CS 61A Topical Review

Tail Recursion

Albert Xu

Slides: albertxu.xyz/teaching/cs61a/

The Cost of Recursion

ain't no free lunch

```
(define (factorial n)
  (if (= n 0)
    1
    (* n (factorial (- n 1)))))
```

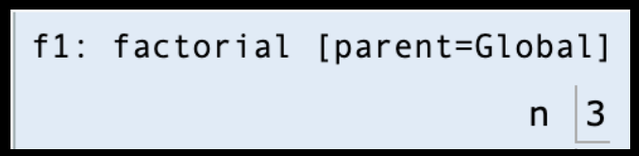
```
scm> (factorial 3)
```

The Cost of Recursion

ain't no free lunch

```
(define (factorial n)
  (if (= n 0)
    1
    (* n (factorial (- n 1)))))
```

```
scm> (factorial 3)
```

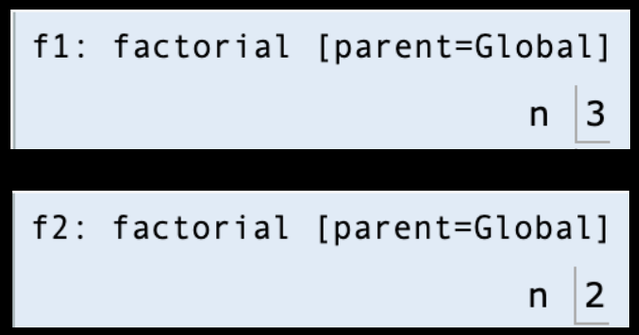


The Cost of Recursion

ain't no free lunch

```
(define (factorial n)
  (if (= n 0)
    1
    (* n (factorial (- n 1)))))
```

```
scm> (factorial 3)
```

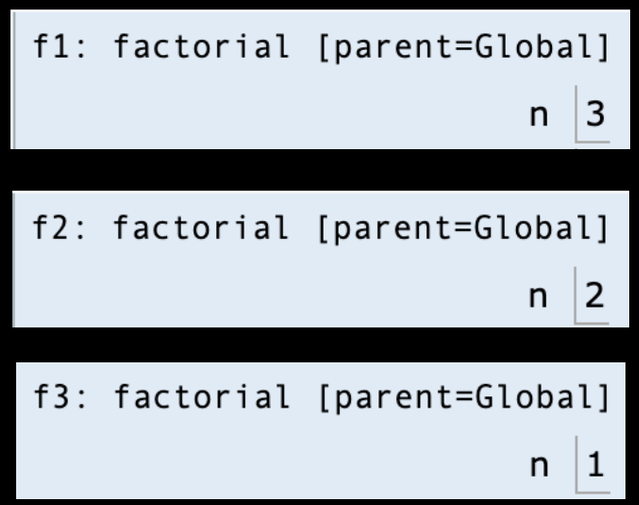


The Cost of Recursion

ain't no free lunch

```
(define (factorial n)
  (if (= n 0)
    1
    (* n (factorial (- n 1)))))
```

```
scm> (factorial 3)
```

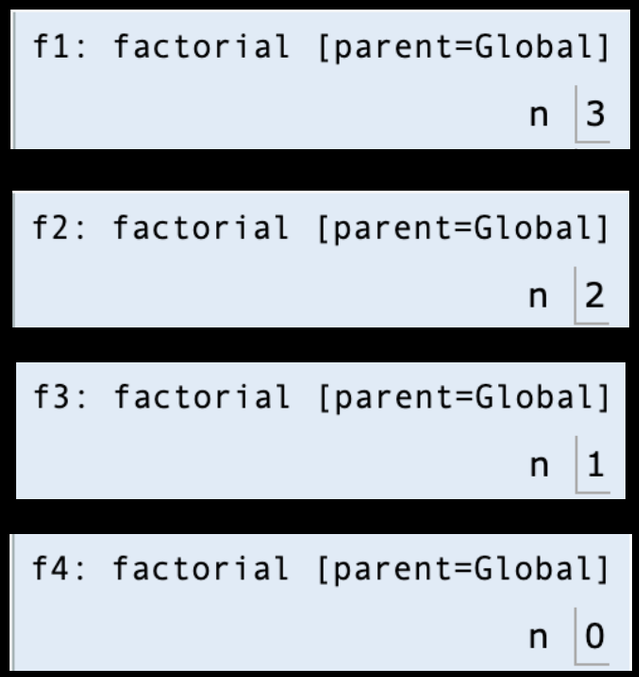


The Cost of Recursion

ain't no free lunch

```
(define (factorial n)
  (if (= n 0)
    1
    (* n (factorial (- n 1)))))
```

```
scm> (factorial 3)
```

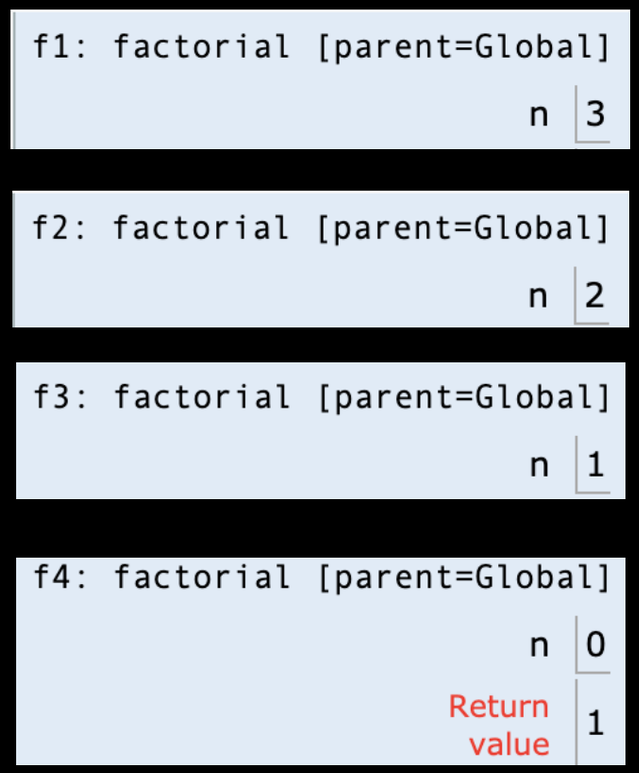


The Cost of Recursion

ain't no free lunch

```
(define (factorial n)
  (if (= n 0)
    1
    (* n (factorial (- n 1)))))
```

```
scm> (factorial 3)
```

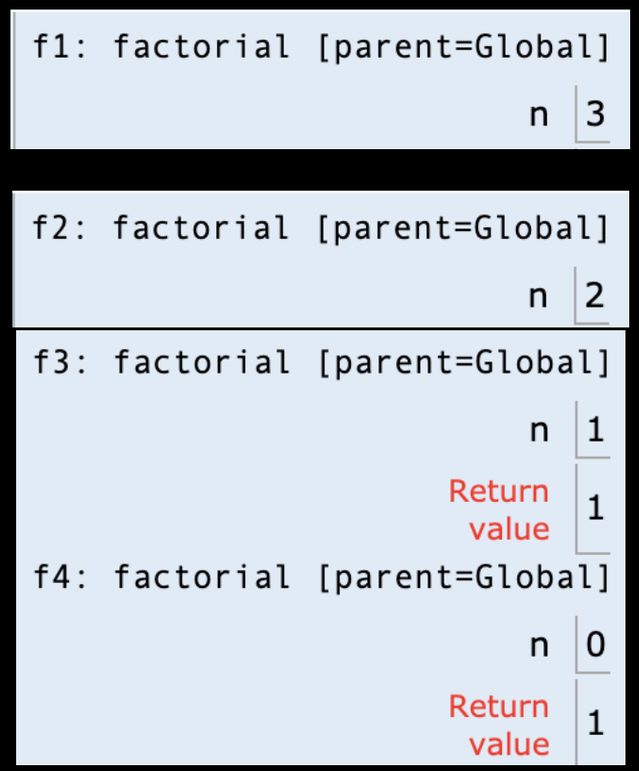


The Cost of Recursion

ain't no free lunch

```
(define (factorial n)
  (if (= n 0)
    1
    (* n (factorial (- n 1)))))
```

```
scm> (factorial 3)
```

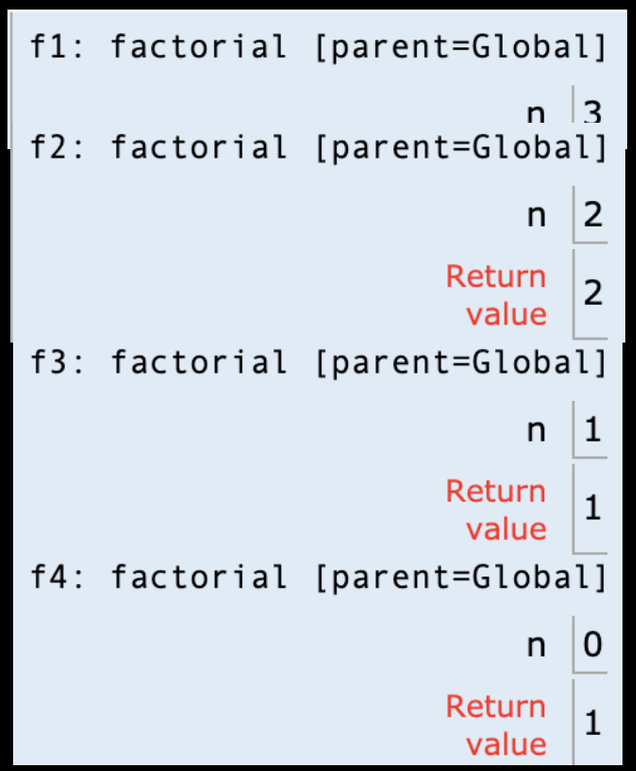


The Cost of Recursion

ain't no free lunch

```
(define (factorial n)
  (if (= n 0)
    1
    (* n (factorial (- n 1)))))
```

```
scm> (factorial 3)
```

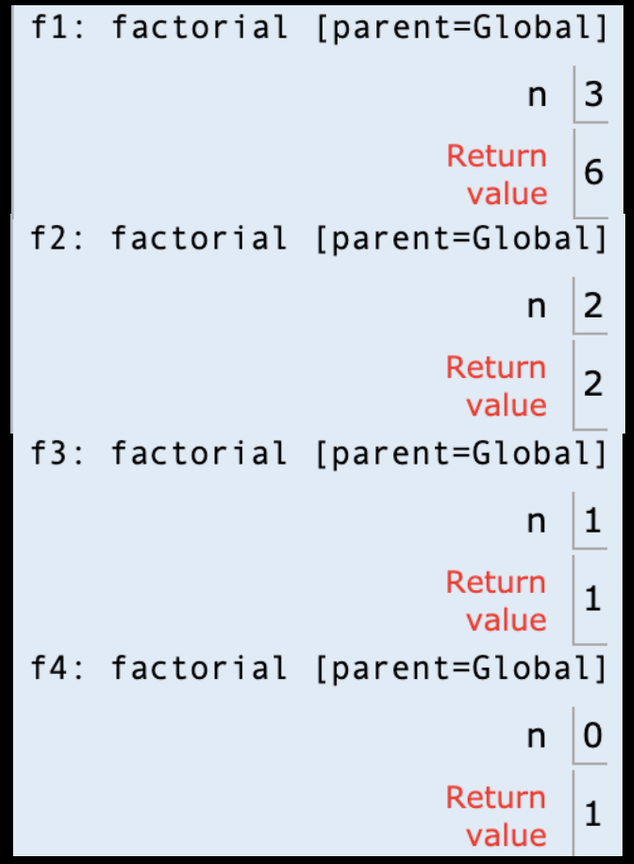


The Cost of Recursion

ain't no free lunch

```
(define (factorial n)
  (if (= n 0)
    1
    (* n (factorial (- n 1)))))
```

```
scm> (factorial 3)
```



The Cost of Recursion

ain't no free lunch

```
(define (factorial n)
  (if (= n 0)
    1
    (* n (factorial (- n 1)))))
```

```
scm> (factorial 3)
```

```
f1: factorial [parent=Global]
                        n | 3
                   Return | 6
                    value |
f2: factorial [parent=Global]
                        n | 2
                   Return | 2
                    value |
f3: factorial [parent=Global]
                        n | 1
                   Return | 1
                    value |
f4: factorial [parent=Global]
                        n | 0
                   Return | 1
                    value |
```

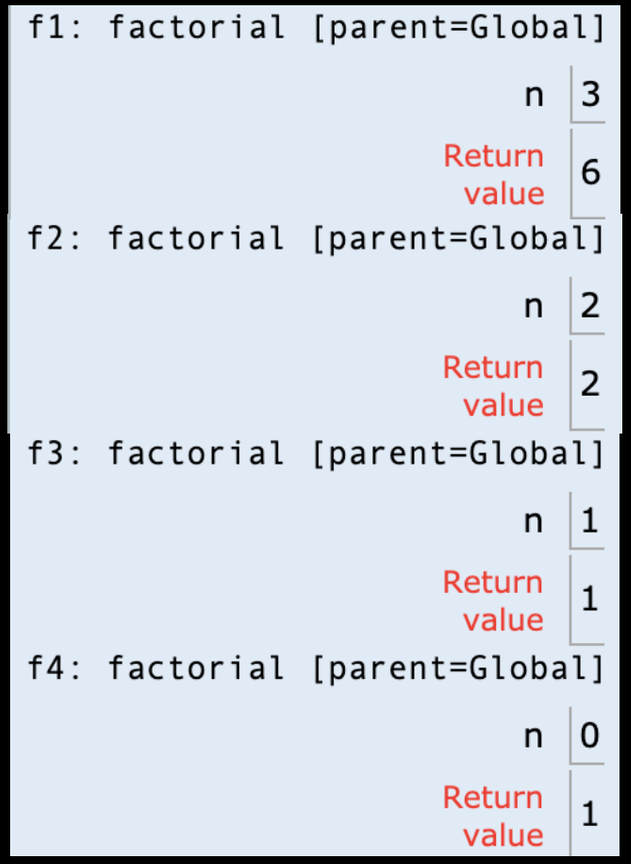
```
scm> (factorial 10)
```

The Cost of Recursion

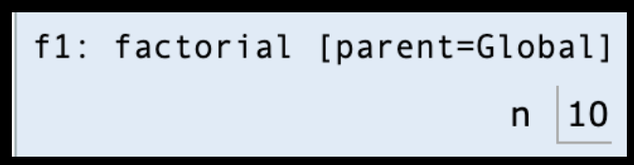
ain't no free lunch

```
(define (factorial n)
  (if (= n 0)
    1
    (* n (factorial (- n 1)))))
```

```
scm> (factorial 3)
```




```
scm> (factorial 10)
```

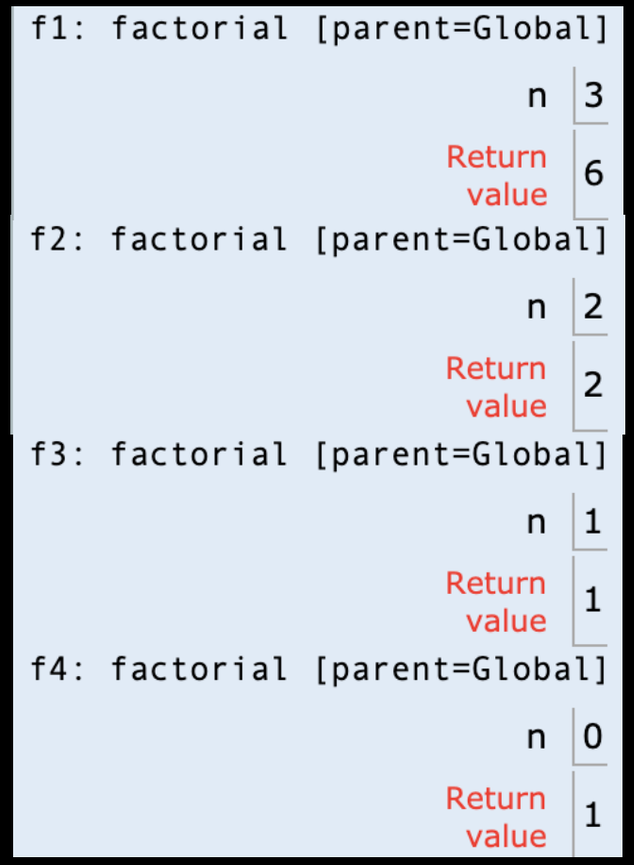


The Cost of Recursion

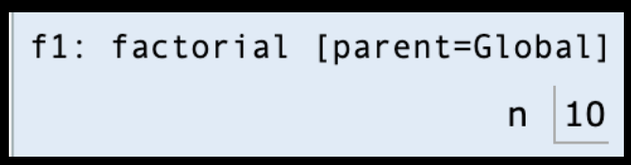
ain't no free lunch

```
(define (factorial n)
  (if (= n 0)
    1
    (* n (factorial (- n 1)))))
```

```
scm> (factorial 3)
```




```
scm> (factorial 10)
```



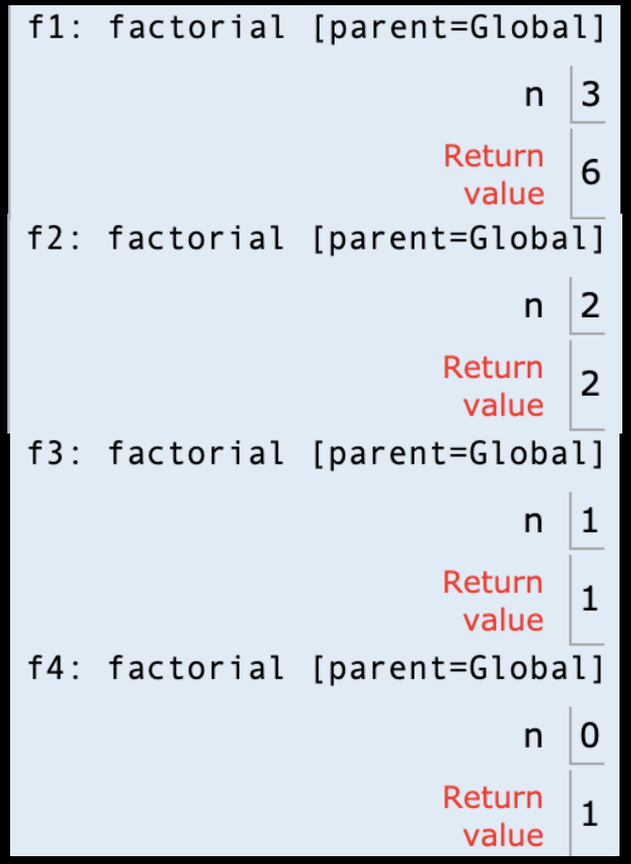
...10 frames!

The Cost of Recursion

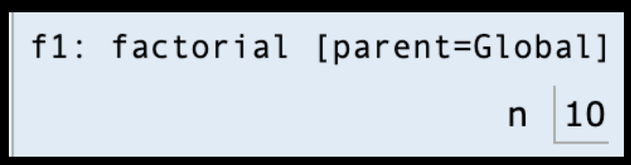
ain't no free lunch

```
(define (factorial n)
  (if (= n 0)
    1
    (* n (factorial (- n 1)))))
```

```
scm> (factorial 3)
```




```
scm> (factorial 10)
```



...10 frames!

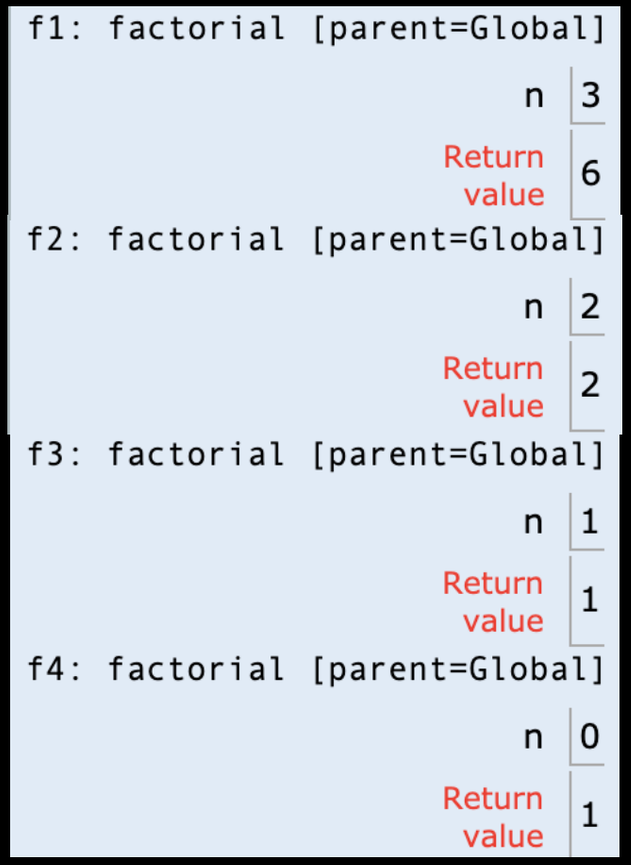
How much memory does `factorial` take? Use big-theta notation!

The Cost of Recursion

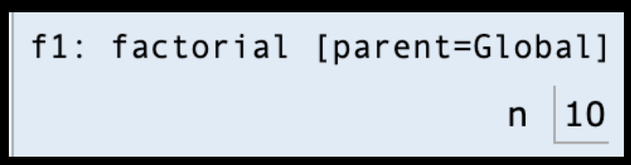
ain't no free lunch

```
(define (factorial n)
  (if (= n 0)
    1
    (* n (factorial (- n 1)))))
```

```
scm> (factorial 3)
```




```
scm> (factorial 10)
```



...10 frames!

How much memory does `factorial` take? Use big-theta notation! $\Theta(n)$

Tail Call Optimization

ain't no free lunch

```
(define (factorial n sofar)
  (if (= n 0)
    sofar
    (factorial (- n 1) (* sofar n)))))
```

Tail Call Optimization

ain’t no free lunch

```
(define (factorial n sofar)
  (if (= n 0)
    sofar
    (factorial (- n 1) (* sofar n)))))
```

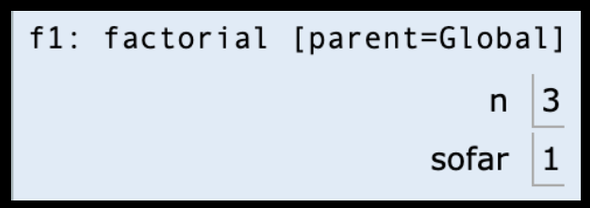
```
scm> (factorial 3 1)
```

Tail Call Optimization

ain’t no free lunch

```
(define (factorial n sofar)
  (if (= n 0)
    sofar
    (factorial (- n 1) (* sofar n)))))
```

```
scm> (factorial 3 1)
```

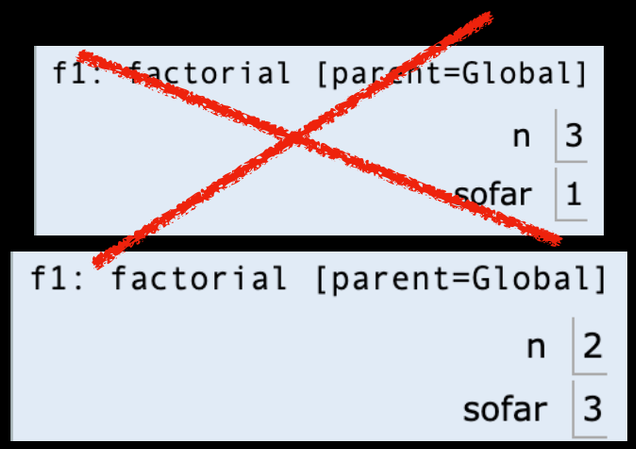


Tail Call Optimization

ain't no free lunch

```
(define (factorial n sofar)
  (if (= n 0)
    sofar
    (factorial (- n 1) (* sofar n)))))

scm> (factorial 3 1)
```

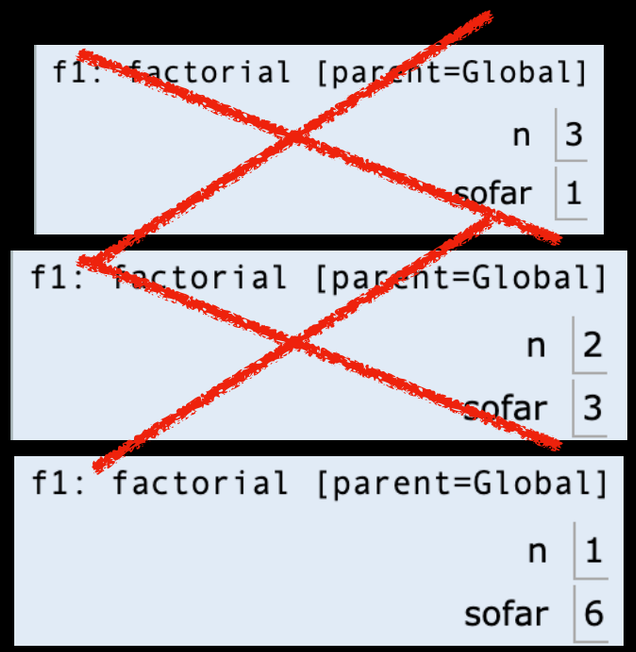


Tail Call Optimization

ain't no free lunch

```
(define (factorial n sofar)
  (if (= n 0)
    sofar
    (factorial (- n 1) (* sofar n)))))
```

```
scm> (factorial 3 1)
```



Tail Call Optimization

ain't no free lunch

```
(define (factorial n sofar)
  (if (= n 0)
    sofar
    (factorial (- n 1) (* sofar n)))))
```

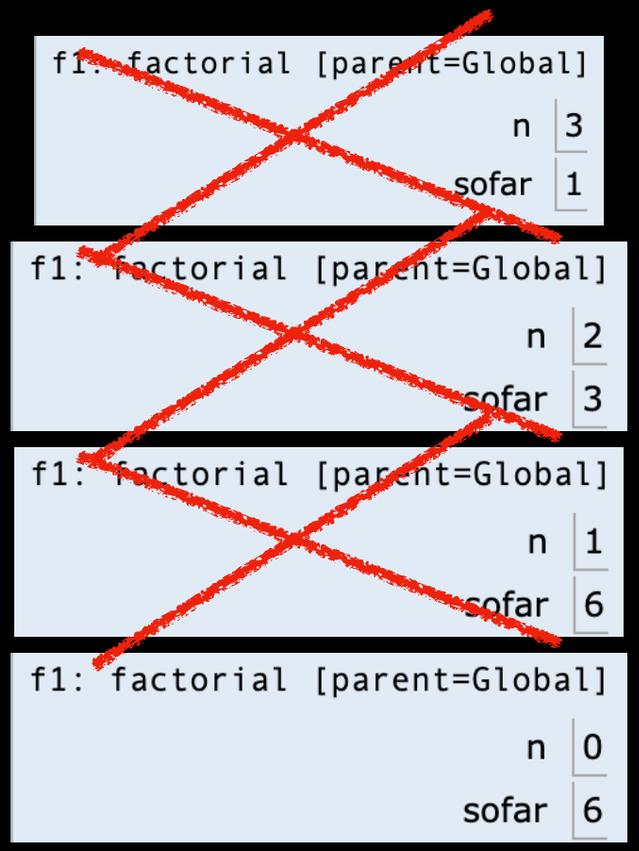
```
scm> (factorial 3 1)
```

Tail Call Optimization

ain't no free lunch

```
(define (factorial n sofar)
  (if (= n 0)
    sofar
    (factorial (- n 1) (* sofar n)))))
```

```
scm> (factorial 3 1)
```



How much memory does `factorial` take now? Use big-theta notation!

`θ(1)`

Is Tail Call Optimization Worth it?

Discuss: what are the benefits and what are the drawbacks of tail-call optimization?

Is Tail Call Optimization Worth it?

Discuss: what are the benefits and what are the drawbacks of tail-call optimization?

\+ More memory efficient! Constant, instead of linear space.

Is Tail Call Optimization Worth it?

Discuss: what are the benefits and what are the drawbacks of tail-call optimization?

\+ More memory efficient! Constant, instead of linear space.

\- Impossible to trace errors back.

Is Tail Call Optimization Worth it?

Discuss: what are the benefits and what are the drawbacks of tail-call optimization?

\+ More memory efficient! Constant, instead of linear space.

\- Impossible to trace errors back.

Because of this drawback, Python does not perform tail-call optimization.

Tail Contexts

Tail contexts are locations in Scheme expressions where recursive calls would be the last operation performed in a frame! Why is this important?

…recursive calls in tail contexts are called tail calls!

```
(define (factorial n sofar)
  (if (= n 0)
    sofar
    (factorial (- n 1) (* sofar n)))))
```

Question:
What makes a Scheme function tail recursive - in the context of tail calls? Check out this example if you're not sure…

Identifying Tail Contexts

How can you tell what is a tail context?

Identifying Tail Contexts

How can you tell what is a tail context?

Given that each of the following expressions is the last expression in the body of the function, the following expressions are tail contexts:

- the second or third operand in an `if` expression
- any of the non-predicate sub-expressions in a `cond` expression (i.e. the second expression of each clause)
- the last operand in an `and` or an `or` expression
- the last operand in a `begin` expression's body
- the last operand in a `let` expression's body

...some examples

Practice!

2 Tail Calls

Questions

2.1 For the following procedures, determine whether or not they are tail recursive. If they are not, write why not and rewrite the function to be tail recursive on the right.

```
; Multiplies x by y
(define (mult x y)
    (if (= 0 y)
        0
        (+ x (mult x (- y 1)))))

; Always evaluates to true
; assume n is positive
(define (true1 n)
    (if (= n 0)
        #t
        (and #t (true1 (- n 1)))))
```

Practice!!

```
; Always evaluates to true
; assume n is positive
(define (true2 n)
    (if (= n 0)
        #t
        (or (true2 (- n 1)) #f)))

; Returns true if x is in lst
(define (contains lst x)
    (cond
        ((null? lst) #f)
        ((equal? (car lst) x) #t)
        ((contains (cdr lst) x) #t)
        (else #f)))
```

Practice!!

2.2 Tail recursively implement **sum-satisfied-k** which, given an input list **lst**, a predicate procedure **f** which takes in one argument, and an integer **k**, will return the sum of the first **k** elements that satisfy **f**. If there are not **k** such elements, return 0.

```
; Doctests
scm> (define lst `(1 2 3 4 5 6))
scm> (sum-satisfied-k lst even? 2)  ; 2 + 4
6
scm> (sum-satisfied-k lst (lambda (x) (= 0 (modulo x 3))) 10)
0
scm> (sum-satisfied-k lst (lambda (x) #t) 0)
0

(define (sum-satisfied-k lst f k)









)
```

Practice!!

2.3 Tail-recursively implement **remove-range** which, given one input list **lst**, and two nonnegative integers **i** and **j**, returns a new list containing the elements of **lst** in order, without the elements from index **i** to index **j** inclusive. For example, given the list (0 1 2 3 4), with $\mathbf{i} = 1$ and $\mathbf{j} = 3$, we would return the list (0 4). You may assume $\mathbf{j} > \mathbf{i}$, and **j** is less than the length of the list. (Hint: you may want to use the built-in **append** function, which returns the result of appending the items of all lists in order into a single well-formed list.)

```
; Doctests
scm> (remove-range '(0 1 2 3 4) 1 3)
(0 4)

(define (remove-range lst i j)
```

Thanks for coming.

Have a great rest of your week! :)

Slides: albertxu.xyz/teaching/cs61a/