Lecture 10: Analysis, Continued
10:00 AM, Sep 26, 2018

Contents

1 Transcript 1
2 Summary 5
   2.1 Another Look at Recursion ........................................... 5
   2.2 Mathematical Functions ............................................. 5

1 Transcript

Last time: we analyzed English2cow:

- \((\text{English2cow empty})\) took 7 operations
- \((\text{English2cow (quote (a b)))}\) took 23 operations, which we can break up as 16 operations plus the operations for the recursive call with the empty list.
- If we tried applying \(\text{English2cow}\) to a two-element list, we would get 39. We would have 16 operations for executing most of the body, plus 23 for the recursive call on the one-element list.

Maybe you can see the pattern here. The number of operations is 7 + 16\(n\).

Functions

The proper mathematical way to come up with such formulas is using recursively defined mathematical functions.

I’m not going to spend too much time on mathematical functions. We will focus on functions where the domain (the set of allowed inputs) is the set of nonnegative integers: \(\{0, 1, 2, \ldots\}\). A function is a way of specifying, for each domain element, exactly one “output”. You could represent a function by a table like this one:

<table>
<thead>
<tr>
<th>input</th>
<th>output</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>7</td>
</tr>
<tr>
<td>1</td>
<td>23</td>
</tr>
<tr>
<td>2</td>
<td>39</td>
</tr>
<tr>
<td>\vdots</td>
<td>\vdots</td>
</tr>
</tbody>
</table>

Of course, you wouldn’t want to write down the whole table because it would have an infinite number of rows!

The tradition in mathematics is to use a single letter to denote a function, often the letter \(f\).

How is a mathematical function different from a procedure?
• In the case of a mathematical function, there is an output corresponding to every legal input. This is not true of a procedure, as we will see.

• A procedure tells you how to compute the output from the input; a function does not.

• The notation for applying a function $f$ to an input, say 2, is $f(2)$. (However, many programming language use the same notation for procedure application.)

An aside: One thing that is common to both functions and to the procedures you write in CS17: for each, the output depends only on the input. This is not true for procedures in general; the main reason that what we are learning CS17 is called functional programming is this fact: there are no “hidden” inputs that influence the procedure.

Using a function to analyze English2cow

In this course, we will use functions to describe how many operations a procedure executes. The input to such a function will usually be the size of the procedure’s input. (For a procedure that takes a number as input, sometimes the input to the function is just the input to the procedure.)

Consider the analysis of English2cow

We start by writing the following:

“We define the function $f$ as follows.
Let $f(n)$ be the number of operations executed by English2cow when given an input of size $n$.”

I really want you to write these sentences when you do analyses. You need to convince me that you are clear on what the function represents.

Now that we have defined the function, we want to understand it better. We use what we know about English2cow to state some equations:

\[
\begin{align*}
f(0) &= 7 \\
f(1) &= 23 \\
f(2) &= 39
\end{align*}
\]

These equations tell us the output of $f$ for very specific input sizes. However, it doesn’t tell us the output for all input sizes, which is what we want. Instead, we write just two equations. The first one is $f(0) = 7$. The second one is more interesting:

\[
f(n) = 16 + f(n - 1) \text{ for } n > 0
\]

This is called a recurrence relation. The relation part just refers to the fact that it is an equation (we’ll see other kinds of relations soon). The recurrence part is a way of saying that the equation is recursive. In the equation, the function appears twice with two different (but related) inputs.

Just as recursion is a valid way to define a procedure, the equations

\[
\begin{align*}
f(0) &= 7 \\
f(n) &= 16 + f(n - 1) \text{ for } n > 1
\end{align*}
\]
completely define the function $f$.

Because the recursive structure of a procedure can be mimicked in the recursive structure of a recurrence relation, it is easy to go from procedure to recurrence relation. Let’s try some more.

### Analyzing double

```
(define double
  (lambda (L)
    (cond
      ((empty? L) empty)
      (#true
       (cons (car L) (cons (car L) (double (cdr L))))))))
```

“We define the function $f$ as follows.
Let $f(n)$ be the number of operations executed by double on an input of size $n$.”

First we write an equation derived from the procedure’s base case. This requires that we count the operations executed when the base case holds, i.e. (in this case) when the input list is empty. I counted 6.

$$f(0) = 6$$

Next we write an equation derived from the recursive case.

$$f(n) = 26 + f(n - 1)$$ for $n > 0$

### Analyzing parity

```
(define parity
  (lambda (n)
    (cond
      ((zero? n) (quote even))
      ((equal? (parity (- n 1)) (quote even)) (quote odd))
      ((equal? (parity (- n 1)) (quote odd)) (quote even)))))
```

“We define the function $f$ as follows.
Let $f(n)$ be the number of operations executed by parity on input $n$.”

Note the difference: “input $n$” versus “inputs of size $n$.”

First, the equation corresponding to the zero input:

$$f(0) = 6$$

There are two other cases. For $n$ even, the number of operations is $20 + f(n - 1)$.

For $n$ odd, the number of operations is $34 + f(n - 1) + f(n - 1)$.

So we write those as equations:

$$f(0) = 6$$
$$f(n) = 20 + f(n - 1)$$ for $n$ even, $n > 0$
$$f(n) = 34 + f(n - 1) + f(n - 1)$$ for $n$ odd, $n > 0$
Analyzing `even`

\[
\text{(define even?} \\
\quad \text{(lambda} \ (n) \\
\qquad \text{(cond} \\
\qquad \quad \text{((equal? n 0) #true)} \\
\qquad \quad \text{(#true (not (even? (- n 1))))}) \text{))}
\]

“We define the function \(f\) as follows.

Let \(f(n)\) be the number of operations executed by `even?` on input \(n\).”

In the case when \(n\) is zero, only 8 operations executed. Otherwise, 19 operations not including the recursive call.

\[
\begin{align*}
  f(0) &= 8 \\
  f(n) &= 19 + f(n - 1) \text{ for } n > 0
\end{align*}
\]

Solving a recurrence relation

The original reason for analyzing procedures is to predict how long they will take. Knowing a recurrence relation doesn’t by itself help. We want to derive a formula. There are a variety of techniques for this, including using power series and analysis of functions on complex numbers. For this class, you will be taught just a few basic techniques that address just a few basic recurrence relations.

Theorem: For any numbers \(a\) and \(b\), if

\[
\begin{align*}
  f(0) &= a \\
  f(n) &= b + f(n - 1) \text{ for } n > 0
\end{align*}
\]

then \(f(n) = a + bn\)

This tells us:

- `English2cow` takes \(7 + 16n\) operations on inputs of size \(n\).
- `double` takes \(6 + 26n\) operations on inputs of size \(n\).
- `even` takes \(8 + 19n\) operations on inputs of size \(n\).

Consider the graph of \(y = 6 + 26n\). You know from high school that this is a line. In general, for any numbers \(a\) and \(b\), the graph of \(y = a + bn\) is a line. For this reason, if there are numbers such that a procedure requires \(a + bn\) operations on inputs of size \(n\), we say the procedure is a linear-time algorithm.

Often having a linear-time procedure is a reason to celebrate. We’ll discuss this more later.

What about our `parity` procedure?
2 Summary

2.1 Another Look at Recursion

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.

2.2 Mathematical Functions

We can use mathematical functions to describe the speed of a recursive procedure. A recurrence relation can be written for procedure \((\texttt{foo x})\) as follows:

\[
f(n) = \begin{cases} 
    a & \text{if } n = 0 \\
    b + f(n - 1) & \text{if } n > 0
\end{cases}
\]

The blurb above the recurrence relation describing what it means is important to include. When writing your own recurrence relation, you should include this description, although you should alter the function name and what property of the input \( n \) represents. As you analyze more complicated procedures, the recurrence relations of these procedures may be more complicated.

**Theorem:** for any \( a, b \), if:

\[
\begin{align*}
    f(0) &= a \\
    f(n) &= b + f(n - 1) \text{ if } n > 0
\end{align*}
\]

then \( f(n) = a + bn \). If this is true about a procedure, then that procedure is a **linear time procedure**.

Keep in mind that being off by some number of operations does not matter. At the end of the day, the specific number of operations does not matter that much.

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).