CS 61A Topical Review

Tail Recursion and Macros

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)))))
```

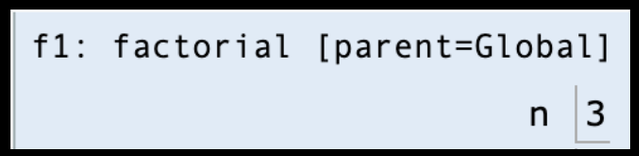
```
scm> (factorial 3)
```

The Cost of Recursion

ain't no free lunch

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

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

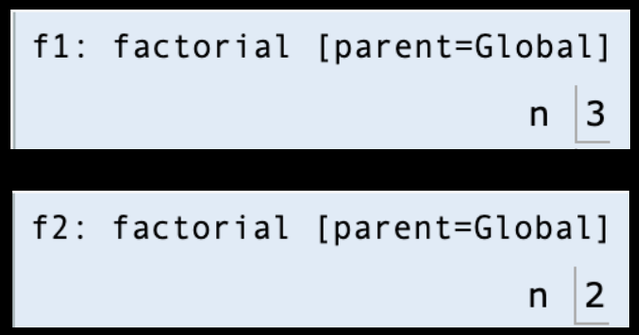


The Cost of Recursion

ain't no free lunch

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

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

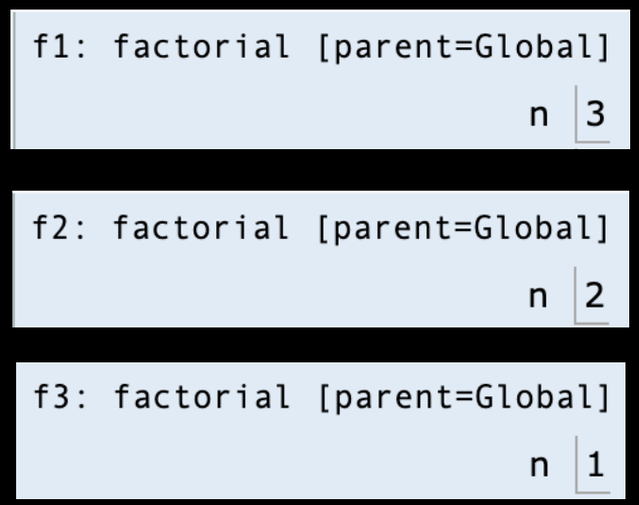


The Cost of Recursion

ain't no free lunch

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

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

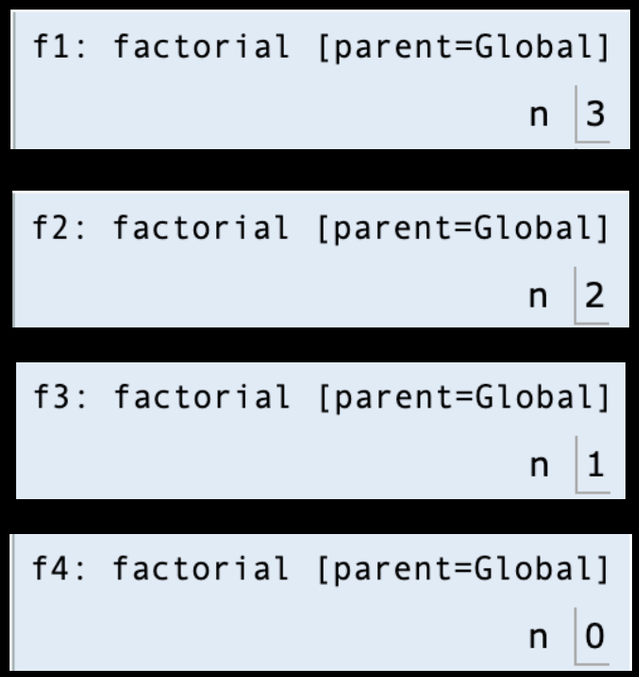


The Cost of Recursion

ain't no free lunch

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

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

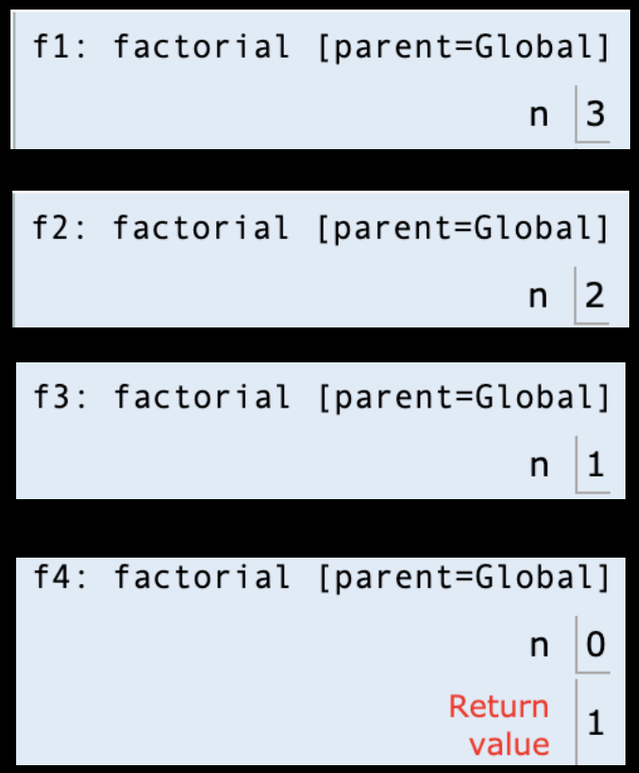


The Cost of Recursion

ain't no free lunch

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

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

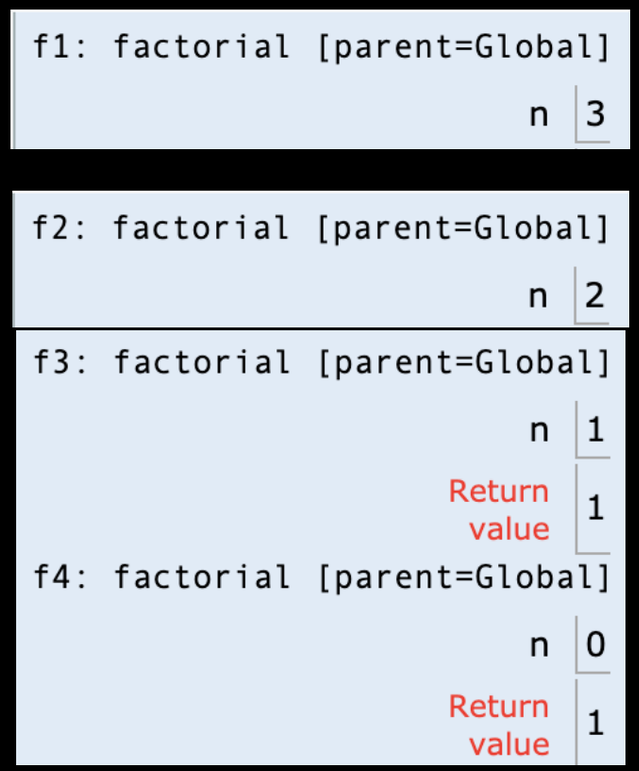


The Cost of Recursion

ain't no free lunch

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

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

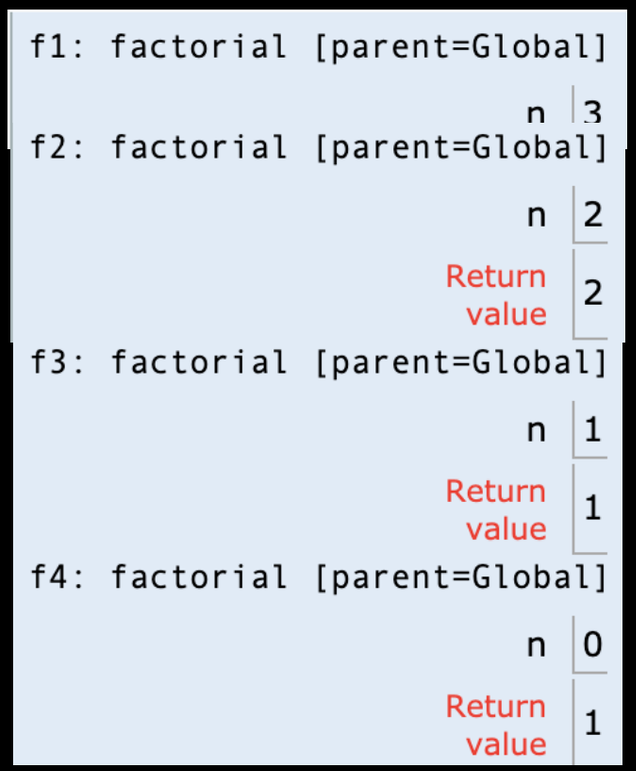


The Cost of Recursion

ain't no free lunch

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

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

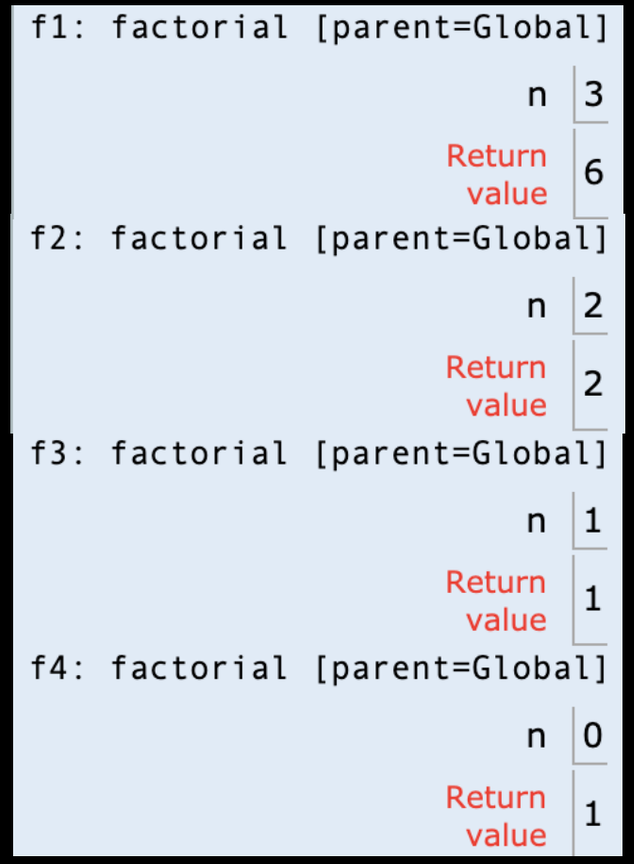


The Cost of Recursion

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 |
```

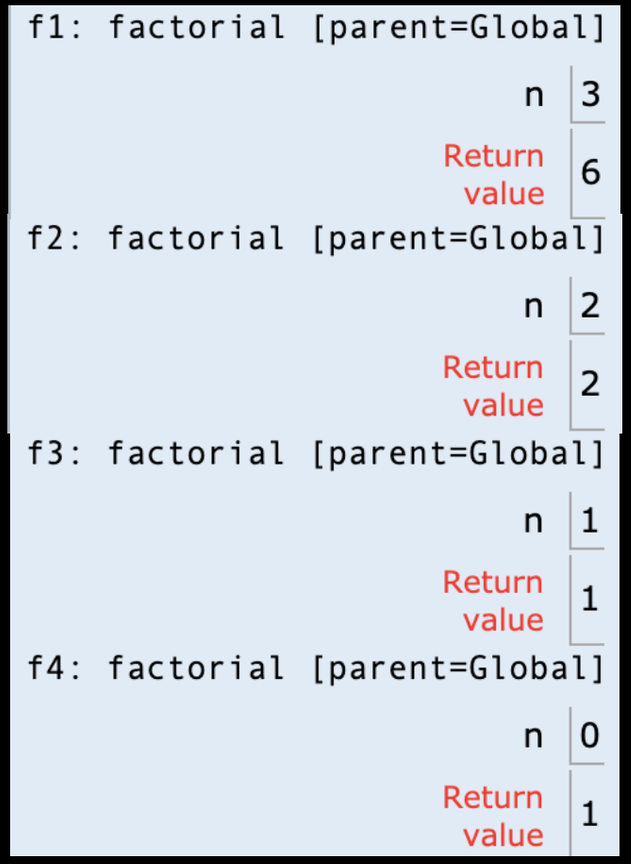
```
scm> (factorial 10)
```

The Cost of Recursion

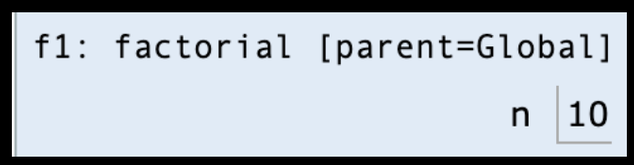
ain't no free lunch

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

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




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

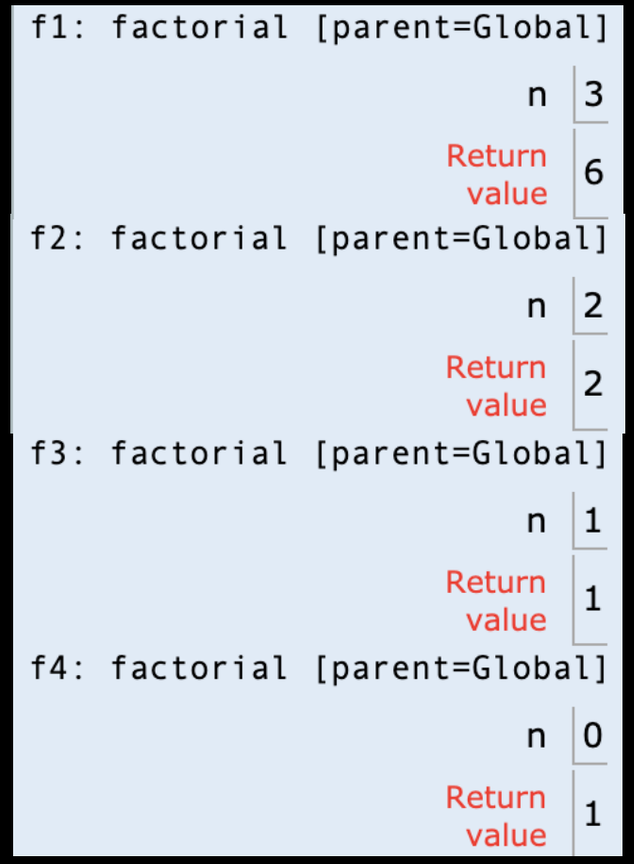


The Cost of Recursion

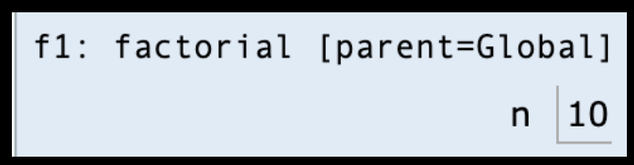
ain't no free lunch

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

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




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



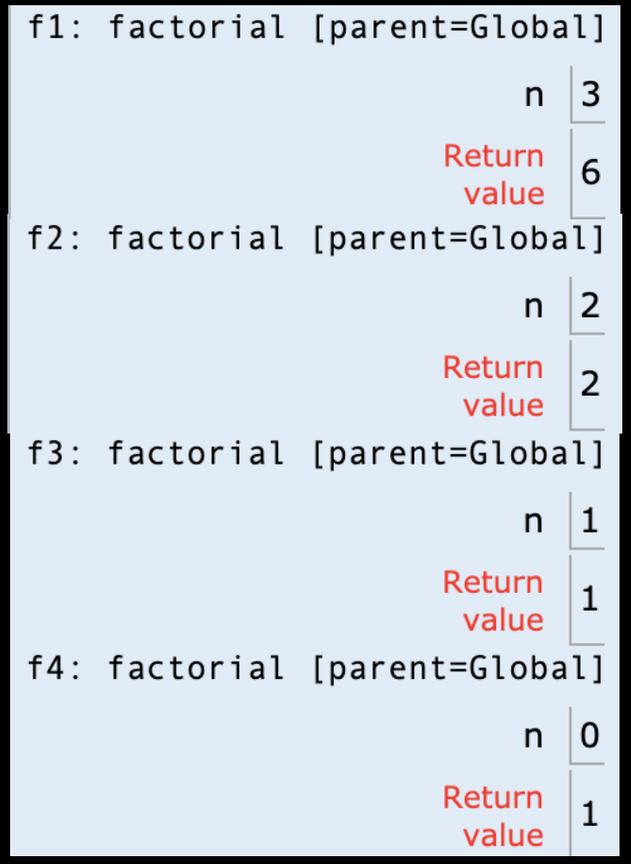
...11 frames!

The Cost of Recursion

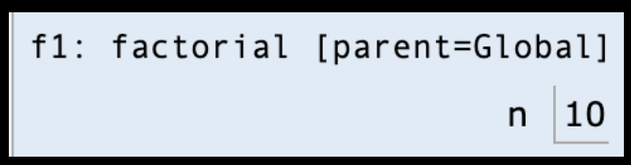
ain't no free lunch

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

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




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



...11 frames!

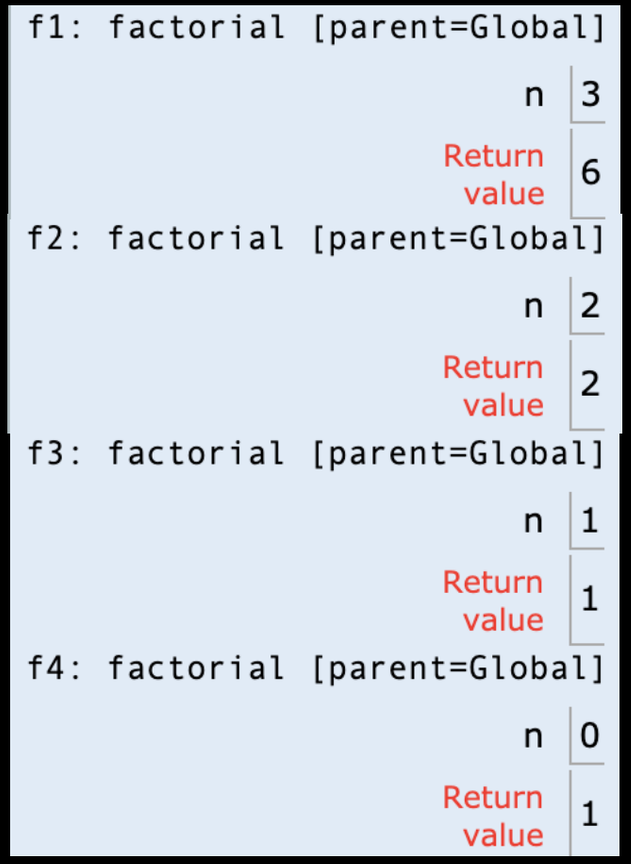
How much memory does `factorial` take? Use big-theta notation!

The Cost of Recursion

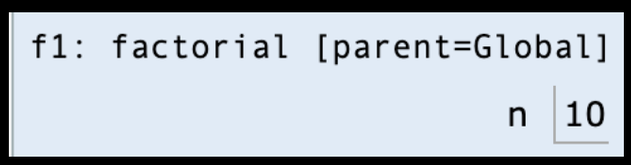
ain't no free lunch

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

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




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



...11 frames!

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

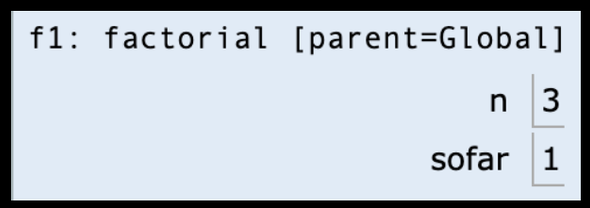
```
scm> (factorial 3 1)
```

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)
```

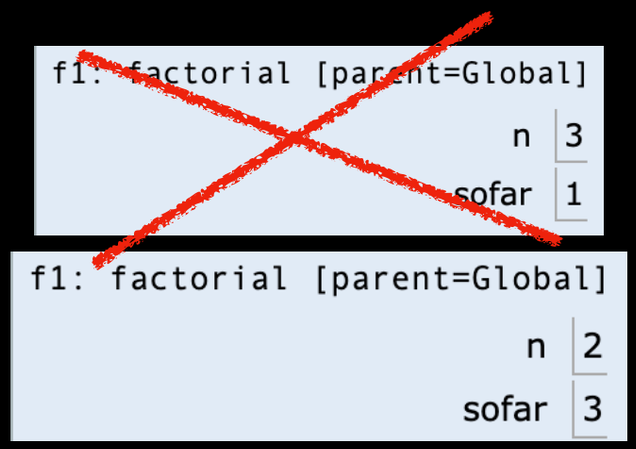


Tail Call Optimization

ain't no free lunch

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

scm> (factorial 3 1)
```

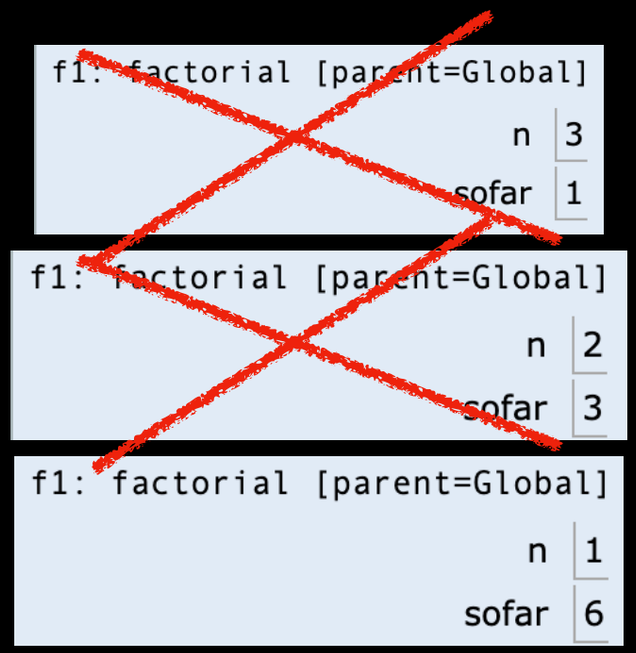


Tail Call Optimization

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)))))
```

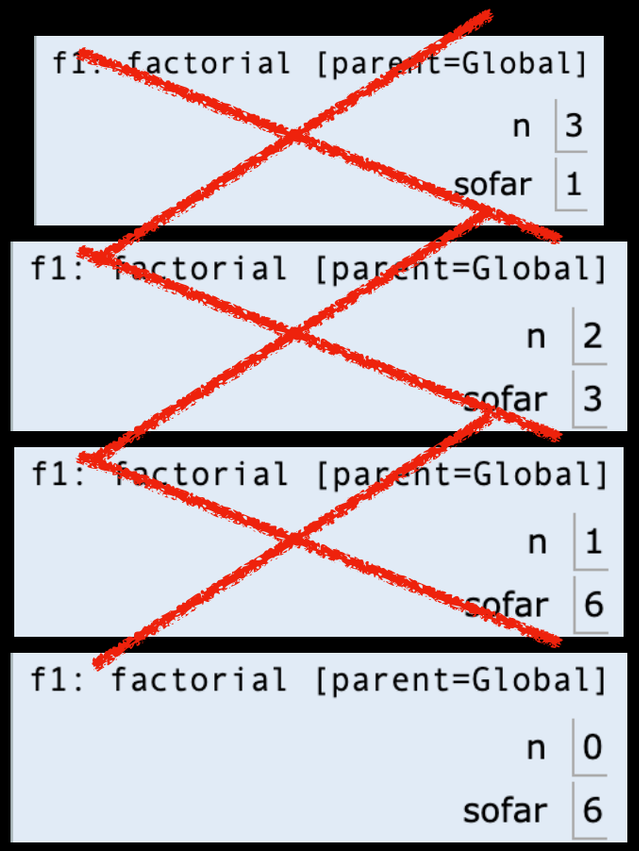
```
scm> (factorial 3 1)
```

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