1 Class Update

Good news! You also no longer need to provide examples of atomic data (integers, booleans, strings, etc) as part of the design recipe. Also, a quick note about what Spike calls generic lists. These are denoted by using the notation 'a list, and essentially say that this is a list whose elements are of type 'a, or alpha. Alpha can be any type: num, boolean, num list, boolean list, etc. (this will become much more clear once we switch to ReasonML and start having to specify the types of our input in the code). As Spike mentioned, there are several procedures (such as length) that can act on any list, no matter the type of its elements. However, something like right-max can only operate on certain kinds of lists (namely, num lists) due to operations within the procedure (in this case, +). What you really need to understand is that 'a list means "a list whose elements are of type 'a".

Additionally, instead of writing lists as:

{(cons 10 (cons 15 (cons 4 (cons 3 (cons -22 empty))))})

You can write:

(list 10 15 4 3 2)

To create an empty list, simply write (list).
To reflect this change, change your Racket language to “Beginning Student with List Abbreviations.” To do this, select Language, Choose Language, and then Beginning Student with List Abbreviations. Once you make the change, printed lists will also be displayed in the new format.

2 Another Look at Recursion

Spike covered recursion from a new angle.
The basic idea that he was trying to get across was that when you run a recursive function on a list with \( k \) items, there is a “contract” that the internal recursive call will return the correct value for \( k - 1 \) items.

To walk through the example, check out the slides!

3 Introduction to Analysis

A big question for computer scientists is, “As the size of a program grows, how does resource consumption change?” There are many ways to answer this question: we could look at the space in memory that the program requires, the bandwidth it uses, the battery power it consumes, or the time it takes for it to run, for example. In this class, we focus on time: it’s the easiest to learn first, and will give you a solid foundation that you can build on for other types of analysis in other classes.

At this point in the class, Spike ran the right-max procedure two times to demonstrate this point: first, he ran it on a list of 4 elements, and it spat out the answer pretty quickly. The second time he ran it on a list of 25 zeros \((\text{list } 0 0 0 \ldots 0)\), which some would intuitively think would not take a long time. However, it took over 2 minutes for the program to finish. What this demonstrates is that, depending on the procedure, as the input size changes, the amount of time required for computation can also change. Estimating how long a program will take to run, how much space it will use, etc. is an important part of computer science. You will soon be analyzing procedures you write yourself and using your knowledge of the various costs of certain operations to write more efficient programs.

However, measuring time is tricky: let’s say we both write an implementation of a function, and run it on our own computers with the same input. Let’s say it takes 0.1 seconds to run on yours, and 3 seconds on mine. This may mean that your function is better than mine, but it may also mean that your computer is better than mine. For all we know, my function could be ten times more efficient, but running on an Apple 1.

The way we analyze efficiency across all computers is by measuring the number of elementary operations performed in a function in the worst-case scenario. As such, we can make a promise about our function: for some input of length \( n \), I promise that my function will do no more than \( x \) operations.

We cover the worst-case scenario because taking special circumstances into consideration can get very messy, very quickly. For instance, consider the \( \text{contains17?} \) function. If seventeen is the first element of the input list, we’re done quickly. However, if seventeen is not in the list, the function could take a very, very long time, depending on the length of the input list. In order to write a
general formula for how well our function runs, we want to be able to make a promise that will hold true about our function for all input lists of the same length. Therefore, we assume the scenario that would take the longest time to run, and calculate the runtime from there. That way, we can promise that contains17? will perform at most $x$ operations, and give our users some idea about what they’re getting themselves into.

3.1 Writing an Analysis

The general format of writing an analysis is:

1. Define a mathematical function, $F(n) : \mathbb{N} \rightarrow \mathbb{N}$, that calculates the worst-case number of operations for a procedure for an input list of length $n$

2. Write a recurrence relation for the mathematical function (covered in lecture 12)

3. Use the recurrence relation to say something useful about the function we’re analyzing (covered soon!)

To define the mathematical function, we use the following format:

“Let $T(n)$ be the largest number of elementary operations involved in applying our procedure to any list of length $n$.”

**Note:** When you’re writing your own proofs, replace $T$ with the first letter of your function name in your program.

In most cases, we can’t actually count the number of operations that occur in $T(n)$. Instead, we prove that $T(n)$ will be less than or equal to some natural number.

3.2 Elementary Operations

The recurrence relation depends on how many elementary operations will occur during evaluation of the procedure. We’ll discuss recurrence relations more later on, but for now, let’s talk about elementary operations.

The different types of elementary operations are:

- Evaluate a number-expression
- Evaluate a string-expression
- Evaluate a bool-expression
- Evaluate a name-expression (i.e. look up a binding in an environment)
- Evaluate `empty`
- Apply `cons` to two values
- Apply `+`, `*`, `−`, `/` to two values
- Apply `cons?`, `empty?`, `zero?`, `>`, `<`, `=`, etc.
For example, let's look at the number of elementary operations in evaluating 
\((\text{cons} \ 13 \ (\text{cons} \ 4 \ \text{empty}))\):

1. Look up \text{cons}
2. Evaluate 13
3. Look up \text{cons} (in the nested expression)
4. Look up 4
5. Evaluate \text{empty}
6. Apply \text{cons} to 4 and \text{empty}
7. Apply the first \text{cons} to 13 and \((\text{cons} \ 4 \ \text{empty})\)

So, this expression is evaluated with seven operations.

That being said, keep in mind that being off by some number of elementary operations *does not matter*. At the end of the day, the specific number of operations (for instance, if we counted 6 or 8 in the example above instead of 7) won’t end up impacting overall program runtime that much. Stay tuned for the next few lectures to find out why!

### 4 Introduction

Today’s class is a mix of lots of material. I’ll get to the last part of the lecture Phil did for me, but near the end. Before that, I’ll talk about operation-counting, naming new types, and non-empty list types, and after talking about list-reversal, I’ll also discuss generic types for a little while, if we have time.

### 5 Operation counting

Recall that I listed earlier all the elementary operations, and we counted that computing the length of the empty list took about 9 operations. If we work through counting operations for computing the length of a list of length one, it takes about \(19 + 9\) operations. I’ve written the number of operations that way because in the middle of doing the counting, we end up needing to count the
number of operations involved in computing \(\text{length } \text{empty}\), which we already know is 9. So there are 19 new steps, in addition to the 9 we knew about from before.

If we do the same for a list of length 2, we end up doing 19 new steps, and then all the work of computing the length of a list of length one, i.e., it takes \(19 + (19 + 9)\) steps in total.

And we start to see a pattern, which we can summarize like this:

Let \(L(n)\) be the largest number of steps involved in evaluating \(\text{length } \text{lst}\), where \(\text{lst}\) is any list of length \(n\).

Then

\[
\begin{align*}
L(0) &= 9 \\
L(n) &= 19 + L(n - 1) \quad \text{for } n \geq 1
\end{align*}
\]  

The function \(L\) is called an operation counting function for the length procedure (or sometimes an “op-count” function, for brevity), and the resulting equation is called a “recurrence equation for \(L\)”. It’s a set of properties that the function \(L\), whatever it may be, must have for inputs of a specified size.

Deriving a recurrence for the op-counting function for a procedure is the first step of analysis of the procedure, so it’s an important skill.

There’s one thing that’s really helpful to know: most of the actual operation-counts in the recurrence will not matter. And it’s best to replace them with constants. So our typical recurrence for the length procedure’s op-counting function would be written instead:

\[
\begin{align*}
L(0) &= A \\
L(n) &= B + L(n - 1) \quad \text{for } n \geq 1
\end{align*}
\]

The other thing to know is that the second line is almost always (for functions other than length) an inequality, with a less-than-or-equal rather than an equals-sign. The inequality is almost always preferable because, for most procedures, we the number of elementary operations done in the recursive case is not guaranteed to be the same for all lists of size \(n\), like it is for length. For instance consider how many operations the contains17? procedure would take on two lists of length 5, \((\text{list } 17 \ 1 \ 1 \ 1 \ 1)\) and \((\text{list } 1 \ 1 \ 1 \ 1 \ 17)\). Running contains17? on the first list would take far fewer operations, because 17 was the first element. The second list would take far more operations, because we’d end up having to traverse the full list to find the 17. The second list is a great example of what a “worst case scenario” input looks like for this procedure.

In some procedures, there are multiple base cases, so you may see something like

\[
\begin{align*}
H(0) &= A \\
H(1) &= B \\
H(n) &\leq C + H(n - 1) \quad \text{for } n \geq 2
\end{align*}
\]

rather than the simpler form.

There’s a generic set of steps to follow to do this first step of analysis, which we call “deriving the recurrence.”
1. Write “Let $H(n)$ be the largest number of operations involved in evaluating myproc on any list of length $n$”. You may have to vary this slightly for two-argument procedures, but the gist is always the same.

2. Count the number of steps in each base-case. This will almost always lead to an equality of the form

$$H(0) = A$$

3. Count the number of steps in the recursive case. Do this in three steps.

(a) Count the number of operations in the recursive case not including those in recursive calls. For example, this might be a constant, or it might depend on $n$.

(b) Identify the size of the argument to the recursive call. If you have $(\text{my-proc (rest aloi)})$, then this size is $n - 1$. But if the recursive call operates on a list consisting of the first, third, fifth, etc. elements of the original list, then this size is more like $n/2$; it’s actually $\lceil n/2 \rceil$, which means “round up $n/2$”.

(c) Write down the work done in the recursive call using your op-counting function. For a single recursive call on the rest of a list, this is $H(n - 1)$.

(d) Sum up these two op-counts to get the right-hand side of the recursive part of the recurrence relation.

4. Be certain to write the recursive-case recurrence with some “condition” on $n$, typically “for $n \geq 1$” if the base case was $n = 0$, or “for $n \geq 2$” if there were two base cases $n = 0, 1$.

For example, a recurrence relation for the procedure `contains17?`, which looks like

```scheme
(define (contains17? aloi)
  (cond
    [(empty? aloi) false]
    [(cons? aloi) (or (= 17 (first aloi))
                  (contains17? (rest aloi)))]))
```

would be: Let $C(n)$ be the largest number of operations involved in evaluating `contains17?` on any list of length $n$.

$$C(0) = A$$
$$C(n) \leq A + C(n-1) \quad \text{for } n \geq 1$$

For the procedure `foo`, which looks like

```scheme
(define (foo alod)
  (cond
    [(empty? alod) 0]
    [(empty? (rest alod)) 4]
    [(cons? (rest alod)) (+ 7 (foo (rest (rest alod))))]))
```

the recurrence relation would be: Let $F(n)$ be the largest number of operations involved in evaluating `foo` on any list of length $n$.

$$F(0) = A$$
$$F(1) = B$$
$$F(n) \leq Q + F(n-2) \quad \text{for } n \geq 2$$
6 Summary

Ideas

- We covered another way to approach recursion, based on the idea that when you run a recursive function on a list with \( k \) items, there’s a “contract” that the internal recursive call will return the correct value for \( k - 1 \) items.

- We also introduced recurrence relations, a structured way of analyzing procedures that we will be using throughout CS17. Don’t worry if you don’t fully understand them right now, there will be much more explanation of and practice with recurrence relations in these next few weeks and for the remainder of the class.

- We’re going to be analyzing the runtime of functions by calculating the worst-case number of elementary operations performed by the function.

Skills

- We can now use the keyword `list`! Make sure you update your Racket language accordingly

- We can define a mathematical function to use in analysis

- We can count the elementary operations involved in evaluating a procedure

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)