Lecture 6: Programming methodology

CS 61A
Summer 2006
Instructor: Kevin Lin
July 5, 2006

Administrative stuff

- HW 1A, 1B due tonight!
- Project 1 due tomorrow night!
- Please fill out survey if you haven't already.
  - Thanks to those who filled out the survey during the last lecture. Your comments were very helpful to me!
- HW 2B will be posted tonight.
- Any questions on anything related to homework/projects?
- MIDTERM: FRIDAY, 7/14, 4-6 PM, 534 Davis

Plan for the next couple of lectures

- Efficiency review, iteration/recursion review.
- Programming methodology.
  - How to prove/verify that your programs are correct.
  - How to comment your code.
  - How to debug your code.
    - You'll see a lot of this in lab today.
- Data abstraction / hierarchical data

Efficiency review

We use Θ notation. (Greek letter "Theta".)

For our current purposes, assume that all built-in procedures require constant time (with the exception of keep and every). That is, they have order of growth in time Θ (1). (We will later see that this is not actually true in all cases, but it is true for all cases that we have considered so far.)

Efficiency review

Suppose the order of growth in time of proc1 is $\Theta(n^2)$, where n is its input, and the order of growth of proc2 is $\Theta(n^3)$, where n is its input. Then what is the order of growth of mystery?

```
(define (mystery n)
  (* (proc1 (- n 20))
     (proc2 (- n 205))))
```


Efficiency review

Suppose the order of growth in time of proc1 is $\Theta(n^2)$, where n is its input. Then what is the order of growth in time of the following procedure foo, in terms of its input m? (You may assume that `se` is Θ(1).)

```
(define (foo m)
  (define (nums x)
    (if (= x 0)
        '()
        (se m (nums (- x 1)))))
  (every proc1 (nums m)))
```

Iteration/recursion review

Write a procedure exp that takes two arguments a and b and returns the number a^b, that is, a multiplied by itself b times. Write this procedure so that it generates a recursive process.

```
(define (exp a b)
  (if (= b 0)
      1
      (* a (exp a (- b 1)))))

; What's the order of growth in time?
; In space?
```


Iteration/recursion review

Now rewrite exp so that it generates an iterative process.

```
(define (exp a b)
  (define (helper count result)
    (if (= count b)
        result
        (helper (+ count 1) (* a result))))
  (helper 0 1))

; What's the order of growth in time?
; In space?
```


Fast-exp

```
(define (fast-exp b n)
  (cond ((= n 0) 1)
        ((even? n) (square (fast-exp b (/ n 2))))
        (else (* b (fast-exp b (- n 1))))))
```

[Aside: This procedure is Theta($\log_2(n)$) time (the logarithm is base 2). You don't have to understand why. Compare the number of multiplications needed for (exp 2 6) versus (fast-exp 2 6).]

How can we prove that fast-exp always returns the correct answer?

The Principle of Induction

We can *prove* that fast-exp always returns the correct answer by a mathematical principle known as induction.

Suppose we are given a bunch of dominoes all lined up, and suppose we know the following two statements are true:

1. The first domino has been knocked down.
2. If the 1st, 2nd, 3rd, …, (k-1)st dominoes have all been knocked down, then the kth domino will be knocked down.

What can we conclude from these two statements?

That *all* dominoes will eventually be knocked down.

The Principle of Induction

The general principle states the following:

IF a proposition P_1 is true,
AND if whenever P_1, P_2, … P_{n-1} are true, then P_n is true.
THEN P_1, P_2, P_3, P_4, P_5, P_6, … are all true.

Fast-exp again

```
(define (fast-exp b n)
  (cond ((= n 0) 1)
        ((even? n) (square (fast-exp b (/ n 2))))
        (else (* b (fast-exp b (- n 1))))))
```

We can *prove* that fast-exp always returns the correct answer by induction.

When n=0, fast-exp clearly returns the correct answer. It is correct by definition.

Otherwise, now assume that fast-exp returns the correct answer for n=1, n=2, …, n=k-1. We want to verify that (fast-exp b k) gives the correct answer.

There are two cases. If k is even, then by assumption (fast-exp b (/ k 2)) gives the correct answer $b^{k/2}$. We return the square of that number, so (fast-exp b k) is $(b^{k/2})^2 = b^k$.

If k is odd, then by assumption (fast-exp b (- k 1)) gives the correct answer b^{k-1}. We return the product of that number and b, so (fast-exp b k) is $b(b^{k-1}) = b^k$, which is correct.

Loop invariants

Here's an iterative version of fast-exp:

```
(define (fast-exp base num)
  (define (iter a b n)
    (cond ((= n 0) a)
          ((even? n) (iter a (square b) (/ n 2)))
          (else (iter (* a b) b (- n 1)))))
  (iter 1 base num))
```

How can we prove that this version of fast-exp is correct? We use loop invariants.

Claim: A loop invariant of this version of fast-exp is $a*b^n$.

Loop invariants

```
(define (fast-exp base num)
  (define (iter a b n)
    (cond ((= n 0) a)
          ((even? n) (iter a (square b) (/ n 2)))
          (else (iter (* a b) b (- n 1)))))
  (iter 1 base num))
```

To see that $a*b^n$ is a loop invariant, we need to check that this quantity remains unchanged in all the invocations of iter for any specific problem.

If n is even, we square b and divide n by 2, so we have $a*(b^2)^{n/2} = a*b^n$.

If n is odd, then we have $(ab)*(b^{n-1}) = ab^n$.

Loop invariants

```
(define (fast-exp base num)
  (define (iter a b n)
    (cond ((= n 0) a)
          ((even? n) (iter a (square b) (/ n 2)))
          (else (iter (* a b) b (- n 1)))))
  (iter 1 base num))
```

So $a*b^n$ is a loop invariant.

We can use this to show that fast-exp is correct. Initially, $a*b^n = 1*base^{num} = base^{num}$.

At the end of the procedure, $a*b^n = a*b^0 = a$.

The loop invariant is constant, so the return value a is equal to $base^{num}$, our desired result.

Loop invariants

It only makes sense to talk about loop invariants of procedures that generate *iterative* processes.

More examples in homework.

Debugging: domain and range

The absolutely most important thing you can do in debugging is to keep straight the domain and range of each procedure you write. Keep this in mind during project 1!

Consider: make-adder.

```
(define (make-adder n) ; correct
  (lambda (x) (+ x n)))

(define (make-adder n x) ; incorrect
  (+ x n))
```


Debugging: domain and range

Where is the error in the following count procedure?

```
(define (count sent) ; incorrect
   (if (empty? sent)
       '()
       (+ 1 (count (butfirst sent)))))
```

The range of count is integers, not sentences!

Debugging: Tracing

How to detect the error in the previous incorrect version of count?

```
STk> (trace count)
okay
STk> (count '(hello there))
.. -> count with sent = (hello there)
.... -> count with sent = (there)
...... -> count with sent = ()
...... -> count returns ()
*** Error:
    +: not a number
Current eval stack:
--------------
 0 (apply fn (map maybe-num args))
 1 ...
STk>
```

19

Debugging: Read the error message!

How to detect the error in the previous incorrect version of count?

```
STk> (count '(hello there))
*** Error:
    +: not a number
Current eval stack:
----------
 0 (apply fn (map maybe-num args))
 1 (+ 1 (count (butfirst sent)))
STk>
```

So the error must be coming from trying to do addition on something that's not a number. The + procedure only appears in the recursive step. When does count ever return something that's not a number? Oops, the base case returns a sentence.

20

Debugging: Read the error message!

Here's another incorrect version of count.

```
(define (count sent)
  (if (empty? sent)
      0
      (+ 1 (count (butfirst snt)))))

STk> (count '(hello there))
*** Error:
    unbound variable: snt
Current eval stack:
...
```

This means that somewhere in your program you used the name snt. You can find instances of "snt" by eye, or you can use the Emacs search command.

21

Debugging: Replacement modeler, stkdb

You'll get practice with the replacement modeler and stkdb in lab.

I'll do a demonstration of the replacement modeler if we have time.

22