Lecture Outline and Reading Assignments

In the following chart, the readings refer to section numbers in the Abelson and Sussman text. Remember, the reading should be done *before* the week indicated.

week	Monday	Wednesday	Friday	reading
1	**holiday**	functional programming 1/19	1/21	1.1
2	1/24	higher-order procedures UI (Kay) 1/26	1/28	1.3
3	1/31	UI (Kay) recursion and iteration 2/2	2/4	1.2.1–4
		Project 1 due Monday, 2/7		
4	2/7	data abstraction, sequences calculator 2/9	2/11	2.1, 2.2.1
		Midterm Wednesday 2/16, 7–9pm		
5	2/14	hierarchical data interpreter 2/16	2/18	2.2.2–3, 2.3.1,3
		Project 2 due Tuesday, 2/22		
6	**holiday**	generic operators 2/23	2/25	2.4–2.5.2
		GCD: 5pm Monday 2/28, MT1, Proj1, HW1–5		
7	2/28	object-oriented programming 3/2	3/4	OOP (reader)
		Midterm Wednesday 3/9, 7–9pm		
8	3/7	assignment, state, environments 3/9	3/11	3.1, 3.2
		Project 3a due Monday, 3/14		
9	3/14	mutable data vectors 3/16	3/18	3.3.1–3
		spring break		
		Project 3b due Monday, 3/28		
		GCD: 5pm Monday 3/28, MT2, Proj2, HW6–8		
10	3/28	client/server concurrency 3/30	4/1	3.4
11	4/4	streams Therac 4/6	4/8	3.5.1–3, 3.5.5, Therac
		Midterm Wednesday 4/13, 7–9pm		
12	4/11	metacircular eval. mapreduce 4/13	4/15	4.1.1–6
13	4/18	mapreduce analyzing, lazy evals. 4/20	4/22	4.1.7, 4.2
		Project 4a due Monday, 4/25		
		GCD: 5pm Monday 4/25, MT3, Proj3, HW9–12		
14	4/25	logic programming review 4/27	4/29	4.4.1–3
		Project 4b due Monday, 5/2		
		GCD: 5pm Monday 5/16, Proj4a, HW13–14		
		GCD: 5pm Tuesday, 5/10, Proj4b		
		Final Tuesday, 5/11, 11:30am–2:30pm		

Note: *GCD* = Grading Complaint Deadline.

CS 61A Spring 2011 Week 1

Topic: Functional programming

Monday, 1/17 is a holiday!

Lectures: Wednesday 1/19, Friday 1/21

Reading: Abelson & Sussman, Section 1.1 (pages 1–31)

Note: With the obvious exception of this first week, you should do each week's reading *before* the Monday lecture. So also start now on next week's reading, Abelson & Sussman, Section 1.3

Homework due noon Monday, 1/24:

People who've taken CS 3: Don't use the CS 3 higher-order procedures such as `every` in these problems; use recursion.

1. Do exercise 1.6, page 25. This is an essay question; you needn't hand in any computer printout, unless you think the grader can't read your handwriting. If you had trouble understanding the square root program in the book, explain instead what will happen if you use `new-if` instead of `if` in the `pigl` Pig Latin procedure.

2. Write a procedure `squares` that takes a sentence of numbers as its argument and returns a sentence of the squares of the numbers:

```
> (squares '(2 3 4 5))
(4 9 16 25)
```

3. Write a procedure `switch` that takes a sentence as its argument and returns a sentence in which every instance of the words `I` or `me` is replaced by `you`, while every instance of `you` is replaced by `me` except at the beginning of the sentence, where it's replaced by `I`. (Don't worry about capitalization of letters.) Example:

```
> (switch '(You told me that I should wake you up))
(i told you that you should wake me up)
```

4. Write a predicate `ordered?` that takes a sentence of numbers as its argument and returns a true value if the numbers are in ascending order, or a false value otherwise.

5. Write a procedure `ends-e` that takes a sentence as its argument and returns a sentence containing only those words of the argument whose last letter is E:

```
> (ends-e '(please put the salami above the blue elephant))
(please the above the blue)
```

Continued on next page.

Week 1 continued...

6. Most versions of Lisp provide `and` and `or` procedures like the ones on page 19. In principle there is no reason why these can't be ordinary procedures, but some versions of Lisp make them special forms. Suppose, for example, we evaluate

```
(or (= x 0) (= y 0) (= z 0))
```

If `or` is an ordinary procedure, all three argument expressions will be evaluated before `or` is invoked. But if the variable `x` has the value `0`, we know that the entire expression has to be true regardless of the values of `y` and `z`. A Lisp interpreter in which `or` is a special form can evaluate the arguments one by one until either a true one is found or it runs out of arguments.

Your mission is to devise a test that will tell you whether Scheme's `and` and `or` are special forms or ordinary functions. This is a somewhat tricky problem, but it'll get you thinking about the evaluation process more deeply than you otherwise might.

Why might it be advantageous for an interpreter to treat `or` as a special form and evaluate its arguments one at a time? Can you think of reasons why it might be advantageous to treat `or` as an ordinary function?

Unix feature of the week: `man`

Emacs feature of the week: `C-g`, `M-x apropos`

There will be a "feature of the week" each week. These first features come first because they are the ones that you use to find out about the other ones: Each provides documentation of a Unix or Emacs feature. This week, type `man man` as a shell command to see the Unix manual page on the `man` program. Then, in Emacs, type `M-x` (that's meta-`X`, or `ESC X` if you prefer) `describe-function` followed by the Return or Enter key, then `apropos` to see how the `apropos` command works. If you want to know about a command by its keystroke form (such as `C-g`) because you don't know its long name (such as `keyboard-quit`), you can say `M-x describe-key` then `C-g`.

You aren't going to be tested on these system features, but it'll make the rest of your life a *lot* easier if you learn about them.

CS 61A Spring 2011 Week 1 Lab
Wednesday 1/19 afternoon, Thursday 1/20, or Friday 1/21 morning
Try to get as much done as possible, but don't panic if you don't finish everything.

1. (15 minutes.) Start the Emacs editor, either by typing `emacs` in your main window or by selecting it from the alt-middle mouse menu. (Your TA will show you how to do this.) From the `Help` menu, select the Emacs tutorial. You need not complete the entire tutorial at the first session, but you should do so eventually.

(Parts 2–4: 15 minutes.)

2. Use Emacs to create a file called `pigl.scm` in your directory containing the Pig Latin program shown below:

```
(define (pigl wd)
  (if (pl-done? wd)
      (word wd 'ay)
      (pigl (word (bf wd) (first wd)))))

(define (pl-done? wd)
  (vowel? (first wd)))

(define (vowel? letter)
  (member? letter '(a e i o u)))
```

Make sure you are editing a file whose name ends in `.scm`, so that Emacs will know to indent your code correctly!

3. Now run Scheme by typing meta-S in your Emacs window. You are going to create a transcript of a session using the file you just created, like this:

```
(transcript-on "lab1")       ; This starts the transcript file.
(load "pigl.scm")            ; This reads in the file you created earlier.
(pigl 'scheme)               ; Try out your program.
                             ; Feel free to try more test cases here!
(trace pigl)                 ; This is a debugging aid. Watch what happens
(pigl 'scheme)               ; when you run a traced procedure.
(transcript-off)
(exit)
```

4. Use `lpr` to print your transcript file.

Continued on next page.

Week 1 lab continued:

5. (15 minutes.) In the shell, type the command

```
cp ~cs61a/lib/plural.scm .
```

(Note the period at the end of the line!) This will copy a file from the class library to your own directory. Then, using emacs to edit the file, modify the procedure so that it correctly handles cases like `(plural 'boy)`.

6. (20 minutes.) Define a procedure that takes three numbers as arguments and returns the sum of the squares of the two larger numbers.

7. (15 minutes.) Write a procedure `dupls-removed` that, given a sentence as input, returns the result of removing duplicate words from the sentence. It should work this way:

```
> (dupls-removed '(a b c a e d e b))
(c a d e b)
> (dupls-removed '(a b c))
(a b c)
> (dupls-removed '(a a a a b a a))
(b a)
```