PAIRS AND LISTS 6

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

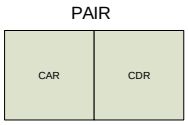
June 29, 2010

1 Pairs

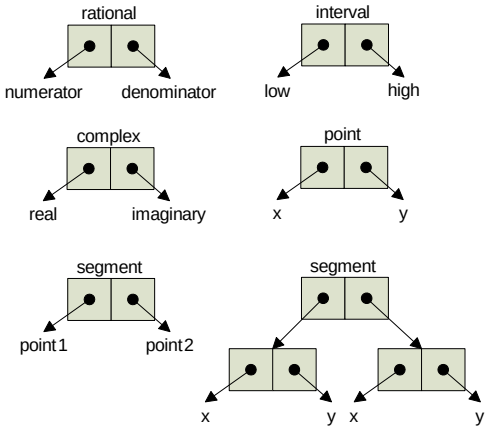
1.1 Overview

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`.



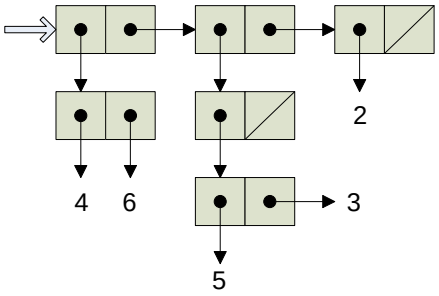
The book uses pairs to represent many different abstract data types: rational numbers (numerator and denominator), complex numbers (real and imaginary parts), points (x and y coordinates), intervals (low and high bounds), and line segments (two endpoints).



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.

1.2 Exercise: `cons`

Construct the following diagram using `cons`.



1.3 Naming

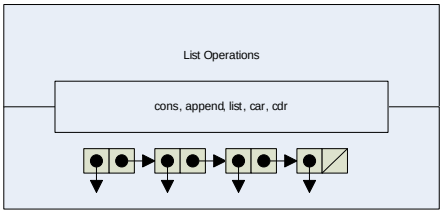
So why the heck is it called car and cdr? Well, turns out that a long time ago, one of the earliest machines, the IBM 704, the machine to implement them had a memory which contained something called a address register and a decrement register. Therefore, car stands for the Contents of Address Register, and cdr stands for Contents of Decrement Register.

But there's a very useful reason to using things like this, and it's used when we are going to combine our pairs into bigger pairs. The important thing is that we can combine `car` and `cdr` in a useful way, such that we get something like `caddar` which is the `(car (cdr (cdr (car ...))))` of something. How do you pronounce that out? Well, load `~cs61a/lib/prounounce.scm`, and it will tell you. This is the most important part of Scheme! If you want to be a lisp wizard, you have to be able to say it right.

1.4 Exercise: Pair Accessors

Explain that if `a` points to the upper left of the diagram above, how to extract each of the different numbers.

2 Sequences: Lists



Well, building things like this is rather dull. What if we want to represent a sequence of things? We may want to use a data structure called a list. Although these typically contain data that is very similar, they don't have to be. You are used to sequential data structures in the form of sentences, and these are essentially the same thing. However, these have a few more bits of power.

1. They can contain other lists.
2. They can contain Procedures.
3. They can contain **anything**.

And they are elegantly defined. They are *recursively* defined:

A list is a pair whose cdr is a list. An empty list is a list.

2.1 Implementation

Let's look at their implementation, before moving on to the selectors and constructors. The basic implementation is straightforward. It's using the pairs we're used to. Essentially, you just connect together a bunch of pairs together via the `cdr`, and at the very end, affix an empty list. Typically, in a flat list, the `car` contains all the information, while the `cdr` is another list. This is analogous to `first` and `butfirst`.

2.2 Selectors

It's just `car` and `cdr`. The reason for this is that a list is just a pair. (Note, the converse is not true, a pair is not necessarily a list.) Therefore, when we're manipulating a list, it's possible to think about it as we're merely manipulating the first pair.

2.3 Constructors

2.3.1 `cons`

This is the simple one. We know that we can make a pair with cons, so if we want to add an element to the beginning of a list, we just use `cons`.

```
> (cons 'the '(quick brown fox jumps over the lazy dog))
(the quick brown fox jumps over the lazy dog)
```

The one thing we want to be careful about is that we must remember that cons is not unique to lists. Therefore, if we cons together a list and a list, we do not get a flat list, we get a deep list, as follows. Note that we will study deep lists in detail later this week.

```
> (cons '(the) '(quick brown fox jumps over the lazy dog))
((the) quick brown fox jumps over the lazy dog)
```

This turns out to be the most useful one, because most of the time when we're doing a recursive transformation, we're building a list one pair at a time. Let's look at `map`, the list analogy to `every`.

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

2.3.2 `append`

This procedure takes two lists as its argument and returns a new list that is a copy of the old one with the last pair pointing to the new list. One student described this to me as basically writing them next to each other and then just erasing the two parens in between them. We will only deal with this procedure as being applied to two lists.

```
> (append '(the quick brown fox) '(jumps over the lazy dog))
(the quick brown fox jumps over the lazy dog)
```

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.

2.3.3 `list`

This is the easiest to understand, but unfortunately, also the easiest to get wrong. In short, it creates *spine pairs* and points one of them to each of the arguments.

For example:

```
> (list '(a list) 'word 87 #4 +)
((a list) word 87 #t +)
```

In practice it turns out to not be very useful, because you can only really use it when you know exactly how many things you want in a list. A lot of list programming deals with sequences of arbitary length. Thus, 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))
```

2.4 Exercises: Naming

Come up with better names:

```
(define (mystery x)
  (if (equal? x '())
      #t
      (if (not (pair? x))
         #f
         (mystery (cdr x)))))
(define (mystery x y)
  (if (null? x)
```

```
      y
      (cons (car x)
            (mystery (cdr x) y))))
```

3 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.

Here's a slightly simplified version of sentences given what you now know about lists:

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

The point is this. `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. `Se` always returns a flat list, but lists accept inputs that are not flat.

3.1 Example: Reverse

Say you want 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)) ))
```

4 Box and Pointer Diagrams

Here are a few things that students often get wrong:

1. An arrow can't point to half of a pair. Remember, the pair is essentially a data structure telling you where to find two pieces of information. It doesn't make sense to point to a a single half of a box, because that's like a mailing address pointing to another envelope. Thus, 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 pair point to the three elements of the top-level list.