class 08 Sections M and R
February 20 and 27 2024

1.

We wrapped up last time by sketching the design of a downcounting exponentiation program, thinking that such a design might allow us to eliminate a parameter from expt-iter: if we start count at n and finish when count = 0, we no longer need the parameter n in expt-iter.

Recalling that a 'design idea', as we use the term in 335, consists in large part of a GI, we start today by giving a second development of a GI for this program.

Part of the design idea → what about a GI? We need a certain kind of relation which makes a connection between the program variables and the desired end result. Here – between count, result and b^n.

When the program stops, we want

$$result = b^n$$

when the program starts, we have count=n and we need to have a value for result which is 'easily' computed.

Suppose result = 1 to start. Then

$$b^n = ?? * result$$

So: $b^n = b^{count} * result$ is weak enough

If we make this b^{count} Then, on start, we would have

$$b^n = b^{count=n} * result^{=1}$$

Moreover, when count=0, this equation would give us

$$b^n = b^0 * result$$

ie, $result = b^n$.

and it is strong enough

b and n are fixed

The only thing left to check is whether it is preservable? The plan is to decrement count — and so we need to adjust result to keep the equation true. Clearly, if

$$b^n = b^{count} * result$$

then also

$$b^n = b^{count-1} * (b * result)$$

So: we just need to multiply result by b.

So we have guess-code for an alternate design:

Note: the n parameter has been removed.

```
(define (expt-iter b count result)
  (cond ((zero? count) result)
        (else (expt-iter b (- count 1)
                         (* b result)))))
```

With this design there is no need to compare and n.

Then test ... and you're done unless the test turns up problems.

Let's look now at an iterative program for adding two numbers a and b.

We'll assume $a \geq 0$ is an integer [but you will certainly want to see about removing $a \geq 0$ restriction]

Design idea: addition by repeated incrementation. We will start with a and b, and (simultaneously) decrement a and increment b until $a = 0$.

Since we decrement a at the same time as we increment b, a likely GI is:

$$a + b \text{ is constant}$$

How does this help? If we had a convenient way of referring to the initial values of a and b —

"ghost variables" say as $\overrightarrow{A}$ and $\overrightarrow{B}$, respectively

Then we could write

$$A + B = a + b$$

ghost variables never change - here they are just names for the initial values of a and b

Why → strong enough? On termination, our plan is that $a = 0$. So the eqn then implies

$$A + B = 0 + b = b$$

and we could simply return b to get the desired sum.

→ weak enough? On start, $a = A$ and $b = B$. So certainly

$$A + B = a + b$$

→ preservable?

$A + B = a + b \Longrightarrow$

$A + B = (a - 1) + (b + 1)$

So — bang! — we have code

```
(define (myplus a b)
  (cond ((zero? a) b)
        (else (myplus (- a 1)
                      (+ b 1)))))
```

What about the recursive procedure for the same task, given as myplus-version-1 in A&S?

How could we use a divide and conquer strategy to solve the problem?

→ how will we break the problem up?

$$a + b = 1 + ((a-1) + b)$$

"smaller problem of the same kind"

Tentative IH

adjustment necessary to leverage the result from the smaller problem to solve the entire problem

Tentative IS →

Tentative Basis: $(0 + b) = b$

Terminating When $a = 0$

It appears we can organize the construction and proof as an induction on a

Reality check: is a something which can be inducted on? Well yes - $a \geq 0$ is an integer.

So we're ready for guess code —

```
(define (myadd a b)
  (cond ((zero? a) b)
        (else (+ 1 (myadd (- a 1) b)))
  ))
```

IH: assume this computes $(a-1)+b$ provided: the precondition holds at the time of the call

WE TS: we add 1 to $(a-1)+b$