Dieses Dokument ist vom 30.10.2025. Die aktuelle Version des Dokuments kannst du im moodle oder direkt von GitHub herunterladen.

Dieses Dokument enthält Fragen, kleine Aufgaben und andere Ressourcen zum Thema Deklarative Programmierung. Die Inhalte dieses Dokuments sollen dir helfen, dein Verständnis über Haskell und Prolog zu prüfen.

Größere Aufgaben haben wir als Challenges markiert. Diese Aufgaben benötigen öfter mehrere Konzepte und führen zusätzlich Konzepte ein, die nur für das Lösen der Aufgabe wichtig sind.

Wenn du Anmerkungen oder weitere Ideen für Inhalte für dieses Dokument hast, schickt uns diese gerne über z.B. mattermost - oder stellt eine PR auf GitHub.

Funktionale Programmierung

Test 1 Was bedeutet es, wenn eine Funktion keine Seiteneffekte hat?

Test 2 Haskell ist eine streng getypte Programmiersprache. Was bedeutet das?

Test 3 Wenn du eine Schleife in Haskell umsetzen möchtest, auf welches Konzept musst du dann zurückgreifen?

Test 4 In imperativen Programmiersprachen sind Variablen Namen für Speicherzellen, deren Werte zum Beispiel in Schleifen verändert werden können. Als Beispiel betrachte

```
def clz(n):
  k = 0
  while n > 0:
    n //= 2
    k += 1
  return 64 - k

def popcnt(n):
  k = 0
  while n > 0:
    if n % 2 == 1:
      k += 1
    n //= 2
  return k
```

In Haskell sind Variablen keine Namen für Speicherzellen. Wie können wir dieses Programm in Haskell umsetzen? Wo wandert das `k` hin?

Test 5 Auf was müssen wir achten, wenn wir eine rekursive Funktion definieren? Die Antwort ist abhängig von dem, was die Funktion berechnen soll. Denke über die verschiedenen Möglichkeiten nach.

Test 6 Gegeben sei das folgende Haskell-Programm.

```
even :: Int -> Bool
even 0 = True
even n = odd (n - 1)

odd :: Int -> Bool
odd 0 = False
odd n = even (n - 1)
```

- Berechne das Ergebnis von `odd (1 + 1)` händisch.
- Wie sieht der Auswertungsgraph für den Ausdruck `odd (1 + 1)` aus?

- Welcher Pfad entspricht deiner händischen Auswertung?
- Welcher Pfad entspricht der Auswertung wie sie in Haskell stattfindet?
- Welcher Pfad entspricht der Auswertung wie sie in Python sinngemäß stattfindet?

Test 7 Es wird als sauberer Programmierstil angesehen, Hilfsfunktionen, die nur für eine Funktion relevant sind, nicht auf der höchsten Ebene zu definieren. Mithilfe welcher Konstrukte kannst du diese lokal definieren?

Test 8 Das Potenzieren einer Zahl x (oder eines Elements einer Halbgruppe) mit einem natürlich-zahligen Exponent n ist in $\mathcal{O}(\log n)$ Laufzeit möglich[1]. Dafür betrachten wir

$$x^n = \begin{cases} \left(x^{\frac{n}{2}}\right)^2 & \text{falls } n \text{ gerade} \\ x \cdot x\left(x^{\frac{n-1}{2}}\right)^2 & \text{sonst} \end{cases}$$

Implementiere eine Funktion, die diese Variante des Potenzierens umsetzt.

Test 9 Gegeben ist folgender Ausdruck.

```
let v = 3
    w = 5
    x = 4
    y = v + x
    z = x + y
 in y
```

Welche Belegungen der Variablen werden tatsächlich berechnet, wenn wir `y` ausrechnen?

Test 10 Ist der folgende Ausdruck typkorrekt?

```
if 0 then 3.141 else 3141
```

Test 11 Wie werden algebraische Datentypen in Haskell definiert?

Test 12 Was ist charakterisierend für Aufzählungstypen, einen Verbundstypen und einem rekursiven Datentypen? Gebe Beispiele für jeden dieser Typarten an.

Test 13 Geben ist der Typ `IntList` mit `data IntList = Nil | Cons Int IntList`. Weiter kann mithilfe der Funktion

```
lengthIntList :: IntList -> Int
lengthIntList Nil         = 0
lengthIntList (Cons _ xs) = 1 + lengthIntList xs
```

die qLänge einer solchen Liste berechnet werden. Du möchtest nun auch die Längen von Listen berechnen, die Buchstaben, Booleans oder Gleitkommazahlen enthalten. Was stört dich am bisherigen Vorgehen? Kennst du ein Konzept mit dessen Hilfe du besser an dein Ziel kommst?

Test 14 Wie ist die Funktion `lengthIntList :: IntList -> Int` aus dem vorherigen Test definiert?

Test 15 Du hast einen Datentypen definiert und möchtest dir Werte des Typen nun z.B. im GHCi anzeigen lassen. Was kannst du tun, um an dieses Ziel zu kommen?

Test 16 Wie definieren wir Funktionen?

[1] Binäre Exponentiation – Wikipedia

Test 17 Gebe ein Listendatentypen an, für den es nicht möglich ist, kein Element zu enthalten.

Test 18 In Programmiersprachen wie Java greifen wir Daten komplexer Datentypen zu, indem wir auf Attribute von Objekten zugreifen oder getter-Methoden verwenden. Wie greifen wir auf Daten in Haskell zu?

Test 19 Wie sieht eine Datentypdefinition im Allgemeinen aus?

Test 20 Welchen Typ haben
- `(:)` und `[]`,
- `Just` und `Nothing`?

Test 21 Was ist parametrischer Polymorphismus?

Test 22 Welche Typkonstruktoren des kinds `* -> *` kennst du?

Test 23 Welchen kind hat `Either a`?

Test 24 Beim Programmieren vernachlässigen redundante Syntax. Gibt es einen Unterschied zwischen `f 1 2` und `f(1, 2)`

Test 25 Welches Konzept erlaubt es uns, dass wir Funktionen auf Listen nicht für jeden konkreten Typen angeben müssen?

Test 26 Wie gewinnt man aus einem Typkonstruktor einen Typ?

Test 27 Visualisiere `[1, 2, 3]` als Baum, wie du es in der Vorlesung kennengelernt hast. Zur Erinnerung: die inneren Knoten sind Funktionen und die Blätter Werte, die nicht weiter ausgerechnet werden können.

Test 28 Ist `[32, True, "Hello, world!"]` ein valider Haskell-Wert? Warum ja oder nein?

Test 29 Was ist der Unterschied zwischen einem Typ und einem Typkonstruktor?

Challenge 1
- Der größte gemeinsamen Teiler (ggT) zweier Ganzzahlen kann mithilfe des euklidschen Algorithmus berechnet werden. Implementiere das Verfahren.

$$\gcd(x, y) = \begin{cases} |x| & \text{falls } y = 0 \\ \gcd(y, x \bmod y) & \text{sonst} \end{cases}$$

- Alternativ kann der ggT auch berechnet werden, indem wir das Produkt des Schnittes der Primfaktorzerlegung der beiden Zahlen betrachten, also

$$\prod(\mathrm{PF}(x) \cap \mathrm{PF}(y))$$

  wobei PF die Menge der Primfaktoren der gegebenen Zahl (mit entsprechenden Mehrfachvorkommen) beschreiben soll. Implementiere diesen Ansatz.

Challenge 2 Die Ableitung einer Funktion $f : \mathbb{R} \to \mathbb{R}$ kann mithilfe des Differenzenquotienten $\frac{f(x+h)-f(x)}{h}$ für kleines h approximiert werden. Ein andere Methode zur Berechnung der Ableitung ist symbolisches Differenzen und ähnelt dem, wie wir analytisch Ableitungen berechnen. Eine Funktion sei dargestellt durch den folgenden Typ:

```haskell
data Fun = X            -- x     (Variable x)
         | E            -- e     (Euler's constant)
         | Num Double   -- c     (Constant)
         | Ln Fun       -- ln    (Natural logarithm)
         | Fun :+: Fun  -- f + g (Addition)
```

```
          | Fun :-: Fun  -- f - g (Subtraction)
          | Fun :*: Fun  -- f * g (Multiplication)
          | Fun :/: Fun  -- f / g (Division)
          | Fun :<: Fun  -- f o g (Composition)
          | Fun :^: Fun  -- f ^ g (Exponentiation)

-- Example
f :: Fun
f = (E :^: X) :<: (X :*: X)  -- (e^x) o (x * x) = e^(x^2)

-- Example
g :: Fun
g :: let x = X
         x2 = x :*: x
         x3 = x2 :*: x
     in x3 :+: x2 :+: x :+: Num 1.0  -- x^3 + x^2 + x + 1
```

- Implementiere eine Funktion `($$) :: Fun -> Double -> Double`, die eine gegebene Funktion in einem gegebenen Punkt auswertet.
- Implementiere eine Funktion `derive :: Fun -> Fun`, die eine gegebene Funktion ableitet.[2]

Challenge 3 In Einführung in die Algorithmik hast du verschiedene Varianten des `mergesort`-Algorithmus kennengelernt. Eine davon hat ausgenutzt, dass in einer Eingabeliste bereits aufsteigend sortierte Teillisten vorkommen können, um den Algorithmus zu beschleunigen.[3] Implementiere diese Variante in Haskell.

Für den Anfang kannst du annehmen, dass die Eingabelisten vom Typ `[Int]` sind. Wenn wir Typklassen behandelt haben, kannst du `Ord a => [a]` nutzen.

Challenge 4 Entwickle einen Datentyp `Ratio` um rationale Zahlen

$$\frac{p}{q} \in \mathbb{Q}, \quad p \in \mathbb{Z}, q \in \mathbb{N}, p \text{ und } q \text{ teilerfremd}$$

darzustellen. Implementiere die Operationen: Addition, Subtraktion, Multiplikation, Divison. Implementiere weiter eine Funktion, die die rationale Zahl als reelle Zahl mit einer festen Anzahl von Nachkommastellen darstellt.

Test 30 Wie können wir es hinkriegen, dass die invalide Liste `[32, True, "Hello, world!"]` ein valider Haskell-Wert wird? Mithilfe welches Hilfstypen kriegen das hin?

Test 31 Du hast bereits viele Funktionen kennengelernt, die in der Haskell `base`-library implementiert sind. Anstatt eine konkrete Liste dieser Funktionen anzugeben, möchten wir dich motivieren, folgende Dokumentationen verschiedener Module anzuschauen.

- Prelude
- Data.List

Wenn du merkst, die Implementierung einer bekannten Funktion fällt dir ad hoc nicht ein, nehme dir Zeit und überlege, wie du sie implementieren könntest.

Test 32 Hier ist eine fehlerhafte Implementierung eines Datentyps für einen knotenbeschrifteten Binärbäumen.

```
data Tree a = Empty | Node Tree a Tree
```

[2] Zusammenfassung der Ableitungsregeln

[3] Falls du interessiert bist: In der Haskell `base`-library wird `sort` aus `Data.List` sehr ähnlich implementiert: Data.List.sort.

Was ist der Fehler?

Test 33 In imperativen Programmierung iterieren wir über Listen oft in folgender Form (in Java).

```
List<Integer> a = new ArrayList<>();
a.add(3); a.add(1); a.add(4); a.add(1); a.add(5);

List<Integer> b = new ArrayList<>();
for (int i = 0; i < a.size(); i++) {
  b.add(2 * a.get(i));
}
```

Wenn wir diesen Code naiv in Haskell übersetzen, könnten wir z.B.

```
double :: [Int] -> Int -> [Int]
double xs i | i < length xs = 2 * xs !! i : double xs (i + 1)
            | otherwise     = []
```

Das wollen wir niemals so tun.

- Wie unterscheiden sich die Laufzeiten?
- Optimiere die Funktion `double`, sodass sie lineare Laufzeit in der Länge der Liste hat.

Test 34 Die `(!!)`-Funktion ist unsicher in dem Sinne, dass sie für invalide Listenzugriffe einen Fehler wirft. Die Funktion `(!?) :: [a] -> Int -> Maybe a` ist eine sichere Variante von `(!!)`. Sie macht den Fehlerfall explizit durch die Wahl des Ergebnistypen. Was tut diese Funktion voraussichtlich? Implementiere diese Funktion.[4]

Test 35 `(++) :: [a] -> [a] -> [a]` wird verwendet, um zwei Listen aneinander zu hängen. Wenn wir eine Funktion induktiv über den Listentypen definieren wie z.B. `square :: [Int] -> [Int]`, die jeden Listeneintrag quadrieren soll, dann können wir das wie folgt tun.

```
square :: [Int] -> [Int]
square []     = []
square (x:xs) = [x * x] ++ square xs
```

Dann ist Funktion zwar korrekt aber nicht Haskell-idiomatisch. Was müssten wir an der Funktion ändern, damit sie idiomatisch ist.

[4] Diese Funktion ist auch bereits vorimplementiert: `(!?)` in `Data.List`.

Hinweise zu Tests und Challenges

Hinweis zu Challenge 1 Zur Darstellung der Multimengen eignen sich sortierte Listen gut.

Hinweis zu Challenge 1 Zur Berechnung des Schnittes können zwei sortierte Listen parallel durchlaufen werden. Wenn zwei gleiche Elemente zu Beginn der Liste stehen, wird eines der Elemente zum Ergebnis hinzugefügt. Im anderen Fall überspringen wir das jeweils kleinere Element der beiden.

Weitere Ressourcen

Wenn du auf der Suche nach weiteren Übungsaufgaben bist, mit denen du deine Programmierkenntnisse in Prolog verbessern möchtest, bietet sich die Liste P-99: Ninety-Nine Prolog Problems an. Lösungen sind ebenso auf der Seite verfügbar. Für Haskell gibt es eine ähnliche Seite H-99: Ninety-Nine Haskell Problems.

Weitere Links:

- Learn You A Haskell
- Haskelite: Ein Schritt-für-Schritt Interpreter für (eine Teilmenge von) Haskell
- Haskell Cheatsheet