CSCI 275: Programming Abstractions

Lecture 8: Tail Recursion & Higher Order Start

Fall 2024

Stephen Checkoway, **Oberlin College**
Slides gratefully borrowed from Molly Q Feldman

Functional Language of the Week: Elm

- Purely functional language for reactive web programming
- Benefit is static types, so very few web runtime errors
- Began in 2012 (new!)
- “Outside of industry” origins: Undergraduate thesis and then used/sponsored work from companies
  - https://elm-lang.org/assets/papers/concurrent-frp.pdf
- Best known for a strong user community & the best error messages of any programming language!
- Fun fact: no generic map ☺

Functional Language of the Week: Elm

```
Detected errors in 1 module.


-- ALIAS PROBLEM ------------------------------------------------ ./Maze.elm

This type alias is recursive, forming an infinite type!

21|>type alias Node =
22|>    { x : Int
23|>    , y : Int
24|>    , children : List Node
25|>    }

When I expand a recursive type alias, it just keeps getting bigger and bigger.
So dealiasing results in an infinitely large type! Try this instead:

    type Node
        = Node { x : Int, y : Int, children : List Node }

This is kind of a subtle distinction. I suggested the naive fix, but you can
often do something a bit nicer. So I would recommend reading more at:
<https://github.com/elm-lang/elm-compiler/blob/0.17.0/hints/recursive-alias.md>
```

https://twitter.com/GregorySchier/status/732830868562182144/photo/1

Functional Language of the Week: Elm

Recommended reading about learning a new language and some of the ways Elm has recently addressed it!

https://elm-lang.org/news/the-syntax-cliff

Tail Recursion, or how to be efficient

Loops and efficiency

Compare a C (or Java) function to compute the factorial

```
int fact(int n) {
  int product = 1;
  while (n > 0) {
    product *= n;
    n -= 1;
  }
  return product;
}
```

versus a recursive Racket implementation

```
(define (fact n)
  (if (<= n 1)
      1
      (* n
         (fact (- n 1)))))
```

How do these differ?
Specifically think about the *number of function calls.*

What does tail recursion mean?

A function is tail-recursive if **the last thing it does in the recursive case(s) is to recurse** and return the result of that recursion

Example:

```
(define (foo x y)
  (if (zero? x)
    y
    (foo (sub1 x) (+ x y))))
```

When the condition is satisfied, `y` is returned, otherwise `foo` is called again with some different parameters and that value is returned

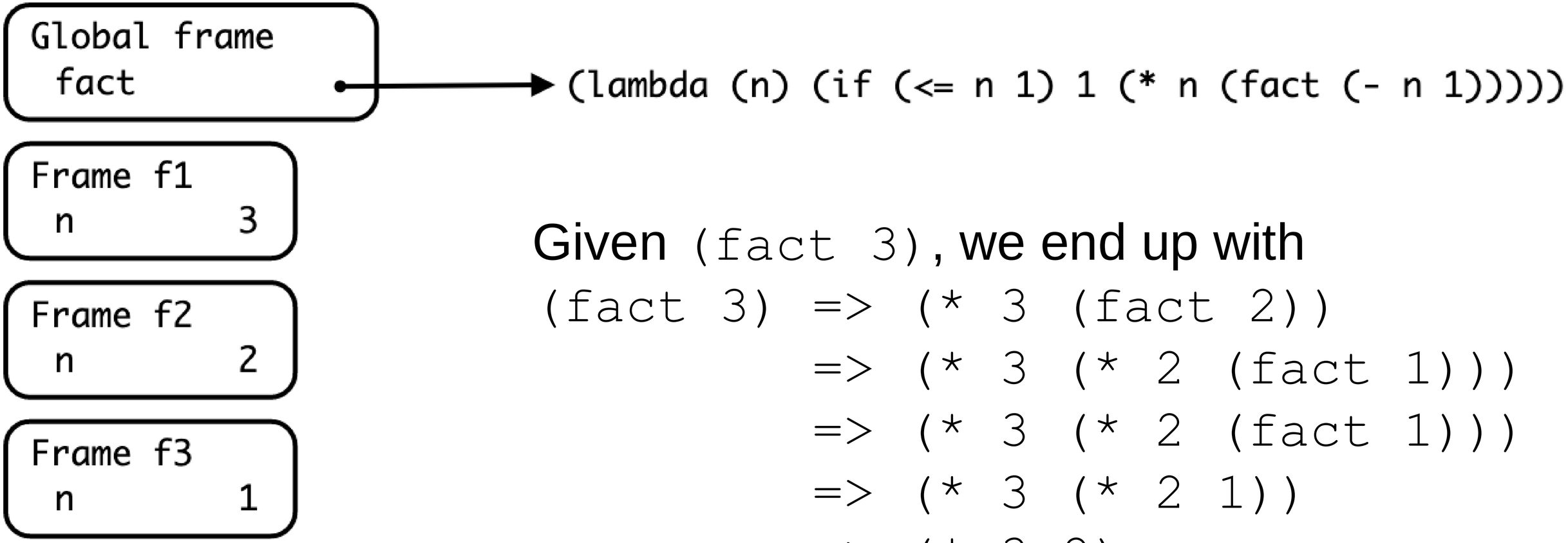
To be efficient, Racket internally converts all **tail-recursions** into loops

Our factorial example is *not* tail recursive

```
(define (fact n)
  (if (<= n 1)
      1
      (* n (fact (- n 1)))))
```

The last thing fact does is perform a multiplication

The recursion happens *before* the multiplication



Given `(fact 3)`, we end up with

```
(fact 3) => (* 3 (fact 2))
         => (* 3 (* 2 (fact 1)))
         => (* 3 (* 2 (fact 1)))
         => (* 3 (* 2 1))
         => (* 3 2)
         => 6
```

Our factorial is *not* tail recursive

Is this procedure tail recursive?

```
(define (length lst)
  (cond [(empty? lst) 0]
        [else (+ 1 (length (rest lst)))]))
```

A. Yes

B. No

C. It depends on how long the list is

Solution: Use an "accumulator"

```
(define (fact-a n acc)
  (if (<= n 1)
      acc ; return the accumulator
      (fact-a (sub1 n) (* n acc))))


(define (fact2 n)
  (fact-a n 1))
```

Four things to notice:

- We defined a recursive helper function that takes an additional param
- We provide an initial value for the accumulator in `fact2`'s call to `fact-a`
- The base case returns the accumulator
- `fact-a` is tail-recursive

fact2 is tail-recursive

```
(define (fact-a n acc)
  (if (<= n 1)
      acc ; return the accumulator
      (fact-a (sub1 n) (* n acc))))

(define (fact2 n)
  (fact-a n 1))
```

```
(fact2 4) => (fact-a 4 1)
          => (fact-a 3 4)
          => (fact-a 2 12)
          => (fact-a 1 24)
          => 24
```

BTW: we can use `letrec` instead of two defines

```
(define (fact-a n acc)
  (if (<= n 1)
      acc ; return the accumulator
      (fact-a (sub1 n) (* n acc))))

(define (fact2 n)
  (fact-a n 1))

(define (fact-3 n)
  (letrec ([fact-a (lambda (n acc)
                     (if (<= n 1)
                         acc
                         (fact-a (sub1 n) (* n acc))))])
    (fact-a n 1)))
```

Benefit: fact-a is *not* exportable! It's a "private" definition.

Practice! Some other tail-recursive procedures

`(sum lst)` — Add all the numbers in the `lst`

`(reverse lst)` — Reverses the list `lst`

`(remove* x lst)` — Remove all instances of x from `lst`

As an exercise for the reader:

`(remove x lst)` — Remove the first instance of x from `lst`

Is this procedure tail recursive?

```
; Return the nth element of lst

(define (list-ref lst n)
  (cond [(empty? lst)
         (error 'list-ref "List too short")]
        [(zero? n) (first lst)]
        [else (list-ref (rest lst) (sub1 n))]))
```

How to throw errors in Racket

A. Yes

B. No

C. I have no idea!

“What’s the point?”

- There are numerous ways to solve computational problems
- Language design and features allow us to solve problems differently (or more easily)
- *Pattern matching* in CS
- These are all tools in your toolbox
  - e.g. iteration, recursion, tail recursion

So how does this become a loop?

Use variables for the parameters and update them each time through the loop

```
(define (fact-a n acc)
  (if (<= n 1)
      acc ; return the accumulator
      (fact-a (sub1 n) (* n acc))))
```

becomes (pseudocode)

```
def fact-a(n, acc):
  loop:
    if n <= 1:
      return acc
    n, acc = n - 1, n * acc
```

Another tool: map

Motivation

You have a list of data `lst` and you have a procedure `f` and you want to call `f` on every element of `lst`, getting a new list back containing the results

E.g., you have `'(1 2 3)` and you want

```
(list (f 1) (f 2) (f 3))
```

Example: Changing HTTP to HTTPS

Imagine we had a list of URLs like

```
(define urls
  `("http://cs.oberlin.edu"
    "http://thelocal.se"
    "http://duckduckgo.com"))
```

and we want to change them all to secure HTTP (`https://`) URLs

```
  `("https://cs.oberlin.edu"
    "https://thelocal.se"
    "https://duckduckgo.com")
```

we could write a procedure turn a list of URLs into a list of different URLs

Example: Changing HTTP to HTTPS

```
(define (securify lst)
  (cond [(empty? lst) lst]
        [else
         (cons (string-replace (first lst) "http" "https")
                               (securify (rest lst)))]))
```

Example: List of courses

We have a list of courses (represented as a list) like

```
(define COURSES
  '((CSCI 150 "Professor Emily")
    (CSCI 151 "Professor Eck")
    (CSCI 210 "Professor Cynthia")
    (MATH 220 "Professor Bosch")))
```

and we want just a list of course numbers `'(150 151 241 220)`

We can write a procedure to turn a list of courses into a list of numbers

Example: List of courses

```
(define (course-numbers lst)
  (cond [(empty? lst) empty]
        [else (let* ([course (first lst)]
                     [num (second course)]
                     [others (course-numbers (rest lst))])
                (cons num others))]))
```

What similarities did you notice between the previous examples?

Similarities

In each case, we have a list of elements of type α

We have an operation we want to apply that takes a value of type α and returns a value of type β

We want to *apply that operation to each element of our list* to get a list of elements of type β

URLs: α = http URL, β = https URL (both were strings here)
Courses: α = course (as a list), β = number

Similarities

In each case, we have:

A list of α

An operation $\alpha \rightarrow \beta$

And our output is a list of β

```
(define (NAME lst)
  (cond [(empty? lst) empty]
        [else (cons (SOMETHING APPLIED TO FIRST)
                    (NAME (rest lst)))]))
```

Enter: Map `(map proc lst)`

`map` calls the procedure `proc` on every element in list `lst`

```
(map f '(1 2 3 4)) =>
     (list (f 1) (f 2) (f 3) (f 4))

(map sub1 '(10 15 20)) =>
       '(9 14 19)

(map (lambda (x) (list x x)) '(a b c))
       => '((a a) (b b) (c c))

(map first '((a 5) (b 6) (c 7))) => '(a b c)
```

Rewriting our examples with map

```
(define (securify lst)
  (map (lambda (url)
         (string-replace url “http” “https”))
       lst))

(define (course-numbers lst)
  (map second lst))
```

What is the result of this?

```
(map rest '((a 5) (b 6) (c 7)))
```

A.'((5) (6) (7))

B.'(5 6 7)

C.'((b 6) (c 7))

D.'(5) '(6) '(7)

E.'(b c)

What is the result of this?

```
(map (lambda (lst) (cons (first lst) lst))
     '((1 2) (3 4)))
```

A.'(1 3)

B.'((1 1 2) (3 3 4))

C.'((1 (1 2)) (3 (3 4)))

D.'((1 4) (2 3))

E.'((1 3) (2 4))

Next Up

Reminder about readings as another resource!

Homework 2 is due **Friday** at 11:59pm via Github