DATA ABSTRACTION 5

GEORGE WANG
gswang.cs61a@gmail.com
Department of Electrical Engineering and Computer Sciences
University of California, Berkeley

June 28, 2010

1 Administrivia

We apologize for all the corrections done for the homework. We didn't do a good enough job beta testing the homework before we released it. We've learned our lesson, and we're aiming to do a much better job for your future homeworks.

1.1 Addendum to Cheating Policy

Sorry for not making this clear. I wanted you to know how the cheating policy for this course worked. Although I spoke at length about what constituted cheating and how we're trying to prevent you from cheating, I didn't talk about what the penalties were for cheating. If you cheat on the homework, we'll give your first offense -4/0. After that, it's an automatic F. If you cheat on projects or exams, it's an automatic F.

We want you to know that we are practically always available one way or another (via email if nothing else), and that if you start early, there is no reason you should need to resort to cheating. Furthermore, you are *encouraged to collaborate on homework* so long as you note where you are getting help. The first project, however, is an individual programming assignment, just so you know.

2 Review from Last Week

We talked about the Syntax of Scheme. I'll list all the functions you should know by today, but if you're missing any take time to review the other lecture notes later. This is in no particular order.

`lambda, let, every, keep, accumulate, first, butfirst, word, bf, last, butlast, bl, word, quote, equal?, if, cond , member?, +, -, /, *, repeated, count, empty?`

2.1 Runtime of an Algorithm

Consider the insertion sort algorithm you did as homework. What we want to make sure we understand is the runtime of each section of it. So, let's look at insert first. What is the runtime there?

Well, whatever it is, we'll multiply that runtime by the length of the unsorted-sent. To see why, imagine we are trying to do something that takes us 8 hours per day. Say we work for 30 days. How long have we spent? Well, for each of those 30 days, we do 8 hours of work, so we do $30 \times 8 = 240$hours of work. This is the same thing. For each element of the unsorted sentence, we must do an insertion's worth of work.

```
(define (insert new-num sorted-sent)
  (if (empty? sorted-sent)
      (se new-num)
      (if (<= new-num (first sorted-sent))
          (se new-num sorted-sent)
          (se (first sorted-sent)
              (insert new-num (bf sorted-sent))))))
(define (insertion-sort-2 unsorted-sent)
  (if (empty? unsorted-sent)
      '()
      (insert (first sent) (insertion-sort-2 (bf sent)))))
```

3 Data Abstraction

3.1 Definition

What is data abstraction? To put it simply, it's the idea that we can arbitrarily make something into something else by constructing nothing but a mental barrier. To be honest, it's a little bit like looking at clouds and saying that "look, that's a lion" or "that's a keyhole". So to kind of illustrate this point, let's look at something called the Treachery of Images.

So the text in French says "This is not a pipe." This is a famous drawing, and the point it's trying to make, is that this is merely a drawing of a pipe, not a pipe itself. However, let's think about what we're trying to say here. Nothing in a computer are things themselves. For example, with the 21 project you guys are working on, there's obviously no tiny deck of cards inside the CPU.

Therefore, we have something called data abstraction. We have to think of data as things to *represent* other things. In other words, the word 2h actually represents the two of hearts. And thus, when we look at '2h, we have to stop thinking of what they are and what they represent.

An easy way to really think about this is that, well, '2h doesn't actually exist on the machine either, right? You guys have heard of binary, a stream of 0's and 1's that really represent everything in a computer. But even that is represented by an electrical signal, which is represented by a bunch of singular electrons, and so on and so forth.

The whole idea is that you need to be able to be very comfortable adapting to where you are in the abstraction heirarchy. You need to be very flexible, being able to switch easily between different layers.

3.2 An Example: Total

Let's look at what I mean by the difference between what they are and what they represent. This example also illustrates a practical purpose to this. By doing so, you will make your code easier to read. Remember, bits have no meaning. It's the programmer's responsibility to bestow meaning as well as interpret that meaning.

Below is a simplified version of part of what you're working on for project 1. This is a procedure that is supposed to take a bunch of cards and sum together the rank of each card.

```
;;;;; 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
```

So let's look at what the heck this is doing. The first line gets the rank of a single card from a hand, and adds that together with the total called on the rest of the hand. In other words, the two butlasts are doing completely different things!

Now, let's look at a modified version:

```
;;;;; In file ~cs61a/lectures/2.1/total.scm
(define (total hand)
  (if (no-more-cards? hand)
      0
      (+ (card-rank (one-card hand))
         (total (remaining-cards hand)) )))
(define card-rank butlast)
(define card-suit last)
(define one-card last)
(define remaining-cards butlast)
(define no-more-cards? empty?)
```

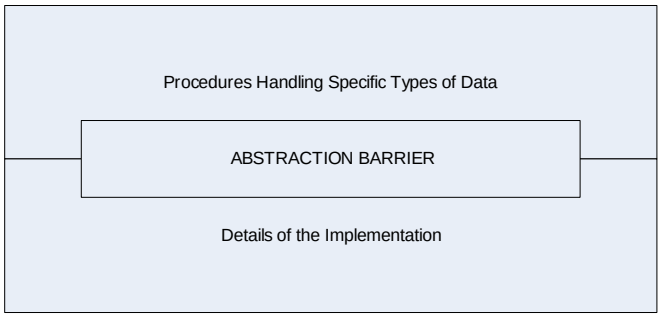
These are called SELECTORS, because they are able to extract a single piece of information from a DATA STRUCTURE. There's one more thing, though. We're assuming that the card is implemented using words with numbers and a letter at the end. But we can make our program even more general. We need CONSTRUCTORS as well. A constructor is a function that creates some piece of data given a few arguments. We can think of a constructor as something to give meaning to a piece of data.

```
;;;;; 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
```

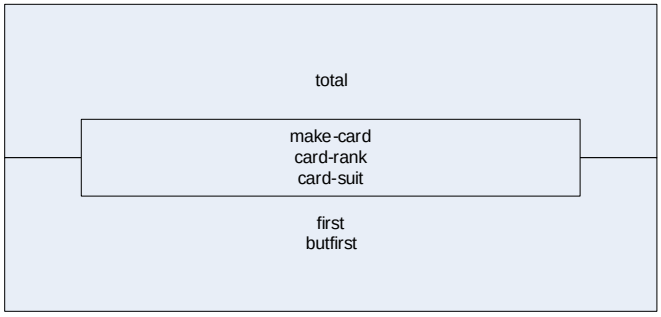
This allows us to completely separate the meaning of the data from the implementation of the data itself. That's something very very important, because sometimes, we may want to change the implementation. For example, say we want to make a different kind of implementation of cards, using numbers instead of words.

```
;;;;; 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 (card-rank card) (remainder card 13))
(define (card-suit card) (nth (quotient card 13) '(heart spade diamond club)))
```

The power of this is that we've just managed to change the entire implementation without touching the functionality of our total procedure. We've changed the implementation without changing the functionality at all.



This is a diagram we draw to illustrate the fact that there is something that separates the functionality of a program from the implementation of things. For the specific example above, we'll draw a specific diagram.



4 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 later.

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`.

5 Data as Functions

This section is somewhat confusing, so if you don't understand this, don't worry about it.

This is I think one of the coolest things. Last week, we said how we could use functions as inputs and outputs to procedures, as data. However, it turns out that the opposite is entirely possible as well.

```
;;;;; 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))
```

As an example, try doing `(cons 2 3)`. How do you get the `car` and `cdr` of it?