# Programming Languages

# Tail Recursion and Accumulators

*Material adapted from*
*Dan Grossman's PL class, U. Washington*

# *Recursion*

Should now be comfortable with recursion:

- No harder than using a loop (Maybe?)
- Often much easier than a loop
  - When processing a tree (e.g., evaluate an arithmetic expression)
  - Avoids mutation even for local variables
- Now:
  - How to reason about *efficiency* of recursion
  - The importance of *tail recursion*
  - Using an *accumulator* to achieve tail recursion
  - [No new language features here]

# *Call-stacks*

While a program runs, there is a *call stack* of function calls that have started but not yet returned

- Calling a function **`f`** pushes an instance of **`f`** on the stack
- When a call to **`f`** to finishes, it is popped from the stack

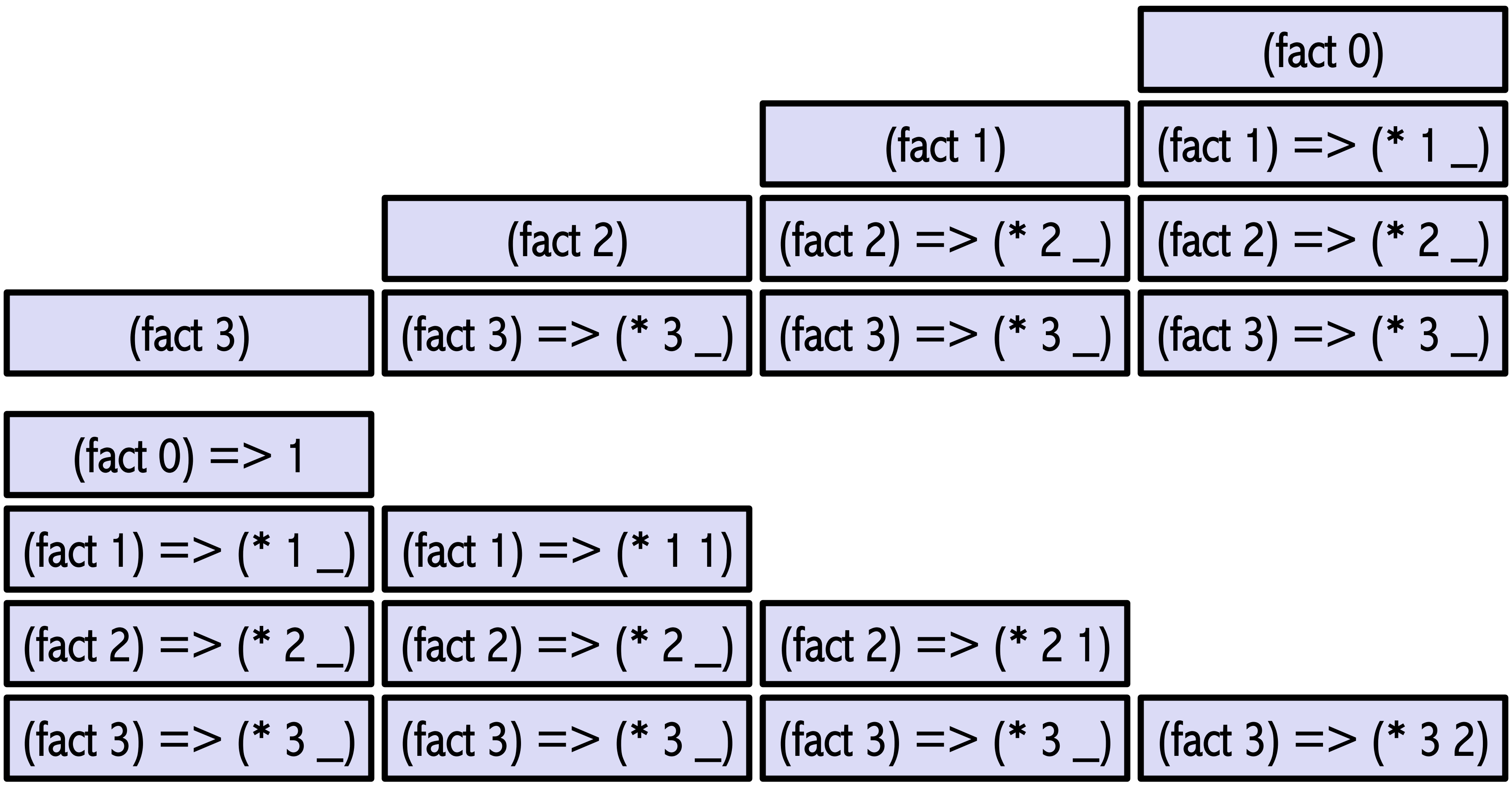
These *stack frames* store information such as

- the values of arguments and local variables
- information about "what is left to do" in the function (further computations to do with results from other function calls)

Due to recursion, multiple stack-frames may be calls to the same function

# *Example*

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



# *What's being computed*

```
(fact 3)
     => (* 3 (fact 2))
     => (* 3 (* 2 (fact 1)))
     => (* 3 (* 2 (* 1 (fact 0))))
     => (* 3 (* 2 (* 1 1)))
     => (* 3 (* 2 1))
     => (* 3 2)
     => 6
```

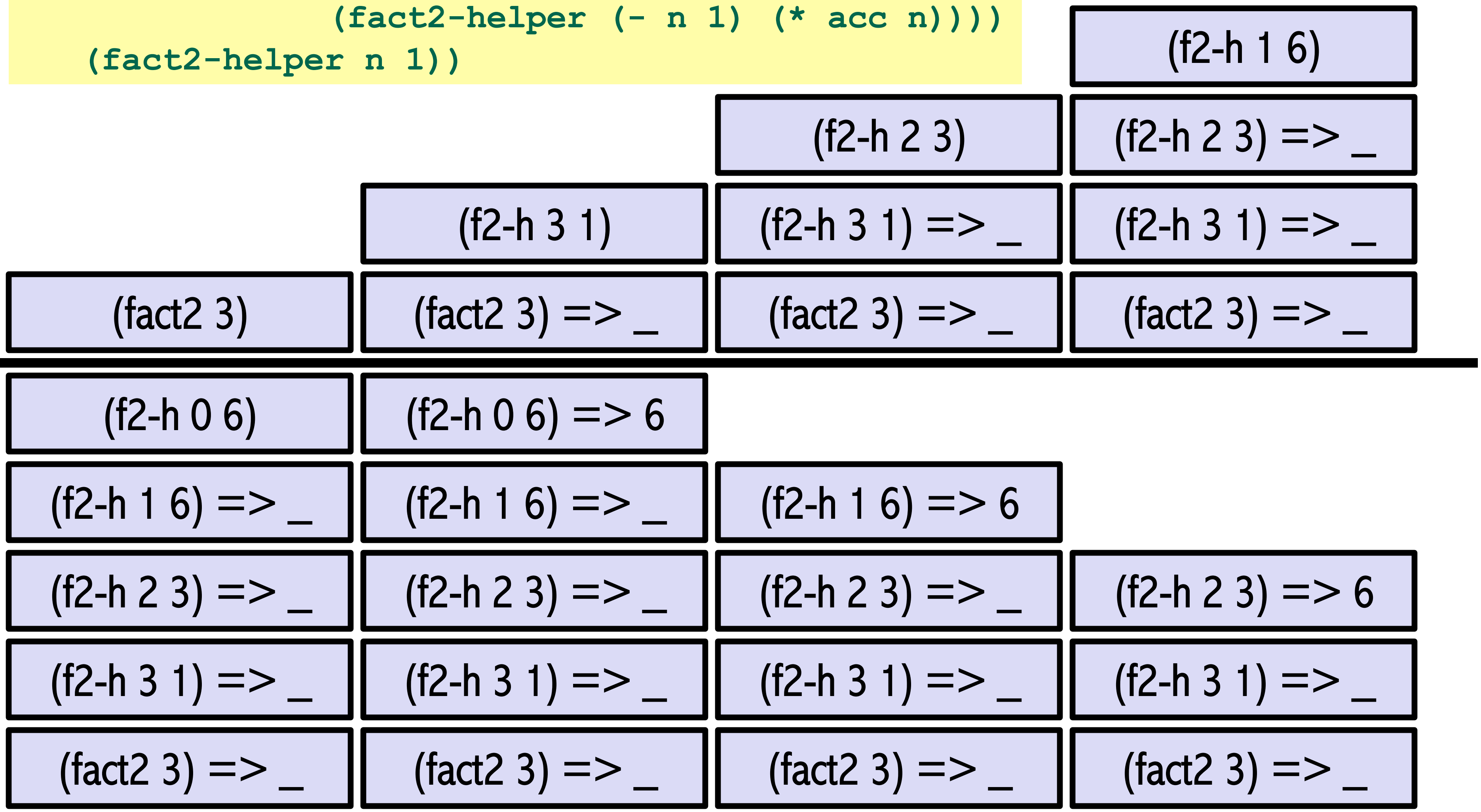
# *Example Revised*

```
(define (fact2 n)

  (define (fact2-helper n acc)
     (if (= n 0) acc
          (fact2-helper (- n 1) (* acc n))))

  (fact2-helper n 1))
```

Still recursive, more complicated, but the result of recursive calls *is* the result for the caller (no remaining multiplication)

# *Example Revised*

```
(define (fact2 n)
   (define (fact2-helper n acc)
        (if (= n 0) acc
                (fact2-helper (- n 1) (* acc n))))
   (fact2-helper n 1))
```



# *What's being computed*

```
(fact2 3)
     => (fact2-helper 3 1)
     => (fact2-helper 2 3)
     => (fact2-helper 1 6)
     => (fact2-helper 0 6)
     => 6
```

# *An optimization*

It is unnecessary to keep around a stack-frame just so it can get a callee's result and return it without any further evaluation

Racket recognizes these *tail calls* in the compiler and treats them differently:

- Pop the caller *before* the call, allowing callee to *reuse* the same stack space
- (Along with other optimizations,) as efficient as a loop

(Reasonable to assume all functional-language implementations do tail-call optimization)

includes Racket, Scheme, LISP, ML, Haskell, OCaml…

# *What really happens*

```
(define (fact2 n)

  (define (fact2-helper n acc)
      (if (= n 0) acc
            (fact2-helper (- n 1) (* acc n))))

  (fact2-helper n 1))
```

| | | | | |
|---|---|---|---|---|
| (fact 3) | (f2-h 3 1) | (f2-h 2 3) | (f2-h 1 6) | (f2-h 0 6) |

# *Moral*

- Where reasonably elegant, feasible, and important, rewriting functions to be *tail-recursive* can be much more efficient
  - Tail-recursive: recursive calls are tail-calls
    - meaning all recursive calls must be the last thing the calling function does
    - no additional computation with the result of the callee
- There is also a methodology to guide this transformation:
  - Create a helper function that takes an *accumulator*
  - Old base case's return value becomes initial accumulator value
  - Final accumulator value becomes new base case return value

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

**Old base case's return value becomes initial accumulator value.**

```
(define (fact2 n)

  (define (fact2-helper n acc)
      (if (= n 0) acc
            (fact2-helper (- n 1) (* acc n))))

  (fact2-helper n 1))
```

**Final accumulator value becomes new base case return value.**

# *Another example*

```
(define (sum1 lst)
  (if (null? lst) 0
      (+ (car lst) (sum1 (cdr lst)))))
```

```
(define (sum2 lst)

  (define (sum2-helper lst acc)
     (if (null? lst) acc
        (sum2-helper (cdr lst) (+ (car lst) acc))))

  (sum2-helper lst 0))
```

# *And another*

```
(define (rev1 lst)
  (if (null? lst) '()
    (append (rev1 (cdr lst)) (list (car lst)))))
```

```
(define (rev2 lst)

  (define (rev2-helper lst acc)
    (if (null? lst) acc
      (rev2-helper (cdr lst) (cons (car lst) acc))))

  (rev2-helper lst '()))
```

# *Actually much better*

```
(define (rev1 lst)     ; Bad version (non T-R)
  (if (null? lst) '()
    (append (rev1 (cdr lst)) (list (car lst)))))
```

- For **`fact`** and **`sum`**, tail-recursion is faster but both ways linear time
- The non-tail recursive **`rev`** is quadratic because each recursive call uses **`append`**, which must traverse the first list
  - And 1 + 2 + … + (length-1) is almost length * length / 2
  - Moral: beware **`append`**, especially if 1st argument is result of a recursive call
- **`cons`** is constant-time (and fast), so the accumulator version rocks

# *Tail-recursion == while loop with local variable*

```
(define (fact2 n)
  (define (fact2-helper n acc)
    (if (= n 0) acc
      (fact2-helper (- n 1) (* acc n))))
   (fact2-helper n 1))
```

```
def fact2(n):
  acc = 1
  while n != 0:
    acc = acc * n
    n = n - 1
  return acc
```

# *Tail-recursion == while loop with local variable*

```
(define (sum2 lst)
  (define (sum2-helper lst acc)
    (if (null? lst) acc
      (sum2-helper (cdr lst) (+ (car lst) acc))))
  (sum2-helper lst 0))
```

```
def sum2(lst):
  acc = 0
  while lst != []:
    acc = lst[0] + acc
    lst = lst[1:]
  return acc
```

# *Tail-recursion == while loop with local variable*

```
(define (rev2 lst)
  (define (rev2-helper lst acc)
    (if (null? lst) acc
      (rev2-helper (cdr lst) (cons (car lst) acc))))
  (rev2-helper lst '()))
```

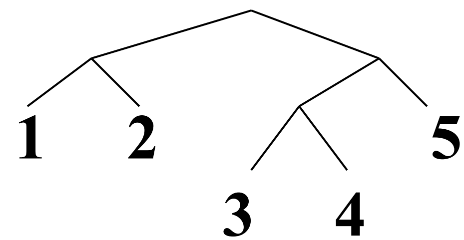
```
def rev2(lst):
  acc = []
  while lst != []:
    acc = [lst[0]] + acc
    lst = lst[1:]
  return acc
```

# *Always tail-recursive?*

There are certainly cases where recursive functions cannot be evaluated in a constant amount of space

Example: functions that process trees

– Lists can be used to represent trees: '((1 2) ((3 4) 5))




In these cases, the natural recursive approach is the way to go

– You could get one recursive call to be a tail call, but rarely worth the complication

# *Precise definition*

If the result of **`(f x)`** is the "return value" for the enclosing function body, then **`(f x)`** is a tail call

i.e., don't have to do any more processing of (f x) to end function

Can define this notion more precisely…

- A *tail call* is a function call in *tail position*
- The single expression (ignoring nested defines) of the body of a function is in tail position.
- If **`(if test e1 e2)`** is in tail position, then **`e1`** and **`e2`** are in tail position (but **`test`** is not). (Similar for cond-expressions)
- If a let-expression is in tail position, then the single expression of the body of the **`let`** is in tail position (but no variable bindings are)
- Arguments to a function call are not in tail position
- …

# *Are these functions tail-recursive?*

```
(define (get-nth lst n)
  (if (= n 0) (car lst)
    (get-nth (cdr lst) (- n 1))))

(define (good-max lst)
  (cond
    ((null? (cdr lst))
      (car lst))
    (#t
      (let ((max-of-cdr (good-max (cdr lst))))
        (if (> (car lst) max-of-cdr)
          (car lst) max-of-cdr)))))
```

# *Try these…*

Write a tail-recursive max function (i.e., a function that returns the largest element in a list).

Write a tail-recursive Fibonacci sequence function (i.e., a function that returns the n'th number of the Fibonacci sequence).

(fib 1) => 1

(fib 2) => 1

(fib 3) => 2

(fib 4) => 3

(fib 5) => 5

In general, (fib n) = (+ (fib (- n 1)) (fib (-n 2)))

```
(define (maxtr lst)
  (define (maxtr-helper lst max-so-far)
    (cond ((null? lst) max-so-far)
          ((> max-so-far (car lst))
            (maxtr-helper (cdr lst) max-so-far))
          (#t (maxtr-helper (cdr lst) (car lst)))))
  (maxtr-helper (cdr lst) (car lst)))
```

```
(define (fib-tr n)

  (define (fib-helper n-minus-1 n-minus-2 current-n)
    (if (= current-n n)
      (+ n-minus-1 n-minus-2)
      (fib-helper (+ n-minus-1 n-minus-2)
                  n-minus-1
                  (+ 1 current-n))))

  (if (< n 3) 1 (fib-helper 1 1 3)))
```