CS 61A Week 1

Topic: Functional programming

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:

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 Week 2

Topic: Higher-order procedures

Reading: Abelson & Sussman, Section 1.3

Note that we are skipping 1.2; we'll get to it later. Because of this, never mind for now the stuff about iterative versus recursive processes in 1.3 and in the exercises from that section.

Don't panic if you have trouble with the half-interval example on pp. 67–68; you can just skip it. Try to read and understand everything else.

Homework:

1. Abelson & Sussman, exercises 1.31(a), 1.32(a), 1.33, 1.40, 1.41, 1.43, 1.46

(Pay attention to footnote 51; you'll need to know the ideas in these exercises later in the semester.)

2. Last week you wrote procedures `squares`, that squared each number in its argument sentence, and saw `pigl-sent`, that `pigl`ed each word in its argument sentence. Generalize this pattern to create a higher-order procedure called `every` that applies an *arbitrary* procedure, given as an argument, to each word of an argument sentence. This procedure is used as follows:

```
> (every square '(1 2 3 4))
(1 4 9 16)
> (every first '(nowhere man))
(n m)
```

3. Our Scheme library provides versions of the `every` function from the last exercise and the `keep` function shown in lecture. Get familiar with these by trying examples such as the following:

```
(every (lambda (letter) (word letter letter)) 'purple)
(every (lambda (number) (if (even? number) (word number number) number))
       '(781 5 76 909 24))
(keep even? '(781 5 76 909 24))
(keep (lambda (letter) (member? letter 'aeiou)) 'bookkeeper)
(keep (lambda (letter) (member? letter 'aeiou)) 'syzygy)
(keep (lambda (letter) (member? letter 'aeiou)) '(purple syzygy))
(keep (lambda (wd) (member? 'e wd)) '(purple syzygy))
```

Continued on next page.

Week 2 continued...

Extra for experts:

In principle, we could build a version of Scheme with no primitives except `lambda`. Everything else can be defined in terms of `lambda`, although it's not done that way in practice because it would be so painful. But we can get a sense of the flavor of such a language by eliminating one feature at a time from Scheme to see how to work around it.

In this problem we explore a Scheme without `define`. We can give things names by using argument binding, as `let` does, so instead of

```
(define (sumsq a b)
  (define (square x) (* x x))
  (+ (square a) (square b)))

(sumsq 3 4)
```

we can say

```
((lambda (a b)
   ((lambda (square)
      (+ (square a) (square b)))
    (lambda (x) (* x x))))
 3 4)
```

This works fine as long as we don't want to use *recursive* procedures. But we can't replace

```
(define (fact n)
  (if (= n 0)
      1
      (* n (fact (- n 1)))))

(fact 5)
```

by

```
((lambda (n)
   (if ...))
 5)
```

because what do we do about the invocation of `fact` inside the body?

Your task is to find a way to express the `fact` procedure in a Scheme without any way to define global names.

Unix feature of the week: `pine`, `mail`, `firefox`

Emacs feature of the week: `M-x info`, `C-x u` (undo)

CS 61A Week 3

Topic: Recursion and iteration

Reading: Abelson & Sussman, Section 1.2 through 1.2.4 (pages 31–47)

Homework:

Note: Programming project 1 will also be due next week.

1. Abelson & Sussman, exercises 1.16, 1.35, 1.37, 1.38

2. A "perfect number" is defined as a number equal to the sum of all its factors less than itself. For example, the first perfect number is 6, because its factors are 1, 2, 3, and 6, and 1+2+3=6. The second perfect number is 28, because 1+2+4+7+14=28. What is the third perfect number? Write a procedure `(next-perf n)` that tests numbers starting with `n` and continuing with `n+1`, `n+2`, etc. until a perfect number is found. Then you can evaluate `(next-perf 29)` to solve the problem. Hint: you'll need a `sum-of-factors` subprocedure.

[Note: If you run this program when the system is heavily loaded, it may take half an hour to compute the answer! Try tracing helper procedures to make sure your program is on track, or start by computing `(next-perf 1)` and see if you get 6.]

3. Explain the effect of interchanging the order in which the base cases in the `cc` procedure on page 41 of Abelson and Sussman are checked. That is, describe completely the set of arguments for which the original `cc` procedure would return a different value or behave differently from a `cc` procedure coded as given below, and explain how the returned values would differ.

```
(define (cc amount kinds-of-coins)
  (cond
    ((or (< amount 0) (= kinds-of-coins 0)) 0)
    ((= amount 0) 1)
    (else ... ) ) )        ; as in the original version
```

4. Give an algebraic formula relating the values of the parameters `b, n, counter,` and `product` of the `expt` and `exp-iter` procedures given near the top of page 45 of Abelson and Sussman. (The kind of answer we're looking for is "the sum of b, n, and counter times product is always equal to 37.")

Continued on next page.

Week 3 continued...

Extra for experts:

1. The partitions of a positive integer are the different ways to break the integer into pieces. The number 5 has seven partitions:

```
5          (one piece)
4, 1          (two pieces)
3, 2          (two pieces)
3, 1, 1          (three pieces)
2, 2, 1          (three pieces)
2, 1, 1, 1          (four pieces)
1, 1, 1, 1, 1          (five pieces)
```

The order of the pieces doesn't matter, so the partition 2, 3 is the same as the partition 3, 2 and thus isn't counted twice. 0 has one partition.

Write a procedure `number-of-partitions` that computes the number of partitions of its nonnegative integer argument.

2. Compare the `number-of-partitions` procedure with the `count-change` procedure by completing the following statement:

`Counting partitions is like making change, where the coins are ...`

3. (Much harder!) Now write it to generate an iterative process; every recursive call must be a tail call.

Unix feature of the week: `mkdir`, `cd`, `pwd`, `ls`

Emacs feature of the week: `C-M-f`, `C-M-b`, `C-M-n`, `C-M-p` (move around Scheme code)

CS 61A Week 4

Topic: Data abstraction

Reading: Abelson & Sussman, Sections 2.1 and 2.2.1 (pages 79–106)

Homework:

Abelson & Sussman, exercises 2.7, 2.8, 2.10, 2.12, 2.17, 2.20, 2.22, 2.23

(Note: "Spans zero" means that one bound is $\leq$ zero and the other is $\geq$ zero!)

• Write a procedure `substitute` that takes three arguments: a list, an *old* word, and a *new* word. It should return a copy of the list, but with every occurrence of the old word replaced by the new word, even in sublists. For example:

```
> (substitute '((lead guitar) (bass guitar) (rhythm guitar) drums)
              'guitar 'axe)
((lead axe) (bass axe) (rhythm axe) drums)
```

• Now write `substitute2` that takes a list, a *list* of old words, and a *list* of new words; the last two lists should be the same length. It should return a copy of the first argument, but with each word that occurs in the second argument replaced by the corresponding word of the third argument:

```
> (substitute2 '((4 calling birds) (3 french hens) (2 turtle doves))
               '(1 2 3 4) '(one two three four))
((four calling birds) (three french hens) (two turtle doves))
```

Note: The first midterm is next week.

Extra for experts:

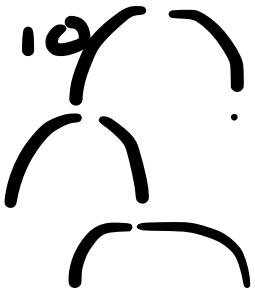
Write the procedure `cxr-function` that takes as its argument a word starting with `c`, ending with `r`, and having a string of letters `a` and/or `d` in between, such as `cdddadaadar`. It should return the corresponding function.

Abelson & Sussman, exercise 2.6. Besides addition, invent multiplication and exponentiation of nonnegative integers. If you're *really* enthusiastic, see if you can invent subtraction. (Remember, the rule of this game is that you have only `lambda` as a starting point.) Read `~cs61a/lib/church-hint` for some suggestions.

Unix feature of the week: `rm`, `mv`, `cp`, `rmdir`, `ln -s`

Emacs feature of the week: `M-%` (find and replace text)

CS 61A Week 5



Topic: Hierarchical data

Midterm Wednesday , 7–9pm.

Reading: Abelson & Sussman, Section 2.2.2–2.2.3, 2.3.1, 2.3.3

Homework:

• Abelson & Sussman, exercises 2.24, 2.26, 2.29, 2.30, 2.31, 2.32, 2.36, 2.37, 2.38, 2.54.

Some of these exercises are harder than they look; don't give up in frustration if your early attempts fail.

• Extend the calculator program from lecture to include words as data, providing the operations `first`, `butfirst`, `last`, `butlast`, and `word`. Unlike Scheme, **your calculator should treat words as self-evaluating expressions** except when seen as the operator of a compound expression. That is, it should work like these examples:

```
calc: foo
foo
calc: (first foo)
f
calc: (first (butfirst hello))
e
```

The program is in `~cs61a/lib/calc.scm`

Note: Programming project 2 is also due next week. It consists of all the exercises in Section 2.2.4 of the text. You can't actually draw anything until you finish the project! To begin, copy the file `~cs61a/lib/picture.scm` to your directory. To draw pictures, once you've completed the exercises:

```
> (cs)
> (ht)
> (===your-painter=== full-frame)
```

For example:

```
> (wave full-frame)
> ((square-limit wave 3) full-frame)
```

Continued on next page.

Week 5 continued...

Extra for experts:

Read section 2.3.4 and do exercises 2.67–2.72.

Programming by example: In some programming systems, instead of writing an algorithm, you give examples of how you'd like the program to behave, and the language figures out the algorithm itself:

```
> (define pairup (regroup '((1 2) (3 4) ...)))
> (pairup '(the rain in spain stays mainly on the plain))
((the rain) (in spain) (stays mainly) (on the))
```

Write `regroup`. Read `~cs61a/lib/regroup.problem` for details.

Unix feature of the week: `head`, `tail`, `more`, `cat`

Emacs feature of the week: `M-x search-forward-regexp`, `M-x query-replace-regexp`

CS 61A Week 6

Topic: Generic Operators

Reading: Abelson & Sussman, Sections 2.4 through 2.5.2 (pages 169–200)

Homework:

Abelson & Sussman, exercises 2.74, 2.75, 2.76, 2.77, 2.79, 2.80, 2.81, 2.83

Note: Some of these are thought-exercises; you needn't actually run any Scheme programs for them! (Some don't ask you to write procedures at all; others ask for modifications to a program that isn't online.)

• If you haven't already finished this week's lab exercises that involve the `scheme-1` interpreter, do it now. Then write a `map` primitive for `scheme-1` (call it `map-1` so you and Scheme don't get confused about which is which) that works correctly for all mapped procedures.

• Modify the `scheme-1` interpreter to add the `let` special form. Hint: Like a procedure call, `let` will have to use `substitute` to replace certain variables with their values. Don't forget to evaluate the expressions that provide those values!

Extra for experts:

Another approach to the problem of type-handling is *type inference*. If, for instance, a procedure includes the expression `(+ n k)`, one can infer that `n` and `k` have numeric values. Similarly, the expression `(f a b)` indicates that the value of `f` is a procedure.

Write a procedure called `inferred-types` that, given a definition of a Scheme procedure as argument, returns a list of information about the parameters of the procedure. The information list should contain one element per parameter; each element should be a two-element list whose first element is the parameter name and whose second element is a word indicating the type inferred for the parameter. Possible types are listed on the next page.

Continued on next page.

Week 6 continued...

`?`	(the type can't be inferred)
`procedure`	(the parameter appeared as the first word in an unquoted expression or as the first argument of `map` or `every`)
`number`	(the parameter appeared as an argument of `+`, `-`, `max`, or `min`)
`list`	(the parameter appeared as an argument of `append` or as the second argument of `map` or `member`)
`sentence-or-word`	(the parameter appeared as an argument of `first, butfirst, sentence,` or `member?`, or as the second argument of `every`)
`x`	(conflicting types were inferred)

You should assume for this problem that the body of the procedure to be examined does not contain any occurrences of `if` or `cond`, although it may contain arbitrarily nested and quoted expressions. (A more ambitious inference procedure both would examine a more comprehensive set of procedures and could infer conditions like "nonempty list".)

Here's an example of what your inference procedure should return.

```
(inferred-types
  '(define (foo a b c d e f)
    (f (append (a b) c '(b c)) (+ 5 d) (sentence (first e) f)) ) )
```

should return

```
((a procedure) (b ?) (c list) (d number)
 (e sentence-or-word) (f x))
```

If you're *really* ambitious, you could maintain a database of inferred argument types and use it when a procedure you've seen is invoked by another procedure you're examining!

Unix feature of the week: `du`, `df`, `quota`

Emacs feature of the week: `M-q` (format paragraphs), `C-M-q` (format Scheme code)

CS 61A Week 7

Topic: Object-oriented programming

Reading:

Read "Object-Oriented Programming—Above-the-line view" (in course reader).

Homework:

Note: To use the OOP language you must first

```
(load "~cs61a/lib/obj.scm")
```

before using `define-class`, etc.

1. For a statistical project you need to compute lots of random numbers in various ranges. (Recall that `(random 10)` returns a random number between 0 and 9.) Also, you need to keep track of *how many* random numbers are computed in each range. You decide to use object-oriented programming. Objects of the class `random-generator` will accept two messages. The message `number` means "give me a random number in your range" while `count` means "how many `number` requests have you had?" The class has an instantiation argument that specifies the range of random numbers for this object, so

```
(define r10 (instantiate random-generator 10))
```

will create an object such that `(ask r10 'number)` will return a random number between 0 and 9, while `(ask r10 'count)` will return the number of random numbers `r10` has created.

2. Define the class `coke-machine`. The instantiation arguments for a `coke-machine` are the number of Cokes that can fit in the machine and the price (in cents) of a Coke:

```
(define my-machine (instantiate coke-machine 80 70))
```

creates a machine that can hold 80 Cokes and will sell them for 70 cents each. The machine is initially empty. `Coke-machine` objects must accept the following messages:

Continued on next page.

Week 7 continued...

`(ask my-machine 'deposit 25)` means deposit 25 cents. You can deposit several coins and the machine should remember the total.

`(ask my-machine 'coke)` means push the button for a Coke. This either gives a `Not enough money` or `Machine empty` error message or returns the amount of change you get.

`(ask my-machine 'fill 60)` means add 60 Cokes to the machine.

Here's an example:

```
(ask my-machine 'fill 60)
(ask my-machine 'deposit 25)
(ask my-machine 'coke)
NOT ENOUGH MONEY
(ask my-machine 'deposit 25)      ;; Now there's 50 cents in there.
(ask my-machine 'deposit 25)      ;; Now there's 75 cents.
(ask my-machine 'coke)
5                                 ;; return val is 5 cents change.
```

You may assume that the machine has an infinite supply of change.

3. We are going to use objects to represent decks of cards. You are given the list `ordered-deck` containing 52 cards in standard order:

```
(define ordered-deck '(AH 2H 3H ... QH KH AS 2S ... QC KC))
```

You are also given a function to shuffle the elements of a list:

```
(define (shuffle deck)
  (if (null? deck)
      '()
      (let ((card (nth (random (length deck)) deck)))
        (cons card (shuffle (remove card deck))) )))
```

A deck object responds to two messages: `deal` and `empty?`. It responds to `deal` by returning the top card of the deck, after removing that card from the deck; if the deck is empty, it responds to `deal` by returning `()`. It responds to `empty?` by returning `#t` or `#f`, according to whether all cards have been dealt.

Write a class definition for `deck`. When instantiated, a deck object should contain a shuffled deck of 52 cards.

Continued on next page.

Week 7 continued...

4. We want to promote politeness among our objects. Write a class `miss-manners` that takes an object as its instantiation argument. The new `miss-manners` object should accept only one message, namely `please`. The arguments to the `please` message should be, first, a message understood by the original object, and second, an argument to that message. **(Assume that all messages to the original object require exactly one additional argument.)** Here is an example using the `person` class from the upcoming adventure game project:

```
> (define BH (instantiate person 'Brian BH-office))

> (ask BH 'go 'down)
BRIAN MOVED FROM BH-OFFICE TO SODA

> (define fussy-BH (instantiate miss-manners BH))

> (ask fussy-BH 'go 'east)
ERROR: NO METHOD GO

> (ask fussy-BH 'please 'go 'east)
BRIAN MOVED FROM SODA TO PSL
```

Extra for experts:

The technique of multiple inheritance is described on pages 9 and 10 of "Object-Oriented Programming – Above-the-line view". That section discusses the problem of resolving ambiguous patterns of inheritance, and mentions in particular that it might be better to choose a method inherited directly from a second-choice parent over one inherited from a first-choice grandparent.

Devise an example of such a situation. Describe the inheritance hierarchy of your example, listing the methods that each class provides. Also describe why it would be more appropriate in this example for an object to inherit a given method from its second-choice parent rather than its first-choice grandparent.

Unix feature of the week: `|` (pipes in the shell)

Emacs feature of the week: `M-x spell-buffer`

Note: The second midterm exam is next week.

CS 61A Week 8

Topic: Assignment, state, environments

Midterm Wednesday , 7–9pm.

Reading: Abelson & Sussman, Section 3.1, 3.2

Also read "Object-Oriented Programming—Below-the-line view" (in course reader).

Homework:

Abelson & Sussman, exercises 3.3, 3.4, 3.7, 3.8, 3.10, 3.11

Note: Part I of programming project 3 is also due next week.

Extra for experts:

The purpose of the environment model is to represent the scope of variables; when you see an `x` in a program, which variable `x` does it mean?

Another way to solve this problem would be to *rename* all the local variables so that there are never two variables with the same name. Write a procedure `unique-rename` that takes a (quoted) lambda expression as its argument, and returns an equivalent lambda expression with the variables renamed to be unique:

```
> (unique-rename '(lambda (x) (lambda (y) (x (lambda (x) (y x))))))
(lambda (g1) (lambda (g2) (g1 (lambda (g3) (g2 g3)))))
```

Note that the original expression had two variables named `x`, and in the returned expression it's clear from the names which is which. You'll need a modified counter object to generate the unique names.

You may assume that there are no `quote`, `let`, or `define` expressions, so that every symbol is a variable reference, and variables are created only by `lambda`.

Describe how you'd use `unique-rename` to allow the evaluation of Scheme programs with only a single (global) frame.

Unix feature of the week: `foreach`, `grep`, `find`

Emacs feature of the week: `C-t` (transpose), `M-c`, `M-u`, `M-l` (change case)

CS 61A Week 9

Topic: Mutable data, vectors

Reading: Abelson & Sussman, Section 3.3.1–3

(If you are a hardware type you might enjoy reading 3.3.4 even though it isn't required.)

Homework:

Abelson & Sussman, exercises 3.16, 3.17, 3.21, 3.25, 3.27

You don't need to draw the environment diagram for exercise 3.27; use a trace to provide the requested explanations. Treat the table procedures `lookup` and `insert!` as primitive; i.e. don't trace the procedures they call. Also, assume that those procedures work in constant time. We're interested to know about the number of times `memo-fib` is invoked.

Vector questions: In all these exercises, don't use a list as an intermediate value. (That is, don't convert the vectors to lists!)

1. Write `vector-append`, which takes two vectors as arguments and returns a new vector containing the elements of both arguments, analogous to `append` for lists.

2. Write `vector-filter`, which takes a predicate function and a vector as arguments, and returns a new vector containing only those elements of the argument vector for which the predicate returns true. The new vector should be exactly big enough for the chosen elements. Compare the running time of your program to this version:

```
(define (vector-filter pred vec)
  (list->vector (filter pred (vector->list vec))))
```

3. Sorting a vector.

(a) Write `bubble-sort!`, which takes a vector of numbers and rearranges them to be in increasing order. (You'll modify the argument vector; don't create a new one.) It uses the following algorithm:

[1] Go through the array, looking at two adjacent elements at a time, starting with elements 0 and 1. If the earlier element is larger than the later element, swap them. Then look at the next overlapping pair (0 and 1, then 1 and 2, etc.).

[2] Recursively bubble-sort all but the last element (which is now the largest element).

[3] Stop when you have only one element to sort.

(b) Prove that this algorithm really does sort the vector. Hint: Prove the parenthetical claim in step [2].

(c) What is the order of growth of the running time of this algorithm?

Note: Part II of programming project 3 is also due next week.

Continued on next page.

Week 9 continued...

Extra for experts:

1. Abelson and Sussman, exercises 3.19 and 3.23.

Exercise 3.19 is incredibly hard but if you get it, you'll feel great about yourself. You'll need to look at some of the other exercises you skipped in this section.

Exercise 3.23 isn't quite so hard, but be careful about the O(1)—i.e. *constant*—time requirement.

2. Write the procedure `cxr-name`. Its argument will be a function made by composing `car`s and `cdr`s. It should return the appropriate name for that function:

```
> (cxr-name (lambda (x) (cadr (cddar (cadar x)))))
CADDDAADAR
```

Unix feature of the week: `alias`, `unalias`

Emacs feature of the week: `C-x 4` (split window)

CS 61A Week 10

Topic: client/server, concurrency

Reading: Abelson & Sussman, Section 3.4

Homework:

These exercises use the Instant Message program, found in the following files:

```
~cs61a/lib/im-client.scm
~cs61a/lib/im-server.scm
~cs61a/lib/im-common.scm
```

1. Invent the capability to send a message to a list of clients as well as to a single client. Do this entirely in the client program, so what actually goes to the server is multiple requests.

2. Invent the capability to broadcast a message to every client. Do this by inventing a `broadcast` command that the server understands.

3. Could #1 have been done with the server doing part of the work? Could #2 have been done entirely in the client code? Compare the virtues of the two approaches.

4. Invent the capability of refusing messages from specific people. The sender of a refused message should be notified of the refusal. Decide whether to do it entirely in the client or with the server's cooperation, and explain why.

5. Why is the 3-way handshake necessary when connecting to the server?

• Abelson & Sussman, exercises 3.38, 3.39, 3.40, 3.41, 3.42, 3.44, 3.46, 3.48

Extra for experts:

Using the Instant Message program as a starting point, write a mail server and client. The mail server should maintain a database of messages for all users. (This can just be a list; don't worry about efficient lookup.) The client should be able to do the following:

```
(mail username message)
(get-mail)
```

`Get-mail` should return a list of messages, which should be deleted from the server.

If you want, you can improve this in several ways: Make deletion from the server be explicitly requested by the client, invent a subject header (another argument to `mail`) and have the client show just headers in `get-mail` and provide another command to read the text of a specific message, and so on.

• Read Section 3.3.5 and do exercises 3.33–3.37.

Unix feature of the week: `&`, `^Z`, `fg`, `bg`, `jobs`, `kill`

Emacs feature of the week: `M-x abbrev-mode`, `M-x add-mode-abbrev`

CS 61A Week 11

Topic: Streams

Reading: Abelson & Sussman, Section 3.5.1–3, 3.5.5

Homework:

1. Abelson & Sussman, exercises 3.50, 3.51, 3.52, 3.53, 3.54, 3.55, 3.56, 3.64, 3.66, 3.68

2. Write and test two functions to manipulate nonnegative proper fractions. The first function, `fract-stream`, will take as its argument a list of two nonnegative integers, the numerator and the denominator, in which the numerator is less than the denominator. It will return an infinite stream of decimal digits representing the decimal expansion of the fraction. The second function, `approximation`, will take two arguments: a fraction stream and a nonnegative integer `numdigits`. It will return a list (not a stream) containing the first `numdigits` digits of the decimal expansion.
`(fract-stream '(1 7))` should return the stream representing the decimal expansion of $\frac{1}{7}$, which is 0.142857142857142857...
`(stream-car (fract-stream '(1 7)))` should return 1.
`(stream-car (stream-cdr (stream-cdr (fract-stream '(1 7)))))` should return 2.
`(approximation (fract-stream '(1 7)) 4)` should return `(1 4 2 8)`.
`(approximation (fract-stream '(1 2)) 4)` should return `(5 0 0 0)`.

Note: Next week is the third midterm.

Extra for experts:

1. Do exercises 3.59–3.62.

2. Consider this procedure:

```
(define (hanoi-stream n)
  (if (= n 0)
      the-empty-stream
      (stream-append (hanoi-stream (- n 1))
                     (cons-stream n (hanoi-stream (- n 1))))))
```

It generates finite streams; here are the first few values:

```
(hanoi-stream 1)    (1)
(hanoi-stream 2)    (1 2 1)
(hanoi-stream 3)    (1 2 1 3 1 2 1)
(hanoi-stream 4)    (1 2 1 3 1 2 1 4 1 2 1 3 1 2 1)
```

Notice that each of these starts with the same values as the one above it, followed by some more values. There is no reason why this pattern can't be continued to generate an infinite stream whose first $2^n - 1$ elements are `(hanoi-stream n)`. Generate this stream.

Unix feature of the week: `diff`, `wc`

Emacs feature of the week: `M-a`, `M-e`, `M-{`, `M-}`, `M-<`, `M->` (move around buffer)

CS 61A Week 12

Topic: Metacircular evaluator

Midterm Wednesday , 7–9pm.

Reading:

Abelson & Sussman, 4.1.1–6

MapReduce paper in course reader.
(metacircular evaluator: `~cs61a/lib/mceval.scm`)

Homework:

Some students have complained that this week's homework is very time-consuming. Accordingly, with some reluctance, I've marked a few exercises as optional; these are the ones to leave out if you're really pressed for time. But it's much better if you do all of them! The optional ones have * next to them.

1. A&S exercises 4.3, 4.6, 4.7*, 4.10*, 4.11*, 4.13, 4.14, 4.15

2*. Modify the metacircular evaluator to allow *type-checking* of arguments to procedures. Here is how the feature should work. When a new procedure is defined, a formal parameter can be either a symbol as usual or else a list of two elements. In this case, the second element is a symbol, the name of the formal parameter. The first element is an expression whose value is a predicate function that the argument must satisfy. That function should return `#t` if the argument is valid. For example, here is a procedure `foo` that has type-checked parameters `num` and `list`:

```
> (define (foo (integer? num) ((lambda (x) (not (null? x))) list))
    (list-ref list num))
FOO
> (foo 3 '(a b c d e))
D
> (foo 3.5 '(a b c d e))
Error: wrong argument type -- 3.5
> (foo 2 '())
Error: wrong argument type -- ()
```

In this example we define a procedure `foo` with two formal parameters, named `num` and `list`. When `foo` is invoked, the evaluator will check to see that the first actual argument is an integer and that the second actual argument is not empty. The expression whose value is the desired predicate function should be evaluated with respect to `foo`'s defining environment. (Hint: Think about `extend-environment`.)

Extra for experts:

Abelson & Sussman, exercises 4.16 through 4.21

Unix feature of the week: `echo`, `set`, `setenv`, `printenv`

Emacs feature of the week: `M-!` (run shell command)

CS 61A Week 13

Topic: Analyzing evaluator, MapReduce

Reading: Therac paper in course reader.
analyzing evaluator: `~cs61a/lib/analyze.scm`

Homework:

A&S exercises 4.22, 4.23, 4.24

MapReduce exercises:

1(a). Google uses MapReduce for a variety of search-engine-related tasks involving processing large data sets. An example of such a task is building an inverted word index. For each word in a given set of documents, we want to generate a key-value pair, where the key is the word and the value is all documents in which the word appears. Write a procedure that takes a directory name as argument and returns the inverted index stream.

1(b). The list above will include words like "a," "so," "the," etc. Suppose now that we only want "important words." We can use word length as a measure of importance (a real search engine would use a more complex heuristic). Write a procedure that returns an index where all the words are N or more letters long, taking the filename and N as arguments.

2. Google also manages the GMail e-mail service, and they would like to filter out as many spammers as possible. In this problem you will implement a simple spam filtering idea with `mapreduce` so that it can be efficiently applied to the multitudes of e-mail messages in GMail.

We want to produce a "blacklist" of e-mail addresses that are spamming GMail users. Spam messages are sent to many addresses at once, and so a spam message can be identified by having one of the most commonly-used subject lines. If an address sends many messages with the most common subject lines, it is likely a spammer address. We would like to find the top ten addresses that have sent the most messages with frequently-recurring subject lines.

The input data is a collection of e-mail records in the file `/sample-emails`. (These are simulated messages, not actual GMail data!) Each e-mail record is a list with the format

```
(from-address to-address subject body)
```

where each element is a double-quoted string. The mapper function will be applied to each e-mail record. An small example set of e-mail records is below.

```
("cs61a-tb" "cs61a-tc" "mapreduce" "mapreduce is great! lucky students!")
("bot1337" "cs61a-ta" "free ipod now!" "buy herbal ipod enhancer!")
("bot1338" "cs61c-tf" "free ipod now!" "buy herbal ipod enhancer!")
...
```

Problem continues on next page...

Week 13 continued:

2(a). Our first step is to produce a table with subject lines as keys and counts of occurrences as values. Identify the intermediate key-value pairs and write the mapper and reducer functions.

2(b). From our tabulation of subject line occurrences in 2(a), we want to find the most common subject lines in the table. We can perform a sort by using the fact that MapReduce sorts by intermediate keys into the reducer groups. Identify the intermediate key-value pairs and write the mapper and reducer functions.

2(c). Finally, assuming you've moved the ten most common subject lines into a list, we want to make a table with from-addresses as keys and counts of e-mails sent with common subject lines as values. You don't need to sort the table, since the procedures would be identical to those in 2(b). Identify the intermediate key-value pairs and write the mapper and reducer functions.

3. Sometimes either the map or reduce operation is trivial, such as an identity function. Is MapReduce still the best way to handle those cases, or would it be better to provide separate `map` and `groupreduce` functions as we did for the non-parallel toy version? Why?

4. Could you use MapReduce to perform a parallelized Sieve of Eratosthenes? If so, describe the process briefly. If not, why not? (Don't try to solve this problem by quizmanship; there's a case to be made for both answers!)

Note: Part I of project 4 is also due next week.

Unix feature of the week: `sort`, `cut`, `paste`, `join`

Emacs feature of the week: `C-x C-o`, `M-\` (delete blank space)

CS 61A Week 14

Topic: lazy evaluator, nondeterministic evaluator

Reading: Abelson & Sussman, Section 4.2, 4.3

A version of the lazy evaluator is online in `~cs61a/lib/lazy.scm`

A version of the nondeterministic evaluator is online in `~cs61a/lib/ambeval.scm`

A modified version of the nondeterministic evaluator, based on the vanilla metacircular evaluator rather than on the analyzing evaluator, is online in `~cs61a/lib/vambeval.scm`

Homework:

There are a lot of exercises this week, because we are covering a lot of material. These are all great exercises, from which you'll learn a lot, but for those who find the homework time-consuming, we're dividing them into two categories, crucial and less-crucial. Again, I recommend all of them!

crucial: 4.25, 4.26, 4.28, 4.42, 4.45, 4.49, 4.50, 4.52

less crucial: 4.30, 4.32, 4.33, 4.36, 4.47, 4.48

Note: Part II of the fourth programming project is also due next week.

Extra for experts:

Exercise 4.31 in Abelson and Sussman. This exercise doesn't require great brilliance, but it's a lot of work and involves a lot of debugging of details. On the other hand, completing this exercise will teach you a lot about the evaluator.

Fix the `handle-infix` procedure from project 4 to handle infix precedence for arithmetic operators properly. That is, multiplications and divisions should be done before additions and subtractions. Comparison operators like `=` come last of all.

Unix feature of the week: `!`, `history`

Emacs feature of the week: `C-x (`, `C-x )`, `C-x e` (keyboard macros)

CS 61A Week 15

Topic: Logic programming

Reading: Abelson & Sussman, Section 4.4.1–3

We are not assigning section 4.4.4, which discusses the implementation of the query system in detail. Feel free to read it just for interest; besides deepening your understanding of logic programming, it provides an example of using streams in a large project.

Homework:

Abelson & Sussman, exercises 4.56, 4.57, 4.58, 4.65

For all problems that involve writing queries or rules, test your solutions. To run the query system and load in the sample data:

```
% stk
> (load "~cs61a/lib/query.scm")
> (initialize-data-base microshaft-data-base)
> (query-driver-loop)
```

You're now in the query system's interpreter. To add an assertion:

```
(assert! (foo bar))
```

To add a rule:

```
(assert! (rule (foo) (bar)))
```

Anything else is a query.

Extra for experts:

The lecture notes for this week describe rules that allow inference of the `reverse` relation in one direction, i.e.,

```
;;; Query input:
(forward-reverse (a b c) ?what)

;;; Query results:
(FORWARD-REVERSE (A B C) (C B A))
```

Continued on next page.

Week 15 continued...

```
;;; Query input:
(forward-reverse ?what (a b c))

;;; Query results:
```
... infinite loop

or

```
;;; Query input:
(backward-reverse ?what (a b c))

;;; Query results:
(BACKWARD-REVERSE (C B A) (A B C))

;;; Query input:
(backward-reverse (a b c) ?what)

;;; Query results:
```
... infinite loop

Define rules that allow inference of the `reverse` relation in both directions, to produce the following dialog:

```
;;; Query input:
(reverse ?what (a b c))

;;; Query results:
(REVERSE (C B A) (A B C))

;;; Query input:
(reverse (a b c) ?what)

;;; Query results:
(REVERSE (A B C) (C B A))
```

Unix feature of the week: `perl`, `awk`, `sed`

Emacs feature of the week: `M-x shell-command-on-region`