Einleitung: Funktionales Programmieren

in Haskell

Christian Höner zu Siederdissen
christian.hoener.zu.siederdissen@uni-jena.de

Theoretische Bioinformatik, Bioinformatik Uni Jena

Okt 20^{st}, 2022

Raumwechsel!

ab *Donnerstag, 2022-10-27* findet die Vorlesung hier statt:
Jentower, Leutragraben 1, 07743 Jena
Raum: 08N04

- den Jentower betreten
- die Fahrstühle finden, sie befinden sich neben der Rezeption im Erdgeschoss
- vor den Fahrstühlen die "8" drücken
- ein Fahrstuhl

  $$A \ldots F$$

  wird angezeigt, diesen betreten und im 8. Stock aussteigen
- dort gibt es zwei Glastüren, eine davon werde ich euch auf lassen
- den Raum 08N04 finden (nach der Glastür gleich links am Gang)

Bürokratie Zuerst

- das hier wird eine Einführungsveranstaltung
- VL 90 Minuten, kombiniert mit kleinen Übungen
- "Hausaufgaben" werden nicht korrigiert, aber kurz besprochen (und bilden manchmal eine Brücke zur nächsten VL)
- Prüfung: mündlich: Themen der VL, funktionale Algorithmen und Datenstrukturen
- Von Vorteil ist: Notebook / Rechner um Code direkt ausprobieren zu können
- das direkte Ausprobieren ist ein bischen ein "Experiment"
- bitte bei Fragen direkt melden (wobei ich dann gerne noch den Gedanken zu Ende führe)
- Raumwechsel?

Tools, Literatur, etc

- ein installierter `GHC`: `www.haskell.org/ghc` (Version ziemlich egal)
- es gibt auch Online-Tools (nicht getestet von mir) `tryhaskell.org`
- Graham Hutton, Programming in Haskell
- Magic, Deep Magic, Black Magic (Code den wir hinnehmen, aber erst spät, vielleicht, nie besprechen)

Programmierparadigmen

- Prozedural
- Objekt-orientiert
- Logik-basiert
- *Funktional*

Programmieren in Haskell

- Funktional: Funktionen werden auf Argumente angewandt
- Pur: es gibt keine Assignments, kein $x = 1$
- statisch typisiert: Beim Compilieren sind alle Typen klar
- mit: Typinferenz, der Compiler kann *berechnen* welche Funktion welchen Typ hat
- und: lazy Evaluation, berechne nur was berechnet werden muss
- gestattet: leichteres (equational) reasoning

Summe in Python vs Haskell

Python:

```
1  def summe(upto):
2      acc = 0                          # assignment
3      for i in range(1,upto):
4          acc += i                     # assignment
5      return acc
```

Haskell:

```
1  summe upto = sum [1..upto]
```

- Keine Assignments (deeper magic incoming ...)
- Wir kommen gleich noch zurück zu `sum`
- Funktionen: `sum` und `..`

Fibonacci Zahlen

$0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144$

- Wir können Pattern Matching in Funktionen betreiben
- Mehrere Zeilen untereinander werden nach und nach abgearbeitet
- `if then else` erfordert immer alle Fälle
- `error` ist black magic

fibRec an Tafel

Fibonacci Zahlen

1			0	1	1	2	3	5	8	13	21	34	55	
2			1	1	2	3	5	8	13	21	34	55	89	
3	0	1	1	2	3	5	8	13	21	34	55	89	144	
4														
5	1	2	3	4	5	6	7	8	9	10	11	12	13	...

fibMemo

Was tut diese Funktion?

```
1  unbekannt [] = []
2  unbekannt (x:xs) =
3    let ls = [y | y <- xs, y <= x]
4        rs = [y | y <- xs, y >  x]
5    in  unbekannt ls ++ [x] ++ unbekannt rs
```

- Listen
- Leere vs nicht-leere Liste
- Fallunterscheidung / pattern matching
- list comprehensions / syntactic sugar