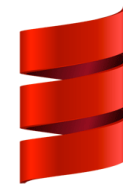
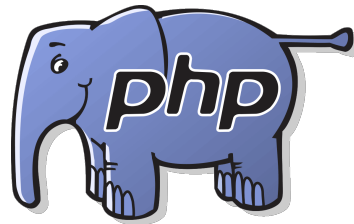


CS 360

Programming Languages

Day 6



Scala



Swift



Today

- Type systems
 - Rules for how types are determined in a language.
- Side effects
 - ...and why they are not used in functional programming.
- Tail recursion
 - Special speed-up built into many functional languages that allows recursive functions (in many cases) to not cause stack overflows.



stack overflow

Type systems

- A ***type system*** is a set of rules that assigns types to variables and expressions.
- The purpose of a type system is to reduce bugs in programs, but also allow for abstraction, documentation, and optimization as well.
- Whole area of ***type theory*** in computer science studies the theoretical underpinnings of type systems.
- We will focus on one aspect of type systems today: **static type checking vs dynamic type checking**.

Declaring functions in C++ vs Python

C++ uses **static type checking**: most code can be checked at compile-time to make sure rules involving types are not violated.

```
int double(int n) {  
    return 2 * n;  
}
```

Python uses **dynamic type checking**: most code cannot be checked for type errors at compile-time; this has to be delayed until run-time.

```
def double(n):  
    return 2 * n
```

Dynamic type checking

- Racket (like most Scheme or Lisp dialects) is dynamically type checked.
- Some characteristics of dynamic type checking:
 - Values have types, but variables do not.
 - A variable can have different types during its lifetime.
 - Most type-error bugs cannot be found before the program is run, and not until the offending line of code is encountered.
 - Possible to write code with type errors that aren't discovered for a long time, if buried in code that isn't executed often.
 - Traditionally (but not always), dynamically-typed languages are interpreted, whereas statically-typed languages are compiled.

Some good things about dynamic type checking

- Enables straight-forward polymorphism (enabling code to handle any data type).
 - Example: Calculating the length of a list.

```
(define (length lst)
  (if (null? lst) 0 (+ 1 (length (cdr lst)))))
```

versus

```
int length_int_linkedlist(int_node* lst) {
  if (lst->next == nullptr) return 0;
  else return 1 + length_int_linkedlist(lst->next);
}
```

Easier to create flexible data structures

- In Racket, it's easy to create a list that can contain any other kind of data structure:
 - List of integers: ' (1 2 3)
 - List of booleans: ' (#f #f #t #f #t)
 - List of strings: ' ("a" "b" "c")
 - List of mixed types: ' ("a" 42 #f)
 - List of really mixed types: ' (1 (3 #f) ("hi") 9 (1 2) ())
- Also, all of these lists will work with our **length** function!
- Mixing types in a single data structure is not easy in statically-typed languages.
- In C++, arrays or vectors must all hold the same type.

"Manual" type-checking

- Dynamically-typed languages often have some way for the programmer to discover the type of a variable.
- In Racket (all of these return `#t` or `#f`):
 - **number?**
 - also **integer?**, **rational?**, **real?**
 - **list?**
 - **pair?**
 - **string?**
 - **boolean?**
- Enables a single function to do different things depending on the type of an argument.

Length of a list vs length of nested lists

```
(define (length-nested lst)
  (cond ((null? lst) 0)
        ((list? (car lst))
         (+ (length-nested (car lst))
            (length-nested (cdr lst))))
        (#t (+ 1 (length-nested (cdr lst)))))
```

Side effects

- In programming, a function has a side effect if it modifies some state or has an observable interaction with functions outside of itself (other functions or the outside world).
- Mutation is an example of a side effect.
 - Also: printing to the screen, modifying files, etc
- Functional programming (in Racket, Scheme, Lisp) traditionally avoids side effects as much as possible.
 - Makes it much simpler to reason about how a program works.
 - Without side effects, calling a function with a fixed set of arguments is guaranteed to always return the same value.

Side effects

- In Racket, function bodies may contain more than one expression, if the extra expressions ***come first and are evaluated only for their side effects.***
 - In "pure" functional programming, you don't have side effects.
 - But it's nice to have this facility at times.
 - For debugging, can use `(displayln expr)` and `(newline)`

- Example:

```
(define (length lst)
  (displayln lst)
  (if (null? lst) 0 (+ 1 (length (cdr lst)))))
```

Tail Recursion and Accumulators

Recursion

Should now be comfortable with recursion (or at least after tomorrow night...):

- 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 stack

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 finishes, it is popped from the stack.
- Common to most programming languages.

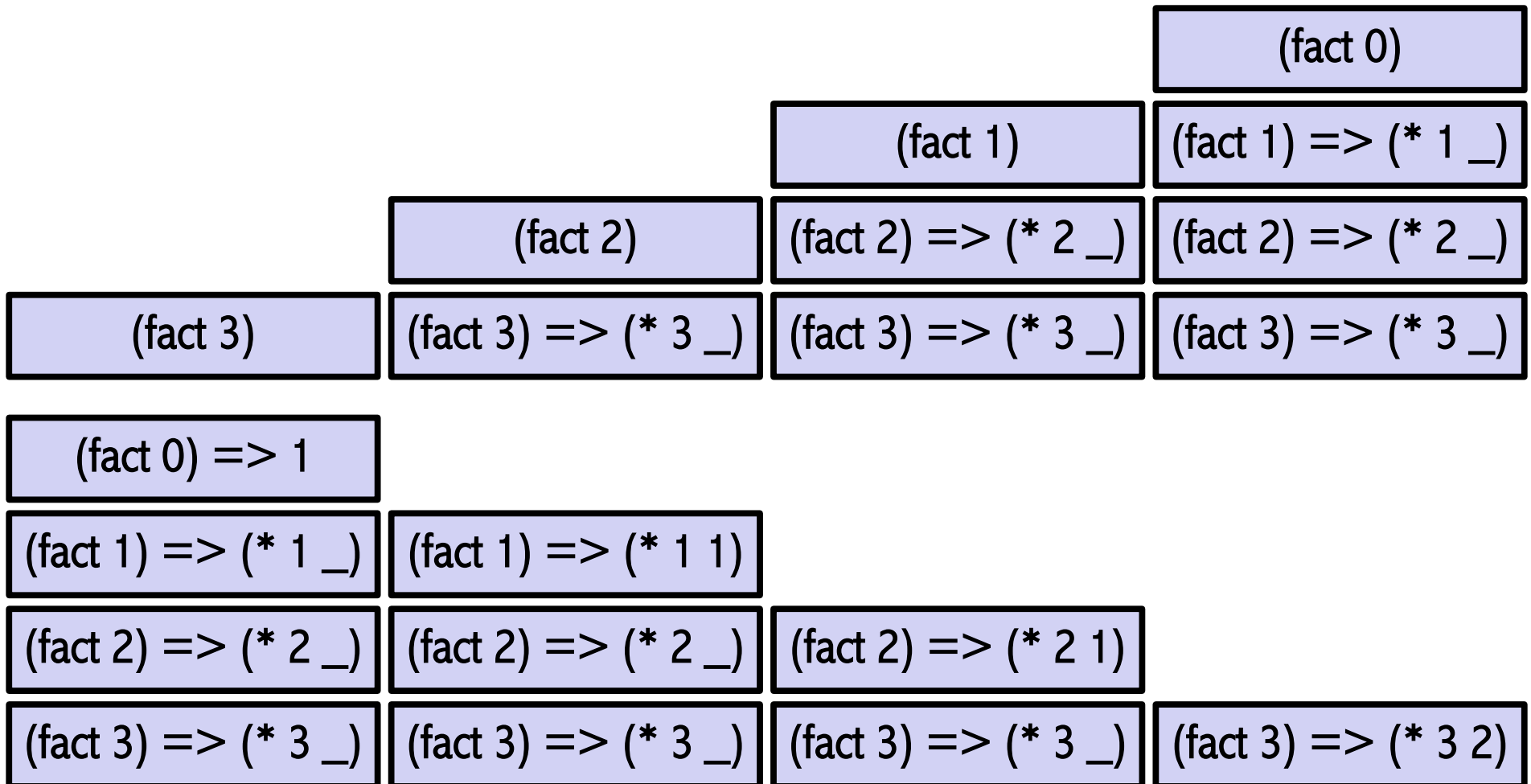
These *stack frames* store information such as

- 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

Compare

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

```
(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)

```

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

```

(f2-h 1 6)

(f2-h 2 3)

(f2-h 2 3) => _

(f2-h 3 1)

(f2-h 3 1) => _

(f2-h 3 1) => _

(fact2 3)

(fact2 3) => _

(fact2 3) => _

(fact2 3) => _

(f2-h 0 6)

(f2-h 0 6) => 6

(f2-h 1 6) => _

(f2-h 1 6) => _

(f2-h 1 6) => 6

(f2-h 2 3) => _

(f2-h 2 3) => _

(f2-h 2 3) => _

(f2-h 2 3) => 6

(f2-h 3 1) => _

(f2-h 3 1) => _

(f2-h 3 1) => _

(f2-h 3 1) => _

(fact2 3) => _

(fact2 3) => _

(fact2 3) => _

(fact2 3) => _

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 situations and treats them differently:

- Pop the caller *before* the call, allowing callee to *reuse* the same stack space.
- Uses same amount of memory as a loop.

Most, if not all functional language implementations do this optimization:
includes Racket, Scheme, LISP, ML, Haskell, OCaml...

What really happens on the call stack

```
(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)

Tail recursion

- In a functional language, rewriting functions to be ***tail-recursive*** can be much more efficient than "normal" recursive functions.
- In a ***tail-recursive*** function, all recursive calls must be the last thing the calling function does.
 - meaning no additional computation is done with the result of the callee (the recursive call).
- Functional languages will automatically optimize these tail-calls so they reuse the same stack space repeatedly.

Key to understanding tail recursion

- Most (singly-)recursive functions involve a recursive call and a computation involving the result of that recursive call.
 - e.g., for factorial, we multiply the result of the recursive call by n .
 - Normally we think about doing the **recursive call first** and the **computation second**.

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

- Tail-recursive functions do the **computation first** and the **recursive call second (last)**.

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

Methodology for tail-recursion

- Straight forward to turn a "normally" recursive function into a tail-recursive one:
 - 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 (length lst)
  (if (null? lst) 0
      (+ 1 (length (cdr lst)))))
```

```
(define (length-tr lst)

  (define (length-tr-helper lst acc)
    (if (null? lst) acc
        (sum-tr-helper (cdr lst) (+ 1 acc))))

  (sum-tr-helper lst 0))
```

And another

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

```
(define (rev-tr lst)

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

  (rev-tr-helper lst '()))
```

Actually much better

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

- For **fact** and **length**, tail-recursive versions are faster than non-TR versions (though both are linear time)
- The non-tail recursive **rev** is quadratic because each recursive call uses **append**, which must traverse the first list to copy it
 - And $1 + 2 + \dots + (\text{length}-1)$ is almost $\text{length} * \text{length} / 2$
 - 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 (fact-tr n)
  (define (fact-tr-helper n acc)
    (if (= n 0) acc
        (fact-tr-helper (- n 1) (* acc n))))
  (fact-tr-helper n 1))
```

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

Tail-recursion == while loop with local variable

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

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

Tail-recursion == while loop with local variable

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

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

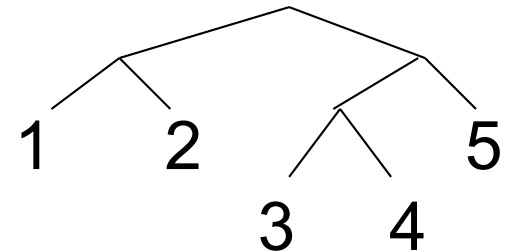
Always tail-recursive?

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 sum function (i.e., a function that takes a list and computes the sum of all the elements).

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)))**

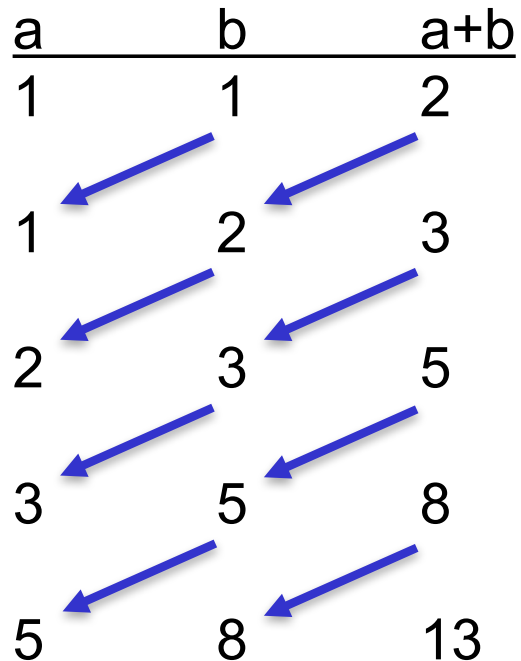
```
(define (sum-tr lst)
  (define (sum-tr-helper lst acc)
    (if (null? lst) acc
        (sum-tr-helper (cdr lst) (+ (car lst) acc))))
  (sum-tr-helper lst 0))
```

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

Fibonacci as a while loop

1 1 2 3 5 8 13

<u>a</u>	<u>b</u>	<u>a+b</u>
1	1	2
1	2	3
2	3	5
3	5	8
5	8	13



```
def fib(n):  
    ctr = 1  
    a = 1  
    b = 2  
    while ctr < n:  
        ctr += 1  
        old_a = a # hold onto value of a  
        a = b  
        b = old_a + b  
    return a
```

Fibonacci as a while loop

```
def fib(n):  
    ctr = 1  
    a = 1  
    b = 2  
    while ctr < n:  
        ctr += 1  
        old_a = a # hold onto value of a  
        a = b  
        b = old_a + b  
    return a
```

```
(define (fib-tr n)
  (define (fib-helper a b ctr)
    (if (= ctr n) a
        (fib-helper b (+ a b) (+ ctr 1))))
  (fib-helper 1 1 1))
```

```
def fib(n):
    ctr = 1
    a = 1
    b = 2
    while ctr < n:
        ctr += 1
        old_a = a # hold onto value of a
        a = b
        b = old_a + b
    return a
```