Lecture 10: Introduction to Analysis
10:00 AM, Sep 27, 2017

Contents

1 Class Update 1
2 Another Look at Recursion 1
3 Introduction to Analysis 1
   3.1 Writing an Analysis 2
   3.2 Elementary Operations 2
4 Summary 4

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. Additionally, instead of writing lists as:

(cons 10 (cons 15 (cons 4 (cons (cons -22 empty) 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's 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 off of for other types of analysis in other classes.

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 `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.`
- Apply `first` or `rest` to a list
- Apply a binding to an environment
- Add a binding to an environment
- Remove a binding from an environment
- Compare a boolean to `true/false`

For example, let’s look at the number of elementary operations in evaluating `(cons 13 (cons 4 empty))`:

1. Look up `cons`
2. Evaluate 13
3. Look up `cons` (in the nested expression)
4. Look up 4
5. Evaluate `empty`
6. Apply `cons` to 4 and `empty`
7. Apply the first `cons` to 13 and `(cons 4 `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 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’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 CS17 document by filling out the anonymous feedback form: [http://cs.brown.edu/courses/cs017/feedback](http://cs.brown.edu/courses/cs017/feedback)