Class 05 February 6 and 8 2024

Recursion & Iteration

① The Visual Distinction Between Recursion and Iteration

(Recursive)

```
(define (fact x)
  (cond ((= x 0) 1)
        (else (* x (fact (- x 1))))))
```

function calls itself, hence syntactically recursive

This mult. op. is said to guard the call to fact

Both are syntactically recursive

wrapper only

```
(define (new-fact x)
  (fact-iter x 0 1))
```

(Iterative)

```
(define (fact-iter x count result)
  (if (= count x)
    result
    (fact-iter x (+ count 1) (* (+ count 1) result))))
```

The call to fact-iter is unguarded.

We need to talk about what constitutes a guard!

② looking more closely, however, you spot a difference — the recursive call to fact is guarded. The guard in fact is pointed at — The cond is NOT counted as a guard. Similarly, the if in fact-iter is not a guard.

Let me note that iterative processes are usually referred to as tail-recursive.

Why is this distinction important?

(3) Operational Difference Between Recursion and Iteration

(consequence of the guard or its absence)

Let's call this the "purely recursive" version

```
(define (fact x)
  (cond ((= x 0) 1)
        (else (* x (fact (- x 1))))))
```

The procedure

The process whose evolution is controlled by the procedure

```
; (fact 6)
; (* 6 (fact 5))
; (* 6 (* 5 (fact 4)))
; (* 6 (* 5 (* 4 (fact 3))))
; (* 6 (* 5 (* 4 (* 3 (fact 2)))))
; (* 6 (* 5 (* 4 (* 3 (* 2 (fact 1))))))
; (* 6 (* 5 (* 4 (* 3 (* 2 (* 1 (fact 0)))))))
; (* 6 (* 5 (* 4 (* 3 (* 2 (* 1 1))))))
; (* 6 (* 5 (* 4 (* 3 (* 2 1)))))
; (* 6 (* 5 (* 4 (* 3 2))))
; (* 6 (* 5 (* 4 6)))
; (* 6 (* 5 24))
; (* 6 120)
; 720
```

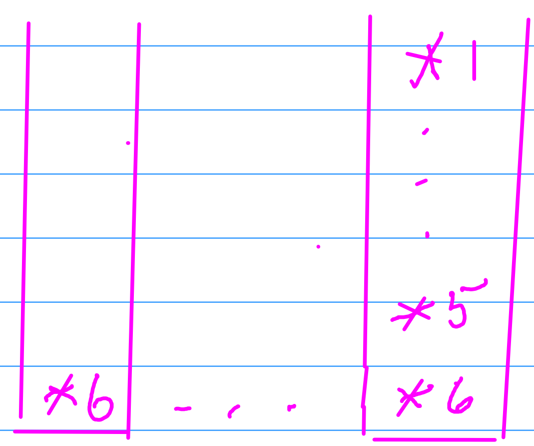
necessarily deferred until (fact 5) is a number

pattern of calls expansion for (fact 6)

deferred (pushed onto the stack) until (fact 4) is computed

The stack is growing

The stack contracts



stack initially

stack at end.

proc. call stack

"mult. promises"

Once (fact 0) returns without any further calls the system can unwind the stack of carrying out of the deferred multiplications — popping each in turn.

(I will sometimes use the phrases "proper recursion" "pure recursion" where the text uses just "recursion")

From the pattern of calls expansion, one sees that (fact n) requires $\Theta(n)$ time (for n multiplications) and also $\Theta(n)$ space (for the stack)

Yes, we will be making use of your background in algorithms (CSC 220 at CCNY)

In contrast, the iterative factorial requires $\Theta(n)$ time but only constant space.

iterative = tail recursive

init vals

```
(define (new-fact x)
  (fact-iter x 0 1))

(define (fact-iter x count result)
  (if (= count x)
      result
      (fact-iter x (+ count 1) (* (+ count 1) result))))
```

remember! computing values, not updating locations

```
; here we see that the work is done by updating the values of count and
; result

;; (new-fact 6)
;; (fact-iter 6 0 1)
;; (fact-iter 6 1 1)
;; (fact-iter 6 2 2)
;; (fact-iter 6 3 6)
;; (fact-iter 6 4 24)
;; (fact-iter 6 5 120)
;; (fact-iter 6 6 720)
```

x count result

explains why this is not just count

will come back to the issue of referential transparency (in the context of procedure calls)

You might want to think of this as a while loop:
while (count < x)
{increment count;
adjust result}

This assumes assignment!

to help understand fact-iter, compare and contrast to the while loop version

Pattern of calls expansion
note: No expansion and contraction.

In fact, the stack is not involved in this computation:

All the work done by the program is done in updating the parameters

No deferred operations!

The constant space required is just that needed for the parameters (up to θ)

Another way of understanding the difference between (proper, pure...) recursion and iteration (tail recursion) comes if you ask "what data needs to be preserved if each process is to be interrupted and then resumed later?"

[perhaps by the operating system]

For fact — one would need to save the instruction pointer as well as the entire call stack. But for fact-iter, the answer is just the instruction pointer and the current values of the parameters.

So it is usually desirable to have iterative programs.

But you will soon discover that it is dramatically easier to write recursive programs than to write iterative programs.

Since human time is almost always the really expensive part of software .... one wants to understand recursion as well as iteration.

Think: rapid prototyping

(As an aside — gcc has a switch allowing users to optimize for tail recursion; smart enough to avoid using a stack when this is unnecessary)

4 Certification Difference Between Recursive and Iterative Programs

But wait / why bother with proofs/certifications? After all, one can try fact on a few values, and see clearly that it works! Right?

Let's have a look → over to DrRacket. We can see in a minute that this simple program can compute some HUGE numbers — it is clearly NOT enough to say

(as is all too often done)
"well, it works on 0, ----, it works on 6 → therefore it works".

What we can gain from program proving is greatly increased confidence that our programs work.

For the fact program — there are a number of distinct factors which might make for errors

* ① → the algorithm might be wrong

② → The BigNum library implementing Schemes infinite precision integer arithmetic might be flawed

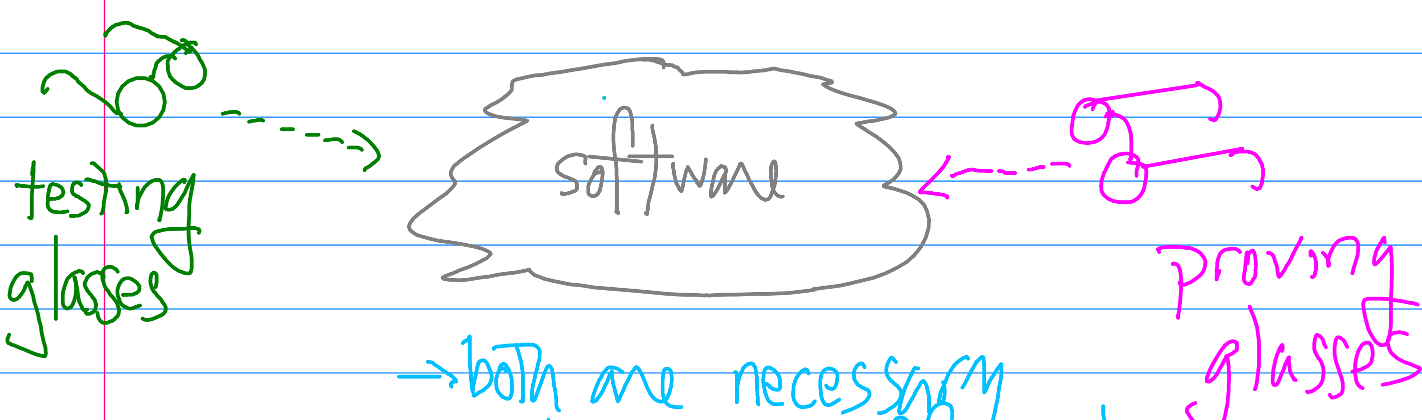
③ → scheme itself might be flawed

④ → The operating system might be flawed

⑤ → The chips might be flawed

We will assume 2 ... 5 do not occur — everything but the algorithm will be assumed to work correctly.

What we'll get



→ both are necessary
→ neither is sufficient

Let's take another look at The proper recursive version of Factorial.

Start from the (math) definition of The factorial:

$$n! = \begin{cases} 1 & \text{if } n = 0 \\ n * (n-1)! & \text{otherwise} \end{cases}$$

Assumptions: $n \geq 0$ and n is an integer

Why isn't this a circular definition? After all, it defines ! in terms of ! The reason is that this is an inductive definition:

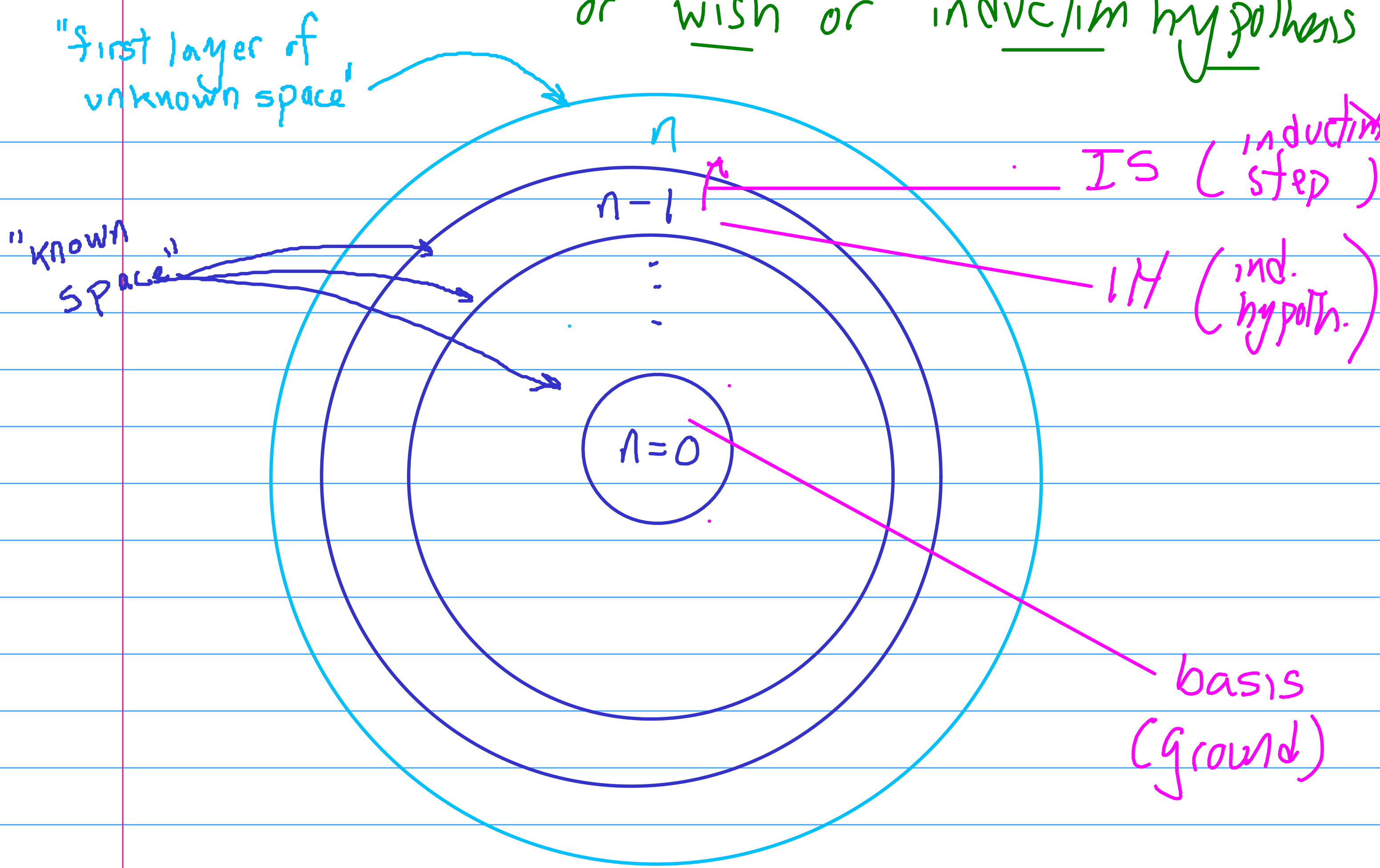
circularity is avoided by making good use of The natural metric on non-neg. integers

ie — $n!$ is defined in terms of ! applied to something smaller than n — ie — $(n-1)$.

We have The ground — or basis — case (always Those cases for which the induction is not necessary).

The working assumption is That $(n-1)!$ is defined

see This as a simple example of divide and conquer — To get The factorial of a big input, we get The fact of a smaller input & use a multiplication 'or <u>wish</u> or <u>induction</u> <u>hypothesis</u>



The scheme version is just a recasting of the function definition

The goal of a pf in this case is to show that our program fact correctly computes factorial - ie - for all $n \in \mathbb{N}$, (fact n) = n!

```
; pre: n ≥ 0 ∧ n is an integer
(define (fact n)
  (cond ((zero? n) 1)
        (else (* n (fact (- n 1))))
  ))
; post: (fact n) = n!
```

The IH is that this call correctly computes (n-1)!

A correctness proof in this course is

given in two parts:

→ partial correctness

"if the program terminates, it gives the correct answer"

This is where we deploy induction →

If both are true then one has total correctness

→ Termination argument

"The program really does return an answer"

will usually be satisfied with a hand-waving argument

in this case: the program starts with input $n \geq 0$. Because n is an integer, 1 can be subtracted only finitely many times before reaching 0, at which point the execution halts.

Assuming that Scheme works as advertized

The induction argument

→ first: what are we inducting on?

We'll induct on n

induction on n depends on $n \geq 0$ being an integer

if we're doing induction, we need a well-founded set; there can be no infinite descending chains.

→ one cannot induct on real numbers

→ one cannot induct on the set of all integers

So we can't induct on

... -10 -9 -8 .... -1 0 1 2 ...

← there are infinite descending chains

What is an example of an infinite descending chain of real numbers between 0 and 1?

→ second: how are we decomposing the problem? Essentially: what is the IH?

For most programs, the IH — to a first approximation — is "we assume that the recursive call works"

This isn't quite enough — a better approximation is

This wish is really the source of power of the technique

"we assume that if the precond is satisfied, then the recursive call works"

```
; pre: n ≥ 0 ∧ n is an integer
(define (fact n)
  (cond ((zero? n) 1)
        (else (* n (fact (- n 1))))
  ))
; post: fact n = n!
```

is it true that $n-1 \geq 0$ when control reaches the recursive call?

let's see: we know ① $n \geq 0$ to start ② (zero? n) is false

hence: $n > 0$ hence $n-1 \geq 0$

(we'd also need to say that n-1 is an integer, but this is immediate from the fact that n is an integer.

→ Third: we need to give the induction step — ie — we need to show that the program does the right thing with the value $(n-1)!$ returned by the call.
Here $(n-1)!$ is multiplied by n — which, according to the def of the factorial function — gives $n!$

It is important to observe what <u>isn't</u> done in this argument — as well as to observe what <u>is</u> done.

Note that there is NOTHING like the pattern of calls expansion — nothing like

(fact n) calls

(fact (- n 1))

calls

(fact (- n 2))

. . .

and so on until 0

Not an induction!

Here's the giveaway: the <u>ellipsis</u>

Rule of Thumb

if your argument proceeds by

"unwinding" the recursion to an indefinite extent [as shown by the ellipsis] then it is NOT an induction.

Induction — properly used — is the most powerful software engineering tool that exists. One simple argument replaces an unbounded number of unwindings. The induction argument MUST make use of the induction hypothesis — the unwinding args have no induction hypothesis

In the context of software engineering
the IH amounts to a black box
which is assumed to magically
solve the smaller problem.