WELLESLEY

Programming Languages

CS 251

Fall 2021

Carolyn Anderson

Recap

What makes a PL functional?

- They provide abstractions over functions
- They treat functions like other values in the language
- They emphasize recursion over iteration
- They do not allow mutation

Side effects

- **Side effect**: any observable effect other than producing a value
- Functional programming languages tend to avoid side effects (mutation is a kind of side effect)
- Side effects make it **harder to reason formally** about a program's behavior
- However, printing is very useful!

Racket printing

- What's the difference between **display**, **write**, and **print**?
- What does **displayln** do?

Making observations

More Racket

Language components

- **Expressions**: bits of the language

(+ 1 2) "cat" (define (foo n) n)

- **Values**: expressions that cannot be reduced any further

"cat" (define (foo n) n)

- **Declarations**: bind variables to values

(define x 4)

More Racket: Values

Functions revisited

Syntax: (**define** (foo id_1, … id_n) e)

parameters **function body**

Practice:

Write a function that takes a list and adds 5 to each item in the list.

Practice:

Write a function that takes a number and counts down to 0 from that number.

```
> (countdown 6)
6
5
4
3
2
1
0
```

Euclid's algorithm for GCD

Find greatest common divisor of r1 and r2:

base case:

If r1 = 0:

    return r2

If r2 = 0:

    return r1

kth step:

If r1 and r2 are greater than 0:

    r1 / r2

    GCD(r2, remainder)

More Racket: Definitions

Local binding

A let expression **binds** a set of variables for use in the body of the let block.

```
(define (greet str)
  (let ((greeting (string-append "hi " str)))
    (display greeting)))
```

Local binding, take two

In a let expression, the right-hand side of a declaration can't refer to the left-hand side.

If we write:

(let ((a (+ a 5))))

if **a** is not defined outside the **scope** of the **let**, the **let** will throw an error.

Practice:

Write a function that takes a list of numbers and returns the sum of their squares

```
> (sum-squares (1 2 3))
14
```

Practice:

Write a function that takes a number and counts up from 0 to that number.

```
> (countup 6)
0
1
2
3
4
5
6
```

First class functions

In Racket, functions are values. This is because Racket has **first class functions**: functions have all the rights and privileges of other values.

Function Bill of Rights:

We the Racketeers hereby declare that functions:

- Do not need to be named (lambdas)
- Can be returned by functions
- Can be arguments to functions

Anonymous Functions

A lambda expression is an anonymous function.
(define (fn)) is really short for (define fn (lambda))

(define (hello-world) (display “hello world!”))
(define hello-world (lambda ( ) (display “hello world!”)))

Lambdas

Lambda: anonymous function

```
(lambda (x y) (+ x y))
```

list of arguments **function body**

Practice: write an anonymous function that returns the second item in a list.

Letrec

This is a problem for declaring recursive functions, since they refer to themselves!
Racket has another local binding construct for this reason: **letrec**.

If we write:

(letrec ((a (+ a 5))))

The **a** in the right-hand side refers to whatever value the **a** on the left-hand side has.

String-reverse using letrec

```
(define (reverse str)
  (letrec ((helper
            (lambda (str x)
              (if (= x (string-length str))
                  ""
                  (string-append
                    (helper str (+ x 1))
                    (string (string-ref str x)))))))
    (helper str 0)))
```

define helper function

helper function arguments

base case

recursive call

call helper function

Practice:

Rewrite **count-up** using **letrec**.

```
(define (count-help x y)
    (display x)
    (if (= x y)
        (void)
        (count-help (+ x 1) y)))

(define (count-up x)
    (count-help 1 x))
```

Recursion versus iteration

How efficient is recursion anyway?

Recursion versus iteration

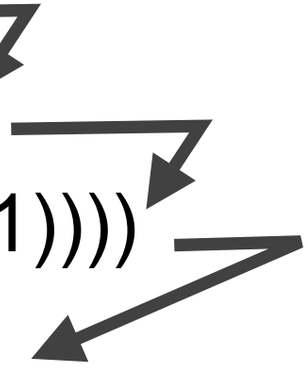
How efficient is recursion anyway?

Iterative

```
> (it-fac 4)
res = res*1
res = res*2
res = res*3
res = res*4
```

Recursive

```
> (fac 4)
(* 4 (fac 3))
(* 4 (* 3 (fac 2)))
(* 4 (* 3 (* 2 (fac 1))))
(* 4 (* 3 (* 2 1)))
```



Tail recursion

There's another way of writing this recursive function!
In **tail recursion**, the multiplication happens inside the recursive call, rather than outside of it.

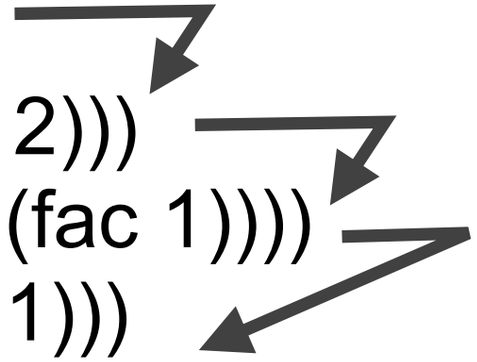
```
(define (tail-fac n)
  (letrec ((helper
            (lambda (n acc)
              (if (= 1 n)
                  acc
                  (helper (- n 1)
                          (* n acc))))))
    (helper n 1)))
```

Tail recursion

How efficient is recursion anyway?

Original version

```
> (fac 4)
(* 4 (fac 3))
(* 4 (* 3 (fac 2)))
(* 4 (* 3 (* 2 (fac 1))))
(* 4 (* 3 (* 2 1)))
```



Tail-recursive version

```
> (tail-fac 4)
(tail-fac 3 (* 4 1))
(tail-fac 2 (* 3 4))
(tail-fac 1 (* 2 12))
(24)
```

Practice:

Write two versions of **string reverse:** a tail-recursive and a non-tail-recursive version.

Practice: Fizzbuzz

Count up from 0 to n in the following way:

- If the number is divisible by 3, print fizz
- If the number is divisible by 5, print buzz
- If the number is divisible by 3 **and** 5, print fizzbuzz
- Otherwise, print the number