CS 61A   Lecture Notes   Week 4

Topic: Data abstraction, sequences

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

Note: The first midterm is next week.

• Big ideas: data abstraction, abstraction barrier.

If we are dealing with some particular type of data, we want to talk about it in terms of its *meaning*, not in terms of how it happens to be represented in the computer.

Example: Here is a function that computes the total point score of a hand of playing cards. (This simplified function ignores the problem of cards whose rank-name isn't a number.)

```
;;;;;                        In file cs61a/lectures/2.1/total.scm
(define (total hand)
  (if (empty? hand)
      0
      (+ (butlast (last hand))
         (total (butlast hand)) )))


> (total '(3h 10c 4d))
17
```

This function calls `butlast` in two places. What do those two invocations mean? Compare it with a modified version:

```
;;;;;                        In file cs61a/lectures/2.1/total.scm
(define (total hand)
  (if (empty? hand)
      0
      (+ (rank (one-card hand))
         (total (remaining-cards hand)) )))

(define rank butlast)
(define suit last)

(define one-card last)
(define remaining-cards butlast)
```

This is more work to type in, but the result is much more readable. If for some reason we wanted to modify the program to add up the cards left to right instead of right to left, we'd have trouble editing the original version because we wouldn't know which `butlast` to change. In the new version it's easy to keep track of which function does what.

The auxiliary functions like `rank` are called *selectors* because they select one component of a multi-part datum.

Actually we're *violating* the data abstraction when we type in a hand of cards as `'(3h 10c 4d)` because that assumes we know how the cards are represented—namely, as words combining the rank number with a one-letter suit. If we want to be thorough about hiding the representation, we need *constructor* functions as well as the selectors:

```
;;;;;                         In file cs61a/lectures/2.1/total.scm
(define (make-card rank suit)
  (word rank (first suit)) )

(define make-hand se)


> (total (make-hand (make-card 3 'heart)
                    (make-card 10 'club)
                    (make-card 4 'diamond) ))
17
```

Once we're using data abstraction we can change the implementation of the data type without affecting the programs that *use* that data type. This means we can change how we represent a card, for example, without rewriting `total`:

```
;;;;;                         In file cs61a/lectures/2.1/total.scm
(define (make-card rank suit)
  (cond ((equal? suit 'heart) rank)
        ((equal? suit 'spade) (+ rank 13))
        ((equal? suit 'diamond) (+ rank 26))
        ((equal? suit 'club) (+ rank 39))
        (else (error "say what?")) ))

(define (rank card)
  (remainder card 13))

(define (suit card)
  (nth (quotient card 13) '(heart spade diamond club)))
```

We have changed the internal *representation* so that a card is now just a number between 1 and 52 (why? maybe we're programming in FORTRAN) but we haven't changed the *behavior* of the program at all. We still call `total` the same way.

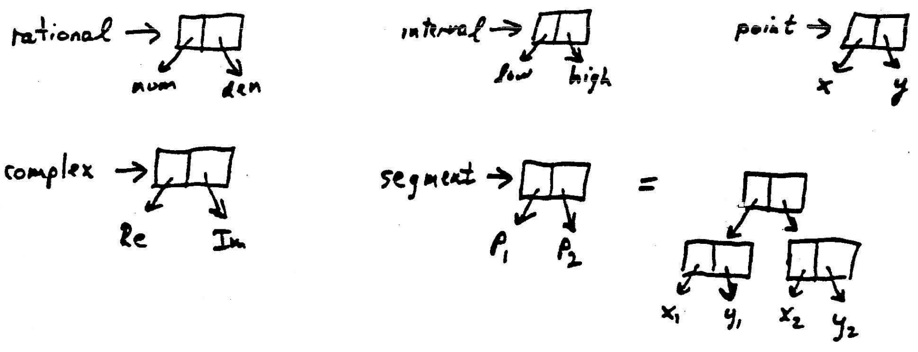
Data abstraction is a really good idea because it helps keep you from getting confused when you're dealing with lots of data types, but don't get religious about it. For example, we have invented the *sentence* data type for this course. We have provided symmetric selectors `first` and `last`, and symmetric selectors `butfirst` and `butlast`. You can write programs using sentences without knowing how they're implemented. But it turns out that because of the way they *are* implemented, `first` and `butfirst` take $\Theta(1)$ time, while `last` and `butlast` take $\Theta(N)$ time. If you know that, your programs will be faster.

• Pairs.

To represent data types that have component parts (like the rank and suit of a card), you have to have some way to *aggregate* information. Many languages have the idea of an *array* that groups some number of elements. In Lisp the most basic aggregation unit is the *pair*—two things combined to form a bigger thing. If you want more than two parts you can hook a bunch of pairs together; we'll discuss this more below.

The constructor for pairs is CONS; the selectors are CAR and CDR.

The book uses pairs to represent many different abstract data types: rational numbers (numerator and denominator), complex numbers (real and imaginary parts), points (x and y coordinates), intervals (low and high bounds), and line segments (two endpoints). Notice that in the case of line segments we think of the representation as *one pair* containing two points, not as three pairs containing four numbers. (That's what it means to respect a data abstraction.)



Note: What's the difference between these two:

```
(define (make-rat num den) (cons num den))
(define make-rat cons)
```

They are both equally good ways to implement a constructor for an abstract data type. The second way has a slight speed advantage (one fewer function call) but the first way has a debugging advantage because you can trace `make-rat` without tracing all invocations of `cons`.

• Data aggregation doesn't have to be primitive.

In most languages the data aggregation mechanism (the array or whatever) seems to be a necessary part of the core language, not something you could implement as a user of the language. But if we have first-class functions we can use a function to represent a pair:

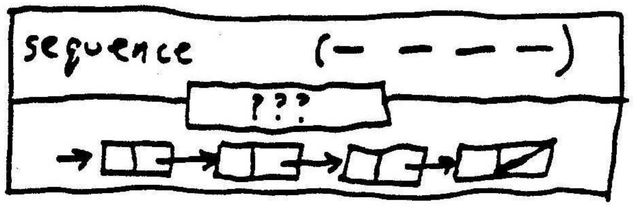
```
;;;;;                          In file cs61a/lectures/2.1/cons.scm
(define (cons x y)
  (lambda (which)
    (cond ((equal? which 'car) x)
          ((equal? which 'cdr) y)
          (else (error "Bad message to CONS" message)) )))


(define (car pair)
  (pair 'car))

(define (cdr pair)
  (pair 'cdr))
```

This is like the version in the book except that they use 0 and 1 as the *messages* because they haven't introduced quoted words yet. This version makes it a little clearer what the argument named `which` means.

The point is that we can satisfy ourselves that this version of `cons`, `car`, and `cdr` works in the sense that if we construct a pair with this `cons` we can extract its two components with this `car` and `cdr`. If that's true, we don't need to have pairs built into the language! All we need is `lambda` and we can implement the rest ourselves. (It isn't really done this way, in real life, for efficiency reasons, but it's neat that it could be.)

• Big idea: abstract data type *sequence* (or *list*).

We want to represent an ordered sequence of things. (They can be any kind of things.) We *implement* sequences using pairs, with each `car` pointing to an element and each `cdr` pointing to the next pair.



What should the constructors and selectors be? The most obvious thing is to have a constructor `list` that takes any number of arguments and returns a list of those arguments, and a selector `nth` that takes a number and a list as arguments, returning the *n*th element of the list.

Scheme does provide those, but it often turns out to be more useful to select from a list differently, with a selector for the first element and a selector for all the rest of the elements (i.e., a smaller list). This helps us write recursive functions such as the mapping and filtering ones we saw for sentences earlier.

Since we are implementing lists using pairs, we ought to have specially-named constructors and selectors for lists, just like for rational numbers:

```
(define adjoin cons)
(define first car)
(define rest cdr)
```

Many Lisp systems do in fact provide `first` and `rest` as synonyms for `car` and `cdr`, but the fact is that this particular data abstraction is commonly violated; we just use the names `car`, `cdr`, and `cons` to talk about lists.

This abstract data type has a special status in the Scheme interpreter itself, because lists are read and printed using a special notation. If Scheme knew only about pairs, and not about lists, then when we construct the list `(1 2 3)` it would print as `(1 . (2 .(3 . ())))` instead.

• List constructors.

Sentences have a very simple structure, so there's just one constructor for them. Lists are more complicated, and have three constructors:

`List` is the simplest to understand. It takes any number of arguments, each of which can be anything, and returns a list containing those arguments as its elements:

```
> (list '(a list) 'word 87 #t)
((a list) word 87 #t)
```

This seems very straightforward, but in practice it's not the most useful constructor, because it can be used only when you know exactly how many elements you want in the list. Most list programming deals with arbitrary sequences, which could be of any length. `List` is useful when you use a fixed-length list to represent an abstract data type, such as a point in three-dimensional space:

```
(define (point x y z)
  (list x y z))
```

`Cons` adds one new element at the front of an existing list:

```
> (cons '(new element) '(the old list))
((new element) the old list)
```

(Of course, `cons` really just takes two arguments and makes a pair containing them, but if you're using that pair as the head of a list, then the effect of `cons` in terms of the list abstraction is to add one new element.) This may seem too specific and arbitrary to be useful, but in fact `cons` is the most commonly used list constructor, because adding one new element is exactly what you want to do in a recursive transformation of a list:

```
(define (map fn seq)
  (if (null? seq)
      '()
      (CONS (fn (car seq))
            (map fn (cdr seq)))))
```

`Append` is used to combine two or more lists in a way that "flattens" some of the structure of the result: It returns a list whose elements are *the elements of* the arguments, which must be lists:

```
> (append '(one list) '(and another list))
(one list and another list)
```

It's most useful when combining results from multiple recursive calls, each of which returns a subset of the overall answer; you want to take the union of those sets, and that's what `append` does.

• Lists vs. sentences.

We started out the semester using an abstract data type called *sentence* that looks a lot like a list. What's the difference, and why did we do it that way?

Our goal was to allow you to create aggregates of words without having to think about the structure of their internal representation (i.e., about pairs). We do this by deciding that the elements of a sentence must be words (not sublists), and enforcing that by giving you the constructor `sentence` that creates only sentences.

Example: One of the homework problems this week asks you to reverse a list. You'll see that this is a little tricky using `cons`, `car`, and `cdr` as the problem asks, but it's easy for sentences:

```
(define (reverse sent)
  (if (empty? sent)
      '()
      (se (reverse (bf sent)) (first sent)) ))
```

To give you a better idea about what a sentence is, here's a version of the constructor function:

```
;;;;;                         In file cs61a/lectures/2.2/sentence.scm
(define (se a b)
  (cond ((word? a) (se (list a) b))
        ((word? b) (se a (list b)))
        (else (append a b)) ))

(define (word? x)
  (or (symbol? x) (number? x)) )
```

`Se` is a lot like `append`, except that the latter behaves oddly if given words as arguments. `Se` can accept words or sentences as arguments.

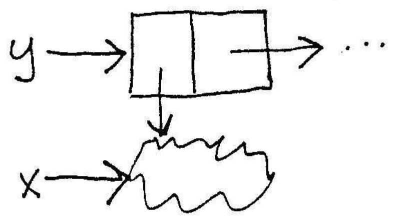
• Box and pointer diagrams.

Here are a few details that people sometimes get wrong about them:

1. An arrow can't point to half of a pair. If an arrowhead touches a pair, it's pointing to the entire pair, and it doesn't matter exactly where the arrowhead touches the rectangle. If you see something like

```
(define x (car y))
```

where `y` is a pair, the arrow for `x` should point to *the thing that the* `car` *of* `y` *points to*, not to the left half of the `y` rectangle.



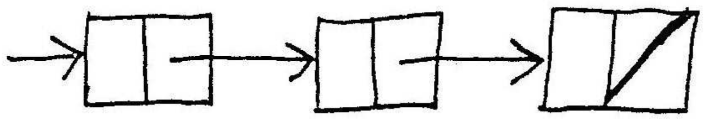
2. The direction of arrows (up, down, left, right) is irrelevant. You can draw them however you want to make the arrangement of pairs neat. That's why it's crucial not to forget the arrowheads!

3. There must be a top-level arrow to show where the structure you're representing begins.

How do you draw a diagram for a complicated list? Take this example:

```
((a b) c (d (e f)))
```

You begin by asking yourself how many elements the list has. In this case it has three elements: first `(a b)`, then `c`, then the rest. Therefore you should draw a three-pair *backbone*: three pairs with the `cdr` of one pointing to the next one. (The final `cdr` is null.)



Only after you've drawn the backbone should you worry about making the `car`s of your three pairs point to the three elements of the top-level list.

• MapReduce

In the past, functional programming, and higher-order functions in particular, have been considered esoteric and unimportant by most programmers. But the advent of highly parallel computation is changing that, because functional programming has the very useful property that the different pieces of a program don't interfere with each other, so it doesn't matter in what order they are invoked. Later this semester, when we have more sophisticated functional mechanisms to work with, we'll be examining one famous example of functional programming at work: the `MapReduce` programming paradigm developed by Google that uses higher-order functions to allow a programmer to process large amount of data in parallel on many computers.

Much of the computing done at Google consists of relatively simple algorithms applied to massive amounts of data, such as the entire World Wide Web. It's routine for them to use clusters consisting of many thousands of processors, all running the same program, with a distributed filesystem that gives each processor local access to part of the data.

In 2003 some very clever people at Google noticed that the majority of these computations could be viewed as a `map` of some function over the data followed by an `accumulate` (they use the name `reduce`, which is a synonym for this function) to collect the results. Although each program was conceptually simple, a lot of programmer effort was required to manage the parallelism; every programmer had to worry about things like how to recover from a processor failure (virtually certain to happen when a large computation uses thousands of machines) during the computation. They wrote a library procedure named `MapReduce` that basically takes two functions as arguments, a one-argument function for the `map` part and a two-argument function for the `accumulate` part. (The actual implementation is more complicated, but this is the essence of it.) Thus, only the implementors of `MapReduce` itself had to worry about the parallelism, and application programmers just have to write the two function arguments.*

`MapReduce` is a little more complicated than just

```
(define (mapreduce mapper reducer base-case data)    ; Nope.
  (accumulate reducer base-case (map mapper data)))
```

because of the parallelism. The input data comes in pieces; several computers run the `map` part in parallel, and each of them produces some output. These intermediate results are rearranged into groups, and each computer does a `reduce` of part of the data. The final result isn't one big list, but separate output files for each reducing process.

To make this a little more specific, today we'll see a toy version of the algorithm that just handles small data lists in one processor.

Data pass through the program in the form of *key-value pairs:*

```
(define make-kv-pair cons)
(define kv-key car)
(define kv-value cdr)
```

A list of key-value pairs is called an *association list* or *a-list* for short. We'll see a-lists in many contexts other than `MapReduce`. Conceptually, the input to `MapReduce` is an a-list, although in practice there are several a-lists, each on a different processor.

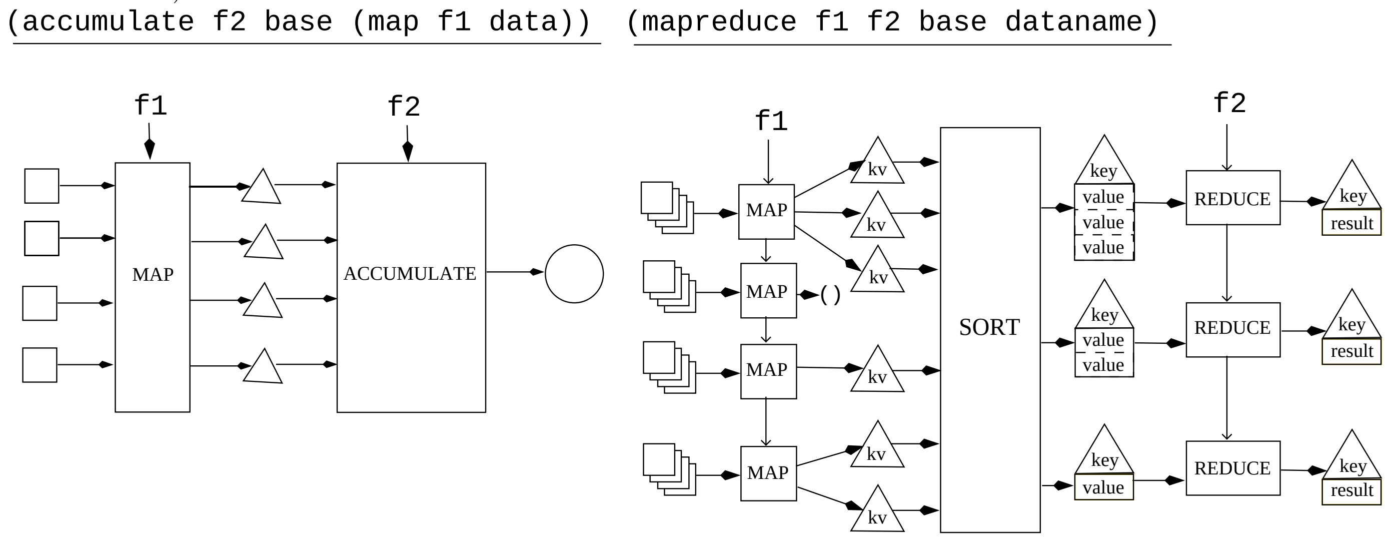
Any computation in `MapReduce` involves two function arguments: the *mapper* and the *reducer.* (Note: The Google `MapReduce` paper in the course reader says "the `map` function" to mean the function that the user writes, the one that's applied to each datum; this usage is confusing since everyone else uses "`map`" to mean

* We are indebted to Google for funding both the development of software and curriculum for this piece of the course and the cost of the cluster of parallel processors you'll use in the lab later. This should not be taken as an endorsement of Google's contemptible business practices, in which their income depends on massive violations of their users' privacy.

the higher-order function that controls the invocation of the user's function, so we're calling the latter the *mapper*:

```
(map mapper data)
```

Similarly, we'll use `reduce` to refer to the higher-order function, and `reducer` to mean the user's accumulation function.)



(accumulate f2 base (map f1 data))
(mapreduce f1 f2 base dataname)
f1
f2
MAP
ACCUMULATE
f1
MAP
MAP
()
MAP
MAP
kv
SORT
key
value
REDUCE
result
f2

The argument to the mapper is always one kv-pair. Keys are typically used to keep track of where the data came from. For example, if the input consists of a bunch of Web pages, the keys might be their URLs. Another example we'll be using is Project Gutenberg, an online collection of public-domain books; there the key would be the name of a book (more precisely, the filename of the file containing that book). In most uses of a-lists, there will only be one kv-pair with a given key, but that's not true here; for example, each line of text in a book or Web page might be a datum, and every line in the input will have the same key.

The value returned by the mapper must be *a list of* kv-pairs. The reason it's a list instead of a single kv-pair, as you might expect, is twofold. First, a single input may be split into smaller pieces; for example, a line of text might be mapped into a separate kv-pair for each word in the line. Second, the mapper might return an empty list, if this particular kv-pair shouldn't contribute to the result at all; thus, the mapper might also be viewed as a filterer. The mapper is not required to use the same key in its output kv-pairs that it gets in its input kv-pair.

Since `map` handles each datum independently of all the others, the fact that many `map`s are running in parallel doesn't affect the result; we can model the entire process with a single `map` invocation. That's not the case with the `reduce` part, because the data are being combined, so it matters which data end up on which machine. This is where the keys are most important. Between the mapping and the reduction is an intermediate step in which the kv-pairs are sorted based on the keys, and all pairs with the same key are reduced together. Therefore, the reducer doesn't need to look at keys at all; its two arguments are a value and the result of the partial accumulation of values already done. In many cases, just as in the accumulations we've seen earlier, the reducer will be a simple associative and commutative operation such as `+`.

The overall result is an a-list, in which each key occurs only once, and the value paired with that key is the result of the `reduce` invocation that handled that key. The keys are guaranteed to be in order. (This is the result of the 61A version of `MapReduce`; the real Google software has a more complicated interface because each computer in the cluster collects its own `reduce` results, and there are many options for how the reduction tasks are distributed among the processors. You'll learn more details in 61B.) So in today's single-processor simulation, instead of talking about `reduce` we'll use a higher order function called `groupreduce` that takes a *list of a-lists* as argument, with each sublist having kv-pairs with the same key, does a separate reduction for each sublist, and returns an a-list of the results. So a complete `MapReduce` operation works roughly like this:

```
(define (mapreduce mapper reducer base-case data) ; handwavy approximation
  (groupreduce reducer base-case
               (sort-into-buckets (map mapper data))))

(define (groupreduce reducer base-case buckets)
  (map (lambda (subset) (make-kv-pair
                         (kv-key (car subset))
                         (reduce reducer base-case (map kv-value subset))))
       buckets))
```

As a first example, we'll take some grades from various exams and add up the grades for each student. This example doesn't require `map`. Here's the raw data:

```
(define mt1 '((cs61a-xc . 27) (cs61a-ya . 40) (cs61a-xw . 35)
              (cs61a-xd . 38) (cs61a-yb . 29) (cs61a-xf . 32)))
(define mt2 '((cs61a-yc . 32) (cs61a-xc . 25) (cs61a-xb . 40)
              (cs61a-xw . 27) (cs61a-yb . 30) (cs61a-ya . 40)))
(define mt3 '((cs61a-xb . 32) (cs61a-xk . 34) (cs61a-yb . 30)
              (cs61a-ya . 40) (cs61a-xc . 28) (cs61a-xf . 33)))
```

Each midterm in this toy problem corresponds to the output of a parallel `map` operation in a real problem.

First we combine these into one list, and use that as input to the `sortintobuckets` procedure:

```
> (sort-into-buckets (append mt1 mt2 mt3))
(((cs61a-xb . 40) (cs61a-xb . 32))
 ((cs61a-xc . 27) (cs61a-xc . 25) (cs61a-xc . 28))
 ((cs61a-xd . 38))
 ((cs61a-xf . 32) (cs61a-xf . 33))
 ((cs61a-xk . 34))
 ((cs61a-xw . 35) (cs61a-xw . 27))
 ((cs61a-ya . 40) (cs61a-ya . 40) (cs61a-ya . 40))
 ((cs61a-yb . 29) (cs61a-yb . 30) (cs61a-yb . 30))
 ((cs61a-yc . 32)))
```

In the real parallel context, instead of the `append`, each `map` process would sort its own results into the right buckets, so that too would happen in parallel.

Now we can use `groupreduce` to add up the scores in each bucket separately:

```
> (groupreduce + 0 (sort-into-buckets (append mt1 mt2 mt3)))
((cs61a-xb . 72) (cs61a-xc . 80) (cs61a-xd . 38) (cs61a-xf . 65)
 (cs61a-xk . 34) (cs61a-xw . 62) (cs61a-ya . 120) (cs61a-yb . 89)
 (cs61a-yc . 32))
```

Note that the returned list has the keys in sorted order. This is a consequence of the sorting done by `sort-into-buckets`, and also, in the real parallel `mapreduce`, a consequence of the order in which keys are assigned to processors (the "partitioning function" discussed in the `MapReduce` paper).

Similarly, we could ask *how many* midterms each student took:

```
> (groupreduce (lambda (new old) (+ 1 old)) 0
               (sort-into-buckets (append mt1 mt2 mt3)))
((cs61a-xb . 2) (cs61a-xc . 3) (cs61a-xd . 1) (cs61a-xf . 2) (cs61a-xk . 1)
 (cs61a-xw . 2) (cs61a-ya . 3) (cs61a-yb . 3) (cs61a-yc . 1))
```

We could combine these in the obvious way to get the average score per student, for exams actually taken.

Word frequency counting. A common problem is to look for commonly used words in a document. For starters, we'll count word frequencies in a single sentence. The first step is to turn the sentence into key-value pairs in which the key is the word and the value is always 1:

```
> (map (lambda (wd) (list (make-kv-pair wd 1))) '(cry baby cry))
((cry . 1) (baby . 1) (cry . 1))
```

If we group these by key and add the values, we'll get the number of times each word appears.

```
(define (wordcounts1 sent)
  (groupreduce + 0 (sort-into-buckets (map (lambda (wd) (make-kv-pair wd 1))
                                           sent))))

> (wordcounts1 '(cry baby cry))
((baby . 1) (cry . 2))
```

Now to try the same task with (simulated) files. When we use the real `mapreduce`, it'll give us file data in the form of a key-value pair whose key is the name of the file and whose value is a line from the file, in the form of a sentence. For now, we're going to simulate a file as a list whose car is the "filename" and whose cdr is a list of sentences, representing the lines of the file. In other words, a file is a list whose first element is the filename and whose remaining elements are the lines.

```
(define filename car)
(define lines cdr)
```

Here's some data for us to play with:

```
(define file1 '((please please me) (i saw her standing there) (misery)
                (anna go to him) (chains) (boys) (ask me why)
                (please please me) (love me do) (ps i love you)
                (baby its you) (do you want to know a secret)))
(define file2 '((with the beatles) (it wont be long) (all ive got to do)
                (all my loving) (dont bother me) (little child)
                (till there was you) (roll over beethoven) (hold me tight)
                (you really got a hold on me) (i wanna be your man)
                (not a second time)))
(define file3 '((a hard days night) (a hard days night)
                (i should have known better) (if i fell)
                (im happy just to dance with you) (and i love her)
                (tell me why) (cant buy me love) (any time at all)
                (ill cry instead) (things we said today) (when i get home)
                (you cant do that) (ill be back)))
```

We start with a little procedure to turn a "file" into an a-list in the form `mapreduce` will give us:

```
(define (file->linelist file)
  (map (lambda (line) (make-kv-pair (filename file) line))
       (lines file)))

> (file->linelist file1)
(((please please me) i saw her standing there)
 ((please please me) misery)
 ((please please me) anna go to him)
 ((please please me) chains)
 ((please please me) boys)
 ((please please me) ask me why)
 ((please please me) please please me)
 ((please please me) love me do)
 ((please please me) ps i love you)
 ((please please me) baby its you)
 ((please please me) do you want to know a secret))
```

Note that `((please please me) misery)` is how Scheme prints the kv-pair `((please please me) . (misery))`.

Now we modify our `wordcounts1` procedure to accept such kv-pairs:

```
(define (wordcounts files)
  (groupreduce + 0 (sort-into-buckets
                    (flatmap (lambda (kv-pair)
                               (map (lambda (wd) (make-kv-pair wd 1))
                                    (kv-value kv-pair)))
                             files))))
```

```
> (wordcounts (append (file->linelist file1)
                      (file->linelist file2)
                      (file->linelist file3)))
((a . 4) (all . 3) (and . 1) (anna . 1) (any . 1) (ask . 1) (at . 1)
 (baby . 1) (back . 1) (be . 3) (beethoven . 1) (better . 1) (bother . 1)
 (boys . 1) (buy . 1) (cant . 2) (chains . 1) (child . 1) (cry . 1)
 (dance . 1) (days . 1) (do . 4) (dont . 1) (fell . 1) (get . 1) (go . 1)
 (got . 2) (happy . 1) (hard . 1) (have . 1) (her . 2) (him . 1) (hold . 2)
 (home . 1) (i . 7) (if . 1) (ill . 2) (im . 1) (instead . 1) (it . 1)
 (its . 1) (ive . 1) (just . 1) (know . 1) (known . 1) (little . 1)
 (long . 1) (love . 4) (loving . 1) (man . 1) (me . 8) (misery . 1) (my . 1)
 (night . 1) (not . 1) (on . 1) (over . 1) (please . 2) (ps . 1) (really . 1)
 (roll . 1) (said . 1) (saw . 1) (second . 1) (secret . 1) (should . 1)
 (standing . 1) (tell . 1) (that . 1) (there . 2) (things . 1) (tight . 1)
 (till . 1) (time . 2) (to . 4) (today . 1) (wanna . 1) (want . 1) (was . 1)
 (we . 1) (when . 1) (why . 2) (with . 1) (wont . 1) (you . 7) (your . 1))
```

(If you count yourself to check, remember that words in the album titles don't count! They're keys, not values.)

Note the call to `flatmap` above. In a real `mapreduce`, each file would be mapped on a different processor, and the results would be distributed to `reduce` processes in parallel. Here, the `map` over files gives us a list of a-lists, one for each file, and we have to append them to form a single a-list. `Flatmap` flattens (appends) the results from calling `map`.

We can postprocess the groupreduce output to get an overall reduction to a single value:

```
(define (mostfreq files)
  (accumulate (lambda (new old)
                (cond ((> (kv-value new) (kv-value (car old)))
                       (list new))
                      ((= (kv-value new) (kv-value (car old)))
                       (cons new old))        ; In case of tie, remember both.
                      (else old)))
              (list (make-kv-pair 'foo 0))        ; Starting value.
              (groupreduce + 0 (sort-into-buckets
                                (flatmap (lambda (kv-pair)
                                           (map (lambda (wd)
                                                  (make-kv-pair wd 1))
                                                (kv-value kv-pair)))
                                         files)))))
```

```
> (mostfreq (append (file->linelist file1)
                    (file->linelist file2)
                    (file->linelist file3)))
((me . 8))
```

(Second place is "you" and "I" with 7 appearances each, which would have made a two-element a-list as the result.) If we had a truly enormous word list, we'd put it into a distributed file and use another `mapreduce` to find the most frequent words of subsets of the list, and then find the most frequent word of those most frequent words.

Searching for a pattern. Another task is to search through files for lines matching a pattern. A *pattern* is a sentence in which the word `*` matches any set of zero or more words:

```
> (match? '(* i * her *) '(i saw her standing there))
#t
> (match? '(* i * her *) '(and i love her))
#t
> (match? '(* i * her *) '(ps i love you))
#f
```

Here's how we look for lines in files that match a pattern:

```
(define (grep pattern files)
  (groupreduce cons '()
               (sort-into-buckets
                (flatmap (lambda (kv-pair)
                           (if (match? pattern (kv-value kv-pair))
                               (list kv-pair)
                               '()))
                         files))))

> (grep '(* i * her *) (append (file->linelist file1)
                               (file->linelist file2)
                               (file->linelist file3)))
(((a hard days night) (and i love her))
 ((please please me) (i saw her standing there)))
```

Summary. The general pattern here is

```
(groupreduce reducer base-case
             (sort-into-buckets
              (map-or-flatmap mapper data)))
```

This corresponds to

```
(mapreduce mapper reducer base-case data)
```

in the truly parallel `mapreduce` exploration we'll be doing later.