1 Transcript

1.1 The bad, the ugly, and the good

Let’s talk about Bad Code.

I don’t mean Bad Coders, and I don’t mean to imply that people who occasionally write Bad Code are Bad Coders. We all write Bad Code sometimes. We just want to get the job done and don’t care.

Well, here at Team CS17, it’s our job to make you care or at least pretend to. No, seriously, I am confident that, as loyal CS17 students, you would be delighted to find ways to improve your code.
Three principles that date back to the misty beginnings of CS17:

1. Get it right, then get it working.
2. Don’t be afraid to throw code away and start over.
3. Elegance pays.

These are all basically the same idea.

You would think that code that runs is better than code that does not run. That is indeed the philosophy in some places (let’s not name names of companies or even courses but see https://xkcd.com/1428/). It is our experience that this is a counterproductive philosophy.

It’s satisfying to have running code, and sometimes it seems that a good strategy is to start with running code and gradually adding more to cover more cases. In this course, the programs we write are just not that long. It’s usually better to take the lessons you learned from your attempt, and just start over but with your newfound wisdom.

Developing taste in code can take time. In this semester, the hallmarks of good code are brevity and clarity. Bad code, ergo, reflects writing something that is too long and unclear. Generally, if you can replace your code with something shorter, that’s an improvement. (The exception is that sometimes there is a conflict between brevity and clarity on the one hand and efficiency on the other. You sometimes have to choose.)

Let’s look at some small examples of how one can make code more concise.

### 1.2 Boolean expressions and conditionals

#### 1.2.1 Use of Boolean literals

(equal? #true (equal? (quote doctor) (car L)))

The value of this expression is #true if the car of the list L is the symbol doctor, and is #false otherwise. Can we think of a more concise expression for the same thing?

(equal? (quote doctor) (car L))

Similarly, how about this?

(if (equal? (quote doctor) (car L)) #true #false)

The value of this expression, similarly, is #true if the car of the list L is the symbol doctor, and is #false otherwise. This expression should similarly be replaced by the more concise expression.

Occurrences of #true and #false (Boolean literals) in your code should be extremely rare except for use of #true as the final condition in a cond, and should be cues for you to take a second or third look at your code.
1.2.2

Let’s look at more if/cond examples.

(define big-or-small
  (lambda (L)
    (cond
      ((empty? L)
        empty)
      ((equal? (car L) 1)
        (cons (quote small) (big-or-small (cdr L))))
      ((equal? (car L) 2)
        (cons (quote small) (big-or-small (cdr L))))
      (#true
        (cons (quote big) (big-or-small (cdr L)))))))

(define big-or-small
  (lambda (L)
    (cond
      ((empty? L)
        empty)
      ((or (equal? (car L) 1) (equal? (car L) 2))
        (cons (quote small) (big-or-small (cdr L))))
      (#true
        (cons (quote big) (big-or-small (cdr L)))))))

(define big-or-small
  (lambda (L)
    (cond
      ((empty? L)
        empty)
      (#true
        (cons (if (or (equal? (car L) 1) (equal? (car L) 2))
                  (quote small)
                  (quote big))
          (big-or-small (cdr L)))))))

(define big-or-small3
  (lambda (L)
    (if ((empty? L)
         empty
      (cons (if (or (equal? (car L) 1) (equal? (car L) 2))
              (quote small)
              (quote big))
            (big-or-small (cdr L)))))))
Lessons:

- look for repetition of an expression or part of an expression and see if there is a way to eliminate the repetition.
- Consider using `if` when only two cases and second condition is just `#true`

Other issues:

- Superfluous base cases
- condition that we already know must be true by that time.
- nested ifs instead of `cond`
- formatting/whitespace

1.3 Analysis

I will use the phrase running time of a procedure as a synonym for the number of operations performed by the procedure.

1.3.1 Order of growth

We said that a linear-time procedure is one such that there are constants $a$ and $b$ for which the number of operations on inputs of size $n$ is at most $a + bn$, for all values of $n$.

We said that a quadratic-time procedure is one such that there are constants $a, b, c$ for which the number of operations on inputs of size $n$ is at most $a + bn + cn^2$, for all values of $n$.

We can go on defining things like this, but we don’t want to have so many different definitions. There is a convenient concept that covers much of this and, once you master it, will simplify your life.

The definition of linear-time procedure reflects two value judgements:

1. The fact that we don’t care about the particular constant $a$ means that we don’t care how long the procedure takes on very small inputs.

2. The fact that we don’t care about the particular constant $b$ means that are unwilling to distinguish between running-time functions that differ by a fixed multiplier.

Why Value Judgement 1? One can imagine putting lots of effort into crafting a procedure that worked extremely well on small inputs. This is indeed a good strategy if most of the time is spent running the procedure on small inputs. However, in that case, one is arguably analyzing the wrong thing: there is a higher-level procedure that calls this procedure, and one should really analyze that one. A tiny amount of time is generally not worth worrying about. It won’t be noticeable.

Why Value Judgement 2? One reason is that our predictor of time spent, number of operations, is just not all that robust or accurate. The other reason is more methodological and indirect. It turns
out that as far as enabling technology to work well, the most important step is getting a linear-time algorithm as opposed to, say, a quadratic-time algorithm. Not carrying about the constant $b$ sort of frees up the algorithm designer to be more creative and to use cleverer techniques. We’ll see examples of that.

That is not to say that people don’t try to analyze or reduce the number $b$. It’s just not usually the most important thing.

These value judgements are reflected in a notion called order of growth.

Let $g(n)$ be a function. We say that another function $f(n)$ is $O(g(n))$ if there exist nonnegative constants $n_0$ and $c$ such that, for every value of $n$ that is greater than $n_0$, $f(n)$ is less than or equal to $c \cdot g(n)$.

For example, suppose $g$ is the function that on input $n$ returns as output $n$, i.e. we write $g(n) = n$.

I’ll prove that this is the same as saying that a procedure with running-time function $f(n)$ is a linear-time function.

**Proof** We have some procedure whose worst-case running time is $f(n)$.

First, suppose $f(n)$ is $O(g(n))$. That means there are nonnegative constants $n_0$ and $c$ such that $f(n) \leq c \cdot n$. Let $a$ be the maximum running time for the procedure on inputs of size at most $n_0$.

It easily follows that $f(n) \leq a + c \cdot n$ for all values of $n$. Why?

- For values of $n$ such that $n \leq n_0$, $f(n) \leq a$ so clearly $f(n) \leq a + c \cdot n$.
- For values of $n$ such that $n > n_0$, $f(n) \leq c \cdot n$, so clearly $f(n) \leq a + c \cdot n$.

Conversely, suppose there are constants $a$ and $b$ such that $f(n) \leq a + b \cdot n$ for all values of $n$. Let $c = a + b$. Let $n_0 = a/b$, and let $c = 2b$. Now consider any value of $n$.

$$f(n) \leq a + bn \leq bn_0 + bn$$

If $n > n_0$ then $bn_0 < bn$, so $bn_0 + bn \leq 2bn$, and $2bn = cn$. This shows that, for all values of $n$, if $n > n_0$ then $f(n) \leq cn$. That means that $f(n)$ is $O(b(n))$. 

5
2 Condensed Notes

2.1 Style/Elegance: Good and Bad

2.1.1 Three Principles of Writing Good Code

1. Get it right, then get it working.
2. Don’t be afraid to throw out code. Oftentimes it is best to start fresh - this will lead to a cleaner, more elegant solution.
3. Elegance pays!

2.1.2 Literal Booleans

Consider the following code:

\[(\text{equal? } \#\text{true} \quad \text{(equal? (quote doctor) (car L))})\]

The value of this expression is \#true if the car of the list L is the symbol doctor, and is \#false otherwise. Is there a more concise way to accomplish this?

First we can remove the outer (equal? \#true) because this is not affecting the functionality of our code. Therefore the above code can be rewritten more concisely as:

\[(\text{equal? (quote doctor) (car L))}\]

Take a moment to convince yourself that these lines of code will always evaluate to the same thing. Now let’s try another example:

\[(\text{if (equal? (quote doctor) (car L)) \#true \#false}\rangle\]

The value of this expression is similarly \#true if the car of the list L is the symbol doctor, and \#false otherwise. We do not need the if statement at all, and this code can also be replaced by the more concise expression above.

In general, if you see a literal boolean (\#true or \#false), there is almost always an opportunity to simplify! There is one exception - you are permitted to use \#true for the last clause of your cond expression.
2.1.3 if/cond

(define big-or-small
  (lambda (L)
    (cond
      ((empty? L) empty)
      ((= (car L) 1)
        (cons (quote small) (big-or-small (cdr L))))
      ((= (car L) 2)
        (cons (quote small) (big-or-small (cdr L))))
      (#true
        (cons (quote big) (big-or-small (cdr L))))))))

Take a look at what happens when the first element of the list is 1 or 2. In both of these clauses, we are doing the same thing - cons-ing (quote small) onto our recursive call. Instead of having two separate cases, we can use an or statement to combine them into one case:

(define big-or-small
  (lambda (L)
    (cond
      ((empty? L) empty)
      ((or (= (car L) 1) (= (car L) 2))
        (cons (quote small) (big-or-small (cdr L))))
      (#true
        (cons (quote big) (big-or-small (cdr L))))))))

We can still do better! In both the second and third cases, we are cons-ing something onto our recursive call. Rather than making two separate calls, we can use an if statement to choose what we want to be cons-ing and then make the recursive call. This leaves us with only two cases in our cond. A cond with only two cases can always be expressed in the form of an if statement.

The simplified code ends up looking like:

(define big-or-small
  (lambda (L)
    (if (empty? L)
      empty
      (cons (if (or (= (car L) 1) (= (car L) 2))
        (quote small)
        (quote big))
      (big-or-small (cdr L))))))
2.1.4 Takeaways: General Guidelines of What to Look Out For

- repetition of an expression or part of an expression
- instances of using cond on only two cases - consider converting to an if statement
- superfluous base cases that can be removed
- conditions that we already know are true by the time they are evaluated
- nested ifs - consider replacing with a cond
- cond cases that evaluate to the same thing - these can be condensed into one clause using or
- literal booleans (unless using #true in the last clause of your cond statement)
- repeatedly writing out the same recursive call - you can usually cons the output of an if statement onto the recursive call rather than having two recursive calls explicitly written out
- formatting/proper whitespace

2.2 Analysis

A linear time procedure is one for which there are constants $a$ and $b$ such that the running time is at most $a + bn$.

A quadratic time procedure is one for which there are nonnegative constants $a,b$, and $c$ such that the running time is at most $a + bn + cn^2$.

1. When analyzing a procedure, we don’t care about the specific value of $a$ because we aren’t concerned with the running time for small input sizes. When the input is small, the runtime will be very short anyways, and it is not worth analyzing a function that takes only a small portion of the overall time taken by our program.

2. We also don’t care about the specific value of $b$. The multiplier itself is not nearly as important as the overall growth rate of the function. This trend is determined by whether the function is linear, quadratic, exponential, etc.

2.2.1 Order of Growth

We say a function $f(n)$ is $O(g(n))$ if there exist nonnegative constants $n_o$ and $c$ such that, for every value of $n$ that is greater than $n_o$,

$$f(n) \leq c \cdot g(n)$$

(1)

Consider a function $g(n) = n$. We say $f(n)$ is $O(g(n))$ if there are constants $n_o$ and $c$ such that for all $n > n_o$, $f(n) \leq c \cdot g(n) \Rightarrow f(n) \leq c \cdot n$. 
Suppose \( f(n) \) is the running time of a linear-time procedure. Then there are constants \( a \) and \( b \) such that \( f(n) \leq a + bn \) for all \( n \). Let \( c = a + b \) and \( n_0 = \frac{a}{b} \). Consider any value of \( n \).

\[
\begin{align*}
  f(n) &\leq a + bn & \text{(2)} \\
      &\leq bn_0 + bn & \text{(3)} \\
      &\leq bn + bn & \text{(4)} \\
      &= 2bn & \text{(5)} \\
      &= cn & \text{(6)}
\end{align*}
\]

Thus, \( f(n) \) is \( O(g(n)) \).

Please let us know if you find any mistakes, inconsistencies, or confusing language in this or any other CS 17 document by filling out the anonymous feedback form: [http://cs.brown.edu/courses/csci0170/feedback](http://cs.brown.edu/courses/csci0170/feedback).