Prüfung vom 6. November 2018

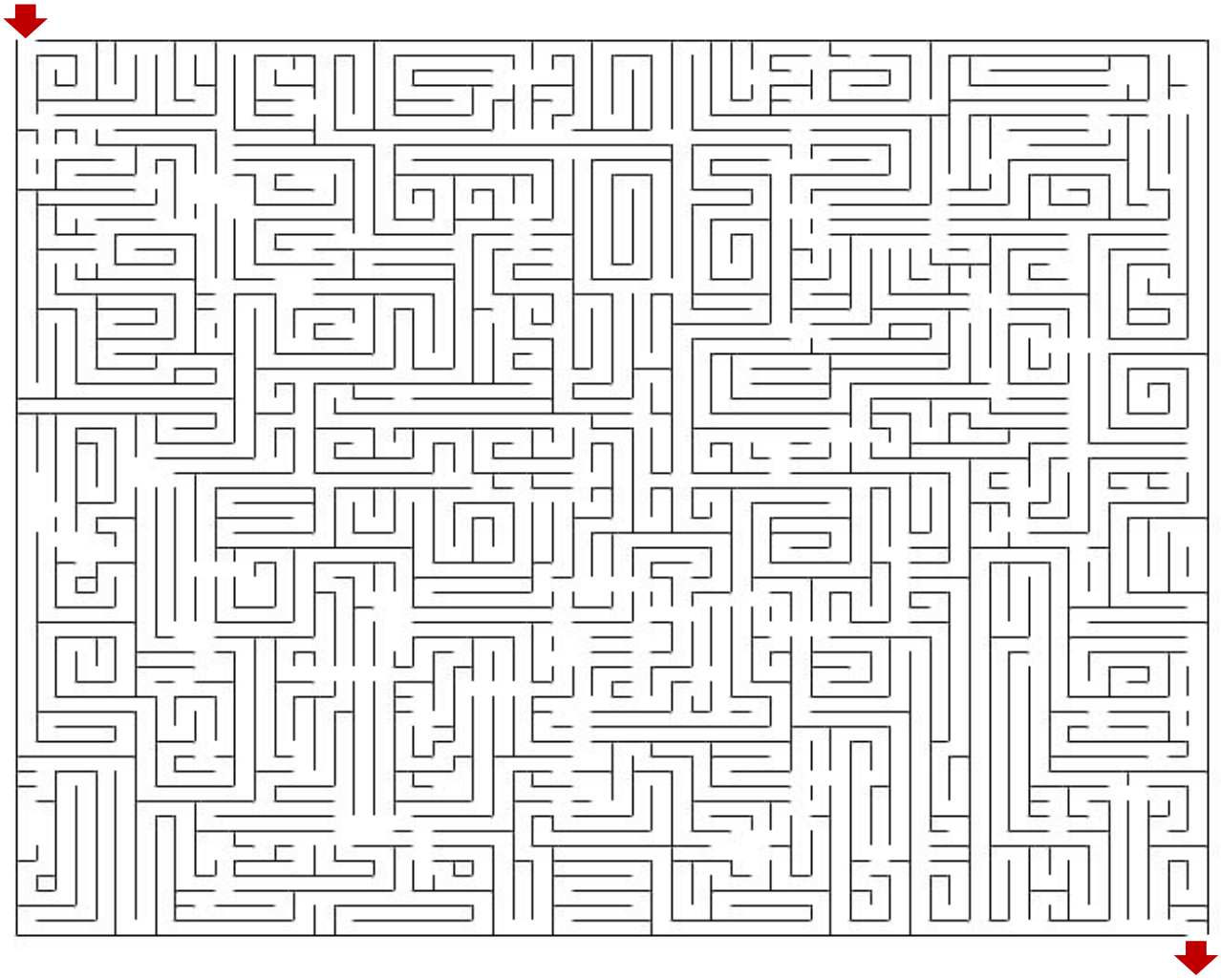
Name, Vorname: _____

Allgemeine Hinweise:

1) Schreiben Sie Ihren Namen auf diese Blatt.
2) Sie dürfen ein beidseitig beschriebenes/bedrucktes DIN-A4 Blatt mit Notizen verwenden. Sonst ist ausser einem Stift nichts erlaubt.
3) Bitten beantworten Sie die Fragen direkt auf den Aufgabenblättern. Notizen schreiben Sie auf die jeweils leeren Rückseiten.
4) Lesen Sie die Prüfung zuerst durch und beginnen Sie mit den Aufgaben, die Ihnen am meisten liegen.
5) Sie haben 60 Minuten Zeit.
6) Bleiben Sie bitte Ruhig an Ihrem Platz bis die Zeit um ist.
7) Ihre Notizen werden ebenfalls eingezogen. Sie bekommen Sie mit der korrigierten Prüfung zurück.

Viel Erfolg!

*Falls Sie frühzeitig fertig sind, können Sie sich gerne mit folgendem Labyrinth beschäftigen. Es ist **nicht** Notenrelevant.*



Aufgabe 1: Functional Programming Basics **(4 Punkte)**

Kreuzen Sie für alle folgenden Aussagen entweder *Richtig* oder *Falsch* an. Aussagen ohne Kreuz, mit zwei Kreuzen oder unklar angekreuzte Felder werden neutral mit 0 Punkten bewertet. Lesen Sie die Aussagen genau durch!

(0.5 Punkte pro richtige Antwort, *0.5 Punkte Abzug pro falsche Antwort*, min. 0 Punkte)

Aussage	Richtig	Falsch
Mit Haskell kann man keine Dateien einlesen.		X
`data A = B | C` definiert die zwei Typen B und C.		X
Die Anzahl Komponenten eines Tuples kann man an seinem Typ erkennen.	X	
Funktionskomposition bindet stärker als Funktionsapplikation.		X
`f :: Eq a => a -> a -> Bool` In obiger Signatur bedeutet das Eq a, dass Werte vom Typ a vergleichbar sein müssen.	X	
Funktionen kann man auch in Tuples packen. Z.B. folgender Wert ist legal: `(head, tail, not)`	X	
Haskell Listen sind immutable. Funktionen auf Listen verändern die ursprüngliche Liste nicht, sondern erzeugen eine neue Liste als Resultat.	X	
Operatoren kann man prefix schreiben, wenn man sie in Klammern packt: (+) 1 2	X	
In Haskell hat jedes `if` auch ein `else`.	X	
Wenn man Funktionsanwendungen nicht klammert, probiert Haskell alle möglichen Klammerungen durch und nimmt dann jene Version, die mit den wenigsten Klammern legal ist.		X

Aufgabe 2: Typen **(6 * 2 = 12 Punkte)**

Gesucht ist der Typ des jeweiligen Ausdrucks. Verwenden Sie `Int` für allfällige numerische Typen.
Wenn Sie keine oder mehrere Felder ankreuzen, gilt die Aufgabe als falsch, genauso unklar angekreuzte Felder. (2 Punkte pro korrekte Antwort)

Gegeben sind folgende Definitionen:

```
f :: (Char -> Int) -> Bool
g :: Char -> Bool
h :: [a] -> a
not :: Bool -> Bool
map :: (a -> b) -> [a] -> [b]
filter :: (a -> Bool) -> [a] -> [a]
head :: [a] -> a
```

`g ' '`
☐ `Char -> Bool`
☐ `(Chat -> Bool) -> Bool`
X `Bool`
☐ Ungültiger Ausdruck

`(\a -> not (f a))`
X `(Char -> Int) -> Bool`
☐ `Char -> Int -> Bool`
☐ `Bool`
☐ Ungültiger Ausdruck

`map tail []`
☐ `[]`
☐ `[a]`
X `[[a]]`
☐ Ungültiger Ausdruck

`g.h.h`
X `[String] -> Bool`
☐ `String -> Bool`
☐ `Char -> [Bool]`
☐ Ungültiger Ausdruck

`head [head]`
X `[a] -> a`
☐ `[[a] -> a] -> [a]`
☐ `[[a] -> a] -> a`
☐ Ungültiger Ausdruck

`map (\p -> p "what?") [tail]`
☐ `String`
X `[String]`
☐ `[String -> b]`
☐ Ungültiger Ausdruck

Aufgabe 3: Werte **(7 * 1 = 7 Punkte)**

Gesucht ist jeweils der Wert von a. Schreiben Sie den Wert auf die dafür vorgesehene Linie.
(1 Punkt pro Aufgabe)

a)

```
f (x:y:z:zs) = y * x + z

a = f [4,3,2,1]
```

Wert von a : _____14_____

b)

```
a = head [\x -> x - x, (+2), (*3)] 3
```

Wert von a : _____0_____

c)

```
a = (\t -> tail (fst t)) ("abc", "def")
```

Wert von a : _____"bc"_____

d)

```
f (x,(y,(z1:z2:zs))) = zs

a = f (1,(2,[3,4]))
```

Wert von a : _____[]_____

e)

```
a = (tail . head . tail) [[1,2,3], [4,5,6]]
```

Wert von a : _____[5,6]_____

f)

```
a = curry snd 3 4
```

Wert von a : _____4_____

g)

```
data ABC = X | Y | Z deriving (Eq)

a = let f = sum . map snd . filter ((==Y).fst)
    in f [(X, 17), (Y, 2), (Y, 40)]
```

Wert von a : _____42_____

Aufgabe 4: Listen **(4 Punkte)**

Kreuzen Sie für alle folgenden Werte an, ob sie legale Haskell Listen sind oder illegale. Aussagen ohne Kreuz, mit zwei Kreuzen oder unklar angekreuzte Felder werden neutral mit 0 Punkten bewertet.
(0.5 Punkte pro richtige Antwort, 0.5 Punkte Abzug pro falsche Antwort, min. 0 Punkte)

Wert	Legal	Illegal
`["Hmm"]`	X	
`[False, not]`		X
`[]:[]`	X	
`'[':"]"`	X	
`"What?"`	X	
`[filter (\a -> a), map not]`	X	
`[[[]]]`	X	
`[(")[","](")]`	X	

Aufgabe 5: Generische Funktion **(4 Punkte)**

Implementieren Sie eine Funktion mit folgender Typsignatur:

```
f :: a -> ((a,b) -> c) -> (a -> b) -> c
```

Die zu definierende Funktion muss legal terminieren. D.h. die Verwendung der Funktionen `error` und `undefined` sowie die Verwendung von Rekursion die nicht abbricht, ist nicht erlaubt. Ebenso nicht erlaubt sind vordefinierte Funktionen. Verwenden Sie allenfalls Patternmatching.

Hinweis: Es gibt nur eine einzige Lösung. Schauen Sie genau auf die Typen und überlegen Sie sich, was als Parameter reinkommt und was Sie damit tun können.

f a fabc fab = fabc (a, fab a)
(a | drei Params)
(b | fab auf a)
(c | Tuple bauen mit a und b)
(d | fabc auf tuple anwenden)

Aufgabe 6: Webshop **(1 + 4 + 4 = 9 Punkte)**

In dieser Aufgabe implementieren Sie Funktionen für einen Computer Hardware Shop. Folgende Kategorien von Produkten werden angeboten:

```
data Category = Storage | CPU deriving Eq
```

Ein Produkt besteht aus einem Titel (String), dessen Produkt Kategorie und dem Preis in CHF (Int).

```
type Product = (String, Category, Int)
```

Der Shop verwaltet alle Produkte in einer Liste. Hier ein Beispiel:

```
products :: [Product]
products = [
  ("2 TB SSD", Storage, 300),
  ("Intel Core I7", CPU, 800),
  ("AMD Ryzen", CPU, 700)]
```

Gegeben:

```
map :: (a -> b) -> [a] -> [b]
filter :: (a -> Bool) -> [a] -> [a]
```

```
sum :: [Int] -> Int
```

Aufgaben:

<u>Eigene rekursive Implementierungen sind in dieser Aufgabe nicht erlaubt!</u>

a) Implementieren Sie die Funktion `price`, die den Preis eines Produktes extrahiert:
`price :: Product -> Int`
Beispiel:
`price ("Intel Core I7", CPU, 800) == 800`

```
price (_, _, p) = p -- (a) Pattern mit 3 Komponenten und die dritte zurückgeben
```

b) Gesucht ist die Funktion `productsByCategory` die alle Produkte der gesuchten Kategorie zurückgibt:
`productsByCategory:: Category -> [Product] -> [Product]`
Beispiel:
`productsByCategory CPU products == [("Intel Core I7",CPU,800),("AMD Ryzen",CPU,700)]`

```
productsByCategory cat ps = filter (\(_, c, _) -> c == cat) ps

(a | verwenden von filter)
(b | category c extrahieren im Prädikat)
(c | cat mit c vergleichen)
(d | ps als zweiter Param)
```

c) Gesucht ist die Funktion `total`, die für alle Produkte der gesuchten Kategorie die Summe deren Preise zurückgibt: `total :: Category -> [Product] -> Int`
Beispiel:

```
total Storage products == 300
total CPU products == 1500 -- (800 + 700)
```

Verwenden Sie die in Aufgabe b) definierte Funktion um alle Produkte der gesuchten Kategorie zu finden und die in Aufgabe a) definierte Funktion um die Preise der Produkte zu extrahieren.
Hinweis: Eine Liste mit Zahlen lässt sich leicht mit der `sum` Funktion aufaddieren.

```
total  cat ps = sum (map price (productsByCategory cat ps))

(a | productsByCategory mit cat und ps)
(b | map price
(c | map mit Resultat von a(=)
(d | sum auf [Int] der Preise)
```

Aufgabe 7: Rekursion **(6 + 6 + 5 = 16 Punkte)**

In dieser Aufgabe programmieren Sie selbst rekursive Funktionen.

Die Verwendung von vordefinierten Listenfunktionen ist in dieser Aufgabe nicht erlaubt. Verwenden Sie Patternmatching um die Listen zu zerlegen. head, tail und length sind ebenfalls nicht erlaubt.

a) Gesucht ist die rekursive Implementierung der Funktion `dropElem`. Sie nimmt ein Wert vom Typ a und eine Liste mit Elementtyp a und entfernt jedes Vorkommnis des Elements in der Liste. Die Funktion hat folgende Signatur:

```
dropElem :: Eq a => a -> [a] -> [a]

Beispiele:
dropElem 1 [2,1,4] == [2,4]
dropElem 'b' ['a','b','c','b'] == ['a','c']


dropElem _ [] = []
(a | Basecase [] mit [] als Resultat)

dropElem a (b:bs) | a == b    = dropElem a bs
(b | List mit Pattern zerlegen)
(c | Case a == b)
(d | b wegwerfen und recurse)

                  | otherwise = b:dropElem a b
(e | Case a/= b)
(f | b behalten und recurse)
```

b) Gesucht ist die rekursive Implementierung der Funktion `separate`. Die Funktion hat folgende Typsignatur: `separate :: String -> String -> String`
Die Funktion nimmt einen ersten String und einen zweiten String (Separator) und setzt zwischen jede zwei Chars des ersten Strings den Separator.

Beispiele:

```
separate "abc" "<>" == "a<>b<>c"
separate "" "+" == ""
separate "a" "X" == "a"


separate [] _  = []
(a | Basecase [] mit Resultat [])
separate [a] _ = [a]
(b | case 1 Element)
(c | List mit dem einzelnen Element zurückgeben)
separate (a:as) sep = a : sep ++ separate as sep
(d | match auf head)
(e | recursive auf Tail)
(f | a : sep ++ zusammenbauen)
```

c) Gesucht ist die rekursive Implementierung der Funktion `compose`. Die gesuchte Funktion hat folgende Typsignatur:
`compose :: [a->a] -> (a->a)`

Sie nimmt eine Liste von Funktionen und baut daraus eine einzige Funktion, nämlich die Komposition der enthaltenen Funktionen.
`compose [f,g,h]` soll folgende Funktion konstruieren: `f . (g . h)`

Für die leere Liste soll die Identitätsfunktion `id` zurückgegeben werden:

```
id :: a -> a
id a = a
```

Beispiele:
```
compose [(*2), (+1), (*5)] 2 == 22
compose [tail, filter (>3)] [6,5,4,3,2,1] == [5,4]
```

Verwenden Sie Patternmatching um die Input-Liste zu zerlegen. Verwenden Sie zudem den Kompositionsoperator `(.)` um die Funktionen zu verketten.
Zur Erinnerung:

```
(.) :: (b -> c) -> (a -> b) -> (a -> c)
```

```
compose [] = id
(a | Basecase [] mit id)
compose [f] = f
(b | Basecase [f] mit f)

compose (f:fs) = f . compose fs
(c | Case mit head / tail zerlegen)
(d | compose auf tail)
(e | f . Resultat)
```