `list1` at this point — note that one of your continuation options is `:C` which says to return from the break. Typing `:C` will cause evaluation to continue until another break is encountered. When you are in a break, you can enter any lisp form and it will be evaluated as if it had been encountered at this point in the evaluation of the function that has been broken. For example, if you type the form `(length list1)`, the response will be `3` since there are three elements in `list1`. Let us now type `:C` to return from the break and continue evaluation. Notice that you encounter another break when the function `Multply` is called again, and this time the binding for `list1` is `(8 2)`. Type `:C` again, and you will encounter another break when `Multply` is entered with `list1` bound to `(2)`. Continue typing `:C` until the function has been fully evaluated and `64` is returned as the value of `(Multply '(4 8 2))`

You can also trace functions with the trace command.

3.2 Lisp Listener/Editor/Debugger

This section describes the Lisp Listener, Editor, and Debugger. You can skip the rest of this handout.

Enter lisp by typing **/opt/bin/lisp** from the Unix shell. After everything's loaded, start the X-windows interface via the lisp form `(env:start-environment)`.

After dismissing the LCL "Common Lispworks" environment dialog box, the Lisp monitor and Lispworks Podium windows appear. The Lisp monitor has four buttons labelled

Break to tty
Interrupt Lisp
Quit GcMonitor
Idle

It shows you the Garbage Collection status and allows asyncronous garbage collection (not something to worry about in this course). The bottom button 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 :-). The Podium allows you to start any other Lispworks Tool (like an Editor or Lisp Listener). It has five menu selectors at the top, labelled *Works*, *File*, *Mail*, *Tools*, and *Help*.

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 a browser. If you already have that 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; it is brought up by selecting *Listener* from the *Tools* menu. 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.

3.2.1 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))
```

3.2.2 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 one will compile everything, and then evalute/interpret the few functions that one is 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** selected from the **File** menu, 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 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. In this buffer, create a definition for a simple procedure, by typing in the following (verbatim):

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

Now, highlight the definition you just typed (using the mouse) and click on **Expression—Evaluate Region** or put the curser in the definition and click on **Definitions—Evaluate**. (You will get some feedback from the editor telling you that it is evaluating the definition.)

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 current process "Listener 1"

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 3, 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 **:?** in the Listener window 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. . 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` (meta-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.

3.2.3 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 that 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 `~carberry/debug.lisp` into your own directory. Visit (open) the file. This file contains definitions of the following procedures `p1`, `p2`, `p3`, and `multply`:

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

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

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

(defun multply (list1)
   (break "entering list1")
```

```
   (cond ((null list1) 1)
         (t (* (car list1) (multply (cdr list1))))))
```

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"

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.

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.

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.