Episode I

The Haskell Menace

Learning outcomes

- Describe the difference between FP and OOP
- Model simple problems using types
- Write simple Haskell programs
  - Using constants, functions and lists

An imperative program

```
public int sum(int from, int to) {
    int total = 0;
    for(int i = from; i <= to; ++i) {
        total += i;
    }
    return total;
}
```

The same program, functionally

```
sum from to
  | from <= to = from+sum (from+1) to
  | otherwise  = 0
```

What's the difference?

- **One is based on mutation**
  - `++i, sum += i, etc.`
- **The other on equations and recursion**
  - `from + sum (from+1) to`

Functional programming

- Describing the *what*
  - Imperative programs describe the *how*
- Functions
- Recursion
- Immutable data

Don't all languages have functions?

- Imperative languages have “functions”
  - Results may depend on time, data, phases of the moon, etc.
  - May have *side effects*
    - Output text, erase your data, steal your dog
- Functional languages have *pure* functions
  - Result *only* depends on its parameters
  - No side effects
  - Easy to test and reason about

Haskell

- A purely functional language
  - Only pure functions
- Lazy evaluation
  - Computation only happens when needed
- Strict, expressive type system
  - Side effects controlled by the type system

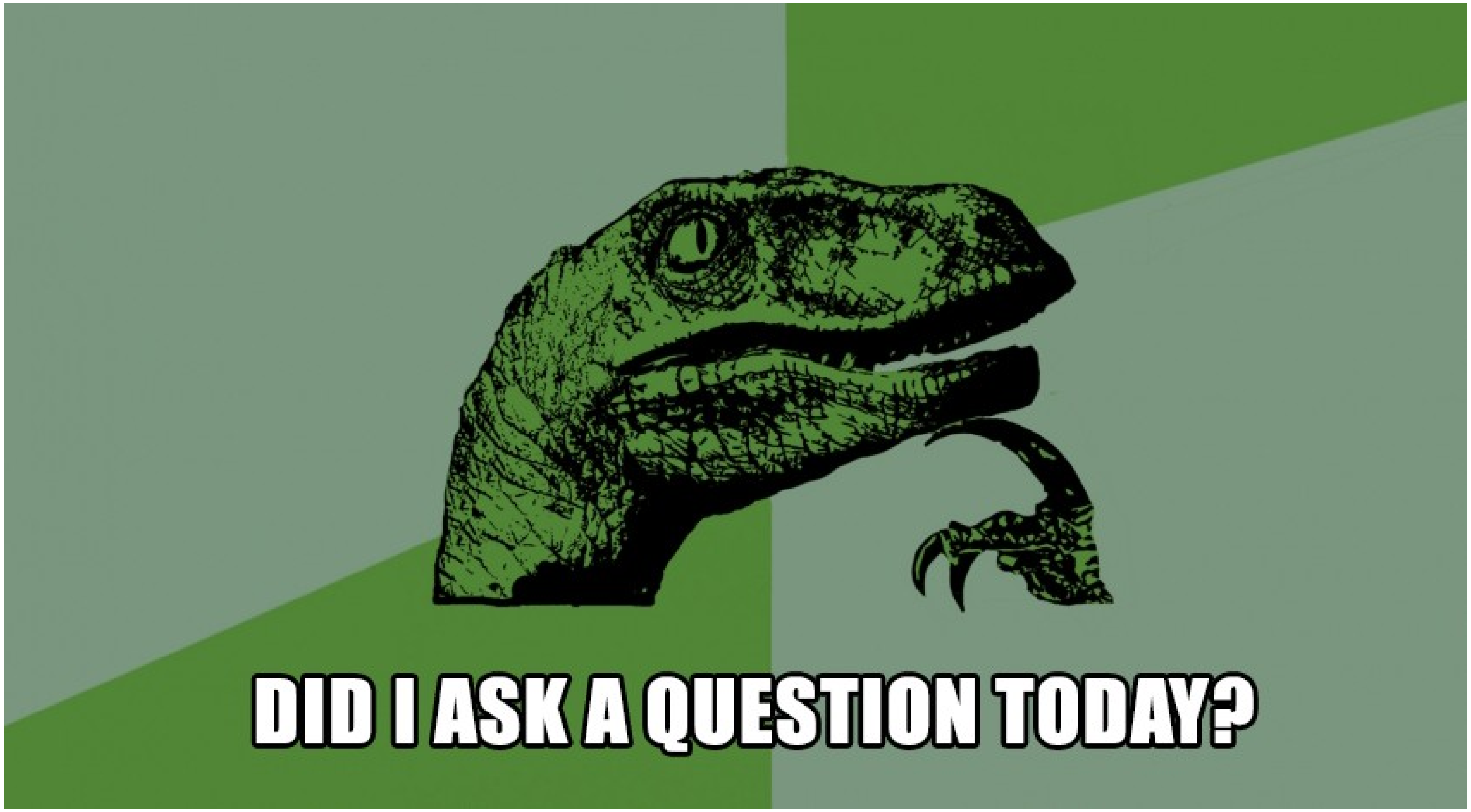
Why Haskell?

- Easy to test and reason about
- Strong mathematical connection
- Cool, highly paid jobs
- Write less code, go home early!

Self studies

- http //learnyouahaskell.com
  - Pedagogical, beginner friendly
- http //book.realworldhaskell.org
  - Pragmatical, geared towards programmers
- http //haskell.org/hoogle
  - API docs
- http //haskell.org/platform
  - Install Haskell on your own computer

Ask questions!



Exercise some statistics

- **Write a function** `average` **to calculate the mean of a list of integers**
- **What is the type of** `average`**?**
- **Write a function** `almostAverage` **to calculate the mean of a list of integers, excluding the largest and smallest element**
  - `almostAverage [2,1,4,3] == average [2,3]`

Episode 2

Attack of the Recursive Types

Type synonyms

- `type Company = String`
- `type Model = String`
- `type Version = Int`

Rolling your own types

- **Structure the data the way you want**
- **Model your problem domain using types**

  -
    ```
    data Phone = Android Company Model
               | IPhone Version
               | OldPhone
    ```
  - `myPhone = Android “Sony” “Z1 Compact”`
  - `dadsPhone = OldPhone`
  - `yourPhone = IPhone 6`

Pattern matching

- **Constructors can both construct and destruct**
- `hasPhone :: PersonName → Phone → String`

```
hasPhone person OldPhone =
  person ++ “ has an old phone :( “

hasPhone person (IPhone v) =
  person ++ “ has an IPhone “ ++ show v

hasPhone person (Android maker model) =
  person ++ “ has a “ ++ model
         ++ “ from “ ++ maker
```

Recursive types and functions

- Functions can refer to themselves

  -
    ```
    sum (x:xs) = x + sum xs
    sum []     = 0
    ```
- Types can too!

  -
    ```
    data List a
      = Empty
      | OneMore a (List a)
    ```

Exercise model your family tree

- What does a family tree look like?
- Express the structure as your own type
- Try to use both `type`, `data`, type variables and records!
  - Hint for type variables different use cases may require different information; sometimes a person may be just a name, some times you need more/other information
- Can you express your own family tree using your type?