Episode I

The Haskell Menace

Ett imperativt program

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

Samma program, funktionellt

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

Vad skiljer?

- Det första bygger på mutation och iteration
  - `++i, sum += i, etc.`
- Det andra bygger på ekvationer och rekursion
  - `from + sum (from+1) to`

Funktionell programmering

- Beskriver vad saker *är*
  - Imperativa program beskriver vad de *gör*
- Funktioner
- Rekursion
- Omuterbar data

Alla språk har väl funktioner?

- Imperativa språk har “funktioner”
  - Kan ge olika resultat beroende på tid, plats, etc.
  - Kan ha *sidoeffekter*
    - skriva ut text, radera din hårddisk, avfyra missiler, etc.
- FP har *rena* funktioner
  - En ren funktions resultat beror *endast* på dess indata
  - Rena funktioner har inga sidoeffekter
  - Enkla att testa och resonera kring

Haskell

- Ett rent funktionellt språk
  - Har bara rena funktioner
- Lat evaluering
  - Beräkning sker endast vid behov
- Strikt, uttrycksfullt typsystem
  - Sidoeffekter begränsade av typsystemet

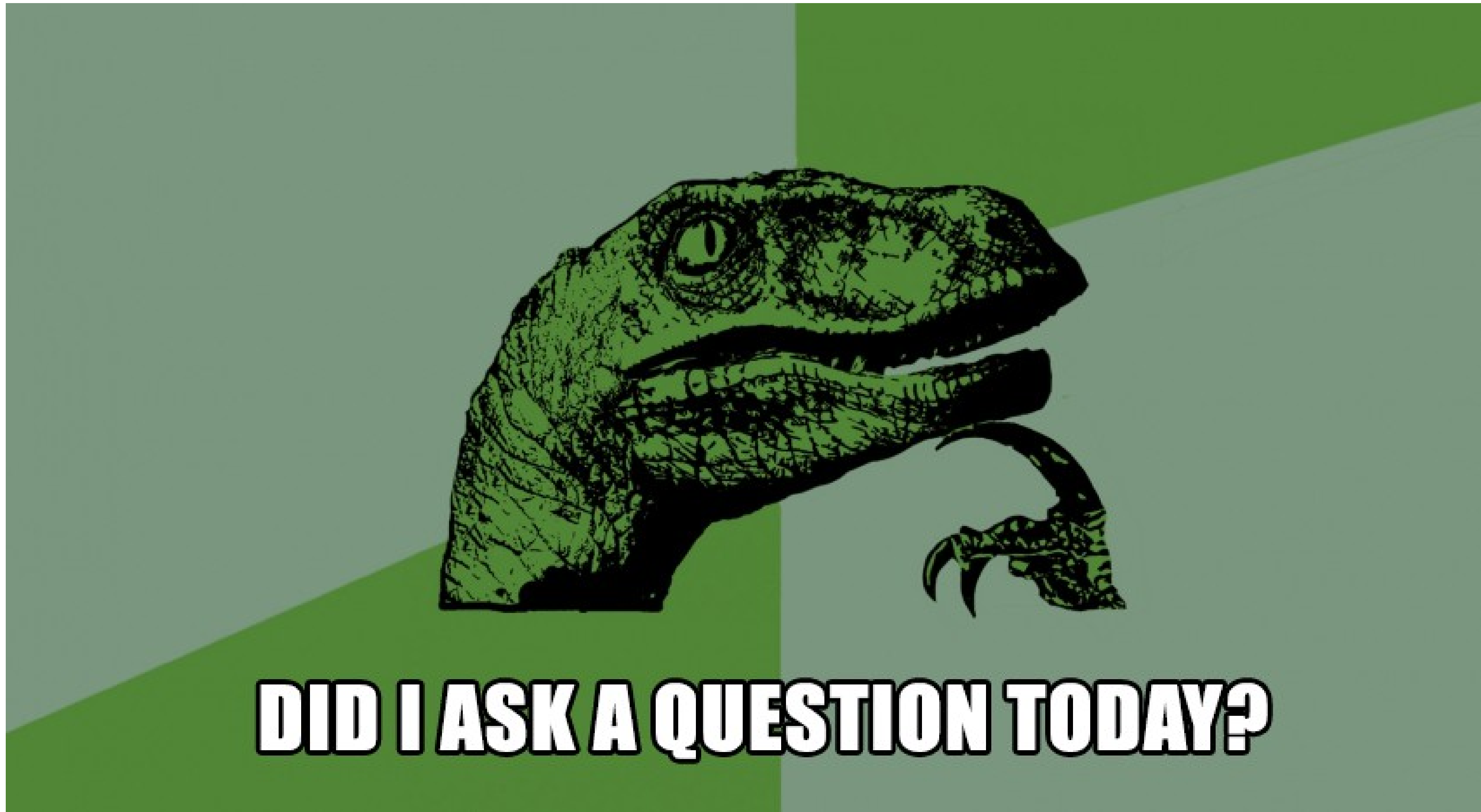
Varför Haskell?

- Lätt att testa och resonera kring
- Stark matematisk koppling
- Coola, välbetalda jobb
- Skriv mindre kod, gå hem tidigare!

Självstudier

- learnyouahaskell.com
  - Pedagogisk, lättsam, för alla
- book.realworldhaskell.org
  - Pragmatisk, för den programmeringsvane
- haskell.org/hoogle
  - API-dokumentation
- Haskell.org/platform
  - Installera Haskell på din dator

Ställ frågor!



Övning: statistikexercis

- Skriv en funktion `average` som beräknar medelvärdet av talen i en lista
- Vad har `average` för typ?
- Skriv en funktion `almostAverage` som räknar ut medelvärdet av alla tal i en lista, utom det största och det minsta
  - `almostAverage [2,1,4,3] == average [2,3]`

Episode 2

Attack of the Recursive Types

Typsynonymer

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

Egna datatyper

- Strukturera din data som det passar dig
- Modellera din problemdomän med typer

```
– data Phone = Android Company Model
             | IPhone Version
             | OldPhone
```

```
– myPhone = Android “Samsung” “Galaxy S3”
```

```
– dadsPhone = OldPhone
```

```
– yourPhone = Iphone 5
```

Mönstermatchning

- Konstruktorer kan både sätta ihop och ta isär
- `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
```

Rekursiva typer och funktioner

- Funktioner kan referera till sig själva – rekursion
  - ```
    sum (x:xs) = x + sum xs
    sum []     = 0
    ```
- Typer också – rekursiva typer
  - ```
    data List a
       = Empty
       | OneMore a (List a)
    ```

Övning: modellera ett släktträd

- Fundera: hur ser ett släktträd ut?
- Rita gärna diagram!
- Uttryck strukturen som en egen datatyp
- Försök använda både `type`, `data`, typvariabler och records!
  - Hint för typvariabler: olika tillfällen kräver olika info om personer – ibland bara ett namn, ibland namn + ålder, etc.
- Försök uttrycka ditt (eller någon annans) släktträd med hjälp av din typ – fungerar det?