EMERGENCY HASKELL CRASH COURSE PART 2

Pattern matching

Instead of a big messy function, you can use **patterns** to define a function by cases.

```
-- A cleaner greeting function
customGreeting "Mary" "Sue"  = "Hey Mary!"
customGreeting fname lname   = "Hi " ++ fname ++ " " ++ lname

-- Fibonacci function
fib 0 = 1
fib 1 = 1
fib n = fib (n - 1) + fib (n - 2)
```

You can use **guards** when the cases are more sophisticated:

```
-- Given two numbers, returns a string
compare x y
  | x < y      = "Less"
  | x == y     = "Equal"
  | otherwise = "More"
```

Q: What is `otherwise`?

Data types

A **constructor** is special kind of function or operator for creating values of a certain type.

1. The `Bool` type has two constructors: `True` and `False`. They're both constants.
2. The constructor for pairs is the `,` operator. It takes two arguments `x` and `y` and produces a pair `x, y`.
3. Linked lists have two constructors:
   - `[]` is the constructor for an empty list. It's a constant.
   - `:` is the constructor that adds an element to the front of the list. So `1 : [2, 3]` produces `[1,2,3]`. We can write `[1,2,3]` in terms of just constructors, like this: `1 : (2 : (3 : []))`.

Constructors are special because you can pattern match against them:

```
-- Takes a pair of numbers and adds them.
addPair (x, y) = x + y

-- Returns the negation of a boolean value.
not True = False
not False = True

-- Check whether the list is empty.
isEmpty []       = True
isEmpty (x : xs) = False

-- Double every element in the list
doubleAll [] = []
doubleAll (x : xs) = (2 * x) : (doubleAll xs)

-- Remove all zeros from the list; [1, 0, 0, 3] becomes [1, 3]
getRidOfZeros []       = []
getRidOfZeros (0 : xs) = xs
getRidOfZeros (x : xs) = x : (getRidOfZeros xs)

-- Check if the list has two consecutive zeros in it
hasConsecutiveZeros []           = False
hasConsecutiveZeros [x]          = False
hasConsecutiveZeros (0 : 0 : xs) = True
hasConsecutiveZeros (x : xs)     = hasConsecutiveZeros xs
```

Q: Write a function that checks if there are any negative numbers in a list.

Q: Write a function that removes all negative numbers from a list.

Debugging

Here's a buggy implementation of the fibonacci function. It's very slow. Why?

```
fib n = fib (n - 1) + fib (n - 2)
fib 1 = 1
fib 0 = 1

main = print (fib 5)
```

We can use a special function `trace` in the `Debug.Trace` module. The expression `trace "Hello" exp` will print `"Hello"` to the console right before `exp` gets evaluated. Here's how we can use it:

```
import Debug.Trace

fib n = trace "Adding..." (fib (n - 1) + fib (n - 2))
fib 1 = trace "Base case 1" 1
fib 0 = trace "Base case 0" 1

main = print (fib 5)
```

Q: What does this print?

Q: What's wrong with the `fib` implementation?

NB: Actually, the Haskell runtime is a bit too smart for this example. It detects an infinite loop and exits out of the program early, raising an error.

Laziness

What happens when you run the following Python program?

```
def foo(x, y):
  print("foo")

def bar(x):
  print(x)

foo(bar("x"), bar("y"))
```

Most languages have a **strict** evaluation order. This means that the arguments of a function have to be evaluated before the function is entered.

Haskell has a **lazy** evaluation order. Rather than explain it, let's see it in action by inserting `trace` calls.

What happens when you run the following Haskell program?

```
import Debug.Trace

main =
  let x = trace "x" 1
      y = trace "y" 2
      foo x y = trace "foo" (x + y)
  in
      print (foo x y)
```

What if we replace `(x + y)` with `0`?

```
main =
  let x = trace "x" 1
      y = trace "y" 2
      foo x y = trace "foo" 0
  in
      print (foo x y)
```

What if we replace `(x + y)` with `(x + x + y)`?

```
main =
  let x = trace "x" 1
      y = trace "y" 2
      foo x y = trace "foo" (x + x + y)
  in
      print (foo x y)
```

How about:

```
sum100 n =
  let m = trace "Hello!" (sum [1..100]) in
  n * m

main =
    print (sum100 100) >>
    print (sum100 5)
```

Q: What is lazy evaluation?

Q: What are some advantages of lazy evaluation?

Static Types

Except for the weird function application syntax, Haskell looks a lot like Python. But Python has a **dynamic type system**, while Haskell has a **static type system**. The code below raises an error at compile time.

```
customGreeting "Mary" "Sue" = "Hey Mary!"
customGreeting fname lname  = False
                              ^^^^
    • Couldn't match expected type ‘[Char]’ with actual type ‘Bool’
    • In the expression: False
      In an equation for ‘customGreeting’:
        customGreeting fname lname = False
```

Q: Why is the compiler complaining?

Q: What's an "expected" type? What's an "actual" type?

Q: Define "static" and "dynamic".

Q: How did the compiler know what the type should be if we didn't write down any types??

A program is **well-typed** if it passes the type checker.

Types as documentation

We can tell GHC what type we *expect* a variable to have by adding a **type annotation** immediately above its declaration.

```
x1 :: Bool
x1 = True

x2 :: Integer
x2 = 12

x3 :: [Integer]
```

```haskell
x3 = [1,2,3]

x4 :: [Char]
x4 = 'x'

x5 :: (Char, Integer)
x5 = ('x', 5)

x6 :: String
x6 = "hello"

-- you can also add type annotations to nested declarations
x7 = x ++ "World"
  where
  x :: String
  x = "Hello "
```

`String` is a type alias for `[Char]`; strings are just lists of characters.

Q: If Haskell has type inference, what's the point of type annotations?

Function types look weirder:

```haskell
-- ok, makes sense
not :: Bool -> Bool
not True = False
not False = True

-- ???
compare :: Integer -> Integer -> String
compare x y
  | x < y     = "Less"
  | x == y    = "Equal"
  | otherwise = "More"

-- ????
cons :: a -> [a] -> [a]
cons x xs = (x : xs)
```

Q: Consider a function `f :: Name -> Address -> SSN -> Person`. How many parameters does it have? What are their types? What is the return type? What do you think this function does?

Q: Consider a function `f :: (Integer -> String) -> [Integer] -> [String]`. How many parameters does it have? What are their types? What is the return type? What do you think this function does?

Why do function types look like that? We won't be able to explain why until we learn about **currying** next week…