CIS 194

FUNCTIONS

BUT FIRST…

- Installed Haskell?
- hw1
- Waitlist
- Lingering questions from lec1

SYNTAX

DEFINING FUNCTIONS

- ***name arg1 arg2 … argN = expression***
- `alwaysOne iAmNotUsed = 1`
- `double x = x + x`
- `hello name = "Hello, " ++ name ++ "!"`
- `smaller x y = if x <= y then x else y`

INVOKING FUNCTIONS

- ***name arg1 arg2 … argN***
- `double 5 -> 10`
- `hello "CIS 194!" -> "Hello, CIS 194!"`
- `smaller "abc" "xyz" -> "abc"`

NESTING FUNCTION CALLS

- `double (double 5) -> 20`
- `smaller (alwaysOne "notTwo") (double 0) -> 0`
- ~~`double double 5`~~

LAMBDAS

- *\arg1 arg2 … argN -> expression*
- `\x -> x + 1`
- \str1 str2 -> str1 ++ " " ++ str2

BIG IDEAS

BIG IDEAS

KEY TAKEAWAYS

- Small functions can be combined to do complex things
- Functions transform data
- Functions are themselves data

LOTS OF SMALL FUNCTIONS —> BIG THING

```
isTeen x = 13 <= x && x <= 19
getName x = fst x
getAge x = snd x

head (
  map getName (
    filter (\person -> isTeen (getAge person))
    [ ("Sue", 10), ("Bob", 20), ("Alex", 14) ]
  )
)
-> "Alex"
```

FUNCTIONS TRANSFORM DATA

- Functions are pure
- You provide data
- You get back new data

- `addOne x = x + 1`
- ``isEven x = x `mod` 2 == 0``

FUNCTIONS ARE DATA

- “First-class values”
- Can pass function as an arg to another function
- Functions can return other functions
- Can be stored in data structures

FUNCTIONS ARE DATA

- `applyTwice f x = f (f x)`
- `applyTwice hello "CIS 194"`
  `-> "Hello, Hello, CIS 194!!”`
- Lingo: `applyTwice` is a “higher-order” function

PARTIAL APPLICATION

ALL FUNCTIONS IN HASKELL TAKE ONLY ONE ARGUMENT

The Dirty Truth About Functions

DON'T BELIEVE ME?

- `appendToMyself str = str ++ str`
- `appendToMyself = \str -> str ++ str`

- `add x y = x + y`
- `add x = \y -> x + y`
- `add = \x -> (\y -> x + y)`

SO WHAT IS PARTIAL APPLICATION THEN?

- Well, really just normal function application!
- Call function and get back another function
- You can think of as not providing all arguments

FUNCTION COMPOSITION

COMPOSING FUNCTIONS

```
head (
  map getName (
    filter (\person -> isTeen (getAge person))
    [ ("Sue", 10), ("Bob", 20), ("Alex", 14) ]
  )
)
```

But is this easy to read?

TWO USEFUL OPERATORS

Function composition

```
f . g = \x -> f (g x)
```

Function application

```
f $ x = f x
```

USING COMPOSITION AND APPLICATION OPERATORS

- `f (g x)` becomes `f . g $ x`
- `f (g (h x))` becomes `f . g . h $ x`
- `f (g (h (i x)))` becomes `f . g . h . i $ x`

- `length . filter even . map numberOfFactors`

COMPOSING FUNCTIONS

```
head . map getName . filter (\person ->
  isTeen (getAge person)) $
  [ ("Sue", 10), ("Bob", 20), ("Alex", 14) ]
```

Better? At least less parens to match up.

POINT-FREE STYLE

- `foo x = f x`
  - becomes `foo = f`
- `foo x = f . g . h $ x`
  - becomes `foo = f . g . h`

- But can quickly get out of hand…
- `\a b c -> a*b+2+c`
- `((+) .) . flip flip 2 . ((+) .) . (*)`

RECURSION

HOW CAN WE IMPLEMENT THESE?

- `factorial 5 -> 120`
- `repeatIt 4 "ha" ->"hahahaha"`

OUR FIRST RECURSIVE FUNCTIONS

```
factorial n =
  if n == 0 then 1
  else n * factorial (n - 1)

repeatIt n snippet =
  if n <= 0 then ""
  else snippet ++ repeatIt (n - 1) snippet
```

TIPS FOR RECURSION

- Base case and recursive case
- Don’t think too hard about base case
- Treat recursive call like oracle

ONE MORE EXAMPLE

```
map f xs =
  if null xs then
    []
  else
    f (head xs) : map f (tail xs)
```

CASE PATTERNS

REVISITING AN OLD FRIEND

```
factorial n =
  if n == 0 then 1
  else n * factorial (n - 1)

factorial' 0 = 1
factorial' n = n * factorial (n - 1)
```

Which version do you like better?

WHAT HAPPENS WHEN I FORGET A CASE?

```
countdown n = countdown (n - 1)
tMinusTen = countdown 10
```

Oops!

```
httpCodeIsOk 200 = True
succeeded = httpCodeIsOk 404
```

Oops! (But for a different reason.)

PARTIAL FUNCTIONS

MODELING PARTIAL FUNCTIONS

- When your function does not work on all inputs
- Wrap output in Maybe
- Restrict your input
- Return a default value
- Or… crash!

CRASHING

- Never write functions that crash
- Avoid using Prelude functions that crash
- Notably `head` and `tail`