

CS61A Notes - Week 2A: Pairs and lists, data abstraction

Orders of Growth

When we talk about the efficiency of a procedure (at least for now), we're often interested in how much more expensive it is to run the procedure with a larger input. That is, as the size of the input grows, how do the speed of the procedure and the space its process occupies grow?

For expressing all of these, we use what is called the Big-Theta notation. For example, if we say the running time of a procedure `foo` is in $\Theta(n^2)$, we mean that the time it takes to process the input grows as the square of the size of the input. Here are some common orders of growth:

- $\Theta(1)$: constant time (takes the same amount of time regardless of input size)
- $\Theta(\log n)$: logarithmic time
- $\Theta(n)$: linear time
- $\Theta(n^2)$, $\Theta(n^3)$, etc.: polynomial time
- $\Theta(2^n)$, $\Theta(3^n)$: exponential time ("intractable"; very, very bad)

Note, this 2 and 3 are not constant factors. The base of an exponential changes the order of growth.

The idea of orders of growth also extends to *space*. Space refers to how much information a process must remember before it can complete. If you'll recall, the advantage of an iterative process is that it does not have to keep any information on the stack as it executes. So an iterative process always occupies a constant amount of space: $\Theta(1)$.

For a recursive process, the space it occupies tends to grow with the number of recursive calls. For example, for the factorial procedure, every time before we make a recursive call, we need to "remember" to multiply (`fact (- n 1)`) by n . Since we'll make n recursive calls, we'll need to remember n such facts. So the orders of growth in space for the factorial function is $\Theta(n)$.

In CS61A, we're going to be mostly concerned with *time*. Time basically refers to the number of recursive calls. Intuitively, the more recursive calls we make, the more time it takes to execute the function.

A few things to note:

- If the function contains only primitive procedures like `+` or `*`, then it is constant time - $\Theta(1)$. An example would be `(define (plusone x) (+ x 1))`
- If the function is recursive, you need to:
 - a. Count the number of recursive calls there will be given input n
 - b. Count how much time it takes to process the input *per recursive call*The answer is usually the product of the above two. For example, given a fruit basket with 10 apples, how long does it take for me to process the whole basket? Well, I'll recursively call my `eat` procedure which eats one apple at a time (so I'll call the procedure 10 times). Each time I `eat` an apple, it takes me 30 minutes. So the total amount of time is just $30 * 10 = 300$ minutes!
- If the function contains calls of helper functions that are not constant-time, then you need to take the orders of growth of the helper functions into consideration as well. In general, how much time the helper function takes would be factored into #2 above. (If this is confusing - and it is - try the examples below.)
- When we talk about orders of growth, we don't really care about constant factors. So if you get something like $\Theta(1000000n)$, this is really $\Theta(n)$. We can also usually ignore lower-order terms. For example, if we get something like $\Theta(n^3 + n^2 + 4n + 399)$, we take it to be $\Theta(n^3)$.

QUESTIONS: What is the order of growth in time for:

1.

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

2.

```
(define (fact-iter x answer)
  (if (= x 0)
      answer
      (fact-iter (- x 1) (* answer x))))
```
3.

```
(define (sum-of-facts x n)
  (if (= n 0)
      0
      (+ (fact x) (sum-of-facts x (- n 1)))))
```
4.

```
(define (fib n)
  (if (<= n 1)
      1
      (+ (fib (- n 1)) (fib (- n 2)))))
```
5.

```
(define (square n)
  (cond ((= n 0) 0)
        ((even? n) (* (square (quotient n 2)) 4))
        (else (+ (square (- n 1)) (- (+ n n) 1)))))
```

Pair Up!

Introducing - the only data structure you'll ever need in 61A - pairs.

A pair is a data structure that contains two *things* – the “things” can be atomic values or even another pair. For example, you can represent a point as `(x . y)` and a date as `(July . 1)`. Note Scheme's representation of a pair; the pair is enclosed in parentheses, and separated by a single period.

Note that there's an operator called `pair?` that tests whether something is a pair or not. For example, `(pair? (cons 3 4))` is `#t`, while `(pair? 10)` is `#f`.

You've read about “cons”, “car”, and “cdr”:

`cons`: takes in two parameters and constructs a pair out of them. So `(cons 3 4)` will return `(3 . 4)`.

`car`: takes in a pair and returns the first part of the pair. So `(car (cons 3 4))` will return 3.

`cdr`: takes in a pair and returns the second part of the pair. So `(cdr (cons 3 4))` will return 4.

These will be all we'll ever need to build complex data structures in this course.

QUESTIONS: What do the following evaluate to?

```
(define u (cons 2 3)) (define w (cons 5 6)) (define x (cons u w))
(define y (cons w x)) (define z (cons 3 y))
```

1. `u`, `w`, `x`, `y`, `z` (write them out in Scheme's notation)

2. `(car y)`
3. `(car (car y))`
4. `(cdr (car (cdr (cdr z))))`
5. `(+ (cdr (car y)) (cdr (car (cdr z))))`
6. `(cons z u)`
7. `(cons (car (cdr y)) (cons (car (car x)) (car (car (cdr z)))))`

Then Came Lists

“Lists” are: either the empty list, or a pair whose `car` is an element of the list and whose `cdr` is another list. Note the recursive definition - a list is a pair that contains a list! So then how does it end? Wouldn't there be an infinite number of lists? Not so: an empty list, called `nil` and denoted `'()` is a list containing no elements. And so it is that every list ends with the empty list. To test whether a list is empty, you can use the `null?` operator on a list.

So, to make a list of elements 2, 3, 4, we do this:

```
(define x (cons 2 (cons 3 (cons 4 '() ) ) ) )
```

So then `x` will be represented as:

```
(2 . (3 . (4 . '() ) ) )
```

Now, that looks a bit ugly, so Scheme, the nice, friendly language that it is, sugar-coats the notation a bit so you get:

```
(2 3 4)
```

It's a bit annoying to write so many `cons` to define `x`. So Scheme, the mushy-gushy language that it is, provides an operator `list` that takes in many elements and returns them in a list. So we can also define `x` this way:

```
(define x (list 2 3 4))
```

Note: `(car x)` is 2, `(cdr x)` is `(3 4)`, and `(car (cdr x))` is 3! Well, it's a bit tiresome to write `(car (cdr x))` to get the second element of `x`. So Scheme, again the huggable lovable language that it is, provides a nifty shorthand: `(cadr x)`. This reads *cader*, and means “take the `car` of the `cdr` of”. Similarly, you can use `(caddr x)` - *caderder* - to take the `car` of the `cdr` of the `cdr` of the `x`, which is 4. You can mix and match the ‘a’ and ‘d’ between the ‘c’ and ‘r’ to get the desired element of the list (up to 4 ‘a’ or ‘d’).

You can also append two lists together. `append` takes in any number of lists and outputs a list containing those lists concatenated together. So `(append (list 3 4) (list 5 6))` returns `(3 4 5 6)`.

Don't You Mean sentence?

Oh stop grumbling. A “sentence” is actually a special kind of “list” - more specifically, a “sentence” is a *flat list* - a list without any sublists - whose elements can only be words or numbers. The operators of sentence - `first`, `butfirst`, `se`, etc. - are also much more forgiving in their domains than their list counterparts. For example, here's a list of equivalences:

CS61A Summer 2010 George Wang, Jon Kotker, Seshadri Mahalingam, Steven Tang, Eric Tzeng
Notes developed based on those of Chung Wu and Justin Chen

```

first <=> car: (first '(1 2 3 4)) = (car '(1 2 3 4)) = 1
butfirst <=> cdr: (butfirst '(1 2 3 4)) = (cdr '(1 2 3 4)) = (2 3 4)
empty? <=> null?: (empty? '()) = (null? '()) = #t
sentence <=> list: (se 1 2 3 4) = (list 1 2 3 4) = (1 2 3 4)
sentence <=> cons: (se 1 '(2 3 4)) = (cons 1 '(2 3 4)) = (1 2 3 4)
sentence <=> append: (se '(1 2) '(3 4)) = (append '(1 2) '(3 4)) = (1 2 3 4)
count <=> length: (count '(3 4 1)) = (length '(3 4 1)) = 3
every <=> map: (every square '(1 2)) = (map square '(1 2)) = (1 4)
keep <=> filter: (keep number? '(2 k 4)) = (filter number? '(2 k 4)) = (2 4)

```

Note that while `se` can be used for any combination of single elements and sentences to make another sentence, `list`, `cons`, and `append` are a bit more subtle, and what you pass in as arguments really matters. For example,

```

(se '(1 2) '(3 4)) = (1 2 3 4) which is not the same as (list '(1 2) '(3 4)) = ((1 2) (3 4))
(se '(1 2) 3) = (1 2 3) which is not the same as (cons '(1 2) 3) = ((1 2) . 3)
(se '(1 2) 3) = (1 2 3) which is not the same as (append '(1 2) 3) = Error: 3 not a list!

```

And so on. You must be more careful with what you pass into the list operators! What do you get for all that trouble? Power - you can put *anything* into lists, not just words and numbers. You will now be able to construct deep lists - lists that contain sublists, which allows you to represent all sorts of cool things. You can also store exotic things like procedures in a list. The possibilities are endless!

QUESTIONS

1. Define a procedure `list-4` that takes in 4 elements and outputs a list equivalent to one created by calling `list`, but without using the `list` procedure.
2. Write `num-satisfies` that takes in a sentence and a predicate, and returns the number of elements in the sentence that satisfy the given predicate. USE ONLY HIGHER ORDER FUNCTIONS; DO NOT USE RECURSION.
3. Write `all-satisfies` that takes in a sentence and a predicate, and returns `#t` if and only if all of the elements in the list satisfy the given predicate. USE ONLY HIGHER ORDER FUNCTIONS; DO NOT USE RECURSION.
4. Rewrite questions 2 and 3 to take in a *list* and a predicate. USE ONLY HIGHER ORDER FUNCTIONS; DO NOT USE RECURSION.

5. Write `repeat-evens` that takes in a sentence of numbers and repeats the even elements twice. USE ONLY HIGHER ORDER FUNCTIONS; DO NOT USE RECURSION.

6. (Hard!) Rewrite question 5 to work on a list of numbers. USE ONLY HIGHER ORDER FUNCTIONS; DO NOT USE RECURSION.

7. Suppose we have `x` bound to a mysterious element. All we know is this:


```
(list? x) => #t
(pair? x) => #f
```

 What is `x`?

8. Add in procedure calls to get the desired results. All the parens below represent lists. Blanks can be left blank:

```
( _____ 'a _____ '(b c d e) )
=> (a b c d e)
```

```
( _____ '(cs61a is) _____ 'cool )
=> (cs61a is cool)
```

```
( _____ '(back to) _____ '(save the universe) )
=> ((back to) save the universe)
```

```
( _____ '(I keep the wolf) _____ '((from the door)) )
=> ((I keep the wolf) from the door)
```

9. Define a procedure `(insert-after item mark ls)` which inserts `item` after `mark` in `ls`.

Box and Pointer Diagrams

QUESTIONS: Evaluate the following, and draw a box-and-pointer diagram for each. (Hint: It may be easier to draw the box-and-pointer diagram first.)

1. `(cons (cons 1 2) (cons 3 4))`

2. `(cons '((1 a) (2 o)) '(3 g))`

3. `(list '((1 a) (2 o)) '(3 g))`

4. `(append '((1 a) (2 o)) '(3 g))`

5. `(cdr (car (cdr '((1) 3) (4 (5 6)))))`

6. `(map (lambda (fn) (cons fn (fn 6))) (list square 1+ even?))`

(Slightly) Harder Lists

1. Define a procedure `(depth ls)` that calculates the maximum depth of sublists in `ls`. For example,
 - `(depth '(1 2 3 4)) => 1`
 - `(depth '(1 2 (3 4) 5)) => 2`
 - `(depth '(1 2 (3 4 5 (6 7) 8) 9 (10 11) 12)) => 3`
 Remember that there's a procedure called `max` that takes in two numbers and returns the greater of the two.

2. Define a procedure `(remove item ls)` that takes in a list and returns a new list with `item` removed from `ls`.

3. Define a procedure `(unique-elements ls)` that takes in a list and returns a new list without duplicates. You've already done this with `remove-dups`, and it used to do this:
 - `(remove-dups '(3 5 6 3 3 5 9 8)) ==> (6 3 5 9 8)`
 where the *last* occurrence of an element is kept. We'd like to keep the *first* occurrences:
 - `(unique-elements '(3 5 6 3 3 5 9 8)) => (3 5 6 9 8)`
 Try doing it without using `member?`. You might want to use `remove` above.

4. Define a procedure `(count-of item ls)` that returns how many times a given item occurs in a given list; it could also be in a sublist. So,
 - `(count-of 'a '(a b c a a (b d a c (a e) a) b (a))) => 7`

5. Define a procedure (`count-unique ls`) which, given a list of elements, returns a list of pairs whose `car` is an element and whose `cdr` is its number of occurrences in the list. For example,


```
(count-unique '(a b b b c d d a e e f a a))
=> ((a . 4) (b . 3) (c . 1) (d . 2) (e . 2) (f . 1))
```

 You might want to use `unique-elements` and `count-of` defined above.

6. Define a procedure (`interleave ls1 ls2`) that takes in two lists and returns one list with elements from both lists interleaved. So,


```
(interleave '(a b c d) (1 2 3 4 5 6 7)) => (a 1 b 2 c 3 d 4 5 6 7)
```

7. Write a procedure (`apply-procs procs args`) that takes in a list of single-argument procedures and a list of arguments. It then applies each procedure in `procs` to each element in `args` in order. It returns a list of results. For example,


```
(apply-procs (list square double +1) '(1 2 3 4))
=> (3 9 19 33)
```

Data Abstraction

Let's look at a simple example, where a **student record** is represented as a pair of names and student IDs. Furthermore, a **name** is represented as a pair of words, for first and last names:

```
((brian . harvey) . 176) ;; student records
((justin . chen) . 205)
```

Instead of `caring` and `cdring` to get a specific piece of data, we'd like to be able to use intuitive **selectors** to access the data more naturally:

```
(define (first-name student) ;; domain: student records
  (car (car student)))
(define (last-name student)
  (cdr (car student)))
```

```
(define (student-id student)
  (cdr student))
```

We would also like to use **constructors** to standardize the creation of student records and names instead of calling `(cons (cons first last) sid)` every time:

```
(define (make-name first last)
  (cons first last))
(define (make-student name sid)
  (cons name sid))
```

It's important to note that, as was stated in lecture, the **abstraction barrier** is a completely *voluntary* practice in Scheme. So why do we do it?

1. **It's easy to change the internal representation of data.** Say we wanted to change a student record to be a pair whose `car` is their SID and whose `cdr` is their name. Instead of tracking down every place where we find a name or SID in our code and switch `car` with `cdr`, we can simply change our selectors and constructors to match our new representation.
2. **It makes the code both more human readable as well as less prone to errors.** Instead of forcing the reader to interpret all the `cars` and `cdrs` for different calls, they can simply read well-named selectors and constructors. If you're coding something, then you can use your abstractions to avoid unnecessary mistakes by `caring` or `cdring` incorrectly.

Finally, note that data abstraction means that your code can work perfectly fine but still be wrong. Using `car` instead of `first` on sentences, for example, would functionally do the same thing. However, it would be a data abstraction violation and would be considered wrong. (Evil, maybe, but we're doing it for your benefit. Really.) **Always remember to RESPECT THE ABSTRACTION!**

QUESTIONS:

1. Write a procedure `get-last-names` that takes a list of student records (that we defined above) and returns a sentence of each student's last name. Respect the data abstraction! (Hint: What's the domain of `get-last-names`? What's the range?)
2. Let's say we wanted to change our internal representation for names so that last names come first in the pair. Which selectors and constructors do we have to change? Modify the procedures that require changes so that they work with the new internal representation.
3. Modify the representation of student records to accommodate a student's GPA. You may use `list` as well as `cons` for your new representation. Finally, modify and/or create any new selectors or constructors for your new representation.