Technische Universität Wien

NON-MANDATORY STUDENT NOTES FOR CONSENTING ADULTS

Funktionale Programmierung

Prof. Jens Knoop

notes by
Mirkl Mork der Markur Mer oder Tom Poise

10. Juli 2021


Haskell

Bemerkung

Im nachfolgenden Dokument befinden sich Screenshots aus den Vorlesungsskripten zu den LVAs »Funktionale Programmierung« und »Fortgeschrittene Funktionale Programmierung« von Prof. Jens Knoop. Beide Skripten sind online abrufbar unter:

http://www.complang.tuwien.ac.at

Fast alle sich hier befindenden Inhalte sind also mit ©Jens Knoop zu lesen.

Viel Spaß,
Merkules

Inhaltsverzeichnis

1 :: eBoc Links

Schöne Übersicht für gern benutzte, vordefinierte Funktionen:

http://www.cse.chalmers.se/edu/year/2019/course/TDA555/tourofprelude.html

Das überaus zugängliche beginner tutorial:

http://learnyouahaskell.com/chapters

. . . und natürlich, für fast alle anderen Fragen:

https://hackage.haskell.org/

2 :: Motivation

∅.hs bietet Taschenrechnerfunktionalität (abs, sin, sqrt, . . .) [GCHi, Hugs]. Es folgt etwas Schwurbelei: Funktionale Sprachen

- beruhen auf dem mathematischen Funktionsbegriff.
- Programme sind Systeme von Funktionen, die über Gleichungen, Fallunterscheidungen und Rekursion definiert sind und auf (strukturierten) Daten arbeiten.
- bieten effiziente, anforderungsgetriebene Auswertungsstrategien, die auch die Arbeit mit (potentiell) unendlichen Strukturen unterstützen.

Imperative Programmierung (befehlsorientiert) ist gekennzeichnet durch:

1. Unterscheidung von Ausdrücken und Anweisungen.
2. Ausdrücke liefern Werte; Anweisungen bewirken Zustandsänderungen (Seiteneffekte).
3. Programmausführung ist die Abarbeitung von Anweisungen (dabei müssen auch Ausdrücke ausgewertet werden).
4. Explizite Kontrollflussspezifikation mittels spezieller Anweisungen (sequentielle Komposition, Fallunterscheidung, Schleifen,...)
5. Variablen sind Namen für Speicherplätze: ihr Wert sind die dort gespeicherten Werte; sie können im Verlauf der Programmausführung (beliebig oft) geändert werden.
6. Bedeutung des Programms ist die Beziehung zwischen Anfangs- und Endzuständen, die bewirkte Zustandsänderung.

Funktionale Programmierung (ergebnisorientiert) ist gekennzeichnet durch:

1. Keine Anweisungen! Ausschließlich Ausdrücke!
2. Ausdrücke liefern Werte. Anweisungen fehlen; deshalb: keine Zustandsänderungen, keine Seiteneffekte.

3. Programmausführung ist Auswertung von Ausdrücken.
4. Keine Kontrollflussspezifikation! Allein Datenabhängigkeiten steuern die Auswertung(sreihenfolge).
5. Variablen sind Namen für Ausdrücke: ihr Wert ist der Wert des Ausdrucks, den sie bezeichnen; ein späteres Ändern, Überschreiben oder Neubelegen ist nicht möglich.
6. Bedeutung des Programms ist die Beziehung zwischen Aufrufargumenten von Ausdrücken und ihrem Wert.

Beispiele:

- Imperativ: Wertzuweisung, temporär, abänderbar

  `a := (17+4)*2`

  Wertzuweisung für Speicherzelle a auf Zeit; Wertzuweisungen sind temporäre Wertfestlegungen für benamte Speicherzellen, Festlegungen auf Zeit, bis zum nächsten Überschreiben (abänderbar, engl. mutable).
- Funktional: Wertvereinbarung, dauerhaft, unabänderbar

  `b = (17+4)*2`

  Wertvereinbarung für Ausdrucksnamen b für die gesamte Programmzukunft; Wertvereinbarungen sind permanente Wertfestlegungen für Ausdrucksnamen, Festlegungen für immer, auf alle Zeit (unabänderbar, engl. immutable).

Aufforderung an ein imperatives Programm:

- Führe deine Instruktionen beginnend mit der durch ν_{init} gegebenen Variablenbelegung aus und liefere die finale Variablenbelegung ν_{final}!

Aufforderung an ein funktionales Programm:

- Liefere den Wert von Ausdruck α unter Lösung der Gleichungen deines Gleichungssystems!

Der Unterschied ist offensichtlich u. konzeptuell fundamental:

- Imperativ: Denken in Instruktionen und ihren Effekten.
- Funktional: Denken in Gleichung(ssystem)en und Eigenschaften ihrer Lösungen.

Eine Aufgabe imperativ zu lösen erfordert deshalb eine andere Herangehens- und Denkweise als funktional und umgekehrt!

- Die imperative Wertzuweisung $a := a + 1$ bedeutet: Erhöhe den in Speicherzelle a befindlichen Wert um 1.
- Die funktionale Wertvereinbarung $b = b + 1$ bedeutet: Finde eine Lösung für die Gleichung $b = b + 1$.

  Der Bedeutungsunterschied ist offensichtlich und er ist fundamental.

Programme funktionaler Programmiersprachen (auch Haskell-Programme) sind i.a. Systeme (wechselweiser) rekursiver Rechenvorschriften, die sich hierarchisch oder/und wechselweise aufeinander abstützen.

Schlüsselwörter (21 Stück):
`case class data default deriving do else if import in infix infixl infixr instance let module newtype of then type where`

3 :: Grundlagen

3.1 Vordefinierte Datentypen

Unstrukturierte Datentypen:

1. Ganze Zahlen: `Int, Integer`
2. Gleitkommazahlen: `Float, Double`
3. Wahrheitswerte: `Bool`
4. Zeichen: `Char`

Strukturierte Datentypen:

1. Tupel (Kreuzprodukttyp): Zusammenfassung vorbestimmter Zahl von Werten (mögl. verschiedener Typen) [heterogen].

   vordefinierte Operatoren: (bei Paaren) `fst, snd`

2. Listen: Zusammenfassung nicht vorbestimmter Zahl von Werten gleichen Typs [homogen].

   vordefinierte Operatoren: `(:), (++), (!!). concat, reverse, length, head, tail, listenkomprehension, ...`

   Ad. Listenkomprehension, z.B.:

   ```
   [ square n | n <-- liste ]

   [ id n | n <-- liste, isPowOfTwo n, n>=5 ]

   [ ((m,n),m+n) | m <-- liste1, n <-- tail liste2, m<=2, n<=7 ]
   ```

   oder auch Quicksort:

   ```
   quickSort :: [Integer] --> [Integer]
   quickSort [ ] = [ ]
   quickSort (n:ns) =
   quickSort [ m | m <-- ns, m <= n ] ++ [n] ++ quickSort [ m | m <-- ns, m > n ]
   ```

   Ad. `(!!)`:

   ```
   ‘‘Haskell’’!!3 -->> ‘k‘, (!!) ‘‘Haskell’’ 3 ->> ‘k‘
   ```

3. Zeichenreihen: Zusammenfassung nicht vorbestimmter Zahl von Werten des Typs `char`.

   vordefinierte Operatoren: `(++), head, tail, length, ...` (btw: `string :: [char]`)

3.2 Funktionen

Schreibweisen für Funktionen:

1. Mittels bedingter Ausdrücke
2. Mittels bewachter Ausdrücke
3. Mittels Muster (das sind z.B.: Konstanten eines Typs, Variablen, wild card (_), ...)
4. Mittels case-Ausdruck
5. Mittels Muster und lokaler Deklaration (`where, let/in`)
6. Mittels anderer Funktionen (argumentbehaftet/argumentfrei)
7. Mittels anonymer λ-Abstraktion ($\backslash x$ `--> «bla bla»`)

Vorteile Muster:

+ legen die Struktur des (Argument-) Werts offen

+ legen Namen für die verschiedenen Strukturteile des Werts fest und erlauben über diese Namen unmittelbaren Zugriff auf diese Teile

+ vermeiden dadurch sonst nötige Selektorfunktionen und führen dadurch zu einem

+ Gewinn an Lesbarkeit und Transparenz

Funktionssignaturen, Funktionsterme, Funktionsstelligkeiten, Curry

- Funktionssignaturen: geben den Typ einer Funktion an und sind rechtsassoziativ, z.B.

  ```
  ersetze :: (Txt --> (Vork --> (Alt --> (Neu --> Txt))))
  ```

- Funktionsterme: aus Funktionsaufrufen aufgebaute Ausdrücke und sind linksassoziativ, z.B.

  ```
  ((((ersetze ‘‘Ein alter Text’’) 1) ‘‘alter’’) ‘‘neuer’’)
  ```

- Funktionsstelligkeiten: Funktion sind einstellig, d.h. alle Funktionen konsumieren ein (elementares, komplexes, funktionales) Argument.

Die Deklarationsweisen curryifziert und uncurryifiziert: Erfolgt die Konsumation

- einzeln Argument für Argument (ersetzt Produkt × durch `-->`): **curryfiziert**, z.B.

  ```
  binom :: Integer --> Integer --> Integer mit Funktionsterm (binom 45 6)
  ```

  Btw.: Nur curryfizierte Funktionen respektieren Prinzip partieller Auswertung und damit Prinzip: Funktionen liefern Funktionen als Ergebnis!

- alle auf einmal als Tupel (ersetzt Produkt `-->` durch ×): **uncurryfiziert**, z.B.

  ```
  binom’ :: (Integer, Integer) --> Integer mit Funktionsterm (binom (45,6))
  ```

Partialität

Bzgl. part. Funktionen, etwa `! :: Int --> Int` mit `otherwise = error ‘‘shit’’`. Vgl. Fehlerbehandlung.

3.3 Typsynonyme, Neue Typen, Typklassen

- **Typsynonyme**: erlauben die äußere Semantik von Datentypen durch Wahl eines guten und sprechenden Namens offenzulegen und mitzuteilen. Ermöglichen damit aussagekräftigere und sprechendere Funktionssignaturen und erhöhen dadurch die Lesbarkeit, Verständlichkeit und Transparenz von und in Programmen. Führen keine neuen Typen ein, sondern ausschließlich neue Namen für bereits existierende Typen, sog. Alias-Namen oder Synonyme. Kein Beitrag zur Typsicherheit (Hier ist das Marssondenbeispiel drin: Typfehler)!

- **Neue Typen**: Nutzdaten hinter Datenkonstruktor zu verbergen und schützen, erlaubt zusätzlich zum Anzeigen intendierter Typ- und Wertbenutzungen diese auch durchzusetzen. Sozusagen eine Mischung aus `type` und `data`. (Stichwort: Isomorphie [in der Mathematik])

  Beispiel:

  ```
  newtype Kurs = K Float

  type Niedrigst = Kurs
  ```

  ... (`newtype` ist neuer Typname, `K` ist Konstruktor). (Wie `Kurs` ist auch `Niedrigst` kein Synonyme für `Float`, sondern für den neuen Typ `Kurs`.)

- **Typklassen**: haben Typen als Elemente und legen Menge von Operationen und Relationen fest, die auf Werten ihrer Elemente implementiert werden müssen. (können für diese Operationen und Relationen bereits vollständige Standardimplementierungen oder noch zu vervollständigende Protoimplementierungen vorsehen. Typen werden durch Instanzbildung zu Elementen/Instanzen einer Typklasse.[1]

  Beispiel (vordefiniert):

  ```
  class Eq a where
  (==) :: a --> a --> Bool
  (/=) :: a --> a --> Bool
  x /= y = not (x==y)                          -- Protoimplementierungen
  x == y = not (x/=y)                          -- für (/=) und (==)
  ```

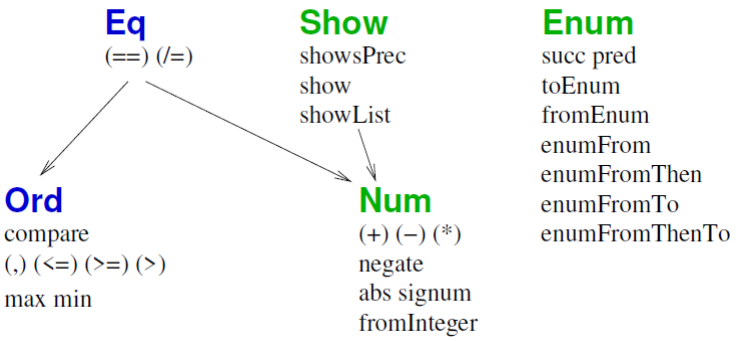
  Beispiel (Instanzbildung):

  ```
  instance Eq Bool where
  True  == True  = True                        -- Überschreiben der
  False == False = True                        -- Proto-Impl. von (==)
      _ == _     = False
  ```

  Und einige prominente Typklassen:

  1. 'Gleichheit' `Eq`: Klasse der Typen mit Gleichheits- `(==)` und Ungleichheitsrelation `(/=)`.
  2. 'Ordnung' `Ord`: Klasse der Typen mit Ordnungsrelationen (`<, <=, ...` etc.).
  3. 'Numerisch' `Num`: Klasse der Typen, deren Werte sich numerisch verhalten (Bsp.: `Int, Integer, Float, Double`)
  4. 'Aufzählung' `Enum`: Klasse der Typen, deren Werte aufgezählt werden können (Bsp.: `[2,4..29] :: Int`).
  5. 'Ausgabe' `Show`: Klasse der Typen, deren Werte als Zeichenreihen dargestellt werden können (Bsp.: `Bool, Int, Integer, [Int], (Bool,Int),...`)

[1] Typklassen sind Sammlungen von Typen, auf deren Werten die in der Typklasse angegebenen Funktionen definiert sind. Durch Instanzbildungen der Typklasse für verschiedene Typen wird die Bedeutung dieser Funktionen überladen und typspezifisch.



Beispiel (oben fortgesetzt):

```
newtype Kurs = K Float

instance Eq Kurs where
K k_1 == K k_2 = k_1 == k_2
```

und:

```
newtype Kurs = Float

instance Ord Kurs where
K k_1 <= K k_2 = k_1 <= k_2
```

Beispiel (automatische Instanzbildung) (oben fortgesetzt):

```
newtype Kurs = K Float deriving (Eq,Ord,Show)
```

(Gib) Faustregel für Typklassen

1. Überlege, welche Operationen/Relationen (semantische Begriffe: plus, gleich) auf Werte dieses Typs anwendbar sind und ob es bereits passende Operatoren/Relatoren (syntaktische Begriffe: `(+)`, `(==)`) in existierenden Typklassen dafür gibt.
2. Mache den neuen Typ zu Instanzen derjenigen Typklassen, in denen diese Operatoren/Relatoren eingeführt sind; oft reichen dafür deriving-Klauseln aus.
3. Sind auf die Werte des neuen Typs Operationen/Relationen anwendbar, für die es keine passenden Operatoren/Relatoren in existierenden Typklassen gibt, so führe eine neue Typklasse mit passenden Operatoren/Relatoren ein (wo möglich, zusammen mit vollständigen Implementierungen oder zu vervollständigenden Protoimplementierungen), wenn anzunehmen ist, dass diese konzeptuell auch für weitere erst noch zu definierende Datentypen relevant sein werden.

Überladene Funktionen

1. Direkt überladene Funktionen

```
(==) :: Eq a => a --> a --> Bool

(>=) :: Ord a => a --> a --> Bool

betrag_in_Muenzen_verschieden_zahlbar :: Waehrung a => a --> Int
```

2. Indirekt überladene Funktionen

```
f :: (Num a, Waehrung a) => a --> a --> a
f a b = ..  betrag_in_Muenzen_verschieden_zahlbar ..
```

3. Nicht überladene Funktionen

```
 fac :: Integer --> Integer

first :: (a,b) --> a

length :: [a] --> Int
```

3.4 Algebraische Datentypen

`data`-Deklarationen zur Definition originär neuer Typen, von Aufzählungs-, Produkt- und Summentypen.

1. Aufzählungstypen : Ausschließlich nullstellige Konstruktoren, z.B.

```
data Jahreszeiten = Fruehling | Sommer| Herbst | Winter
data Geschlecht = Maennlich | Weiblich
```

2. Produkttypen : Exakt ein zwei- oder höherstelliger Konstruktor, z.B.

`data Person = P Vorname Nachname Geschlecht` (echter Produkttyp)

`data Person = P (Vorname,Nachname,Geschlecht)` (unechter Produkttyp)

zu unterscheiden von:

`type Person = (Vorname, Nachname, Geschlecht)` (Tupeltyp; kein Konstruktor)

3. Summentypen : Mindestens zwei Konstruktoren, mindestens ein nicht nullstelliger Konstruktor, z.B.

```
data Baum = Blatt Int | Wurzel Baum Int Baum
```

(Bem.: Ein algebraischer Datentyp mit genau einem einstelligen Konstruktor lässt sich in gleicher Weise als unechter Summen- wie als unechter Produkttyp ansehen. Aufzählungs- und Produkttypen sind Spezialfälle von Summentypen)

Vordefinierte Datentypen:

```
data Ordering = LT | EQ | GT deriving (Eq,Ord, Bounded, Enum, Read, Show)
    data Bool = False | True deriving (Eq,Ord,Bounded,Enum,Read,Show)
      data Maybe a = Nothing | Just a deriving (Eq,Ord,Read,Show)
```

Selbstdefinierte Datentypen:

```
        data Baum = Leer | Wurzel Baum Int Baum deriving (Eq,Ord,Show)
data Baum' = Blatt String | Wurzel' Baum' Int Bool Baum' deriving (Eq,Ord,Show)
  data TBaum = Nichts | Gabel Person TBaum TBaum TBaum deriving (Eq,Ord,Show)
```

oder Suchbäume:

```
type Schluessel = Int
type Information = String
data Suchbaum = Sb Schluessel Information |
Sk Schluessel Information Suchbaum Suchbaum deriving (Eq,Ord,Show)
```

Allgemeines Muster:

`data Typname = Kon_1 t_11 ... t_1k_1 | ...| Kon_n t_n1 ... t_nk_n`, wobei

- Typname: Freigewählter frischer Identifikator als Typname.
- `Kon_i`: Freigewählte frische Identifikatoren als (Datenwert-)Konstruktornamen.
- `k_i`: Stelligkeit des Konstruktors `Kon_i`.
- `t_ij`: Namen bereits existierender Typen.
- Typ- und Konstruktornamen müssen stets mit einem Großbuchstaben beginnen.

Feldsyntax

Für transparente, sprechende Typdeklarationen drei Möglichkeiten:

1. Kommentierung
2. Typsynonyme
3. Feldsyntax (Verbundtypsyntax)

Datentypdeklaration

Datentypdeklarationen in Haskell mittels dreier Sprachkonstrukte:

1. `type` : Typen zusätzliche, neue Namen (Synonyme, Aliase)
2. `newtype` : Typen unverwechselbare, neue Identitäten zu verleihen.
3. `data` : Erlaubt originär neue Typen und ihre Werte einzuführen.

3.5 Muster

Muster für ... Werte elementarer Datentypen sind:

1. Konstanten: `0, 42, 3.14, ‘c‘, True,...`
2. Variablen: `b, m, n, t, e,...`
3. Joker (wild card): _

... für Tupeltypen sind:

1. Konstanten: `(0,0), (0,‘‘Null’’), (3.14,‘‘pi’’,True),...`
2. Variablen: `t, t1,...`
3. Joker (wild card): _
4. Kombination aus denen oben

... für Listentypen sind:

1. Konstanten: `[ ], ‘‘ ’’, [1..50], [‘a‘..‘z‘], [True,False,True,True], ‘‘aeiou’’,...`
2. Variablen: `p, q, ps, qs, ...`
3. Joker (wild card): _
4. Konstruktormuster: (⟨ `muster-listenkopf` ⟩ : ⟨ `muster-listenrest` ⟩), `(p:ps), (p:q:qs),...`

weils so nett is nochmal das Baumbeispiel:

```
type Zett = Int
data Baum a b = Blatt a | Wurzel b (Baum a b) (Baum a b)
data Liste a = Leer | Kopf a (Liste a)

tiefe :: (Baum a b) --> Zett
tiefe (Blatt _) = 1
tiefe (Wurzel _ l r) = 1 + max (tiefe l) (tiefe r)

laenge :: (Liste a) --> Zett
laenge Leer = 0
laenge (Kopf _ xs) = 1 + laenge xs
```

4 :: Applikative Programmierung

Applikative Programmierung: Das tragende Prinzip applikativen Programmierens ist die Bildung von Funktionen durch Abstraktion v. Ausdrücken nach unabh. Variablen und die Applikation v. Funktionen auf elementare Werte mit elementarem Resultat; kurz: Das Rechnen mit elementaren Werten.

Funktionale Programmierung: Das tragende Prinzip funktionalen Programmierens ist die Bildung von Funktionen aus Funktionen mithilfe v. Funktionen höherer Ordnung und die Applikation von Funktionen auf Funktionen; kurz: Das Rechnen mit Funktionen.

Werte von Ausdrücken können

- elementar sein:

```
fac (17+4)*2-39 ->> 6 :: Int
fib (fac (17+4)*2-39) ->> fib 6 ->> 8  ::| Int
map (\n-> n+1) [1,2,3] ->> [2,3,4] :: [Int]
foldr (*) 1 [3,4,5] ->> 60 :: Int
```

- funktional sein:

```
map (\n-> n+1) :: [Int] -> [Int]     -- 1-stellig
foldr (*) :: Int -> [Int] -> Int     -- 2-stellig
foldr (*) 1 :: [Int] -> Int          -- 1-stellig
```

4.1 Rekursion

Rechenvorschrift heißt rekursiv, wenn sie in ihrem Rumpf (direkt oder indirekt) aufgerufen wird:

1. mikroskopische Ebene: (oder auch direkt bzw. unmittelbare Rekursion) betrachtet einzelne Rechenvorschriften und die syntaktische Gestalt der rekursiven Aufrufe

   (a) Repetitive (schlichte, endständige) Rekursion: pro Zweig höchstens ein rekursiver Aufruf und diesen stets als äußerste Operation (sehr kostengünstig). (z.B.: `ggt`, Folie 632)

   (b) Lineare Rekursion: pro Zweig höchstens ein rekursiver Aufruf, davon mindestens einer nicht als äußerste Operation. (z.B.: `powerThree`, Folie 633)

   (c) Baumartige (kaskadenartige) Rekursion pro Zweig können mehrere rekursive Aufrufe nebeneinander vorkommen. (z.B.: `binom`, Folie 634)

   (d) Geschachtelte Rekursion: rekursive Aufrufe enthalten rekursive Aufrufe als Argumente (nicht kostengünstig). (z.B.: `fun91`, Folie 635)

2. makroskopischer Ebene: betrachtet Systeme von Rechenvorschriften und ihre wechselseitigen Aufrufe.

   (e) Indirekte (verschränkte, wechselweise) Rekursion zwei oder mehr Funktionen rufen sich wechselweise auf. (z.B.: `isOdd`, Folie 639)

Aufrufgraphen

Der Aufrufgraph von S (System von Rechenvorschriften) ist ein Graph, der

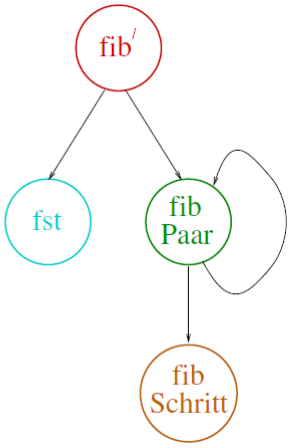
- für jede in S deklarierte Rechenvorschrift einen Knoten mit dem Namen der Rechenvorschrift als Beschriftung enthält,
- eine gerichtete Kante vom Knoten f zum Knoten g genau dann enthält, wenn im Rumpf der zu f gehörigen Rechenvorschrift die zu g gehörige Rechenvorschrift aufgerufen wird.

```haskell
fibSchritt :: (Integer,Integer) -> (Integer,Integer)
fibSchritt (m,n) = (n,m+n)

fibPaar :: Integer -> (Integer,Integer)
fibPaar n =
 | n == 0    = (0,1)
 | otherwise = fibSchritt (fibPaar (n-1))

fib' :: Integer -> Integer
fib' n = fst (fibPaar n)

fst :: (a,b) -> a
fst (x,y) = x
```



wobei

- Direkte Rekursivität einer Funktion: 'Selbstkringel'.
- Wechselweise Rekursivität zweier Funktionen: Kreise mit zwei Kanten.
- Direkte hierarchische Abstützung einer Funktion auf eine andere: Es gibt eine Kante von Knoten f zu Knoten g, aber nicht umgekehrt.
- Indirekte hierarchische Abstützung einer Funktion auf eine andere: Knoten g ist von Knoten f über eine Folge von Kanten erreichbar, aber nicht umgekehrt.
- Indirekte wechselweise Abstützung: Knoten g ist von Knoten f über eine Folge von Kanten erreichbar und umgekehrt
- Unabhängigkeit/Isolation einer Funktion: Knoten f hat (ggf. mit Ausnahme eines Selbstkringels) weder ein- noch ausgehende Kanten.

Komplexitätsklassen

Die classics halt: $\mathcal{O}, \Omega, o, \omega, \Theta$. Dazu nur ein btw.: Ein langsamer Rechner mit asymptotisch gutem Algorithmus gewinnt gegen einen schnellen Rechner mit asymptotisch schlechtem oder auch nur schlechterem Algorithmus bei Probleminstanzen nichttrivialer Größe immer!

4.2 Auswertung einfacher Ausdrücke

Ziel, Ausdrücke soweit zu vereinfachen wie nur irgend möglich und so ihren Wert zu berechnen. (Hier steckt das epic Church-Rosser-Theorem drin) Dafür ist das Zusammenspiel des

- Expandierens (E) (Funktionsterme, Funktionsaufrufe)
  simple x y z := $(x + z) * (y + z)$ also: simple 2 3 4; (E) –» $(2 + 4) * (3 + 4)$
- Simplifizierens (S) (Funktionstermfreie Ausdrücke)
  $3 * (9 + 5)$; (S) –» $3 * 14$; (S) –» 42

von Ausdrücken zu organisieren. Die Hauptauswertungsvorgehensweisen:

1. Applikativ : Argumentauswertung vor Expansion

2. Normal : Argumentauswertung nach Expansion

```
fac n = if n == 0 then 1 else (n * fac (n - 1))
        fac 2
 (E) ->> if 2 == 0 then 1 else (2 * fac (2 - 1))
 (S) ->> if False then 1 else (2 * fac (2 - 1))
 (S) ->> (2 * fac (2 - 1))
 (S) ->> 2 * fac 1
 (E) ->> 2 * (if 1 == 0 then 1 else (1 * fac (1 - 1)))
 (S) ->> 2 * (if False then 1 else (1 * fac (1 - 1)))
 (S) ->> 2 * ((1 * fac (1 - 1)))
 (S) ->> 2 * (1 * fac 0)
 (E) ->> 2 * (1 * (if 0 == 0 then 1
                  else (0 * fac (0 - 1))))
 (S) ->> 2 * (1 * (if True then 1
                  else (0 * fac (0 - 1))))
 (S) ->> 2 * (1 * 1)
 (S) ->> 2 * 1
 (S) ->> 2              ⇝ sog. applikativer Auswertungstil.
```

(a) applikative Auswertung

```
fac n = if n == 0 then 1 else (n * fac (n - 1))
        fac 2
 (E) ->> if 2 == 0 then 1 else (2 * fac (2 - 1))
 (S) ->> if False then 1 else (2 * fac (2 - 1))
 (S) ->> (2 * fac (2 - 1))
 (E) ->> 2 * (if (2-1) == 0 then 1
                else ((2-1) * fac ((2-1)-1)))
(2S) ->> 2 * (if False then 1
                else ((2-1) * fac ((2-1)-1)))
 (S) ->> 2 * (((2-1) * fac ((2-1)-1)))
 (S) ->> 2 * (1 * fac ((2-1)-1))
 (E) ->> 2 * (1 * (if ((2-1)-1) == 0 then 1
                 else ((2-1)-1) * fac (((2-1)-1)-1)))
 (E) ->> 2 * (1 * (if True then 1
                 else ((2-1)-1) * fac (((2-1)-1)-1)))
(3S) ->> 2 * (1 * 1)
(2S) ->> 2              ⇝ sog. normaler Auswertungsstil.
```

(b) normale Auswertung

4.3 Programmentwicklung

Systematische Programmentwicklung für rekursive Programme als 5-schrittiger Prozess (am Beispiel: streiche ersten n Elemente einer Liste):

1. Lege die (Daten-) Typen fest.

   ```
   drop :: Int --> [a] --> [a]
   ```

2. Führe alle relevanten Fälle auf.

   ```
   drop 0 [ ] =
   drop 0 (x:xs) =
   drop (n+1) [ ] =
   drop (n+1) (x:xs) =
   ```

3. Lege die Lösung für die einfachen (Basis-) Fälle fest.

   ```
   drop 0 [ ] = [ ]
   drop 0 (x:xs) = x:xs
   drop (n+1) [ ] = [ ]
   ```

4. Lege die Lösung für die übrigen Fälle fest.

   ```
   drop (n+1) (x:xs) = drop n xs
   ```

5. Verallgemeinere und vereinfache das Lösungsverfahren.

   ```
   5a) drop :: Integral b => b --> [a] --> [a]

   5b) drop 0 xs = xs
   drop (n+1) [ ] = [ ]
   drop (n+1) (x:xs) = drop n xs

   5c) drop 0 xs = xs
   drop _ [ ] = [ ]
   drop (n+1) (_:xs) = drop n xs
   ```

5 :: Funktionale Programmierung

5.1 Funktionen höherer Ordnung

Das sind Funktionen, deren Argumente oder Resultate Funktionen sind (vgl. Operatoren in der Mathematik z.B. $\partial/\partial x$).

- Funktionen mit funktionalen Resultaten:

```
(+) :: Num a => a --> (a --> a)

((+) 1) :: Num a => (a --> a)
```

- Funktionen mit funktionalen Argumenten und Resultaten:

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

Funktionale Abstraktion höher Stufe (Idee): Verknüpfungsvorschriften werden zu funktionalen Parametern einer Funktion, die Funktion dadurch zu einer Funktion höherer Ordnung.

(Beispiel eines) Rekursionsschema(s):

```
rekSchema :: Int -> (Int -> Int -> Int) -> Int -> Int
rekSchema basiswert verknuepfe n
 | n==0 = basiswert
 | n>0  = verknuepfe n (rekSchema basiswert verknuepfe (n-1))
```

Und why that? Gewinn durch funktionale Abstraktion höherer Stufe ist Wiederverwendung und dadurch kürzerer, verlässlicherer, wartungsfreundlicherer Code.

5.2 Polymorphie

...ist in funktionalen Sprachen

- erstrangiges Sprachelement (engl. first-class citizen):

```
map    :: (a -> b) -> [a] -> [b]
foldr  :: (a -> b -> b) -> b -> [a] -> b
flip   :: (a -> b -> c) -> (b -> a -> c)
length :: [a] -> Int
```

Der Typ jeder Funktion ist grundsätzlich von den konkreten Typen der Argumentwerte so weit entkoppelt wie irgend möglich.

...ist in imperativen Sprachen

- zweitrangiges Sprachelement (engl. second-class citizen):

Möglichkeiten zur polymorphen Typspezifikation fehlen oft völlig oder sind im Vergleich zu funktionalen Sprachen wesentlich limitiert.

Wir unterscheiden Polymorphie auf:

1. Datentypen
   (a) Algebraische Datentypen, `data`
   (b) Neue Typen, `newtype`
   (c) Typsynonyme, `type`

Sprachmittel: Typvariablen, Typklassen.

2. Funktionen

   (d) Parametrische Polymorphie (oder echte Polymorphie)
   Sprachmittel: Typvariablen.

   (e) Ad hoc Polymorphie (oder unechte Polymorphie, Überladung)
   Haskell-spezifisches Sprachmittel: Typklassen

Polymorphe Datentypen

Ein algebraischer (Daten-) Typ, neuer Typ oder Typsynonym T heißt **polymorph**, wenn einer oder mehrere Grundtypen der Werte von T in Form einer oder mehrerer Typvariablen als Typparameter angegeben werden, wie z.B.

- `data Baum a b c = Blatt a b | Wurzel (Baum a b c) c (Baum a b c)`

  `data (Eq a, Ord b, Ord c, Num c) => Baum' a b c = Blatt' a b | Wurzel' (Baum' a b c) c (Baum' a b c)`

  (Beispiele polymorpher algebraischer Datentypen)

- `newtype Tripelpaar a b c d = Tp ((a,b,c),(b,c,d))`

  `newtype (Ord a, Ord b) => Relation a b = R [(a,b)]`

  (Beispiele polymorpher neuer Typen)

- `type Assoziationssequenz a b = [(a,b)]`

  `type MeinTyp a = Unverwechselbar_mit_Typ_a a`

  (Beispiele Typesynonyme) [btw.: keine Kontexteinschränkungen erlaubt]

Parametrische Polymorphie auf Funktionen

Eine Funktion heißt **parametrisch polymorph** (oder echt polymorph), wenn die Typen eines oder mehrerer ihrer Parameter angegeben durch Typvariablen Werte beliebiger Typen als Argument zulassen. Diese

1. ermöglicht die Wiederverwendung von Funktionsnamen und Funktionsrümpfen, und
2. ist erkennbar daran: keine Typvariable der Funktionssignatur ist typkontexteingeschränkt.

Beispiele (vordefiniert):

- Auf beliebigen Typen:

```
curry :: ((a,b) --> c) --> (a --> b --> c)
curry f x y = f (x,y) bzw.

uncurry :: (a --> b --> c) --> ((a,b) --> c)
uncurry g (x,y) = g x y

flip :: (a --> b --> c) --> (b --> a --> c)
flip f x y = f y x
```

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

- Auf beliebigen Listentypen:

```
map :: (a --> b) --> [a] --> [b]
f xs = [ f x | x <-- xs ]

filter :: (a --> Bool) --> [a] --> [a]
filter p xs = [ x | x <-- xs, p x ]

foldr :: (a --> b --> b) --> b --> [a] --> b
foldr f e [ ] = e
foldr f e (x:xs) = f l (foldr f e xs)

foldl :: (a --> b --> a) --> a --> [b] --> a
foldl f e [ ] = e
foldl f e (x:xs) = foldl f (f e x) xs

length :: [a] --> Int
length [ ] = 0
length (_:xs) = 1 + length xs

head :: [a] --> a
head (x:_) = x

tail :: [a] --> [a]
tail (_:xs) = xs
```

Ad hoc Polymorphie auf Funktionen

Ist eine schwächere, weniger generelle Form von Polymorphie mit folgenden Synonymen: (1) Unechte Polymorphie bzw. (2) Überladen, Überladung.

Eine Funktion in Haskell heißt **ad hoc polymorph** (oder unecht polymorph oder überladen), wenn die Typen eines oder mehrerer ihrer Parameter angegeben durch Typvariablen durch Typkontexte eingeschränkt (Erkennungsmerkmal) Werte aller durch den Typkontext zugelassenen Typen als Argument zulassen. Tritt in funktionalen wie nichtfunktionalen Sprachen ebenso beiläufig und ubiquitär auf: `(+)`, `(*)`, `(-)`, `(==)`, `(/=)`, `(>)`, `(>=)`, `show`, `...`

1. (Direkt) überladene Funktionen
   (a) vordefinierter Typklassen: Alle in einer vordefinierten Typklasse angegebenen Funktionen (z.B. `(==)`, `(/=)` aus `Eq`, `(<)`, `(>)` aus `Ord`, `(+)`, `(*)` aus `Num`, etc.)
   (b) selbstdefinierter Typklassen: Alle in einer selbstdefinierten Typklasse angegebenen Funktionen
2. Indirekt überladene Funktionen: Alle Funktionen, die sich auf eine überladene Funktion abstützen, ohne selbst in einer Typklasse eingeführt zu sein, z.B.:

```
sum :: Num a => [a] --> a
sum [ ] = 0
sum (x:xs) = x + sum xs
```

Allgemeines Muster einer Typklassendefinition (vereinfacht):

```
class Name' tv => Name tv where
 f_1 :: ...          -- Signaturen der Typklassen-
 f_2 :: ...          -- funktionen f_1,...,f_k
 ...                 -- über tv und anderen Typen.
 f_k :: ...
 f_i = ...               -- Protoimplementierungen
 ...                     -- für null oder mehr der
 f_j = ...               -- Funktionen f_1,...,f_k.
```

- `Name'`: Name einer existierenden Typklasse als Kontext.
- `Name`: Freigewählter Name als Identifikator der Klasse.
- `tv`: Typvariable.

Ad. Typklasseninstanzbildung:

- `data (Eq a, Eq b, Eq c) => Baum a b c = Blatt a b | Wurzel (Baum a b c) c (Baum a b c) deriving Eq` (automatisch)

  gleichbedeutend zu:

- `data (Eq a, Eq b, Eq c) => Baum a b c = Blatt a b | Wurzel (Baum a b c) c (Baum a b c)`

  `instance (Eq a, Eq b, Eq c) => Eq (Baum a b c) where`

  `(Blatt u v) == (Blatt x y) = (u == x) && (v == y)`

  `(Wurzel ltb z rtb) == (Wurzel ltb' z' rtb') = (ltb == ltb') && (z == z') && (rtb == rtb')`

  `_ == _ = False` (manuell)

```
class Info a where
 wert_beispiele  :: [a]        -- Signaturen überla-
 zu_zeichenreihe :: a -> String  -- dener Funktionen
                                 -- in Typklasse Info
 wert_beispiele    = []          -- Protoimplemen-
 zu_zeichenreihe _ = ""              -- tierungen
          -- entspricht: zu_zeichenreihe = \_ -> ""

class Info a => Groesse a where
 groesse :: a -> Int               -- Signatur über-
                                -- ladener Funktion
                             -- in Typklasse Groesse
 groesse = (length . zu_zeichenreihe)  -- Protoimple-
                                       -- mentierung
```

```
instance Info Char where
 wert_beispiele  = ['a','A','z','Z','0','9']
 zu_zeichenreihe = \c -> [c] (entspr.: zu_z. c = [c])
instance Groesse Char where
-- Die Protoimplementierung passte; nichts wäre zu
-- tun Aus Effizienzgründen geben wir dennoch an:
groesse = \_ -> 1        (entspr.: groesse _ = 1)
instance Info Bool where
 wert_beispiele        = [True,False]
 zu_zeichenreihe True  = "Wahr"
 zu_zeichenreihe False = "Falsch"
instance Groesse Bool where
 groesse True  = 4    -- length (zu_zeichenreihe True)
 groesse False = 6    -- length (zu_zeichenreihe False)
```

Überladene Funktionen in selbstdefinierter Typklassen

Automatsche Typklasseninstanzbildung möglich für `Eq`, `Ord`, `Enum`, `Bounded`, `Show`, `Read`.

Grenzen des Überladen

Ist es möglich jeden Typ zu einer Instanz der Typklasse `Eq` zu machen?
Nein: Gleichheit von Funktionen ist nicht entscheidbar, d.h. es gibt keinen Algorithmus, der für zwei beliebig vorgelegte Funktionen stets nach endlich vielen Schritten entscheidet, ob diese Funktionen gleich sind oder nicht. Genauer: Funktionen bestimmter (auch scheinbar universeller Natur) Funktionalität lassen sich i.a. nicht für jeden Typ angeben, sondern nur für eine Teilmenge aller Typen.

5.3 Ad hoc Polymorphie vs. parametrische Polymorphie

- Ad hoc Polymorphie unterstützt Wiederverwendung des Funktionsnamens, nicht jedoch der Funktionsimplementierung. Es gilt Prinzip: ein Name, eine T-spezifische Implementierung pro Instanz T von a

  Parametrische Polymorphie unterstützt Wiederverwendung von Funktionsname und Funktionsimplementierung. Es gilt Prinzip: ein Name, eine Implementierung.

- Ein polymorpher Typ wie `(a --> a)` steht informell für:

  $\forall$ `a` $\in$ "Menge gültiger Typen" . `(a --> a)`.

  Ein ad hoc polymorpher Typ wie `(Num a => a --> a --> a)` steht informell für:

  $\forall$ `a` $\in$ `Num . (a --> a --> a)`

  (Parametrische Polymorphie ist unter dem Aspekt Wiederverwendung echt stärker als ad hoc Polymorphie.)

6 :: Fundierung funktionaler Programmierung

Der λ-Kalkül (Church 1936) ist ein universelles Berechnungsmodell (Grundlage aller funktionalen Programmiersprachen, Folie 1011), wie auch Allgemein rekursive Funktionen (Herbrand 1931, Gödel 1934, Kleene 1936), μ-rekursive Funktionen (Kleene 1936), Turing-Maschinen (Turing 1936), Endliche kombinatorische Prozesse (Post 1936), Markov-Algorithmen (Markov 1951), Registermaschinen (Random Access Machines (RAMs)) (Shepherdson, Sturgis 1963), ...
Wichtig: Die Klasse der Turing-berechenbaren Funktionen stimmt mit der Klasse der intuitiv berechenbaren Funktionen überein. (Church-Turing-These)
Mit Theorem 12.1.1 (Gleichmächtigkeit) (Folie 995) steckt hier dann auch der λ-Kalkül drin.

6.1 Exkurs. λ-Kalkül : Syntax

Die Menge E der wohlgeformten Ausdrücke des reinen λ-Kalküls über einer Menge N von Namen ist induktiv wie folgt definiert:

1. Namen: Jeder Name aus N ist in E.

   Bsp.: $a, b, c, \ldots, x, y, z, \ldots$

2. Abstraktionen: Ist x aus N, e aus E, so ist $(\lambda x.e)$ in E.

   Sprechweise: (Funktions-) Abstraktion mit Parameter x und Rumpf e.

3. Applikationen: Sind f und e aus E, so ist auch $(f\ e)$ in E.

   Sprechweise: Applikation oder Anwendung von f auf e; f heißt auch Rator, e auch Rand.

Analog zu Haskell:
$\lambda x.\lambda y.\lambda z.(x\ (y\ z))$ steht kurz für $(\lambda x.(\lambda y.(\lambda z.(x\ (y\ z)))))$ (Rechtsassoz,)
$e_1 e_2 e_3 \ldots e_n$ steht kurz für $(\ldots((e_1\ e_2)e_3)\ldots e_n)$ (Linksassoz.)

6.2 Exkurs. λ-Kalkül : Semantik

Syntaktische Substitution

Notation: $e'\,[e/x]$ ist der Ausdruck, der aus e' entsteht, in dem jedes freie Vorkommen von x in e' durch e substituiert, ersetzt wird.

Die syntaktische Substitution ist die 3-stellige Abbildung
$\cdot[\cdot/\cdot] : E \to E \to V \to E$ definiert bei Anwendung auf

...Namensterme durch:

$$y\,[e/x] =_{df} \begin{cases} y & \text{falls y aus N mit } y \neq x \\ e & \text{falls y aus N mit } y = x \end{cases}$$

...applikative Terme durch:

$$(f\ g)\,[e/x] \;=_{df}\; (f\,[e/x])\ (g\,[e/x])$$

...Abstraktionsterme durch:

$$(\lambda y.f)\,[e/x] =_{df} \begin{cases} \lambda y.f & \text{falls } y = x \\ \lambda y.(f\,[e/x]) & \text{falls } y \neq x \wedge y \notin \mathit{frei}(e) \\ \lambda z.((f\,[z/y])\,[e/x]) & \text{falls } y \neq x \wedge y \in \mathit{frei}(e), \\ & \quad \text{wobei z frisch aus N: } z \notin \mathit{frei}(e) \cup \mathit{frei}(f) \\ & \quad\quad \text{(Vermeidung von Bindungsfehlern!)} \end{cases}$$

Konversionsregeln

Diese führen abgestützt auf syntaktische Substitution zu einer operationellen Semantik für λ-Ausdrücke in Form maximaler Ausdrucksvereinfachung:

1. α-Konversion (Umbenennung von Parametern)
   $\lambda x.e \longleftrightarrow \lambda y.e\,[y/x]$, wobei $y \notin \mathit{frei}(e)$
2. β-Konversion (Funktionsanwendung)
   $(\lambda x.f)\ e \longleftrightarrow f\,[e/x]$
3. η-Konversion (Elimination redundanter Funktion)
   $\lambda x.(e\ x) \longleftrightarrow e$, wobei $x \notin \mathit{frei}(e)$

Von links nach rechts gerichtete Anwendungen der β – und η–Konversion heißen β–Reduktion und η–Reduktion. Von rechts nach links gerichtete Anwendungen der β–Konversion heißen β–Abstraktion. Btw.: Es gibt auch noch die δ–Reduktion, die das Rechnen mit vordefinierten Funktionen ermöglicht, etwa: $(+)\ 1\ 2 \to_\delta 3$

Reduktionsfolgen

Eine **Reduktionsfolge** für einen λ-Ausdruck ist eine endliche oder nicht endliche Folge von β-, η-Reduktionen und α-Konversionen. Diese heißt **maximal**, wenn höchstens noch α-Konversionen anwendbar sind.

Theorem (diamond property nach Church-Rosser). Wenn e_1, e_2 (λ-Ausdrücke) ineinander konvertierbar sind, d.h. $e_1 \leftrightarrow e_2$, dann gibt es einen gemeinsamen λ-Ausdruck e, zu dem e_1, e_2 reduziert werden können, d.h. $e_1 \to^* e$ und $e_2 \to^* e$. (garantiert, dass die Normalform eines λ-Ausdrucks (bis auf α-Konversionen) eindeutig bestimmt ist, so sie existiert.

Semantik von λ-Ausdrücken

Sei e ein λ-Ausdruck. Die Semantik von e ist seine (bis auf α-Konversionen) eindeutig bestimmte Normalform, wenn sie existiert; die Normalform ist die Bedeutung und der Wert von e. Seine Semantik ist undefiniert, wenn die Normalform nicht existiert.

Takeaway

Haskell beruht auf angewandten typisierten λ-Kalkülen. Diese ordnen jedem wohlgeformten Ausdruck einen Typ zu, z.B.:
$3 : \text{Int}$
$(*) : \text{Int} \to \text{Int} \to \text{Int}$
$(\lambda x.2 * x) : \text{Int} \to \text{Int}$

6.3 Auswertungsanordnung

Auswertungsordnungen legen fest

1. Wo wird in einem komplexen Ausdruck 'gerechnet'?

   Antwort: So weit links wie möglich.

   Beispiel:

   `2^3+fac(fib(squ(2+2)))+3*5+fib(fac(7*(5+3)))+...`

   `-» 8+fac(fib(squ(2+2)))+3*5+fib(fac(7*(5+3)))+...`

2. Wann werden Funktionstermargumente ausgewertet?

   Antwort[2]:

   (a) Applikativ: Argumentauswertung vor Expansion [Auswertungsordnung: Jedes Argument wird genau einmal ausgewertet.]

   Beispiel:

   `squ (2+2)` und `(S) -» squ 4` und `(E) -» 4 * 4` und `(S) -» 16`

   (b) Normal: Argumentauswertung nach Expansion [Auswertungsordnung: Jedes Argument wird so oft ausgewertet, wie es benutzt wird.]

   `squ (2+2)` und `(E) -» (2+2) * (2+2)` und `(S) -» 4 * (2+2)` und `(S) -» 4 * 4` und `(S) -» 16`

   Außerdem die relevanten Operationalisierungen:

   (i) Linksapplikativ: Frühe, fleißige Auswertung (eager evaluation). Linksinnerst: Linkestinnerste Stelle

   (ii) Linksnormal mit Ausdrucksteilung (expression sharing) als Implementierungskniff: Späte, faule Auswertung (lazy evaluation) [Auswertungsordnung: Jedes Argument wird höchstens einmal ausgewertet. (sehr effizient)]. Linksäußerst: Linkestäußerste Stelle.

[2] Haskell arbeitet 'normal' und lazy (späte Auswertung). Haskell kann eine 'früh'-Auswertung vermittels `($!)` manuell erzwingen.

Applikative Funktionstermauswertung

. . . heißt Argumentauswertung vor Expansion. Ein Funktionsterm $(f\ ausd_1 \ldots ausd_n)$ wird ausgewertet, indem:

1. die Argumentausdrücke $ausd_1, \ldots, ausd_n$ werden vollständig zu ihren Werten $w_1, \ldots, w_n$ ausgewertet.
2. $w_1, \ldots, w_n$ werden im Rumpf von f für die Parameter von f eingesetzt.
3. der entstandene expandierte Ausdruck wird ausgewertet.

Normale Funktionstermauswertung

. . . heißt Argumentauswertung nach Expansion: Ein Funktionsterm $(f\ ausd_1 \ldots ausd_n)$ wird ausgewertet, indem:

1. die Argumentausdrücke $ausd_1, \ldots, ausd_n$ werden unausgewertet im Rumpf von f für die Parameter von f eingesetzt.
2. der entstandene expandierte Ausdruck wird ausgewertet.

Beispiele

```
binom3 x y :: Int -> Int -> Int
binom3 x y = (x + y) * (x - y)
binom3 16 ((5-3)*7) ->> ... ->> 60
```

Applikative Funktionstermauswertung:

```
         binom3 16 ((5-3)*7) (erst Arg. vereinfachen...)
(S) ->> binom3 16 (2*7)
(S) ->> binom3 16 14          (...dann expandieren!)
(E) ->> (16 + 14) * (16 - 14)
(S) ->> ...
(S) ->> 60
```

Normale Funktionstermauswertung:

```
         binom3 16 ((5-3)*7)     (sofort expandieren!)
(E) ->> (16 + ((5-3)*7)) * (16 - ((5-3)*7))
(S) ->> ...
(S) ->> 60
```

```
fac n = if n == 0 then 1 else (n * fac (n - 1))
```

...für den Aufruf `fac (1+1)` (statt `fac 2`):

```
          fac (1+1)          (Argument wird sofort ausgewertet)
  (S) ->> fac 2          (Expansion nach max. Argumentvereinf.)
  (E) ->> if 2 == 0 then 1 else (2 * fac (2-1))
  (S) ->> if False then 1 else (2 * fac (2-1))
  (S) ->> 2 * fac (2-1)        (Arg. wird sofort ausgewertet)
  (S) ->> 2 * fac 1         (Exp. nach max. Argumentvereinf.)
  (E) ->> 2 * (if 1 == 0 then 1
                 else (1 * fac (1-1)))
  (S) ->> 2 * (if False then 1
                 else (1 * fac (1-1)))
  (S) ->> 2 * (1 * fac (1-1)) (Arg. wird sofort ausgewertet)
  (S) ->> 2 * (1 * fac 0)      (Exp. nach max. Arg.vereinf.)
  (E) ->> 2 * (1 * (if 0 == 0 then 1
                      else (0 * fac (0-1))))
  (S) ->> 2 * (1 * (if False then 1
                      else (0 * fac (0-1))))
  (S) ->> 2 * (1 * 1)
  (S) ->> 2 * 1
  (S) ->> 2
```

vollst. appl. Auswertung

```
fac n = if n == 0 then 1 else (n * fac (n - 1))
```
...für den Aufruf `fac (1+1)` (statt `fac 2`):

```
          fac (1+1)     (Sofortige Exp., keine vorh. Arg.vereinf.)
  (E) ->> if (1+1) == 0 then 1
             else ((1+1) * fac ((1+1)-1))
  (S) ->> if 2 == 0 then 1
             else ((1+1) * fac ((1+1)-1))
  (S) ->> if False then 1
             else ((1+1) * fac ((1+1)-1))
  (S) ->> ((1+1) * fac ((1+1)-1))
  (S) ->> (2 * fac ((1+1)-1))            (Sofortige Exp.)
  (E) ->> 2 * (if ((1+1)-1) == 0 then 1
             else (((1+1)-1) * fac (((1+1)-1)-1)))
 (3S) ->> 2 * (if False then 1
             else (((1+1)-1) * fac (((1+1)-1)-1)))
  (S) ->> 2 * (((1+1)-1) * fac (((1+1)-1)-1))
  (S) ->> 2 * ((2-1) * fac (((1+1)-1)-1))
  (S) ->> 2 * (1 * fac (((1+1)-1)-1))   – Sofortige Exp.
  (E) ->> 2 * (1 * (if (((1+1)-1)-1) == 0 then 1
     else (((1+1)-1)-1) * fac ((((1+1)-1)-1)-1)))
 (4S) ->> 2 * (1 * (if True then 1
     else (((1+1)-1)-1) * fac ((((1+1)-1)-1)-1)))
  (S) ->> 2 * (1 * 1)
  (S) ->> 2 * 1
  (S) ->> 2
```

vollst. normale Auswertung

Linksapplikative oder linksnormale Auswertung?

Wo ist im Ausdruck auszuwerten?

```
2^3+fac(fib(squ(2+2)))+3*5+fib(fac(7*(5+3)))+fib((5+7)*2)
->> 8+fac(fib(squ(2+2)))+3*5+fib(fac(7*(5+3)))+fib((5+7)*2)
->> 8+fac(fib(squ 4))+3*5+fib(fac(7*(5+3)))+fib((5+7)*2)
->> 8+fac(fib((4*4)))+3*5+fib(fac(7*(5+3)))+fib((5+7)*2)
->> 8+fac(fib 16)+3*5+fib(fac(7*(5+3)))+fib((5+7)*2)
->> 8+fac(fib (16-2)+fib(16-1)) + 3*5 +...
->> 8+fac(fib 14+fib(16-1)) + 3*5 +...
->> ...
```

(a) linksapplikative Auswertung

```
2^3+fac(fib(squ(2+2)))+3*5+fib(fac(7*(5+3)))+fib((5+7)*2)
->> 8+fac(fib(squ(2+2)))+3*5+fib(fac(7*(5+3)))+fib((5+7)*2)
->> 8+(if fib(squ(2+2))==0 then 1 else n*fac(fib(squ(2+2))-1))
                    +3*5+fib(fac(7*(5+3)))+fib((5+7)*2)
->> 8+(if (if squ(2+2)==0 then 0
          else if squ(2+2)==1 then 1
              else fib(squ(2+2)-2)+fib(squ(2+2)-1))==0
     then 1 else n*fac(fib(squ(2+2))-1)) + 3*5 +...
->> 8+(if (if (2+2)*(2+2)==0 then 0
          else if squ(2+2)==1 then 1
              else fib(squ(2+2)-2)+fib(squ(2+2)-1))==0
     then 1 else n*fac(fib(squ(2+2))-1)) + 3*5 +...
->> 8+(if (if 4*(2+2)==0 then 0
          else if squ(2+2)==1 then 1
              else fib(squ(2+2)-2)+fib(squ(2+2)-1))==0
     then 1 else n*fac(fib(squ(2+2))-1)) + 3*5 +...
->> ...
```

(b) linksnormale Auswertung

Wichtig: (Links-) applikative und (links-) normale Auswertungsordnung können sich für einen Ausdruck unterscheiden in:

- Terminierungsverhalten (d.h. Terminierungshäufigkeit)
- Terminierungsgeschwindigkeit (d.h. Performanz)

nicht aber im Ergebnis! (diamond property)

Vergleich

Vorteile später Argumentauswertung:

+ Terminiert mit Normalform, wenn es (irgend-) eine terminierende Auswertungsreihenfolge gibt. Informell: Späte (wie normale und linksnormale) Auswertungsordnung terminieren häufigst möglich!
+ Wertet Argumente nur aus, wenn deren Werte wirklich benötigt werden; und dann nur einmal.

+ Ermöglicht eleganten und flexiblen Umgang mit potentiell unendlichen Werten von Datenstrukturen (z.B. unendliche Listen, Ströme, unendliche Bäume, etc.).

Nachteile später Argumentauswertung

– Konzeptuell u. implementierungstechnisch anspruchsvoller:

– Repräsentation von Ausdrücken in Form von Graphen statt linearer Sequenzen; Ausdrucksauswertung und -manipulation als Graph- statt Sequenzmanipulation.

– Partielle Auswertung von Ausdrücken kann Seiteneffekte bewirken!

– Ein-/Ausgabe nicht in trivialer Weise transparent für den Programmierer zu integrieren.

Vorteile früher Argumentauswertung:

+ Konzeptuell und implementierungstechnisch einfacher.

+ Einfache(re) Integration imperativer Konzepte.

+ Vom mathematischen Standpunkt oft 'natürlicher'.

Ad. normale Auswertungsordnung:

- Argumente werden so oft ausgewertet, wie sie verwendet werden.

  + Kein Argument wird ausgewertet, dessen Wert nicht benötigt wird.
  + Terminiert, wenn immer es eine terminierende Auswertungsfolge gibt; terminiert somit am häufigsten, insbesondere häufiger als applikative Auswertung.
  – Argumente, die mehrfach verwendet werden, werden auch mehrfach ausgewertet; so oft, wie sie verwendet werden implementierungspraktisch deshalb irrelevant; implementierungspraktisch relevant: faule Auswertung.

Ad. Früh, fleißige applikative Auswertungsordnung:

- Argumente werden genau einmal ausgewertet.

  + Jedes Argument wird exakt einmal ausgewertet; kein zusätzlicher Aufwand über die Auswertung hinaus.
  – Auch Argumente, deren Wert nicht benötigt wird, werden ausgewertet; das ist kritisch für Argumente, deren Auswertung teuer ist, gar nicht terminiert (unendlich teuer!) oder auf einen Laufzeitfehler führt.

Ad. Späte, faule Auswertungsordnung :

- Argumente werden höchstens einmal ausgewertet.

  + Ein Argument wird nur ausgewertet, wenn sein Wert benötigt wird; und dann exakt einmal.
  + Kombiniert die Vorteile von applikativer Auswertung (Effizienz) und normaler Auswertung (Terminierung!).
  – Erfordert zusätzlichen Aufwand zur Laufzeit für die Verwaltung der Auswertung von (Teil-) Ausdrücken.

6.4 Typprüfung

Typisierte Programmiersprachen[3]:

- schwach (Typprüfung zur Laufzeit)
- stark (Typprüfung/-inferenz zur Übersetzungszeit)

Haskell ist eine stark getypte Programmiersprache.
Vorteile: Verlässlicherer Code, Effizienterer Code, Effektivere Programmentwicklung.

- Gültige Ausdrücke haben wohldefinierte Typen und heißen wohlgetypt.
- Typen wohlgetypter Ausdrücke können sein:
  - Monomorph
    `fac :: Int -> Int`
    Erkennungszeichen: Keine Typvariablen, nur konkrete Typen in der Signatur.
  - Parametrisch polymorph (uneingeschränkt polymorph)
    `length :: [a] -> Int`
    Erkennungszeichen: Typvariablen, keine Typkontexte.
  - *Ad hoc* polymorph (eingeschränkt polymorph)
    `elem :: Eq a => a -> [a] -> Bool`
    Erkennungszeichen: Typvariablen und Typkontexte.
- Typen können angegeben sein:
  - explizit: Typprüfung (grundsätzlich) ausreichend.
  - implizit: Typinferenz erforderlich.

Typprüfung

1. monomorphe Typprüfung: via Kontextauswertung
2. polymorphe Typprüfung: via Unifikation (= allgemeinste gemeinsame (Typ-) Instanz einer Menge von Typausdrücken und der zugehörigen Substitution.)
3. Polymorphe Typprüfung mit Typklassen: via Typklassenkontextanalyse (= dreistufiger Prozess, bestehend aus: Unifikation, Analyse [einschl. Instanz- und Typklassendeklarationen], Simplifikation)

Typinferenz

Typsysteme sind logische Systeme, die uns erlauben, Aussagen der Form » “expression” ist Ausdruck vom Typ t« zu formalisieren und sie mithilfe von Axiomen und Regeln des Typsystems zu beweisen.

Typinferenz bezeichnet den Prozess, den Typ eines Ausdrucks automatisch mithilfe der Axiome und Regeln des Typsystems abzuleiten.

Typumgebungen sind partielle Abbildungen, die Typvariablen auf Typschemata abbilden.

[3] Lisp ist z.B. nicht getypt.

7 :: Weiterführende Konzepte

7.1 Exkurs. Interaktive Programme: Ein-/Ausgabe

Das Problem: Dialog findet nicht statt!

...unsere Programme sind bislang stapelverarbeitungsorientiert:

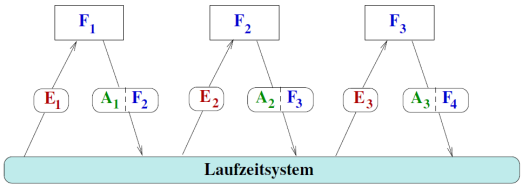
- Eingabedaten müssen zu Programmbeginn vollständig zur Verfügung gestellt werden.
- Einmal gestartet, besteht keine Möglichkeit mehr, mit weiteren Eingaben auf das Verhalten oder Ergebnisse des Programms zu reagieren und es zu beeinflussen.

Peter Pepper. *Funktionale Programmierung.* Springer–Verlag, 2003, S. 245.

...Dialog, Interaktion zwischen Benutzer und Programm findet nicht statt.

Das Ziel: Wir hätten gerne auch

...dialog- und interaktionsorientierte Haskell-Programme:



Peter Pepper. *Funktionale Programmierung.* Springer–Verlag, 2003, S. 253.

Auflösung eines scheinbar unauflöslichen Widerspruchs: Der Umgang m. Seiteneffekten in einer seiteneffektlosen Welt, denn: Konstituierendes Kennzeichen

- rein funktionaler Programmierung:

  Vollkommene Abwesenheit von Seiteneffekten!

- Ein-/Ausgabe:

  Unvermeidbare Anwesenheit von Seiteneffekten![4] (Ein- und Ausgabe, lesen und schreiben verändern den Zustand der äußeren Welt notwendig und irreversibel.)

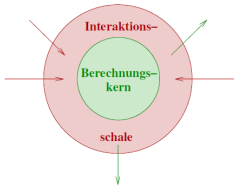
Haskells Lösung

Konzeptuelle E/A-Lösung Haskells

Konzeptuell wird in Haskell ein Programm geteilt in

- einen rein funktionalen Berechnungskern
- eine imperativähnliche Dialog- und Interaktionsschale.

zwischen denen mittels vordefinierter besonderer Ein-/Ausgabefunktionen Daten geschützt ausgetauscht werden können:



Manuel Chakravarty, Gabriele Keller. *Einführung in die Programmierung mit Haskell.* Pearson, 2004, S. 89.

A) Ein (vordef.) polymorpher Datentyp für Ein-/Ausgabe:
- data IO a = ... (Details implementierungsintern versteckt)

B) Vordefinierte primitive E/A-Operationen:
- `getChar :: IO Char`
  `getInt :: IO Int`
  ...
- `putChar :: Char -> IO ()`
  `putInt :: Int -> IO ()`
  ...

C) Ein Operator zur Komposition von E/A-Operationen:
- `(>>=) :: IO a -> (a -> IO b) -> IO b`
- 'Syntaktischer Zucker' für `(>>=)`: do-Notation.

D) Zwei Vermittlungs'operatoren' zwischen Schale und Kern:
- `return :: a -> IO a`
- `'<-' :: IO a -> a'` (⇝ informell!)

[4] Das gilt paradigmenunabhängig! Ein- und Ausgabe erzeugen Seiteneffekte notwendig, unvermeidbar, sind ohne Seiteneffekte nicht vorstellbar!

A) Trennung in rein funktionalen Berechnungskern und imperativartige Dialog- und Interaktionsschale:

Der Datentyp `(IO a)` erlaubt die Unterscheidung von Typen

- des rein funktionalen Berechnungskerns (`Char`, `Int`, `Bool`, etc.)
- der imperativartigen Dialog- und Interaktionsschale (`(IO Char)`, `(IO Int)`, `(IO Bool)`, etc.)

...IO-Werte bleiben in der Schale u. können nicht den rein fkt. Kern 'kontaminieren'.Vereinbarungen wie für `wahr_oder_falsch`, `fun'` sind in Haskell typinkorrekt u. nicht möglich; sie werden vom Typsystem ausgeschlossen und abgewiesen:

```
fun' :: Int -> Int      wert = (17+4)*2 :: Int
fun' n = n   + getInt   wahr_oder_falsch   ((-)-inkompatibel)
         :: Int :: IO Int     = (wert - wert) + (getInt - getInt)
  ((+)-inkompatibel)            :: Int        :: IO Int :: IO Int
```

B) Vordefinierte primitive E/A-Operationen
...als Bausteine, aus denen komplexe(re) Ein-/Ausgabeoperationen gebaut werden können.

C) Festlegung der zeitlichen Abfolge von E/A-Operationen ("Der Benutzer lebt in der Zeit...und kann nicht anders..."):

Der Kompositionsoperator `(>>=)` (oder gleichwertig die do-Notation, s. Kap. 15.4) erlauben die präzise Festlegung der

- zeitlichen Abfolge von E/A-Operationen.

und

D) Verbindung von funktionalem Kern und E/A-Schale

- `return`: Von Kern in Schale (in äußere Welt).
- `<-`: Von Schale (von äußerer Welt) in Kern.

Informell:

- `return` erlaubt rein funktionale Werte (engl. pure values) aus dem funktionalen Kern über die Schale als seiteneffektverursachende Werte (engl. impure values) in die äußere Welt zu transferieren.
  ...'kontaminieren' reiner Werte.
- `<-` erlaubt den 'reinen' Anteil (a-Wert) seiteneffektverursachender Werte (`(IO a)`-Wert) aus der äußeren Welt in den funktionalen Kern zu transferieren.
  ...'dekontaminieren' E/A-verschmutzter Werte.

Kontaminierung, Dekontaminierung noch bildhafter:

- `<-`: Dekontaminierung E/A-verschmutzter Werte $\rightsquigarrow$ aus einem `(IO a)`-Wert wird ein a-Wert:
  `<- :: IO a -> a` (`IO a` = Schale, `a` = Kern)
  `<-` nimmt einen E/A-verschmutzten `(IO a)`-Wert und extrahiert daraus den rein funktionalen sauberen a-Wert, der dadurch für den rein funktionalen Berechnungskern nutzbar wird.
- `return`: Kontaminierung rein funktionaler Werte $\rightsquigarrow$ aus einem a-Wert wird ein `(IO a)`-Wert:
  `return :: a -> IO a` (`a` = Kern, `IO a` = Schale)
  `return` nimmt einen rein funktionalen sauberen a-Wert und verschmutzt ihn zu einem `(IO a)`-Wert, der dadurch für und in der äußeren Welt nutzbar wird.

7.2 Robuste Programme: Fehlerbehandlung

Typische Fehlersituationen:

- Division durch null: `div 1 0`.
- Zugriff auf das erste Element einer leeren liste: `head [ ]`
- ...

Typische Sonderfälle:

- Auseinanderfallen von intendiertem und implementiertem Definitionsbereich einer Funktion, z.B. `(!)` und `fac`
- Umgang mit Argumentwerten außerhalb des intendierten Definitionsbereichs.
- ...

Drei vorgestellte Möglichkeiten:

1. Panikmodus
2. Aufangwerte (default values) (Funktionsspezifisch und Aufrufspezifisch)
3. Fehlertypen, Fehlerwerte, Fehlerfunktionen

Panikmodus

Ziel: Fehler und Fehlerursache melden, sowie fehlerhafte Programmauswertung stoppen. Möglichkeit: Die polymorphe Funktion [5]

```
error :: String --> a
```

```
fac :: Integer --> Integer
fac n
    | n == 0 = 1
    | n > 0 = n * fac (n-1)
    | otherwise = error "bam oida"
```

Vorteile:

+ Generell, einfach, schnell; im Detail
+ Generalität: Stets anwendbar
+ Einfachheit: Sowohl konzeptuell wie umsetzungstechnisch

Nachteil:

– Radikalität, i.e. die Berechnung stoppt unwiderruflich.
– Jegliche Information über den Programmlauf ist verloren, auch sinnvolle.
– Für sicherheitskritische Systeme können die Folgen eines unbedingten Programmabbruchs fatal sein.

Auffangwerte

Ziel: Panikmodus vermeiden und Programmlauf nicht zur Ganze abbrechen, sondern Berechnung möglichst sinnvoll fortführen. Möglichkeit: funktionsspezifische (Variante 1) und aufrufspezifische (Variante 2) Auffangwerte (default values) zur Weiterrechnung im Fehlerfall.

Funktionsspezifische Auffangwerte: im Fehlerfall wird ein funktionsspezifischer Wert als Resultat geliefert.

```
fac :: Integer --> Integer
fac n
| n == 0 = 1
| n > 0 = n * fac (n-1)
| otherwise = -1
```

Vorteile:

+ Panikmodus vermieden, Programmlauf nicht abgebrochen.

Nachteil:

– Oft gibt es einen zwar naheliegenden und plausiblen funktionsspezifischen Auffangwert; jedoch kann dieser das Eintreten der Fehlersituation verschleiern und intransparent machen, wenn der Auffangwert auch als Resultat einer regulären Berechnung auftreten kann.

[5] Liefert Programmfehler: Funktion f: Ungültige Eingabe - und stoppt Programm.

– Oft fehlt ein naheliegender und plausibler Wert als Auffangwert; die Wahl eines Auffangwerts ist in diesen Fällen willkürlich und unintuitiv.

– Oft fehlt ein funktionsspezifischer Auffangwert gänzlich; Auffangwertvariante 1 ist in diesen Fällen nicht anwendbar.

Aufrufspezifische Auffangwerte: im Fehlerfall wird ein aufrufspezifischer Auffangwert als Resultat geliefert.

```
fac :: Integer --> Integer
fac n
| n == 0 = 1
| n > 0 = n * fac (n-1)
| otherwise = n
```

Vorteile:

+ Panikmodus vermieden, Programmlauf nicht abgebrochen.

+ Generalität, stets anwendbar.

+ Flexibilität, aufrufspezifische Auffangwerte ermöglichen variierende Fehlerwerte und Fehlerbehandlung.

Nachteil:

– Transparente Fehlerbehandlung ist nicht gewährleistet, wenn aufrufspezifische Auffangwerte auch reguläres Resultat einer Berechnung sein können.

– In diesen Fällen Gefahr ausbleibender Fehlerwahrnehmung mit (möglicherweise fatalen) Folgen durch.

Fehlertypen, Fehlerwerte, Fehlerfunktionen

Ziel: Systematisches Erkennen, Anzeigen und Behandeln von Fehlersituationen. Möglichkeit: Dezidierte Fehlertypen, Fehlerwerte und Fehlerfunktionen statt schlichter Auffangwerte.

Maybe: Anzeigbarkeit von Fehlern wird erreicht durch Übergang von Typ `a` zum (Fehler-) Datentyp `Maybe a`:

```
data Maybe a = Just a
             | Nothing
             deriving (Eq, Ord, Read, Show)
```

Beispiel:

```
div' :: Int -> Int -> Maybe Int
div' n m
 | m /= 0 = Just (div n m)
 | m == 0 = Nothing

div' 13 5 ->> Just 2            (Division geklappt)
div' 13 0 ->> Nothing         (Division gescheitert)
```

Dazu:

- `map_Maybe`: Erkennen und weiterreichen von Fehlern.

```
map_Maybe :: (a --> b) --> Maybe a --> Maybe b
map_Maybe f Nothing = Nothing -- (Fehlerwert Nothing von map_Maybe durchgereicht)
map_Maybe f (Just u) = Just (f u)
```

- `maybe`: Fangen und behandeln von Fehlern.

```
maybe :: b --> (a --> b) -> Maybe a --> b
maybe x f Nothing = x  -- (Auffangwert x als Resultat geliefert)
maybe x f (Just u) = f u
```

Vorteile:

+ Fehler können erkannt, angezeigt, weitergereicht und schließlich gefangen und (im Sinn von Auffangwertvariante behandelt werden.

+ Systementwicklung ist ohne explizite Fehlerbehandlung möglich (z.B. mit nichtfehlerbehandelnden Funktionen wie `div`).

+ Fehlerbehandlung kann nach Abschluss durch Ergänzung der fehlerbehandelnden Funktionsvarianten zusammen mit den Funktionen `map_Maybe` und maybe umgesetzt werden.

Nachteil:

– Geänderte Funktionalität: `Maybe b` statt `b`.

7.3 Module, Modularisierung von Programmen

. . . Zerlegung von Programmen in überschaubare, (oft) getrennt übersetzbare Programmeinheiten als wichtige programmiersprachliche Unterstützung der *Programmierung im Großen*. Allgemeine Vorteile:

1. Arbeitsphysiologisch: Unterstützung arbeitsteiliger Programmierung.

2. Softwaretechnisch: Unterstützung der Wiederbenutzung von Programmen und Programmteilen.

3. Implementierungstechnisch: Unterstützung getrennter Übersetzung (engl. separate compilation).

4. Insgesamt: Höhere Effizienz der Softwareerstellung bei gleichzeitiger Qualitätssteigerung (Verlässlichkeit) und Kostenreduktion.

Wichtige Eigenschaften:

1. Kohäsion (modullokal, intramodular): beschäftigt sich mit dem inneren Zusammenhang von Modulen, mit Art und Typ der in einem Modul zusammengefassten Funktionen.

2. Koppelung (modulübergreifend, intermodular): beschäftigt sich mit dem äußeren Zusammenhang von Modulen, dem Import-/Export- und Datenaustauschverhalten.

Ziele guter Modularisierung

...von einer technischen Programmperspektive aus:

Modullokal (intramodular): Module sollen

- einen klar umrissenen, unabhängig von anderen Modulen verständlichen Zweck besitzen.
- nur einer Abstraktion entsprechen.
- einfach zu testen sein.

Modulübergreifend (intermodular): Modular entworfene Programme sollen

- Auswirkungen von Designentscheidungen (z.B. Einfachheit vs. Effizienz einer Implementierung)
- Abhängigkeiten von anderen Programmen oder Hardware

...auf (möglichst) wenige Module beschränken.

Ziele guter Modularisierung

...von einer semantischen, inhaltl. Programmperspektive aus:

Modullokal (intramodular):

- Funktionale Kohäsion: Fasse Funktionen gleicher Funktionalität zusammen, z.B. Sortierverfahren, Ein-/Ausgabe,...
- Datenkohäsion: Fasse Funktionen zusammen, die auf den gleichen Datenstrukturen arbeiten, z.B. Funktionen auf trigonometrischen Daten,...

Modulübergreifend (intermodular):

- Schwache funktionale Koppelung: Strebe nach wenigen, wohlbegründeten funktionalen Beziehungen und Abhängigkeiten zwischen Modulen.
- Feste Datenkoppelung: Strebe nach Kommunikation modulverschiedener Funktionen durch Wertübergabe: Ergebnisse einer Funktion werden Argumente einer anderen.

Haskellmodule

... unterstützt den Import und Export von Datentypen, Typsynonymen, Typklassen und Funktionen. Im einzelnen:

- Import: Selektiv/nicht selektiv, Qualifiziert, mit Umbenennung
- Export: selektiv/nicht selektiv händischer Reexport, nicht unterstützt – Automatischer Reexport

Moduldateien werden eingeleitet von der Zeile:

```
module M where
```

gefolgt von Deklarationen/Definitionen von:

1. Typen (algebraische Typen, Neue Typen, Typsynonyme)
2. Typklassen
3. Funktionen

```
module M where              -- Moduldefinition
data D_1 ...    = ...       -- Algebraische Typen
...
data D_n ...    = ...
newtype N_1 ... = ...       -- Neue Typen
...
newtype N_m ... = ...
type T_1 ...    = ...       -- Typsynonyme
...
type T_p ...    = ...
class C_1 ...               -- Typklassen
...
class C_q ...
f_1 :: ...                  -- Funktionen
f_1 ... = ...
...
f_r :: ...
f_r ... = ...
```

Schematischer Aufbau von Haskellmodulen

Import

Nicht selektiver Import (schematisch)

```
module M1 where
...

module M2 where
import M1
...
```

- Modul M2 importiert aus Modul M1 alle (global sichtbaren) Bezeichner und Definitionen, die danach in M2 verwendet werden können.

Selektiver Import (schematisch)

```
module M1 where
...
module M2 where                          -- Variante 1
import M1 (D_1 (..), D_2, T_1, C_1 (..), C_2, f_5)
...
module M3 where                          -- Variante 2
import M1 hiding (D_1, T_2, f_1)
...
```

- M2 importiert aus M1 ausschließlich die explizit genannten Bezeichner und Definitionen; das sind: D_1 (einschließlich von M1 exportierter Konstruktoren), D_2 (ohne Konstruktoren), T_1, C_1 (..) (einschließlich von M1 exportierter Funktionen), C_2 (ohne Funktionen), f_5.
- M3 importiert aus M1 alle in M1 (sichtbaren) Bezeichner und Definitionen mit Ausnahme der explizit genannten.

Export

Nicht selektiver Export (schematisch)

```
module M1 where
data D_1 ... = ...
...
newtype N_1 ... = ...
...
type T_1 = ...
...
class C_1 ...
...
f_1 :: ...
f_1 ... = ...
...
```

- Alle in M1 eingeführten global sichtbaren Bezeichner und Definitionen sind exportbereit und können von anderen Modulen importiert werden.

  Beachte: Die Zeile `module M1 where...` ist bedeutungsgleich zu `module M1 (module M1) where...`

Selektiver Export (schematisch)

```
module M1 (D_1 (..), D_2, D_3 (Dc_1,...,Dc_k), C_1 (..),
           C_2, C_3 (cf_1,...,cf_l), T_1, f_2, f_5) where
data D_1 ... = ...
...
newtype N_1 ... = ...
...
type T_1 = ...
...
class C_1 ...
...
f_1 :: ...
f_1 ... = ...
...
```

- Nur die explizit genannten Bezeichner, Definitionen aus M1 sind exportbereit und können von anderen Modulen importiert werden. Dabei ist D_1 einschließl. seiner Konstruktoren exportbereit, D_2 ohne, D_3 mit den explizit genannten. Analog für die Klassen C_i.
- Beachte: Selektiver Export unterstützt das Geheimnisprinzip!

Reexport

Reexport (schematisch): Nicht automatisch!

```
module M1 where...

module M2 where
import M1
...
f_M2j
...

module M3 where
import M2
...
```

- M2 importiert nicht selektiv aus M1, d.h. alle in M1 (global sichtbaren) Bezeichner, Definitionen werden von M2 importiert und können in M2 benutzt werden.
- M3 importiert nicht selektiv aus M2, d.h. alle in M2 (global sichtbaren) Bezeichner, Definitionen werden von M3 importiert und können in M3 benutzt werden, nicht jedoch die von M2 aus M1 importierten Namen, d.h. kein automatischer Reexport!

Abhilfe: Händischer Reexport!

...in den zwei Varianten nicht selektiv und selektiv:

```
module M2 (module M1,f_M2_j) where        (nicht selektiv)
import M1
...
f_M2_j
...                                                (selektiv)
module M3 (D_1 (..), D_2, D_3 (Dc_1,Dc_2), C_1 (..), C_2,
           C_3 (cf_1,cf_2,cf_3), f_1,f_M3_k) where
import M1
...
f_M3_k
...
```

- Nicht selektiver Reexport von M1 aus M2: M2 reexportiert jeden aus M1 importierten Namen, sowie das M2-lokale f_M2_j aus M2.
- Selektiver Reexport von M1 aus M3: M3 reexportiert von den aus M1 importierten Namen ausschließlich D_1 (einschließl. Konstruktoren), D_2 (ohne Konstruktoren), D_3 (mit angegebenen Konstruktoren); analog f. d. Klassen C_1, C_2, C_3, f_1 und das M3-lokale f_M3_k.

7.4 Abstrakte Datentypen

Konkrete Datentypen (KDT) (in Haskell: Algebr. Datentypen)

- werden durch die exakte Angabe und Darstellung ihrer Werte spezifiziert, aus denen sie bestehen.
- auf ihnen gegebene Funktionen/Operationen werden zum Definitionszeitpunkt nicht angegeben und bleiben offen.

Abstrakte Datentypen (ADT)

- werden durch ihr Verhalten spezifiziert, d.h. durch die auf ihren Werten definierten Funktionen/Operationen und deren Zusammenspiel.
- die tatsächliche Darstellung der Werte des Datentyps wird zum Definitionszeitpunkt nicht angegeben u. bleibt offen.
- Dem Anwender eines abstrakten Datentyps wird die Darstellung der Werte und die Implementierung der Funktionen darauf nie bekanntgegeben: Geheimnisprinzip!

Grundlegende Idee von ADT-Definitionen

...Festlegung u. Implementierung eines Datentyps in 3 Teilen:

A) Schnittstellenfestlegung: Angabe der auf den Werten des Datentyps zur Verfügung stehenden Operationen in Form ihrer syntaktischen Signaturen (öffentlich).

B) Verhaltensfestlegung: Festlegung der Bedeutung der Operationen durch Angabe ihres Zusammenspiels in Form von Axiomen (sog. Gesetzen), die von jeder (!) Implementierung dieser Operationen einzuhalten sind (öffentlich).

C) Implementierung: Implementierung des ADT durch einen KDT, der A) und B) erfüllt (nicht öffentlich).

Wichtig: In A) und B) wird die Darstellung der Werte des abstrakten Datentyps ausdrücklich nicht festgelegt; sie bleibt verborgen und deshalb für die Implementierung in C) als Freiheitsgrad offen!

Vorteil von ADT-Definitionen: die Trennung von öffentlicher A), B) Schnittstellen- und Verhaltensfestlegung und nicht öffentlicher C) Implementierung erlaubt die

+ Implementierung zu verstecken (Geheimnisprinzip!)

+ Zweckmäßigkeit und Anforderungen (z.B. Einfachheit, Performanz) auszuwählen und in der Einsatzphase bei Bedarf auch auszutauschen.

Konzeptueller Vorteil abstrakter Datentypen das Geheimnisprinzip: Nur die ADT-Schnittstelle ist bekannt, die KDT-Implementierung bleibt verborgen. Damit:

+ Schutz der Datenstruktur vor unkontrolliertem oder nicht beabsichtigtem/zugelassenem Zugriff.

+ Einfache Austauschbarkeit der zugrundeliegenden Implementierung.

+ Unterstützung arbeitsteiliger Programmierung.

A Ausgewählte, ausgearbeitete Beispiele und Prüfungen

A.1 Kurztest für Probetestlauf 1

1. Welches ist die Standardauswertungsordnung von Haskell?

   Antwort: Lazy evaluation, d.h. die besprochene effiziente Implementierungsumsetzung linksnormaler Auswertung, bei der Ergebnisse auszuwertender Ausdrücke nur so weit berechnet werden, wie gerade benötigt.

2. Was ist ein Operatorabschnitt? Geben Sie ein Beispiel dafür an.

   Antwort: Das sind partiell ausgewertete Binäroperatoren (operator sections). Beispiele:

   - $(*2)$ dbl, die Funktion, die ihr Argument verdoppelt $(\lambda x.x * 2)$
   - $(2:)$ headAppend, die Funktion, die 2 an den Anfang einer typkompatiblen Liste setzt $(\lambda xs.(2 : xs))$
   - $(+1)$ inc, die Funktion, die ihr Argument um 1 erhöht $(\lambda x.x + 1)$

3. Was ist mit Funktionen als *first class citizens* in funktionalen Programmiersprachen gemeint?

   Antwort: Funktionen sind erstrangige Sprachelemente, d.h. Ausdrücke funktionalen Werts dürfen (fast überall) stehen, wo auch Ausdrücke elementaren Werts stehen dürfen und umgekehrt, d.h. Funktion können etwa u.A. einer anderen Funktion als Argument übergeben oder von einer anderen Funktion als Wert zurückgegeben werden. [6]

4. Definieren Sie einen Operator `(;)` für die Komposition (d.h. Hintereinanderausführung) von Funktionen von links nach rechts; geben Sie auch die Signatur von `(;)` an.

   Antwort:

   ```
   (;) :: (b --> c) --> (a --> b) --> a --> c
   (g ; f) x = g (f x)
   ```

   bzw.

   ```
   ((;) g f) x = g (f x)
   ```

   und mit η-Reduktion und (.) auch:

   ```
   (g ; f) = g . f
   ```

5. Ist der Typ Ihres Operators `(;)` aus Aufgabe 4 allgemeinstmöglich? Begründen Sie Ihre Antwort knapp, aber präzise.

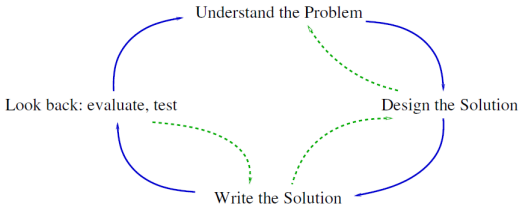
   Antwort: Die Funktion f liefert, angewandt auf ein Argument x :: a, einen Wert y :: b, der als Argument y :: b unter Anwendung von g den Wert z :: c liefert. In jedem Fall haben wir uns nicht eingeschränkt. (Der Compiler macht dies etwa mit `:t` unter Kontextauswertung und Unifikation)

[6] Polymorphie wird auf den Folien auch als *first class citizen* angeführt.

A.2 Kurztest für Probetestlauf 2

1. Was versteht Simon Thompson unter reflektivem Programmieren?

   ...der Programm-Entwicklungszyklus nach Simon Thompson, Haskell: The Craft of Fuctional Programming, 2. Auflage, 1999, Kap. 11 'Reflective Programming':

   ...in jeder der 4 Phasen ist es nützlich, (sich) Fragen zu stellen, zu beantworten und den Lösungsweg ggf. anzupassen.

2. Was kann in Haskell polymorph deklariert werden?

   Antwort:

   (a) Funktionen: Parametrische Polymorphie (oder echte Polymorphie) z.B. `map` oder Ad hoc Polymorphie (oder unechte Polymorphie, Überladung) z.B. `(+)` (direkt überladen) oder `sum` (indirekt überladen)

   (b) Datentypen: Algebraische Datentypen mit `data`, Neue Typen mit `newtype`, Typsynonyme mit `type`.

3. Was ist mit der sog. record-Syntax von Haskell gemeint?

   Antwort: Produkttypen (oder Verbundtypen (engl. record types)), d.h. Typen mit möglicherweise unendlich vielen (Tupel-) Werten. Beispiel:

```
type Ziffernfolge = String
type Zf           = Ziffernfolge
data G            = M | W deriving (Eq,Ord,Show)
newtype Gb        = Gb (Zf,Zf,Zf) deriving (Eq,Ord,Show

data Meldedaten   = Md { vorname    :: String,
                         nachname   :: String,
                         geboren    :: Gb
                         geschlecht :: G,
                         gemeinde   :: String,
                         strasse    :: String,
                         hausnummer :: Int,
                         plz        :: Int,
                         land       :: String
                       } deriving (Eq,Ord,Show)
```

4. Welchen Typ hat das vordefinierte Funktional map auf Listen in Haskell? Geben Sie zwei unterschiedliche Implementierungen für `map` an (z.B. rekursiv, mit einer Listenkomprehension):

```
map :: (a --> b) --> [a] --> [b]
map f [ ] = [ ]
map f (l:ls) = (f l) : map f ls     -- Rekursion

map f ls = [f l | l <-- ls]         -- Listekomrehension
```

5. (1) Was sind Muster in Haskell? (2) Wozu sind sie nützlich?

   Antwort: (1) Das sind (syntaktische) Ausdrücke, die die Struktur von Werten beschreiben.Musterpassung (pattern matching) dient dabei in Funktionsdefinitionen durch Muster festgelegte Alternativen auszuwählen. Beispiel:

```
sortiere :: [Integer] --> [Integer]
sortiere [ ] = [ ]                    -- (Muster der leeren Liste)
sortiere (n:[ ]) = [n]                -- (Muster einelementiger Liste)
sortiere (n:ns) = ...                 -- (Muster zwei-/mehrelementiger Liste)
```

   (2) Eleganz; führen (i.a.) zu knappen, gut lesbaren Spezifikationen; können aber auch zu subtilen Fehlern führen; Programmänderungen/-weiterentwicklungen erschweren.

A.3 WS 20 Übungstestangabe 1

Aufgabe 1. Zwei Zeichenreihen haben Abstand n iff die beiden Zeichenreihen sind von gleicher Länge und unterscheiden sich an genau n Positionen voneinander.

1. Gibt es Fälle ür die Berechnung des Abstandes von Zeichenreihen, die von der obigen Beschreibung nicht erfasst sind und eine besondere Behandlung erfordern? Wenn ja, welche?

   Antwort: Ja, der Fall `length` $\text{string}_1 \neq$ `length` string_2.

2. Wie können etwaige besondere Fälle sinnvoll behandelt werden? Nennen Sie eine Methode dafür und wie genau Sie damit etwaige besondere Fälle hier behandeln.

   Antwort: Zum Beispiel mit "Panikmodus". Wann immer zwei Strings nicht die selbe Länge haben, return "error".

3. Schreiben Sie eine curryfizierte Haskell-Rechenvorschrift abs einschließlich ihrer syntaktischen Signatur, die angewendet auf zwei Zeichenreihen ihren Abstand liefert. Etwaige besondere Fälle sollen von abs so behandelt werden, wie in der vorigen Teilaufgabe beschrieben:

```
type Nat0 = Int

abs :: [Char] --> [Char] --> Nat0
abs s1 s2
    | length s1 == length s2 = distt (zip s1 s2) 0
    | True                   = error "ayy"

distt :: [(Char, Char)] --> Nat0 --> Nat0
distt [ ] i = i
distt (x:xs) i = if fst x == snd x then distt xs i else  distt xs (i+1)
```

4. Erklären Sie knapp, aber gut nachvollziehbar, wie ihre Rechenvorschrift vorgeht.

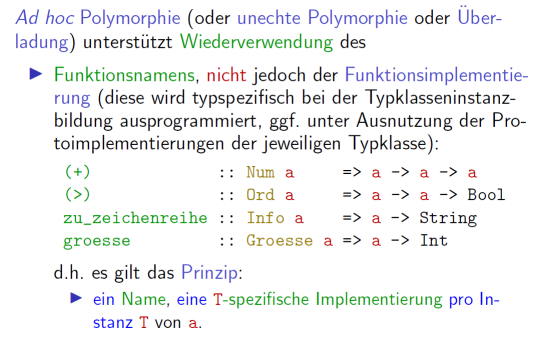
   Antwort: Wenn `length` $\text{string}_1 \neq$ `length` string_2, bauen wir eine Tupelliste und übergeben sie an eine Hilfsfunktion `distt` zusammen mit einem Laufindex i (beginnt bei 0). Die Funktion `dist` überprüft sukzessive die Tupeleinträge auf Gleichheit und inkrementiert den Laufindex, falls Ungleichheit besteht.

Aufgabe 2

1. Wie sind Typklassen in Haskell aufgebaut? Siehe da.
2. Wozu dienen sie?

   Antwort: Siehe darüber und: Konstrukt der funktionalen Programmierung zur Implementierung von Ad hoc Polymorphie.
3. Mit welcher Art von Polymorphie sind Typklassen eng verbunden?

   Antwort: Siehe darüber.
4. Was ermöglicht diese Art von Polymorphie wiederzuverwenden?



Ad hoc Polymorphie (oder unechte Polymorphie oder Überladung) unterstützt Wiederverwendung des
Funktionsnamens, nicht jedoch der Funktionsimplementierung (diese wird typspezifisch bei der Typklasseninstanzbildung ausprogrammiert, ggf. unter Ausnutzung der Protoimplementierungen der jeweiligen Typklasse):
(+) :: Num a => a -> a -> a
(>) :: Ord a => a -> a -> Bool
zu_zeichenreihe :: Info a => a -> String
groesse :: Groesse a => a -> Int
d.h. es gilt das Prinzip:
ein Name, eine T-spezifische Implementierung pro Instanz T von a.

5. Welche synonymen Bezeichnungen gibt es für diese Polymorphieart?

   Antwort: Unecht Polymorphie oder Überladung.

Aufgabe 2

```
f :: Integer --> Integer
f n = if n == 0 then 0 else f (n-1) + n * n
```

1. Was berechnet f?

   Antwort: $f = \sum_{i=1}^{n} i^2$
2. Von welchem Rekursionstyp ist f?

   Antwort: Lineare Rekursion.
3. Schreiben Sie die Funktion f bedeutungsgleich

   (a) mithilfe bewachter Ausdrücke.

   ```
   f2 :: Integer --> Integer
   f2 n
      | n == 0 = 0
      | True   = f (n-1) + n * n
   ```

   (b) argumentfrei mithilfe einer anonymen λ-Abstraktion.

   ```
   f3 :: Integer --> Integer
   f3 = \n --> (if n == 0 then 0 else f (n-1) + n * n)
   ```

(c) unter (Mit-) Verwendung einer Listenkomprehension.[7]

```
f4 :: Integer --> Integer
f4 n = last [x | x <- list] where list = map f [0..n]
```

Aufgabe 4. Wir betrachten Bäume, deren Blätter eine Benennung und deren Nichtblätter zwei Benennungen tragen und keinen oder beliebig viele Teilbäume besitzen. Ein Wald von Bäumen enthält beliebig viele oder auch gar keinen Baum.

1. Geben Sie möglichst typallgemeine Definitionen für Bäume und Wälder in Haskell an. Verwenden Sie algebraische Datentypen nur, wenn nötig:

```
data Baum x y z = Blatt x
                | Knoten y z [Baum x y z] -- deriving(Eq)
type Wald x y z = [Baum x y z]
```

2. Machen Sie den Baumtyp zu einer Instanz der Typklasse Eq, ohne dafür eine deriving-Klausel zu verwenden. Zwei Bäume sind gleich gdw. die Bäume stimmen in Struktur und Benennungen überein.

```
instance  (Eq a, Eq b, Eq c) => Eq (Baum a b c) where
(Blatt x) == (Blatt y) = x == y
(Knoten y1 z1 baum1) == (Knoten y2 z2 baum2) = y1 == y2 && z1 == z2 && baum1 == baum2
_ == _ = False
```

Aufgabe 5. Was bedeutet die Signatur der Funktion h: `h :: (a --> b) --> [a] --> [b]` in

1. curryfizierter Lesart?

   Antwort: h bildet eine Funktion vom Typ `(a --> b)` auf eine Funktion vom Typ `([a] --> [b])` ab. Entspricht: `(a --> b) --> ([a] --> [b])`.

2. nicht-curryfizierter Lesart?

   Antwort: h bildet ein Argument vom Typ `[a]` auf einen Wert vom Typ `[b]` mithilfe einer Funktion vom Typ `(a --> b)` ab.

A.4 WS 20 Übungstestangabe 2

Aufgabe 1. Zwei Zeichenreihen z und z' heißen anfangsgleich vom Grad n, wenn die ersten n Zeichen von z und z' positionsweise ident sind.

1. Schreiben Sie eine Haskell-Rechenvorschrift ag einschließlich ihrer syntaktischen Signatur, die angewendet:

```
anfangsgleich :: (Eq a) => [a] -> [a] -> Nat0
anfangsgleich [] _ = 0
anfangsgleich _ [] = 0
anfangsgleich l1 l2 = countIndex l1 l2 0

countIndex :: (Eq a) => [a] -> [a] -> Nat0 -> Nat0
```

[7] Eines der retardiertesten Dinge, die ich je geschrieben hab, zumal `map f [0..n]` schon das Ergebnis liefert.

```
countIndex [] _ i = i
countIndex _ [] i = i
countIndex (x:xs) (y:ys) i = if x == y then countIndex xs ys i+1 else i
```

2. Erklären Sie knapp, aber gut nachvollziehbar, wie ihre Rechenvorschrift vorgeht.

   Antwort: Laufindex i und vergleiche Elementweise bis entweder das letzte Paar an Elementen erreicht ist, dann ist grad $n =$ `length` der kürzeren Liste, oder return den Index, wo die erste Ungleichheit auftritt.

Aufgabe 2. Schreiben Sie eine Funktion sum für die Berechnung der Summe der ersten n natürlichen Zahlen mit eins als kleinster natürlicher Zahl. Dabei soll gelten: sum

1. hat den Typ `Int --> Int`.
2. stützt sich für die Berechnung auf eine repetitiv rekursive Funktion sum' ab.
3. sieht eine Panikmodusfehlerbehandlung vor.

Antwort:

```
sum :: Int -> Int
sum n = if not (n >= 0) then error "ayyyo bisch" else sum' [0..n] 0

sum' :: [Int] --> Int --> Int
sum' [ ] m = m
sum' (n:ns) m = sum' ns (n+m)
```

Welchen Wert liefert `sum (2+3) + 5? --> sum (2+3) + 5 = (sum (2+3)) + 5 = 15+5 = 20`.

Aufgabe 3. Siehe da.

Aufgabe 4.

```
h [ ] ys = ys
h (x:xs) ys = h xs (x:ys)
```

1. Welches ist der allgemeinstmögliche Typ von h?

   Antwort: `[a] -> [a] -> [a]`

2. Ist h eine curryfizierte oder uncurryfizierte Funktion?

   Antwort: Uncurryfiziert. Außerdem sehen wir, dass partielle Auswertung nicht unterstützt wird.

3. Von welchem Rekursionstyp ist h und woran erkennt man diesen Rekursionstyp?

   Antwort: Repetitive (schlichte, endständige) Rekursion.

4. Was berechnet h? Was ist seine Bedeutung?

   Antwort: h konsumiert zwei Listen `l1, l2 :: [a]` und hängt die Elemente von `l1` vorne an `l2` an.

5. Was ist die Bedeutung von h mit der leeren Liste als zweitem Argument?

   Antwort: Dann wirkt h wie die Funktion `reverse`.

Aufgabe 5. Gegeben ist die Funktion `f`: `f x y = (x + y) * (x - y)`

1. Welches ist der allgemeinstmögliche Typ von `f`

   Antwort: `Num a => a --> a --> a`

   Warum? Weil wir uns von `(+), (-)` und `(*)` dazu motivieren lassen.

2. Ist dieser Typ monomorph oder polymorph?

   Antwort: Eine Funktion in Haskell heißt ad hoc polymorph (oder unecht polymorph oder überladen), wenn die Typen eines oder mehrerer ihrer Parameter angegeben durch Typvariablen durch Typkontexte eingeschränkt Werte aller durch den Typkontext zugelassenen Typen als Argument zulassen. [Die Funktion `f` ist polymorph]

3. Im Fall von Polymorphie:

   - Wie heißt diese Art von Polymorphie möglichst genau?
   - Gibt es Synonyme für diesen Polymorphiebegriff? Wenn ja, welches oder welche?

   Antwort: Siehe darüber.

Aufgabe 7.

```
type Nat1 = Int
type Artikelnummer = Int
type PreisInEURcent = Nat1
type Warenkatalog = (Artikelnummer --> PreisInEURcent) -- total definiert!
type NeuerPreis = PreisInEURcent
```

1. Angewendet auf einen Warenkatalog, eine Artikelnummer und einen neuen Preis wird der Preis des entsprechenden Artikels auf den neuen Preis gesetzt. Die Preise aller anderen Artikel bleiben unverändert.

   ```
   preisaenderung :: Warenkatalog --> Artikelnummer --> NeuerPreis --> Warenkatalog
   preisaenderung w a n =
   \artikelnummer --> if artikelnummer == a then n else w artikelnummer
   ```

2. Funktionen wie `preisaenderung` gehören zu einer Teilmenge von Funktionen, die wg. ihrer Wichtigkeit einen eigenen Namen haben. Wie heißen die Funktionen dieser Teilmenge und woran erkennt man sie?

   Antwort: Die Funktion konsumiert u.a. eine Funktion als Argument und gibt eine Funktion vom Typ Warenkatalog (curryifzierte Leseart) zurück. In diesem Sinne gehört sie zu den Funktionen höherer Ordnung.

B Selbsteinschätzungstest

Selbsteinschätzungstest 1

KW 45 (02.-06.11.2020)

Stoff: Vortragsteile I und II

Dauer: 10min. (ohne Abgabe, ohne Beurteilung)

0. **Falls Übung in Präsenz:** *Lösen Sie die Sitzplatzplanaufgabe in Tuwel und tragen Sie Ihren Sitzplatz im Hörsaal FAV 1 im freien Textfeld der Aufgabe ein!*

1. Welche der folgenden Symbolfolgen bezeichnen Operatoren in Haskell, welche nicht? Welche Bedeutung haben sie? Welche sind ohne Bedeutung in Haskell?

   ```
   +
   ++
   -
   --
   *
   **
   /
   //
   ```

2. Gegeben ist folgende Haskell-Rechenvorschrift:

   ```
   f [] _           == []
   f _ []           == ()
   f (b::bs) c::cs == (b || c) :: f (bs,cs)
   ```

   (a) Markieren und verbessern Sie alle Syntaxfehler in den definierenden Gleichungen von `f`.

   (b) Welche Typsignatur hat `f` (nach Ausbesserung der Syntaxfehler)?

   (c) Was berechnet `f` (nach Ausbesserung der Syntaxfehler)?

3. Schreiben Sie eine uncurryfizierte rekursive Haskell-Rechenvorschrift `hoch`, die die *Potenz*-Funktion berechnet:

   $$Potenz : IN_1 \times IN_0 \to IN_1$$

4. Was unterscheidet konzeptuell eine Wertzuweisung wie `x := y+z` in einer imperativen Programmiersprache von einer Wertvereinbarung wie `x = y+z` in einer funktionalen Programmiersprache?

5. Gibt es ein Beispiel, das Ihre Antwort auf die vorige Frage besonders gut illustriert? Welches?

Lösungsvorschlag zu

Selbsteinschätzungstest 1

KW 45 (02.-06..2020)

Stoff: Vortragsteile I und II

Dauer: 10min. (ohne Abgabe, ohne Beurteilung)

1. Die Symbolfolgen bezeichen oder bedeuten:

`+`	Addition
`++`	Listenkonkatenation
`-`	Subtraktion, Vorzeichen
`--`	einzeiliger Kommentar
`*`	Multiplikation
`**`	Exponentiation
`/`	Gleitkommadivision
`//`	Ohne (vordef.) Bedeutung in Haskell

   Die Symbolfolgen `--` und `//` bezeichnen keine bzw. keine vordefinierten Operatoren in Haskell.

2. Zu

   (a)
   ```
   f [] _          = []
   f _ []          = []
   f (b:bs) (c:cs) = (b || c) : f bs cs
   ```

   (b) `f :: [Bool] -> [Bool] -> [Bool]`

   (c) `f` berechnet elementweise das 'logische oder' der beiden Argumentlisten (bis zur Länge der kürzeren der beiden Listen).

3.
   ```
   type Nat0 = Int
   type Nat1 = Int

   hoch :: (Nat1,Nat0) -> Nat1
   hoch (m,n)
    | n == 0 = 1
    | n > 0  = m * hoch (m,n-1)
   ```

4. Imperative Wertzuweisungen stellen eine temporäre, nicht dauerhafte Verbindung zwischen dem in Speicherzelle `x` gespeicherten Wert und dem Wert des rechtsseitigen Ausdrucks dar. Dieser ist selbst in der Zeit veränderbar, da er aus den momentanen Werten der in ihm vorkommenen Variablen berechnet wird, d.h. aus den in den zugehörigen Speicherzellen momentan gespeicherten Werte.

   Funktionale Wertvereinbarungen stellen eine dauerhafte, nicht temporäre Verbindung zwischen dem linksseitigen Namen und dem Wert des rechtsseitigen Ausdrucks dar. Der Wert dieses Ausdrucks ist ebenfalls unveränderbar in der Zeit. Eine vom Programm aus ansprechbare Verbindung zwischen Namen und Speicherzellen besteht nicht.

5. Selbstbezügliche Wertzuweisungen bzw. Wertvereinbarungen wie

   `x := x+1`, `x = x+1`

   illustrieren den Unterschied besonders deutlich. Interpretiert als

   - Wertzuweisung ist `x := x+1` wohldefiniert: In die mit `x` bezeichnete Speicherzelle wird der dort gespeicherte um 1 erhöhte Wert geschrieben.
   - Wertvereinbarung ist die Bedeutung von `x = x+1` undefiniert: Konzeptuell: Für keinen Wert von `x` ist die Gleichung erfüllt. Operationell: Die Auswertung des Ausdrucks `x` terminiert nicht (regulär, sondern mit Speicherüberlauf).

6. ***Bitte Teilnehmeranzahl am Übungsgruppentermin notieren!***

Selbsteinschätzungstest 2

KW 46 (09.-13.11.2020)

Stoff: Vortragsteile I, II und III

Dauer: 10min. (ohne Abgabe, ohne Beurteilung)

1. Was haben Typsicherheit und der Verlust einer Marssonde miteinander zu tun? Wie hieß die Marssonde?

2. Gegeben sind folgende Deklarationen:

```
> type UntereSchranke = Int
> type ObereSchranke  = Int
> data Intervall      = IV (UntereSchranke,ObereSchranke)
>                       | Leer
>                       | Ungueltig deriving Show
```

   (a) Können die Zeichen '>' weggelassen werden? Wenn ja, warum? Wenn nein, warum nicht?

   (b) Was liefern folgende Ausdrücke?

```
show (IV (2,5)) ->>
show IV (2,5) ->>
show Leer ->>
show Ungueltig ->>
show (Ungueltig) ->>
show (IV (2-3,2+3)) ->>
show (IV (2-3,(2+)3)) ->>
show (IV (2-3,(+2)3)) ->>
show (IV ((2-)3,2+3)) ->>
show (IV ((-2)3,2+3)) ->>
```

3. `type`-, `newtype`- und `data`-Deklarationen sind gegeneinander austauschbar. Richtig oder falsch? Begründen Sie Ihre Antwort.

4. Gegeben ist die Haskell-Rechenvorschrift `hoch`:

```
type Nat0 = Int
type Nat1 = Int
hoch :: ((Nat0 -> ((Nat0 -> Nat1))))
m 'hoch' n
 | (n == 0) = ((1))
 | (n > 0)  = (m * (m 'hoch' (n-1)))
```

   (a) Warum steht `hoch` im Rumpf von `hoch` in Hochkommata?

   (b) Was passiert, wenn `hoch` mit einem negativen Wert für `n` aufgerufen wird?

   (c) Welchen Rekursionstyp hat `hoch`?

   (d) Ist der Ausdruck `hoch 5` syntaktisch korrekt? Wenn nein, warum nicht? Wenn ja, welchen

      i. Typ

      ii. Wert

   hat der Ausdruck `hoch 5`?

   (e) Streichen Sie in in der Deklaration von `hoch` so viele Klammern wie möglich, ohne dass sich die Bedeutung ändert oder die syntaktische Korrektheit verloren geht.

Lösungsvorschlag zu

KW 46 (09.-13.11.2020)

Stoff: Vortragsteile I, II und III

Dauer: 10min. (ohne Abgabe, ohne Beurteilung)

1. Die Marssonde hieß *Mars Climate Orbiter*. Sie ging am 23.09.1998 beim Landeanflug auf den Mars verloren, weil einige Programme mit metrischen, andere mit nichtmetrischen Größen rechneten (vgl. Kap. 4.2.1).

2. (a) Die Zeichen `>` können weggelassen werden. Aus den Deklarationen eines literaten Skripts werden Deklarationen eines gewöhnlichen Skripts.

   (b) Wir erhalten folgende Ausgaben:

```
show (IV (2,5)) ->> "IV (2,5))"
show IV (2,5) ->> Nicht typbar, nicht compilierbar
show Leer ->> "Leer"
show Ungueltig ->> "Ungueltig"
show (Ungueltig) ->> "Ungueltig"
show (IV (2-3,2+3)) ->> "IV (-1,5))"
show (IV (2-3,(2+)3)) ->> "IV (-1,5)"
show (IV (2-3,(+2)3)) ->> "IV (-1,5)"
show (IV ((2-)3,2+3)) ->> "IV (-1,5)"
show (IV ((-2)3,2+3)) ->> Nicht typbar, nicht compilierbar
```

3. Die Aussage ist falsch; `type`-, `newtype`- und `data`-Deklarationen sind nicht gegeneinander austauschbar.

   Mit `type` werden Typsynonyme eingeführt, Alias-Namen für Typen, die bereits (in einem Programm) definiert sind. Originär neue, bisher nicht existierende Typen, können mit `type` nicht eingeführt werden.

   Mit `newtype` wird Werten eines existierendens Typs eine neue Identität verliehen, syntaktisch dadurch ausgedrückt, dass die Werte des bisherigen Typs zu Argumenten des einen (!) Datenwertkonstruktors des Neuen Typs werden. Insofern ist die Möglichkeit, neue, bisher nicht existierende Typen einzuführen, mit `newtype` limitiert.

   Ausschließlich `data`-Deklarationen bieten die Möglichkeit, neue Datentypen mit beliebig strukturierten Werten einzuführen. Insbesondere kann jede `newtype`-Deklaration durch eine `data`-Deklaration ersetzt werden, aber nicht umgekehrt. Allerdings ist damit ein Performanzverlust verbunden, da anders als bei `newtype`-Deklarationen bei `data`-Deklarationen die Datenwertkonstruktoren nicht nur Übersetzungszeit, sondern auch zur Laufzeit einen gewissen Berechnungsaufwand verursachen.

   Anders als `type`-Deklarationen gewähren `newtype`- und `data`-Deklarationen Typsicherheit. In beiden Fällen sind die Werte der damit eingeführten Typen typsicher, geschützt durch programmweit eindeutig benannte Datenwertkonstruktoren.

4. (a) Der Name `hoch` wird im Rumpf als Infixoperator verwendet. Diese Nichtstandardverwendung muss durch Hochkommata angezeigt werden, da die Standardverwendung von `hoch` als Präfixoperator wäre.

   (b) Konzeptuell terminiert die Auswertung von `hoch` nicht, wenn es mit einem negativen Wert für `n` aufgerufen wird; in der Praxis terminiert `hoch` in diesen Fällen irregulär, wenn der gesamte (endliche) Speicher der Maschine aufgebraucht ist.

   (c) Die Funktion `hoch` ist linear rekursiv.

   (d) Der Ausdruck `hoch 5` ist syntaktisch korrekt und hat funktionalen Typ:

```
(hoch 5) :: Nat0 -> Nat1
```

   Sein Wert ist die (math.) Funktion *fuenf_hoch*:

$$fuenf\_hoch : IN_0 \to IN_1$$

$$\forall\, n \in IN_0.\ fuenf\_hoch(n) =_{df} 5^n$$

(e) Folgende Klammerung ist minimal, d.h. weitere Klammern können nicht weggelassen werden, ohne die Bedeutung zu verändern oder syntaktische Korrektheit zu verlieren:

```
hoch :: Nat0 -> Nat0 -> Nat1
m 'hoch' n
 | n == 0 = 1
 | n > 0  = m * m 'hoch' (n-1)
```

5. ***Bitte Teilnehmeranzahl am Übungsgruppentermin notieren!***

Selbsteinschätzungstest 3

KW 47 (16.-20.11.2020)

Stoff: Vortragsteile I, II, III und IV

Dauer: 10min. (ohne Abgabe, ohne Beurteilung)

1. Was ist mit der Aussage, Funktionen seien in funktionalen Programmiersprachen *first-class citizens*, gemeint?
2. Was unterscheidet echte und unechte Polymorphie auf Funktionen voneinander?
3. Welche Synonyme gibt es für echte und unechte Polymorphie?
4. Wozu dienen Typklassen in Haskell?
5. Was ist ein Funktional?
6. Ist die Addition `(+)` in Haskell eine Funktion erster Ordnung oder eine Funktion höherer Ordnung? Begründen Sie Ihre Antwort.
7. Was unterscheidet nach Konrad Hinsens Artikel *"The Promises of Functional Programming"* Funktionen im mathematischen Sinn von Funktionen im Sinn imperativer oder objektorientierter Programmiersprachen?
8. Definieren Sie als algebraischen Datentyp einen Typ Binärbaum, wobei Blätter eines Binärbaums einen Wert, Verzweigungen zwei Werte tragen sollen. Die Typen der Werte in Blättern und Verzweigungen sollen paarweise verschieden sein können.
9. Die Funktion `map` ist wie alle Funktionen in Haskell einstellig. Streichen Sie das heraus, in dem Sie den Signaturausdruck von `map` vollständig, aber nicht überfüssig klammern:

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

10. Etwas salopp darf man `map` auch zweistellig ansehen. Man nennt dies die curryfizierte und die uncurryfizierte Lesart der Signatur von `map`. Wie lauten diese Lesarten?
    (a) Curryfizierte Lesart: `map` ist eine 1-stellige Funktion, die ... abbildet auf ...
    (b) Uncurryfizierte Lesart: `map` ist eine 2-stellige Funktion, die ... abbildet auf ...

Lösungsvorschlag zu

Selbsteinschätzungstest 3

KW 47 (16.-20.11.2020)

Stoff: Vortragsteile I, II, III und IV

Dauer: 10min. (ohne Abgabe, ohne Beurteilung)

1. Mit der Aussage, Funktionen seien *first-class citizens*, ist gemeint, dass grundsätzlich (d.h. von wenigen Ausnahmen abgesehen) überall dort, wo der Wert eines nichtfunktionalen Typs stehen darf, auch der Wert eines funktionalen Typs stehen darf, oder kürzer und salopper: Überall dort, wo ein elementares Datum stehen darf, darf auch eine Funktion stehen. Anders ausgedrückt: Funktionen sind Daten; in gleicher Weise, wie Werte elementarer, nicht funktionaler Typen Daten sind. Funktionen können deshalb insbesondere Argument und Resultat von Funktionen sein.

2. Für echt poplymorphe Funktionen gibt es genau eine Implementierung. Diese Implementierung 'funktioniert' für alle konkreten Typen, die für die Typvariablen in der Signatur eingesetzt werden. Für unecht polymorphe Funktionen gibt es für jeden Typ, auf dessen Werten die Funktion arbeiten können soll, eine eigene typspezifische Implementierung. Diese typspezifische Implementierung erfolgt durch den Programmierer bei der Instanzbildung eines Typs für die Typklasse, die diese unecht polymorphe Funktion enthält.

3. Synonym zu echter Polymorphie ist *parametrische Polymorphie*; Synonyme zu unechter Polymorphie sind *ad hoc Polymorphie* und *Überladung*.

4. Typklassen sind das Sprachmittel von Haskell, Funktionen (genauer: Operator- und Relatorsymbole) zu überladen und typspezifisch zu implementieren. Ist z.B. ein Typ `A` Element der Typklasse `Num`, so kann sich der Programmierer darauf verlassen, dass auf `A`-Werten alle in `Num` aufgeführten Funktionen (u.a. `(+)`, `(-)`, `(*)`,...) definiert sind, aber auch alle Funktionen, die in Typklassen aufgeführt sind, von denen die Typklasse `Num` erbt, nämlich `Eq` und `Show`. Der Programmier kann sich also auch darauf verlassen, dass `A`-Werte durch Aufruf von `show` auf dem Bildschirm ausgegeben werden können, oder `A`-Werte mithilfe der Relatoren `(==)` und `(/=)` auf Gleichheit und Ungleichheit verglichen werden können.

5. Ein Funktional oder synonym eine *Funktion höherer Ordnung* ist eine Funktion, die Funktionen als Argument erwartet oder/und als Resultat abliefert.

6. Der Typ des Operationssymbols `(+)` ist in der vordefinierten Typklasse `Ord` curryfiziert vereinbart als eine 1-stellige Funktion auf `a`-Werten mit der Menge der 1-stelligen Funktionen auf `a`-Werten als Bildbereich: `(+) :: Num a => a -> a -> a`. Daher bezeichnet das Operationssymbol `(+)` eine Funktion höherer Ordnung, ein Funktional; entsprechend ist `(+)` mit der Bedeutung Addition auf Zahlbereichen eine Funktion höherer Ordnung.

7. Funktionen im mathematischen Sinn sind rechtseindeutige Relationen zwischen Argument- und Bildbereich: Ist eine Funktion für ein Argument definiert, gibt es genau einen Bildwert, einen Funktionswert für dieses Argument. Im programmiersprachlichen Jargon sind Funktionen im mathematischen Sinn seiteneffektfrei. Für Funktionen einer imperativen oder objektorientierten Programmiersprache gilt das nicht. Funktionen einer imperativen oder objektorientierten Programmiersprache sind Subroutinen, die einen Wert zurückliefern und dabei Seiteneffekte bewirken können. Sie können also über das Liefern eines Werts hinaus etwas tun und bewirken (vgl. Artikel von Konrad Hinsen, S. 87, linke Spalte, Anfang von Abschnitt "Basics"). Angewendet auf dasselbe Argument liefern sie deshalb i.a. nicht dasselbe Resultat. Für Funktionen im mathematischen Sinn ist das nicht möglich.

8. Folgende mehrfach polymorphe algebraische Datentypdeklaration leistet das Geforderte:

```
data BB a b c = Blatt a
              | Verzweigung (BB a b c) b c (BB a b c)
```

9. Vollständig, aber nicht überfüssig geklammerter Signaturausdruck:

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

10. Die beiden Lesarten der Signatur von `map`:

    (a) Curryfizierte Lesart: `map` ist eine 1-stellige Funktion, die Funktionen, die `a`-Werte auf `b`-Werte abbilden, auf Funktionen abbildet, die Listen von `a`-Werten auf Listen von `b`-Werten abbilden.

    (b) Uncurryfizierte Lesart: `map` ist eine 2-stellige Funktion, die Funktionen, die `a`-Werte auf `b`-Werte abbilden, und Listen von `a`-Werten auf Listen von `b`-Werten abbildet.

11. ***Bitte Teilnehmeranzahl am Übungsgruppentermin notieren!***

Selbsteinschätzungstest 4

KW 48 (23.-27.11.2020)

Stoff: Vortragsteile I, II, III und IV

Dauer: 10min. (ohne Abgabe, ohne Beurteilung)

1. Welchen Typ hat `filter`? Leiten Sie aus der Implementierung von `filter` den allgemeinstmöglichen Typ (d.h. so polymorph wie möglich) für `filter` her:

```
filter p [] = []
filter p (l:ls)
 | p l  = l : filter p ls
 | True = filter p ls
```

2. Was ist funktionale Abstraktion
   (a) 1. Stufe?
   (b) höherer Stufe?
3. Was versteht man unter λ-Lifting?
4. Schreiben Sie eine rekursive Haskell-Rechenvorschrift `sum_kubik`, die die Funktion f berechnet:

$$f : IN_1 \to IN_1$$

$$\forall\, n \in IN_1.\ f(n) =_{df} \Sigma_{k=1}^{n} k^3$$

5. Schreiben Sie eine rekursive Haskell-Rechenvorschrift `streiche` mit der Signatur:

```
streiche :: String -> Char -> Char -> String
```

   Angewendet auf eine Zeichenreihe `s` und zwei Zeichen `'c'` und `'d'` streicht `streiche` von links nach rechts alle Vorkommen der Teilzeichenreihe `['c']++['d']` in `s`.
6. Geben Sie die Aufrufgraphen der Rechenvorschriften aus den Aufgaben 1, 4 und 5 an.
7. Von welchem Rekursionstyp sind die Funktion `filter` aus Aufgabe 1 und Ihre Funktionen `sum_kubib` und `streiche` aus Aufgabe 4 und 5?
8. Implementieren Sie das Funktional `map :: (a -> b) -> [a] -> [b]`
   (a) rekursiv
   (b) mit einer Listenkomprehension
   (c) argumentfrei mit einer anonymen λ-Abstraktion.

Bemerkung: Das Symbol $=_{df}$ steht für 'definitionsgemäß gleich'.

Lösungsvorschlag zu

Selbsteinschätzungstest 4

KW 48 (23.-27.11.2020)

Stoff: Vortragsteile I, II, III und IV

Dauer: 10min. (ohne Abgabe, ohne Beurteilung)

1. Die Signatur des Funktionals `filter` ist wie folgt:

```
filter :: (a -> Bool) -> [a] -> [a]
filter p [] = []
filter p (l:ls)
 | p l  = l : filter p ls
 | True = filter p ls
```

2. Funktionale Abstraktion

   (a) 1. Stufe bezeichnet informell das Ersetzen eines Ausdrucks (oder einer Menge strukturell gleicher Ausdrücke) durch eine Funktion und entsprechende Funktionsaufrufe. Der Ausdruck wird dabei zum Rumpf der Funktion, die Operanden des Ausdrucks zu Parametern der Funktion. Die Ausdrucksstruktur, d.h. die Verknüpfungsvorschrift der Operanden des Ausdrucks, wird dadurch wiederverwendbar, indem die Funktion für verschiedene Argumente aufgerufen werden kann. Beispiel:

```
Ausdruecke: 7*2+4*(5-2), 8*3+2*(7-3), usw.
Funktion:   f :: Int -> Int -> Int -> Int -> Int
            f a b c d = a*b+c*(d-b)
Aufrufe:    f 7 2 4 5 ->> 7*2+4*(5-2) ->>...,
            f 8 3 2 7 ->> 8*3+2*(7-3) ->>..., usw.
```

   (b) höherer Stufe bezeichnet zusätzlich zum Herausziehen der Argumente auch das Herausziehen der Struktur möglicher Ausdrücke über diesen Argumenten, d.h. möglicher Verknüpfungsvorschriften der Argumente. Die Funktion wird so zu einer Funktion höherer Ordnung. Beispiel:

```
Ausdruecke: 7*2+7*(5-2), 8-3*8+2*(3-8), usw.
Funktion:   f :: (Int -> Int -> Int -> Int) -> Int -> Int -> Int -> Int
            f g = g
Aufrufe:    f h 7 2 5 ->> h 7 2 5 ->> 7*2+7*(5-2) ->>...,
            f k 8 3 2 ->> k 8 3 2 ->> 8-3*8+2*(3-8) ->>..., usw.
            -- mit z.B.:
            h a b c = a*b+a*(c-b)
            k a b c = a-b*a+c*(b-a)
```

3. Als λ-Lifting bezeichnet man die Konstruktion einer konstanten Funktion zu einem gegebenen Wert w, die diesen Wert als einzigen Funktionswert hat (und deshalb eine konstant(wertige) Funktion ist. Ein Beispiel:

```
w = 42 :: Int
konstant_wertige_Funktion_42 :: a -> Int
konstant_wertige_Funktion_42 = \_ -> w
```

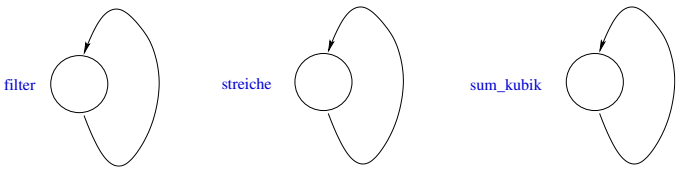
4. Die Funktion `sum_kubik` kann wie folgt implementiert werden:

```
type Nat1 = Int
sum_kubik :: Nat1 -> Nat1
sum_kubik 1 = 1
sum_kubik n = n^3 + sum_kubik (n-1)
```

5. Die Funktion `streiche` kann wie folgt implementiert werden:

```
streiche :: String -> Char -> Char -> String
streiche [] _ _ = []
streiche (x:[]) _ _ = [x]
streiche (x:y:zs) c d
 | (x==c && y==d) = streiche zs c d
 | True           = x : (streiche (y:zs) c d)
```

6. Die Aufrufgraphen der Rechenvorschriften aus den Aufgaben 1, 4 und 5 sind strukturident:



   Die Aufrufgraphen von `map`, `streiche` und `sum_kubik` unterscheiden sich also in der Struktur nicht. Das zeigt, dass Aufrufgraphen eine Projektion oder Abstraktion der im Programmtext enthaltenen Infomationen darstellen. Auf der Ebene des Programmtexts unterscheidbare Funktionen wie `map`, `streiche` und `sum_kubik` werden auf strukturell gleiche Graphen abgebildet und sind auf dieser Ebene nicht mehr unterscheidbar.

7. Die Funktionen `filter`, `sum_kubik` und `streiche` sind linear rekursiv.

8. Das Funktional `map` kann folgendermaßen implementiert werden:

   (a)
   ```
   map :: (a -> b) -> [a] -> [b]
   map f []     = []
   map f (x:xs) = (f x) : map f xs
   ```

   (b)
   ```
   map :: (a -> b) -> [a] -> [b]
   map f xs = [f x | x <- xs]
   ```

   (c)
   ```
   map :: (a -> b) -> [a] -> [b]
   map = \f xs -> [f x | x <- xs]         -- uncurryfiziert

   map :: (a -> b) -> [a] -> [b]
   map = \f -> (\xs -> [f x | x <- xs])   -- curryfiziert
   ```

9. ***Bitte Teilnehmeranzahl am Übungsgruppentermin notieren!***

Selbsteinschätzungstest 5

KW 49 (30.11.-04.12.2020)

Stoff: Vortragsteile I, II, III, IV und V

Dauer: 10min. (ohne Abgabe, ohne Beurteilung)

1. Gegeben sind Deklarationen von zwei Funktionen und ein Ausdruck über diesen Funktionen:

```
type Nat0 = Integer
type Nat1 = Integer

fac :: Nat0 -> Nat1
fac n = if n == 0 then 1 else n * fac (n-1)

potenz :: Nat0 -> Nat0 -> Nat1
potenz m n = if n == 0 then 1 else m * potenz m (n-1)

ausdruck =
 42 * (potenz (2^3) (fac (17+4))) + 21 - (potenz 47 11) + (fac (potenz (2+3) (fac (21*2))))
```

   Werten wir den Ausdruck `ausdruck` aus, sieht er nach dem ersten Auswertungsschritt folgendermaßen aus:

```
ausdruck ->>
 42 * (potenz (2^3) (fac (17+4))) + 21 - (potenz 47 11) + (fac (potenz (2+3) (fac (21*2))))
        ->> ...
```

   Wie sieht dieser Ausdruck nach dem nächsten und übernächsten Auswertungschritt aus, wenn beide Schritte gemäß

   (a) normaler Auswertungsordnung an linkest-möglicher Stelle erfolgen?

   (b) applikativer Auswertungsordnung an rechtest-möglicher Stelle erfolgen?

2. Manche funktionale Sprachen sind nach dem Prinzip *fleißiger Auswertung* (engl. *eager evaluation*) implementiert, andere nach dem Prinzip *fauler Auswertung* (engl. *lazy evaluation*).

   Welche Gründe könnnen Protagonisten dieser Sprachen für die jeweilige Implementierungsentscheidung anführen? Anders ausgedrückt: Was sind Vorteile von fleißiger, was von fauler Auswertung? Was sind mögliche Nachteile?

3. Welche Reduktionsregeln gibt es im reinen λ-Kalkül? Was ist ihre informelle Bedeutung? Wozu dienen sie?

4. Was versteht man unter einer punktfreien (oder: argumentlosen) Funktionsdeklaration? Geben Sie eine punktfreie und eine nichtpunktfreie Haskell-Deklaration der Fibonacci-Funktion an.

5. Für einige Typklassen können in Haskell Instanzbildungen automatisch vorgenommen werden. Welche sind das? Was geschieht bei der automatischen Instanzbildung? Warum ist die automatische Instanzbildung nicht für alle Typklassen möglich?

6. Gilt folgende Gleichheit für alle natürlichen Zahlen $n \in IN_1$?

$$\sum_{k=1}^{n} k^3 = (\sum_{k=1}^{n} k)^2$$

   Schreiben Sie eine Haskell-Rechenvorschrift `test`, die angewendet auf eine natürliche Zahl als Argument den Wert `True` ausgibt, wenn die Gleichung für diese Zahl erfüllt ist, sonst `False`.

Lösungsvorschlag zu

Selbsteinschätzungstest 5

KW 49 (30.11.-04.12.2020)

Stoff: Vortragsteile I, II, III, IV und V

Dauer: 10min. (ohne Abgabe, ohne Beurteilung)

1. Nach den nächsten beiden Auswertungsschritten sieht der Ausdruck jeweils folgendermaßen aus:

   (a) Bei normaler Auswertungsordnung an linkest-möglicher Stelle:

   ```
   ausdruck ->>
   42 * (potenz (2^3) (fac (17+4))) + 21 - (potenz 47 11) + (fac (potenz (2+3) (fac (21*2))))
   ->> 42 * (if (fac (17+4)) == 0 then 1 else (2^3) * (potenz (2^3) ((fac (17+4))-1))) + 21...
   ->> 42 * (if (if (17+4) == 0 then 1 else (17+4) * fac ((17+4)-1)) == 0
                then 1 else (2^3) * (potenz (2^3) ((fac (17+4))-1))) + 21...
   ->> ...
   ```

   (b) Bei applikativer Auswertungsordnung an rechtest-möglicher Stelle:

   ```
   ausdruck ->>
   42 * (potenz (2^3) (fac (17+4))) + 21 - (potenz 47 11) + (fac (potenz (2+3) (fac (21*2))))
   ->> ... + (fac (potenz (2+3) (fac 42)))
   ->> ... + (fac (potenz (2+3) (if 42 == 0 then 1 else 42 * fac (42 -1))))
   ->> ...
   ```

2. Vorteile fleißiger Auswertung sind eine einfachere Implementierung und oft eine schnellere Terminierung; Funktionsargumente werden genau einmal ausgewertet. Nachteilig ist, dass auch Funktionsargumente ausgewertet werden, die im Funktionsrumpf nicht verwendet werden; solche Argumente werden unnötig ausgewertet. Aus mathematischer Perspektive erscheint fleißige Auswertung oft natürlicher, weil die Undefiniertheit eines Funktionsarguments die Undefiniertheit des Funktionswerts zur Folge hat (Striktheit!), was bei fauler Auswertung i.a. nicht so ist.

   Vorteil fauler Auswertung ist, dass die Auswertung so häufig wie nur irgend möglich terminiert. Ausdrücke werden nur ausgewertet, wenn ihr Wert zum Wert des Gesamtausdrucks beiträgt, und dann nur genau einmal. Nachteil fauler Auswertung ist eine technisch anspruchsvollere Implementierung, um das Teilen gemeinsamer Ausdrücke sicherzustellen, um das unnötige Mehrfachauswerten von Ausdrücken zu vermeiden. Faule Auswertung vereint damit die Vorteile applikativer Auswertung (Effizienz) und normaler Auswertung (Terminierungshäufigkeit).

3. Im reinen λ-Kalkül gibt es drei Reduktionsregeln, die α-, β- und η-Reduktionsregeln. Informelle Bedeutung bzw. Zweck der α-Reduktion ist die gebundene Umbennenung von Funktionsparametern, um Bindungsfehler bei naiver Anwendung der β-Reduktionsregel zu vermeiden; Bedeutung der β-Reduktion ist die Funktionsanwendung und der η-Reduktion die Elimination unnötiger Funktionen.

4. Eine punktfreie Funktionsdeklaration verzichtet auf die Argumente bei der Definition des Funktionsrumpfes, eine nichtpunktfreie nennt die Argumente ausdrücklich. Beispiel anhand der Fibonacci-Funktion:

   (a) Nichtpunktfreie (oder: argumentbehaftete) Deklaration von `fib`:

   ```
   type Nat0 = Integer
   fib :: Nat0 -> Nat0
   fib n = if n <= 1 then n else fib (n-2) + fib (n-1)
   ```

   (b) Punktfreie (oder: argumentlose) Deklaration von `fib` (mithilfe (i) einer anonymen λ-Abstraktion, (ii) von `fib` und (iii) von `fib`$'$):

   ```
   fib' :: Integer -> Integer
   fib' = \n -> if n <= 1 then n else fib' (n-2) + fib' (n-1)     -- (i)

   fib'' :: Integer -> Integer
   ```

```
fib'' = fib                                                    -- (ii)

fib''' :: Integer -> Integer
fib''' = fib'                                                  -- (iii)
```

5. Automatische Instanzbildungen können ausschließlich für die vordefnierten Typklassen `Eq`, `Ord`, `Enum`, `Bound`, `Read` und `Show` vorgenommen werden. Die automatische Instanzbildung nimmt dabei die manuelle Instanzbildung in 'naheliegendster' Weise vor. Das erfordert ein Vorwissen der 'naheliegendsten' Bedeutung der in einer Klasse vorgesehenen Funktionen. Dieses Vorwissen ist nicht für alle Typklassen für den Compiler ersichtlich oder erschließbar, insbesondere nicht für selbstdefinierte Typklassen. Deshalb ist automatische Instanzbildung nicht für alle Typklassen möglich.

6. Zum Test auf Gleichheit:

```
type Nat1 = Int

-- A) sum und sum_kubik unmittelbar implementiert
sum :: Nat1 -> Nat1
sum 1 = 1
sum n = n + sum (n-1)

sum_kubik :: Nat1 -> Nat1
sum_kubik 1 = 1
sum_kubik n = n^3 + sum_kubik (n-1)

test :: Nat1 -> Bool
test n = sum_kubik n == (sum n)^2

-- B) sum und sum_kubik mithilfe einer Funktion hoeherer Ordnung implementiert
type Basiswert    = Nat1
type Verknuepfung = (Nat1 -> Nat1 -> Nat1)

fho_rek :: Basiswert -> Verknuepfung -> Nat1 -> Nat1
fho_rek basiswert verknuepfe n
 | n == 1 = basiswert
 | n > 1  = verknuepfe n (fho_rek basiswert verknuepfe (n-1))

sum' :: Nat1 -> Nat1
sum' = fho_rek 1 (+)

sum_kubik' :: Nat1 -> Nat1
sum_kubik' = fho_rek 1 (\n m -> n^3 + m)

test' :: Nat1 -> Bool
test' n = sum_kubik' n == (sum' n)^2

-- C) Punktfrei mit Operatorabschnitt (square) und Funktionskomposition (test''):
square :: Nat1 -> Na1
square = (^2)

test'' = sum_kubik'  == (square . sum')
```

7. ***Bitte Teilnehmeranzahl am Übungsgruppentermin notieren!***

Selbsteinschätzungstest 6

KW 50 (07.-11.12.2020)

Stoff: Vortragsteile I, II, III, IV, V und VI

Dauer: 10min. (ohne Abgabe, ohne Beurteilung)

1. Was besagen die Church-Rosser-Theoreme informell? Warum und wofür sind sie wichtig?
2. Was ist ein Bindungsfehler naiv angewandeter syntaktischer Substitution? Geben Sie ein Beispiel an, das Ihre Antwort illustriert.
3. Was sind α-, β-, η-Konversion? Was β- und η-Reduktion?
4. Zur Normalform von λ-Ausdrücken:
   (a) Was ist die Normalform eines λ-Ausdrucks?
   (b) Wozu dienen Normalformen im λ-Kalkül?
   (c) Hat jeder λ-Ausdruck eine Normalform?
5. Welchen Typ hat das Funktional "Falten von links"? Rekonstruieren Sie den Typ allgemeinstmöglich (d.h. so polymorph wie möglich) aus der unten angegebenen Implementierung von `foldl`:

   ```
   foldl f e [] = e
   foldl f e (l:ls) = foldl f (f e l) ls
   ```

6. Welche Vorteile führen Konstantin Läufer und George K. Thiruvathukal in ihrem Artikel *"The Promises of Typed, Pure, and Lazy Functional Programming: Part II"* für statische gegenüber dynamischer Typisierung von Programmiersprachen an?
7. Warum stellt Ein- und Ausgabe eine Herausforderung für rein funktionale Programmiersprachen dar? Wie wird diese Herausforderung in Haskell gelöst?
8. Listen in Haskell sind homogen, d.h. die Elemente von Listen sind alle von ein und demselben Typ. Konstantin Läufer und George K. Thiruvathukal zeigen in ihrem Artikel *"The Promises of Typed, Pure, and Lazy Functional Programming: Part II"*, wie man in Haskell dennoch quasi heterogene Listen implementieren kann. Wie gehen sie dafür vor?
9. Was ist die Idee reflektiver Programmierung bzw. eines reflektiven Programmierstils?
10. Zur Fehlerbehandlung:
    (a) Was versteht man unter Panikmodus bei der Fehlerbehandlung?
    (b) Welche Vor- und Nachteile sind damit verbunden?
    (c) Was sind bessere Möglichkeiten zur Fehlerbehandlung?

Lösungsvorschlag zu

Selbsteinschätzungstest 6

KW 50 (07.-11.12.2020)

Stoff: Vortragsteile I, II, III, IV, V und VI

Dauer: 10min. (ohne Abgabe, ohne Beurteilung)

1. Zu den Aussagen und der Bedeutung der Church/Rosser-Theoreme:

   (a) Das erste Church/Rosser-Theorem besagt informell: Wenn die Normalform eines λ-Ausdrucks existiert, ist sie bis auf gebundenes Umbenennen von Namen eindeutig bestimmt (s. Theorem 12.3.4.1, Konfluenz-, Diamant-, Rauteneigenschaft).

   (b) Das zweite Church/Rosser-Theorem besagt informell: Wenn irgendeine Reduktionsfolge zur Normalform eines λ-Ausdruck führt, dann auch die normale Reduktionsordnung. Anders ausgedrückt: Die Auswertung gemäß normaler Reduktionsordnung terminiert am häufigsten (s. Theorem 12.3.4.2, Standardisierung).

   (c) Die Church/Rosser-Theorem machen die Semantikdefinition für λ-Ausdrücke in Form ihrer Normalform sinnvoll, indem sie garantieren: Wenn die Normalform eines λ-Ausdrucks existiert, ist sie bis auf irrelevantes Umbenennen von Namen eindeutig bestimmt und kann mittels normaler Reduktion berechnet werden. Damit ist λ-Ausdrücken in Form ihrer Normalform in eindeutiger Weise eine Bedeutung gegeben.

2. Naiv angewendet, d.h. ohne geeignete Umbenennung von definierenden und der an sie gebundenen angewandten Namen in λ-Abstraktionen, kann die syntaktische Substitution dazu führen, dass vormals freie Variablen im einzusetzenden Ausdruck im Rumpf der λ-Abstraktion eingefangen und gebunden werden:

   $$(\lambda\, x.\ x\ y)\ [x/y] \overset{naiv}{\longrightarrow} (\lambda\, x.\ x\ x)$$

   Das vormals freie Vorkommen von x, das durch die Substitution für y im Rumpf der λ-Abstraktion einzusetzen ist, ist nach naiver Einsetzung ohne Umbenennung eingefangen, da es nach Einsetzen an das definierende Vorkommen von x der λ-Abstraktion gebunden ist.

   Die in der Definition der β-Reduktion festgelegte Umbenennung des definierenden Vorkommens von x stellt sicher, dass dieses Einfangen von x nicht passiert:

   $$(\lambda\, x.\ x\ y)\ [x/y] \longrightarrow (\lambda\, z.\ x\ y)\ [z/x])\ [x/y] \longrightarrow (\lambda\, z.\ z\ y)[x/y] \longrightarrow \lambda\, z.\ z\ x$$

3. α-, β- und η-Konversion bezeichnen 3 Regeln zur syntaktischen Transformation von λ-Ausdrücken. Wendet man die β- und η-Konversionsregeln von links nach rechts an, werden λ-Ausdrücke dadurch vereinfacht, sie werden 'einfacher', 'kürzer'. Man spricht deshalb von der β- und η-Konversionsregel als Reduktionsregel, wenn man anzeigen möchte, diese beiden Regeln ausschließlich von links nach rechts anzuwenden.

4. Zur Normalform von λ-Ausdrücken:

   (a) Ein λ-Ausdruck liegt in Normalform vor, wenn keine β- und η-Reduktionen mehr auf ihn angewendet werden können. λ-Ausdrücke in Normalform sind deshalb bis auf gebundenes Umbenennen von Variablen durch α-Konversionen eindeutig bestimmt.

   (b) λ-Ausdrücke in Normalform dienen dazu, die Semantik, die Bedeutung von λ-Ausdrücken zu definieren: Die Bedeutung eines λ-Ausdrucks ist seine bis auf α-Konversionen eindeutig bestimmte Normalform, wenn sie existiert. Diese Festlegung ist sinnvoll aufgrund der Aussagen der Church/Rosser-Theoreme.

   (c) Nein, nicht jeder λ-Ausdruck hat eine Normalform. Es gibt λ-Ausdrücke, die eine unendliche Folge von z.B. β-Reduktionen induzieren:

   $$\lambda\, x.(xx)\lambda\, x.(xx) \overset{\beta}{\to} \lambda\, x.(xx)\lambda\, x.(xx) \overset{\beta}{\to} \ldots$$

   Die Bedeutung von λ-Ausdrücken ohne Normalform ist undefiniert.

5. Die Signatur des Funktionals “Falten von links” ist wie folgt:

```
foldl :: (a -> b -> a) -> a -> [b] -> a
foldl f e [] = e
foldl f e (l:ls) = foldl f (f e l) ls
```

6. Zur Laufzeit eines Programm einer statisch getypten Programmiersprache sind keinerlei Fehler oder Programmabstürze aufgrund inkompatibler Typung von Ausdrücken, Variablen, etc. möglich. Eine sehr große Gruppe von Fehlermöglichkeiten kann deshalb bereits zur Übersetzungszeit eines Programms mit Sicherheit ausgeschlossen werden. Zur Laufzeit eines Programms sind deshalb auch keine Typüberprüfungen mehr nötig, wie das für Programme dynamisch getypter Programmiersprachen erforderlich ist. Die Programmausführung statisch typgeprüfter Programme ist dadurch performanter (S. 70, linke Spalte, Mitte). Im Unterschied dazu können in Programmen dynamisch getypter Programmiersprachen, Typfehler zur Laufzeit auftreten und zu Programmabbrüchen führen. Im Fall sicherheitskritischer Anwendungen (Flugsicherungssoftware, Kraftwerkssteuerungen, etc.) ist das fatal. In der Praxis führt dies zu der Praxis, den Binärcode installierter Programme zur Fehlerbeseitigung direkt zu verändern (was seinerseits schwierig und fehlerträchtig ist). In Programmen statisch getypter Programmiersprachen können solche Fehler schon zur Übersetzungszeit der Programme aufgedeckt und vor Auslieferung der Programme berichtigt werden (S. 70, linke Spalte, oben). Ein oft ins Feld geführter Einwand, dass der Programmierer höheren Aufwand hat, indem er stets Typen deklarieren muss, ist für viele Sprachen, darunter auch Haskell, entkräftet, da Typen automatisch inferiert werden können und deshalb nicht notwendig vom Programmierer angegeben werden müssen (S. 70, linke Spalte, unten; S. 70, mittlere Spalte, unten, und fortfolgende).

7. Rein funktionale Programmiersprachen sind zu 100% frei von Seiteneffekten und damit *referentiell transparent*. Ein-/Ausgabeoperationen sind inhärent seiteneffektbehaftet. Ihre Aufnahme durchbricht damit die Eigenschaft referentieller Transparenz. In Haskell wird dies dadurch gelöst, dass Werte von Ein-/Ausgabeoperationen durch unterschiedliche Typisierung strikt von Werten reiner funktionaler Typen getrennt werden. Informell kann man sich das als eine Ein-/Ausgabeschale vorstellen, die um den rein funktionalen Kern von Haskell gelegt wird. Werte können aus dem Kern in die Schale hinaus und umgekehrt nur in automatisch sichergestellter Form erfolgen, so dass für den rein funktionalen Kern referentielle Transparenz und damit die Vorteile rein funktionaler Programmierung erhalten bleiben.

8. Konstantin und Läufer schlagen vor, Listen über einem algebraischen Summentyp zu definieren, um quasi heterogene Listen zu definieren. Sie illustrieren ihren Vorschlag an folgendem Beispiel (s.S. 73, linke Spalte, unten):

```
data IntOrString = AnInt Int
                 | AString String

add [] = (0,"")
add (AnInt i : xs)   = let (k,t) = add xs in (i + k, t)
add (AString s : xs) = let (k,t) = add xs in (k, s ++ t)

Prelude> add [AnInt 3, AString "adsf", AnInt 7, AString "quer"]
 (10,"adsfqwer")
```

   Für ‘zweitypige’ Listen würde der vordefinierte polymorphe Typ `Either a b` eine andere Möglichkeit bieten, den Typ `IntOrString` zu definieren:

```
type IntOrString = Either Int String

add [] = (0,"")
add (Left i : xs)  = let (k,t) = add xs in (i + k, t)
add (Right s : xs) = let (k,t) = add xs in (k, s ++ t)

Prelude> add [Left 3, Right "adsf", Left 7, Right "quer"]
 (10,"adsfqwer")
```

9. Unter *reflektiver Programmierung* versteht man die systematische Einflechtung reflektiver Feedback-Schleifen im Programmentwicklungsprozess. Dafür denkt man sich den Entwicklungsprozess in vier große Phasen aufgeteilt, innnerhalb dessen Rücksprünge in frühere Phasen möglich sind:

   (a) Problem verstehen.

   (b) Problemlösung konzipieren.

   (c) Problemlösung implementieren.

   (d) Implementierte Problemlösung testen und evaluieren.

   Für jede dieser Phasen gibt es typische Fragen. Von ihrer Beantwortung hängt es ab, ob mit der nächsten Phase fortgefahren wird oder in die oder eine frühere Phase zurückgesprungen wird. Dieses kontinuierliche Nachdenken und Hinterfragen des aktuellen Ergebnisses des bisherigen Entwicklungsprozesses ist namensgebend für dieses Vorgehen: Reflektive Programmierung.

10. Unter Fehlerbehandlung im Panikmodus versteht man den sofortigen und endgültigen Programmabbruch im Fehlerfall, in Haskell z.B. durch Aufruf der polymorphen Funktion `error` mit Signatur `error :: String -> a`.

    (a) *Vorteile des Panikmodus:* Einfach zu realisieren.

    (b) *Nachteile des Panikmodus:* Der Programmabbruch ist endgültig. Bisher berechnete Teilergebnisse sind verloren. Für sicherheitskritische Systeme können die Folgen solch eines Abbruchs fatal sein.

    (c) (i) Die Verwendung und Weiterrechnung mit vordefinierten Auffangwerten im Fehlerfall. Besonders vorteilhaft dafür sind Auffangwerte, die den Fehler zu erkennen und darauf zu reagieren erlauben. (ii) Die Verwendung spezieller Fehlertypen zur Erkennung, Weiterreichung und Verarbeitung von Fehlerwerten.

Selbsteinschätzungstest 7

KW 51 (14.-18.12.2020)

Stoff: Vortragsteile I, II, III, IV, V, VI und VII

Dauer: 10min. (ohne Abgabe, ohne Beurteilung)

1. Worin unterscheiden sich applikative und normale Auswertungsordnung operationell?
2. Typen, Typnamen:
   (a) Welche Konstrukte bietet Haskell für die Einführung von Typen und Typnamen?
   (b) Was leisten diese Konstrukte? Wozu dienen sie?
   (c) Welche
      i. Möglichkeiten/Vorteile
      ii. Limitierungen/Nachteile

   weisen sie auf (auch im Vergleich zueinander)?
3. Funktionen sind häufig in Form von Fallunterscheidungen definiert. Welche (Haupt-) Möglichkeiten bietet Haskell an, diese Fallunterscheidungen zu definieren? Welche Vor- und Nachteile haben sie?
4. Was hat das *Generator/Selektor*-Prinzip mit *lazy* Evaluation zu tun?
5. Wie erklären Konstantin Läufer und George K. Thiruvathukal in ihrem Artikel *"The Promises of Typed, Pure, and Lazy Functional Programming: Part II"* den Begriff *referential transparency*?
6. Geben Sie einen λ-Ausdruck an, der keine Normalform besitzt.
7. Wie kann in Haskell eine applikativartige Auswertungsordnung erzwungen werden? Welchem Ziel dient das in den meisten Fällen?
8. Was versteht man unter selektivem Import und Export im Zusammenhang mit dem Modulkonzept von Haskell? Was bedeutet es, dass es keinen automatischen Reexport gibt?
9. Worauf ist bei der Definition kaskadenartig-rekursiver Funktionen aus Effizienzgründen besonders zu achten?
10. Welches sind die wissenschaftliche Flaggschiff-Konferenz und Flaggschiff-Zeitschrift für Themen rund um funktionale Programmierung?

Lösungsvorschlag zu

Selbsteinschätzungstest 7

KW 51 (14.-18.12.2020)

Stoff: Vortragsteile I, II, III, IV, V, VI und VII

Dauer: 10min. (ohne Abgabe, ohne Beurteilung)

1. Applikative und normale Auswertungsordnung unterscheiden sich operationell in der Art und Weise der Expansion von Funktionsaufrufen. Applikativ werden zunächst alle Argumente des Aufrufs vollständig ausgewertet, bevor der Funktionsaufruf expandiert wird. Normal wird der Funktionsaufruf sofort expandiert, die Argumente werden bei der Expansion unausgewertet im Rumpf für die Parameter der Funktion eingesetzt.

2. Typen, Typnamen:

   (a) `type`, `newtype`, `data`.

   (b) `type` erlaubt die Einführung von Typnamen, sog. (Typ-) Aliasen; `newtype` erlaubt existierenden Tupeltypen neue Identitäten zu geben; `data` erlaubt die Einführung neuer, bisher nicht existiererender Summentypen. Produkttypen (**ein** mehrstelliger (Daten-) Konstruktor) und Aufzählungstypen (ein oder mehrere nullstellige (Daten-) Konstruktoren) sind Spezialfälle von Summentypen (ein oder mehrere null-, ein- oder mehrstellige (Daten-) Konstruktoren.

   (c) Möglichkeiten/Vorteile und Limitierungen/Nachteile

      i. `type`: erlaubt einem bereits existierenden Typ einen aussagekräftigeren, 'sprechenderen' Namen zu geben (Bsp.: `type EUR = Int; type Wirtschaftswachstum_in_Prozent = Int`), einen Alias. Die Einführung und Verwendung gut gewählter Aliase macht ein Programm einfacher lesbar, einfacher verständlich. Gleichzeitig stehen alle Funktionen, die auf Werten des Grundtyps vorhanden sind, auch unter dem neuen Typnamen zur Verfügung (Grundtyp- und Aliasname bezeichen denselben Typ!). Zusätzliche Typsicherheit gewinnt man dadurch allerdings nicht. (Grund-) Typname und Alias dürfen sich an jeder Stelle wechselweise vertreten (der Vergleich oder die Addition von Euro- und Wirtschaftswachstumswerten ist semantisch sinnlos und von daher untergewollt, führt aber nicht zu einem Typfehler).

      ii. `data`: erlaubt neue, bisher nicht existierende Typen einzuführen, und die (Struktur der) Werte dieser Typen exakt zu beschreiben. Die so eingeführten Typen können konzeptuell in Summen-, Produkt- und Aufzählungstypen eingeteilt werden. Man spricht daher auch von algebraischen Datentypen. Produkt- und Summentypen ergeben sich dabei als Spezial- oder Randfälle (echter) Summentypen, s.a. Antwort zu voriger Teilaufgabe. Mit `data` eingeführte Typen sind typsicher; sie können sich nicht wechselweise vertreten. Alle Funktionen, die Werte eines algebraischen Datentyps manipulieren sollen, müssen selbstgeschrieben werden; dabei müssen die (Daten-) Konstruktoren der Typen sowohl zur Schreib- und Übersetzungszeit wie zur Laufzeit eines Programms beachtet werden; zur Laufzeit eines Programms entsteht dadurch Rechenaufwand (der bei mit `newtype` eingeführten Typen nicht entsteht).

      iii. `newtype`: erlaubt Tupeltypen (entsprechend Produkttypen mit **einem** Datenfeld) eine neue Identität zu verleihen. Das wäre in gleicher Weise mit einer `data`-Deklaration, einem algebraischen Datentyp möglich. Beides gewährleistet Typsicherheit: mittels `newtype` oder/und `data` eingeführte Typen können sich nicht wechselweise vertreten. Vorteil eines mit `newtype` gegenüber eines mit `data` eingeführten Produkttyps mit einem Datenfeld ist, dass der (Daten-) Konstruktor des Produkttyps nur zur Schreibzeit und zur Typprüfung während der Übersetzungszeit eines Programms erforderlich ist verwendet wird. Ist die Typprüfung zur Übersetzungszeit erfolgreich, kann der *genau eine* (Daten-) Konstruktor des stets einstelligen `newtype`-Produkttyps abgestriffen werden. Für `data`-Typen ist dies nicht möglich, weil es i.a. mehr als einen (Daten-) Konstruktor gibt. Der Preis dafür ist, dass Produkttypen mit mehr als einem Datenfeld nicht mit einer `newtype`-Deklaration eingeführt werden können. Dafür ist eine `data`-Deklaration erforderlich.

3. Hauptmöglichkeiten Funktionen in Form von Fallunterscheidungen zu definieren sind:

   (a) *Bedingte Ausdrücke* (`if bedingungsausdruck then ausdruck1 else ausdruck2`)

      i. *Vorteile:* Keine (außer scheinbarer Vertrautheit wegen oberflächlicher syntaktischer Ähnlichkeit mit Konstrukten aus anderen, insbesondere imperativen oder objektorientierten Programmiersprachen; hier ist diese oberflächliche Ähnlichkeit besonders tückisch, weil inhaltlich die Unterschiede groß sind: Fallunterscheidungs**ausdruck** funktional mit einem Wert als Bedeutung vs. Fallunterscheidungs**anweisung** imperativ und objektorientiert mit einer Zustandstransformation als Bedeutung).

      ii. *Nachteile:* Geschachtelte Fallunterscheidungen sind schwer zu lesen und verstehen.

   (b) *Bewachte Ausdrücke* (`| waechterausdruck = ausdruck`)

      i. *Vorteile:* Kurz und klar, wenig syntaktischer Overhead.

      ii. *Nachteile:* Keine.

   (c) *Musterbasierte* Ausdrücke (`(x:y:zs) = ausdruck`)

      i. *Vorteile:* Kurz und klar, wenig syntaktischer Overhead, praktisch vor allem bei strukturierten Argumenten, da Muster diese Struktur offenlegen und die umständliche Verwendung von Selektorfunktionen deshalb vermieden werden kann, um auf die Strukturelemente zuzugreifen.

      ii. *Nachteile:* Sehr änderungsaufwändig, wenn eine Funktion später um zusätzliche Parameter erweitert werden muss, da alle Musterverwendungsstellen angepasst werden müssen.

   (d) *`case`-Ausdrücke* (`case musterausdruck of wert -> ausdruck...`)

      i. *Vorteile:* Keine, auch mit musterbasierten Ausdrücken gemäß 3. erreichbar.

      ii. *Nachteile:* Notationell aufwändiger als musterbasierte Ausdrücke gemäß 3.

4. Das *Generator/Selektor*-Prinzip bedingt *lazy* Evaluation als Auswertungsordnung für Ausdrücke. Informell bewirkt *lazy* Evaluation eine Verzahnung der Auswertung von Generator- und Selektorausdruck; der Generatorausdruck wird nur so weit ausgewertet, bis der Selektorausdruck zuschlagen kann. Führt die Auswertung des Generatorausdrucks auf einen konzeptuell nicht endlichen Wert (z.B. Strom statt Liste, nicht endlicher Baum,...), würde die versuchte Vorwegauswertung des Generatorausdrucks im Stil von *eager* Evaluation nicht terminieren, der Selektorausdruck käme nie zum Zuge. Führt die Auswertung des Generatorausdrucks auf einen endlichen Wert, ist die Vorwegauswertung zwar grundsätzlich möglich, aber i.a. wenig(er) effizient, wenn die Lösung vom Selektor bereits bei teilweiser Auswertung des Generatorausdrucks identifiziert werden kann.

5. Referentielle Transparenz garantiert *"[that] we can replace any expression in a program with its resulting value without changing the program's semantics"* (s. S.72, mittlere Spalte, Mitte) Referentielle Transparenz ist in rein funktionalen Sprachen durch die Abwesenheit von Seiteneffekten garantiert (abgesehen von Ausdrücken für Ein- und Ausgabe, die unvermeidbar seiteneffektbehaftet und in Sprachen wie Haskell deshalb speziell gekapselt sind).

6. Der folgende λ-Ausdruck ist ein sehr einfaches Beispiel eines λ-Ausdrucks ohne Normalform; der Ausdruck reproduziert sich unter fortgesetzter Anwendung von β-Reduktionen selbst:

$$\lambda x.(xx)\lambda x.(xx) \xrightarrow{\beta} \lambda x.(xx)\lambda x.(xx) \xrightarrow{\beta} \ldots$$

7. Eine applikativartige Auswertung kann in Haskell mithilfe des Operators `($!)` erzwungen werden. Informell wird alles, was rechts vom infixangewandten Operator `($!)` steht, ausgewertet, bevor die Auswertung linksseitig vom Operator fortgesetzt wird. Bei Funktionsaufrufen kann so erreicht werden, dass Funktionsargumente erst ausgewertet werden, bevor der umfassende Funktionsaufruf expandiert wird. Die Auswertung ist applikativartig, nicht vollständig applikativ, weil in Abhängigkeit des Argumenttyps die Auswertung nur bis zu einer gewissen Tiefe durchgeführt wird. Eingesetzt wird der Operators `($!)` hauptsächlich zur Vermeidung exzessiven Speicherverbrauchs, der entsteht, wenn rekursionsbedingt immer größere Ausdrücke auf Argumentposition entstehen.

8. Selektiver Import und Export bedeutet, dass der Programmierer festlegen kann, welche Namen (Datentypen, Konstruktoren, Funktionsnamen, Wertenamen, etc.) von anderen Modulen importiert

werden können bzw. aus anderen Modulen importiert werden. Der Verzicht auf automatischen Reexport von Namen bedeutet, dass Namen, die ein Modul aus anderen Modulen importiert hat, nicht automatisch für den Import durch ein anderes Modul zur Verfügung stehen; sie müssen explizit zum Reexport zugelassen werden.

9. Bei kaskadenartig-rekursiven Funktionen wird ein Aufruf in zwei oder mehr Aufrufe aufgespalten. Aus Effizienzgründen ist dabei darauf zu achten, dass in der Folge dieser rekursiven Aufspaltung möglichst selten argumentgleiche Aufrufe entstehen, weil dasselbe Problem sonst mehrfach gelöst wird, wodurch Implementierungen oft sehr ineffizient werden. Die naive Implementierung der Fibonacci-Funktion mit ihrem exponentiellen Berechnungsaufwand ist ein prototypisches Beispiel dafür.

10. Flaggschiff-Konferenz für Themen rund um funktionale Programmierung ist die seit 1996 jährlich stattfindende *ACM SIGPLAN International Conference on Functional Programming*, spezieller rund um Haskell die seit 2008 ebenfalls jährlich stattfindenden *ACM SIGPLAN International Haskell Symposia*. Flaggschiff-Zeitschrift ist das *International Journal for Functional Programming, Cambridge, UK*, dessen erste Ausgabe 1991 erschienen ist.

Fortgeschrittene Funktionale Programmierung

C Fortgeschrittene Funktionale Programmierung

What is ...

1. a Monad?

   Answer: A monad over a category $\mathcal{C}$ is a triple (T, η, μ), where $T : \mathcal{C} \to \mathcal{C}$ is a functor, $\eta : ID_{\mathcal{C}} \to T$ and $\mu : T^2 \to T$ are natural transformations and the following equations hold:

$$\mu \circ T\mu = \mu \circ \mu T$$
$$\mu \circ T\eta = \mu \circ \eta T = ID_T$$

   i.e. »a monad is a monoid in the category of endofunctors[8]«.

   In Haskell:

...monads are instances of the type constructor class Monad obeying the monad laws:

Type Constructor Class Monad

```
class Monad m where
  (>>=)  :: m a -> (a -> m b) -> m b
  return :: a -> m a
  (>>)   :: m a -> m b -> m b
  fail   :: String -> m a

  c >> k  =  c >>= \_ -> k
  fail s  =  error s
```

Monad Laws

```
return x >>= f            = f x              (ML1)
c >>= return              = c                (ML2)
c >>= (\x -> (f x) >>= g) = (c >>= f) >>= g  (ML3)
```

...using the monad laws as example.

A) The monad laws using (>>=) and (>>):

```
return a >>= f             = f a              (ML1)
c >>= return               = c                (ML2)
c >>= (\x -> (f x) >>= g) = (c >>= f) >>= g   (ML3)
```

B) The monad laws using do-notation:

```
do x <- return a; f x      =  f a             (ML1)
do x <- c; return x        =  c               (ML2)
do x <- c; y <- f x; g y   =
            do y <- (do x <- c; f x); g y     (ML3)
```

See https://wiki.haskell.org/Monad

... and https://wiki.haskell.org/Monad_laws

... and https://wiki.haskell.org/All_About_Monads.

2. monadic programming?

   Answer: monadic programming closes a 'functional gap' between function application, sequential function composition and functorial mapping:

Monads in Functional Programming

...the monad notion became particularly popular in the field of functional programming (Philip Wadler, 1992) because (Haskell-style) monads

- allow to introduce some useful aspects of imperative programming such as sequencing into functional programming
- are well suited to smoothly integrate input/output into functional programming, as well as many other programming tasks and domains
- provide a suitable interface between functional programming and programming paradigms with side effects, in particular, imperative and object-oriented programming

...without breaking the functional paradigm!

These Capabilities let Monads

...appear to be a Suisse Knife of Functional Programming!

Monadic programming seems/is perfect for problems involving:

- Global state
  - Updating data during computation is often simpler than making all data dependencies explicit (the state monad).
- Huge data structures
  - No need for replicating a data structure that is not needed otherwise.
- Exception and error handling
  - The Maybe monad.
- ...
- Side-effects, explicit sequencing, evaluation orders
  - Canonical scenario: Input/output operations (the IO monad).

[8] A *endofunctor* is a functor that maps a category to that same category.

Example:

We consider a selection of predefined monads:

- Identity monad
- List monad
- Maybe monad
- Map monad
- State monad
- Input/Output monad

...but there are many more of them predefined in Haskell:

- Writer monad
- Reader monad
- Failure monad
- ...

...when making a 1-ary type constructor a monad, then:

- `(>>=)` will be defined to unpack the value of the first argument, map the second argument over it, and return the packed result this yields.
- `return` will be defined in the most straightforward way to lift the argument value to its monadic counterpart.
- `(>>)` and `fail` are usually not to be implemented afresh. Usually, their default implementations provided in type constructor class `Monad` are just fine.

  If the default implementations of `(>>)` and `fail` are used, this means for
  - `(>>)`: the first argument is evaluated and dropped, the second argument is evaluated and returned as result (makes sense for some monads like the IO-monad).
  - `fail`: the computation stops by calling `error` with some appropriate error message.

Extra:

Monads (i.e., instances of the type constructor class `Monad`) combine features of

- functors and functional composition/sequencing:
  `(>>=) :: m a -> (a -> m b) -> m b`
  `c >>= k >>= k' >>= k'' >>= ...`

Monads are thus well-suited for

- structuring and ordering the steps of a computation

because the monadic sequencing operations `(>>=)` and `(>>)`

- allow specifying the order of computations explicity.
- offer an adequately high abstraction by decoupling the data type forming a monad (instance) from the structure of computation.
- support equational reasoning, e.g., in terms of the monad laws.

...are often considered of being fanned by an aura of something

- mystic, wondrous that is difficult to grasp and lets monads appear the Holy Grail of functional programming (*'once I will have understood monads, I will have understood functional programming'*).

This (slightly odd) image of monads might be due to the origin and ties of the monad notion to (possibly often difficult considered) fields like

- philosophy, category theory, programming languages theory and semantics.

3. stream programming?

Short Answer:

...stream programming together with lazy evaluation enables:

- Higher abstraction: Constraining oneself to finite lists is often more complex, and – at the same time – unnatural.
- Modularization: Streams together with lazy evaluation allow for elegant possibilities of decomposing a computational problem. Most important is the
  - **Generate-Prune Pattern**

  of which the
  - Generate-select
  - Generate-filter
  - Generate-transform pattern

  and combinations thereof are specific instances.
- Boosting performance: Avoiding recomputations and recursion using stream programming combined with:
  - Münchhausen principle (cf. Chapter 2.3.2)
  - memoization (cf. Chapter 2.3.3)

Answer: A stream can be thought of as a function which when invoked, returns some of the result (not all of it) along with a call back function to get the rest when needed. Technically, it gives the entire content, but one chunk at a time, and only if asked for the rest of it. Implementing streams could be done by a new polymorphic data type like: `data Stream a = a :* Stream a` to emphasize the conceptual difference of streams (infinite by

definition) and lists (finite by definition).[9]

Example: Pre-defined:

```
[0..] ->> [0,1,2,3,4,5,...
```

```
[1,3..] ->> [1,3,5,7,9,11,...
```

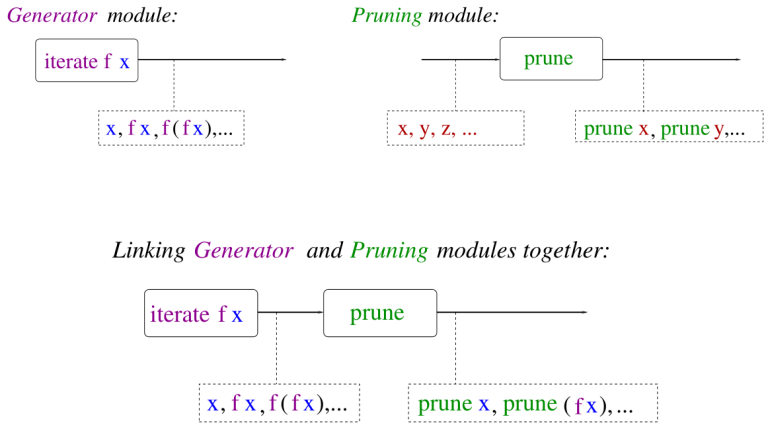
```
[1,1..] ->> [1,1,1,1,1,1,...
```

User-defined, co[10]-recursive[11]:

```
ones = 1 : ones ->> [1,1,1,1,1,1,...
```

```
nats = 0 : map (+1) nats ->> [0,1,2,3,...
```

Moreover, the non-terminating evaluation of stream generating terms can be tamed using the Generate-Prune Pattern which allows conceptually new ways of modularizing lazily evaluated functional programs:



where the replacement of »prune« yields the following three principles (modulo combination):

(a) Generate-Select: E.g. computing the square root, the n-th Fibonacci number.

(b) Generate-Filter: E.g. computing all even Fibonacci numbers.

(c) Generate-Transform: E.g. 'scaling' random numbers.

[9] Pragmatically, however, it is advantageous to model streams (and lists) by ordinary list types `[a]` (omitting for streams the empty list `[]`).

[10] *Co-recursion* is a type of operation that is dual to recursion. Whereas recursion works analytically, starting on data further from a base case and breaking it down into smaller data and repeating until one reaches a base case, corecursion works synthetically, starting from a base case and building it up, iteratively producing data further removed from a base case. Put simply, corecursive algorithms use the data that they themselves produce, bit by bit, as they become available, and needed, to produce further bits of data.

[11] Remind to Münchhausen's famous trick of »sich am eigenen Schopfe aus dem Sumpf zu ziehen!«

Extra: Streams (also called infinite lists or lazy lists) are means which (thanks to lazy evaluation) often allow to solve problems elegantly, concisely, efficiently, gain/improve performance.

Ad. performance: Stream Programming combined with

(a) *Münchhausen principle*. Very very efficient, e.g. the co-recursive stream to compute the Fibonacci numbers:

```
fibs :: [Int]                                          -- Generator
fibs = 0 : 1 : zipWith (+) fibs (tail fibs)
```

(b) *Memoization*. Idea: Replace, where possible, the (costly) computation of a function according to its body by looking up its value in a table, a so-called memo table[12]. E.g.:

```
fib_memolist = [ fib_ml n | n <- [0..] ]
fib_ml :: Int -> Int
fib_ml 0 = 0
fib_ml 1 = 1
fib_ml n = fib_memolist!!(n-1) + fib_memolist!!(n-2)
```

4. abstract data type programming?

Answer: Concrete Data Types (CDTs) are specified by naming their values (There is no need, however, to specify any operation associated with a CDT at the time of defining it.) E.g.:

```
type Forename  = String
...
type Publisher = String
type Edition   = Int

data Vehicle = Bicycle | Motorcycle | Car | Bus
data Tree a  = Nil | Leaf a | Root (Tree a) a (Tree a)
data Person  = P Forename Surname Address
newtype Book = B (Author,Title,Publisher,Edition)

v1 = Bicycle :: Vehicle
v2 = Car :: Vehicle
t1 = Leaf 42 :: Tree Int
t2 = Root Nil True (Leaf False) :: Tree Bool
p  = P "Simon" "Thompson" "unknown" :: Person
b  = B ("Thompson","Haskell","Addison-Wesley",2) :: Book
```

Note: At the time of defining the above CDTs, there is no need to define operations manipulating their values.

Contrariwise, Abstract Data Types (ADTs) are specified by naming their operations (not by naming their values). An ADT and its associated operations are implemented by a CDT and the operations associated with it (invisible to user). The meaning of the operations is precisely specified by means of laws, while the internal structure of the ADT, i.e., the representation of its values and the definition of its associated operations are left open; there is no need to define the internal structure of an ADT at the time of defining it.

Defining and Implementing an ADT is technically a three-stage approach of specification:

(a) Specification (user-visible)

- Interface Specification: Signatures of ADT operations
- Behaviour Specification: Laws for ADT operations

[12] A costly to compute function is replaced by an equivalent memo function using (memo) table look-ups.

(b) Implementation (user-invisible)

- Implementing the ADT values in terms of a CDT
- Implementing the ADT operations as CDT operations

(c) Verification

- Specification: Proving that the ADT laws are consistent and complete (proof obligation of the ADT specificator)
- Implementation: Proving that the implemented CDT operations are sound, i.e., satisfy the ADT laws (proof obligation of the CDT implementor)

Example: Stacks, Queues, Priority Queues, Tables, Sets, Heaps, Trees, Arrays, ... The Challenge: ADTs are not a first-class citizen in Haskell. Therefore, we have to pragmatically make use of Haskell features allowing us to achieve the constituting proper- ties of ADTs of information hiding, of separating their, user-visible specification from their user-invisible implementation as good as possible.

Interface Specification

...of the ADT stack, named Stack (user-visible):

```
module Stack (Stack,empty,is_empty,push,pop,top)
                                                where

-- Interface Spec.: Signatures of stack operations
empty    :: Stack a
is_empty :: Stack a -> Bool
push     :: a -> Stack a -> Stack a
pop      :: Stack a -> Stack a
top      :: Stack a -> a

-- Behaviour Spec.: Laws for stack operations
Laws (1) thru (6)
```

Behaviour Specification

...of the stack operations of the ADT stack (user-visible):

```
Behaviour Spec.: Laws for stack operations

1) is_empty empty      == True
2) is_empty (push v s) == False
3) top empty           == undef
4) top (push v s)      == v
5) pop empty           == undef
6) pop (push v s)      == s
```

Implementation A

...of the ADT stack as an algebraic data type (user-invisible):

```
data Stack a   = Empty | Stk a (Stack a)

empty          = Empty

is_empty Empty = True
is_empty _     = False

push x s       = Stk x s

pop Empty      = error "Stack is empty"
pop (Stk _ s)  = s

top Empty      = error "Stack is empty"
top (Stk x _)  = x
```

Verification

Specifier and implementer of the ADT stack can prove, respectively:

Lemma 8.2.1 (Consistency, Completeness)
The 6 laws of the behaviour specification of the ADT stack are consistent and complete.

Lemma 8.2.2 (Soundness)
The implementations A and B (and C) satisfy the 6 laws of the behaviour specification of the ADT stack.

Why ADTs?

(a) Separation of concerns: Separation of specification (interface and behaviour specification) and implementation of a data type

(b) Information hiding: No disclosure of the internal structure of the CDT, the representation and implementation of its values and the operations working on them.

(c) Security: CDT values implementing their (only) implicitly defined ADT counterparts can exclusively be created, accessed, and manipulated using the ADT operations implemented by their CDT counterparts.

(d) Programming-in-the large: ADTs enable modular program development by separating the responsibilities for specifying and implementing a data type and the operations associated with it.

(e) Reusability, maintainability.

5. generate/prune programming?

   Answer: See stream programming.

6. memoization programming?

   Answer: See stream programming.

7. 'Münchhausen-style' programming?

   Answer: See stream programming.

8. combinator programming?

   Answer: Sketch. Usually there is some type T, some functions for constructing »primitive« values of type T, and some »combinators« which can combine values of type T in various ways to build up more complex values of type T. See Parsing.

9. (functional) 'glue' programming?

   Answer: Functional programming provides two new, especially powerful kinds of glue, namely

   (a) *Higher-order functions* (glueing functions together)
   By decomposing (modularizing) and representing a simple function (`sum` in the example) as a combination of (i) a higher-order function and (ii) some simple specific functions as arguments, we obtained a program frame (`reduce` in the example) that allows us to implement many functions on lists essentially without any further programming effort.

   (b) *Lazy evaluation* (glueing programs together)
   Consider two functions (two complete functional programs) f and g and their composition $(g \,.\, f)$. Applied to input `in`, $(g \,.\, f)$ yields the output `out`: `out` $= (g \,.\, f)$ `in` $= g\ (f$ `in`$)$.
   Task: Implementing the communication between f and g.
   Possible problems: (1) Temporary files could get too large and exceed the available storage capacity. (2) f might not terminate.
   Solution: As functional glue allows a more elegant approach by decomposing a program into a *generator* and *selector* component/module glued together by functional composition and synchronized by lazy evaluation, ensuring: The generator »runs as little as possible« till it is terminated by the selector.

   Extra: The expressiveness of a language depends much on the power of the concepts and primitives allowing to *glue* solutions of subproblems to the solution of an overall problem, i.e., its power to support a modular program design. This is based on the statement that »Modularity is the key to programming in the large« (but: just modules do not suffice).

10. functional array programming?

    Answer: `Data.Array` supports static (or: immutable) arrays. Makes use of Index Type Class `Ix` which extends the type class Ord allowing for functions `range, index, inRange, rangeSize`.

    Extra:

    - Imperative Arrays:
      + Values of an array can be accessed or updated in constant time.
      + The update operation does not need extra space.
      + There is no need for chaining the array elements with pointers as they can be stored in contiguous memory locations.
      − The size is fixed (defined and fixed at declaration time).
    - Functional lists:
      + The size is not fixed. Lists can get, can be arbitrarily long, conceptually even infinitely long.
      − Lists do not enjoy the set of favorable properties of impe- rative arrays; most disturbing, values of a list can not be accessed or updated in constant time:
    - Functional arrays[13] (shall complement functional lists and be designed and implemented to get as close as possible to the favorable properties of imperative arrays):
      + Accessing the i-th element of an array (using `(!)`) shall take a constant number of steps, regardless of i.
      − The size is fixed (defined and fixed at creation time).

11. functional pearls programming?

    Answer (Internet): »Just as natural pearls grow from grains of sand that have irritated oysters, these programming pearls have grown from real problems that have irritated programmers. The programs are fun, and they teach important programming techniques and fundamental design principles. Those guidelines apply too to functional pearls.« – (Jon Bentley. *Programming Pearls*, In: CACM)

    Typical functional pearls consist of:

    (a) an instructive example of program calculation or proof, or

    (b) a nifty presentation of an old or new data structure, or

    (c) an interesting application or programming technique.

[13] Persistent data structures have the property of keeping previous versions of themselves unmodified. On the other hand, structures such as arrays admit a destructive update, that is, an update which cannot be reversed. Once a program writes a value in some index of the array, its previous value can not be retrieved anymore.

Answer (Notes):

1. Pick a combinatorially (highly) complex problem P.
2. Solve P by a conceptually straightforward, simple, and intuitive algorithm, the so-called initial algorithm (IA) implemented by some initial program IP, which is
   - obviously correct
   - typically (hopelessly) inefficient.
3. The Functional Pearl:
   3.1 Transform IP step by step into some final program (FP) which may be
   - conceptually more complex, less intuitive, not at all obviously correct but (much more) efficient than IP (e.g., feasible instead of practically infeasible, logarithmic instead of quadratic, linear instead of quasi linear,...)

   3.2 Prove that every transformation step preserves the semantics of the program it is applied to (ensuring overall equivalence of the initial and the final program and hence the correctness of the latter).

It is important to note: The functional pearl is

- not the finally resulting (efficient) implementation
- but the calculation and proof process leading to it!

The elegance of the calculation and proof process makes the

- beauty of a functional pearl!

The transformation of

- `reverse` into `fast_reverse` together with the proof of the two functions' equality

can be considered a most simple example of a functional pearl.

Example: The Smallest Free Number, Not the Maximum Segment Sum, A Simple Sudoku Solver, ...[14].

12. functional reactive programming?

Answer: Functional reactive programming (FRP) is a programming paradigm for reactive programming (asynchronous dataflow programming) using the building blocks of functional programming (e.g. map, reduce, filter). FRP has been used for programming graphical user interfaces (GUIs), robotics, games, and music, aiming to simplify these problems by explicitly modeling time. See https://en.wikipedia.org/wiki/Functional_reactive_programming.

Reactive programming a declarative programming paradigm concerned with data streams and the propagation of change. With this paradigm, it's possible to express static (e.g., arrays) or dynamic (e.g., event emitters) data streams with ease, and also communicate that an inferred dependency within the associated execution model exists, which facilitates the automatic propagation of the changed data flow. See https://en.wikipedia.org/wiki/Reactive_programming.

13. testing within Haskell?

Answer: There are three means (to test for correctness of programs) are at our disposal:

(a) Correctness by Construction (a priori, e.g. functional pearls)

(b) Verificiation (a posteriori, see Verification)

(c) Testing (a posteriori)

- Ad hoc: Controllable effort but usually no quantifiable quality statement; hence, a questionable overall value.
- Systematically: Controllable effort, quantifiable quality statement.

For testing, we used `QuickCheck`[15]: Simple properties can be defined in terms of Boolean valued functions, so-called predicates. Allows possibility to add precondition via `==>` operator. Using `==>` amounts to a trial-and-error approach for test data generation: 'Generate, then check whether the precondition is matched; if not, drop; repeat.'

[14] https://wiki.haskell.org/Research_papers/Functional_pearls.
[15] Alternatives: `EasyCheck, SmallCheck, Lazy SmallCheck, Hat`.

14. verification (accord. to the lecture notes)?

    Answer: While coinciding in their overall goal, testing and verification (proof) are of different rigor. Testing, even if it can be amazingly effective, is limited to showing the presence of errors; it can not show their absence (except of the most simple scenarios), while verification can prove the absence of errors.

15. higher-order functions programming?

    Answer: A *higher-order function* (HoF) is a function[16] that does at least one of the following:

    (a) takes one or more functions as arguments,

    (b) returns a function as its result.[17]

    Some design principles of algorithms that make use of HoF:

    - *Top-down*: Starting from the initial problem, the algorithm works down to the solution by considering subproblems or alternatives.
      - *Divide-and-conquer.*
        Idea ...: (I) If a problem instance is simple/small enough, (I') solve it, else (II) divide the problem instance into smaller subproblem instances that can be solved. (III) Combine solved instances.
        ... as HoF:
        * `indiv :: p -> Bool`; (I)
        * `solve :: p -> s`; (I')
        * `divide :: p -> [p]`; (II)
        * `combine :: p -> [s] -> s`; (III)

        Example:

        ```
        quickSort :: Ord a => [a] -> [a]
        quickSort ls  = divide_and_conquer indiv solve divide combine ls where
        indiv ls      = length ls <= 1
        solve         = id
        divide (l:ls) = [[ x | x <- ls, x <= l], [ x | x <- ls, x > l]]
        combine (l:_) [l1,l2] = l1 ++ [l] ++ l2
        ```
      - *Backtracking search.*
        Idea ...: Search for a particular solution of the problem by a systematic trial-and-error exploration of the solution space.[18]

[16] All other functions are first-order functions. In mathematics higher-order functions are also termed operators or functionals. https://en.wikipedia.org/wiki/Higher-order_function.

[17] By the way: A *first-class citizen* (also type, object, entity, or value) in a given programming language is an entity which supports all the operations generally available to other entities. These operations typically include being passed as an argument, returned from a function, modified, and assigned to a variable. First-class functions are a necessity for the functional programming style, in which the use of higher-order functions is a standard practice.

[18] Requirements: A set of all possible situations or nodes constituting the search (node) space; these are the potential solutions that need to be explored. A set of legal moves from a node to other nodes, called the successors of that node. An initial node. A goal node, i.e., the solution.

... as HoF: The objective is a HoF `search_dfs`[19] solving suitably parameterized problem instances of kind p (problems) using the 'backtracking' principle, where

```
search_dfs :: (Eq node) => (node --> [node]) --> (node --> Bool)
                                --> node --> [node]
```

where `search_dfs` has the arguments `goal :: node --> Bool` (function checking whether a node is a solution), `succ :: node --> [node]` (function yielding the list of successors of a node) and `node` (representing node information).
Example: The eight-tile problem, Towers of Hanoi, ...

– *Priority-first search.*
Idea ...: Similar to backtracking search, i.e., searching for a partic- ular solution of the problem by a systematic trial-and- error exploration of the search space but the candidate nodes are ordered such that always the most promising node is first (priority-first search/best-first search).[20]
... as HoF: The objective is a HoF `search_pfs` solving suitably parameterized problem instances of kind p (problems) using the 'priority-first/best-first' principle.

```
search_dfs :: (Ord node) => (node --> [node]) --> (node --> Bool)
--> node --> [node]
```

where `search_pfs` has the arguments `goal :: node --> Bool`, `succ :: node --> [node]` and `node`.
Example: The eight-tile problem, ...

– *Greedy search.*
Idea ...: Similar to priority-first/best-first search but limiting the search to immediate successors of a node (greedy search/hill climbing search).[21]
... as HoF: The objective is a HoF `search greedy` solving suitably parameterized problem instances of kind p (problems) using the 'greedy/hill climbing' principle.

```
search_dfs :: (Ord node) => (node --> [node]) --> (node --> Bool)
--> node --> [node]
```

where the arguments are defined as above.
Example: Prim's minimum spanning tree algorithm, The money change problem, ...

- Bottom-up: Starting from small problem instances, the algorithm works up to the solution of the initial problem by combining solutions of smaller problem instances to solutions of larger ones.
  – *Dynamic programming.*[22]

[19] The process is similar to a depth-first graph traversal. Moreover, when exploring the graph, each visited path can lead to the goal node with an equal chance. Sometimes, however, it might be known that the current path will not lead to the solution. In such cases, one backtracks to the next level up the tree and tries a different alternative.

[20] Requirements: A set of all possible situations or nodes constituting the search (node) space; these are the potential solutions that need to be explored. A comparison criterion for comparing and ordering candidate nodes wrt. their (expected) 'quality' to investigate 'more promising' nodes before 'less promising' nodes. A set of legal moves from a node to other nodes, called the successors of that node. An initial node. A goal node, i.e., a solution.

[21] Requirements: A set of all possible situations or nodes constituting the search (node) space; these are the potential solutions that need to be explored. A set of legal moves from a node to other nodes, called the successors of that node. An initial node. A goal node, i.e., a solution. There shall be no local minimums, i.e., no locally best solutions.

[22] Top-down algorithms might suffer from generating a large number of identical subproblems. This replication

Idea ...: (I) Solve (the) smaller instances of the problem first. (II) Save the solutions of these smaller problem instances. (III) Use these results to solve larger problem instances.
... as HoF: The objective is a HoF `dynamic` solving suitably parameterized problem instances of kind p (problems) using the 'dynamic programming' principle.

```
dynamic :: (Ix coord) => (Table entry coord --> coord --> entry) -->
                         (coord,coord) --> (Table entry coord)
```

where `dynamic` has the arguments `compute :: (Ix coord) => Table entry coord --> coord --> entry` (Given a table and an index, compute computes the corresponding entry in the table (possibly using other entries in the table)) and `bnds :: (Ix coord) => (coord,coord)` (The argument `bnds` specifies the boundaries of the table. Since the type of the index is in the class `Ix`, all indices in the table can be generated from these boundaries using the function range.).
Extra: (Dynamic Programming vs. Memoization)
Dynamic programming and memoization enjoy very much the same characterics and offer the programmer quite similar benefits. In practice, differences in behaviour are minor and strongly problem-dependent. In general, both techniques are similarly powerful. Conceptual difference:

* Memoization opportunistically computes and stores argument/result pairs on a by-need basis ('lazy' approach).
* Dynamic programming systematically precomputes and stores argument/result pairs before they are needed ('eager' approach).

Dynamic Programming:

+ Memory efficiency: For some problems the dynamic programming solution can be adjusted to use asymptotically less memory. Limited history recurrence, i.e., only a limited number of preceding values need to be remembered (e.g., two for the computation of Fibonacci numbers) which allows to reuse memory during computation.
+ Run-time performance: The systematic programmer-controlled filling of the argument/result pairs table allows sometimes slightly more efficient (by a constant factor) implementations.

Memoization:

+ Freedom of conceptual overhead: The programmer does not need to think about in what order argument/result pairs need to be computed and how to be stored in the memo table. (In dynamic programming all table entries are computed systematically when needed.)
+ Freedom of computational overhead: Only argument/result pairs are computed and stored when needed. (In dynamic programming they are systematically precomputed when and before they are needed.)

Example (dynamic programming): Shortest paths for all pairs of nodes of a graph, Fibonacci numbers, The travelling salesman problem, ...

of work can severely impair performance. Dynamic programming aims at overcoming this shortcoming by systematically precomputing and reusing results in a bottom-up fashion (from smaller to larger problem instances).

Extra: Many powerful, general algorithmic principles can be encapsulated in a suitable higher-order function . This allows to design a collection or a class of algorithms (instead of designing an algorithm for only a particular application).

Conceptually: emphasizes the essence of the underlying algorithmic principle.

Pragmatically: makes these algorithmic principles easily re-usable.

16. input/output programming?

Answer: Haskell's I/O system is built around a mathematical foundation: the monad, which mediates between the values natural to a functional language and the actions that characterize I/O operations and imperative programming in general.

The Input/Output Monad

```
instance Monad IO where    (Impl. intern. hidden)
 (>>=)  :: IO a -> (a -> IO b) -> IO b
 return :: a -> IO a
 (>>)   :: IO a -> IO b -> IO b
 fail   :: String -> IO a
```

Note:

- IO-values are so-called IO-commands (or commands).
- Commands have a procedural effect (i.e., reading or writing) and a functional effect (i.e., computing a value).
- (>>=): With p, q commands, p >>= q is a composed command that first executes p, thereby performing a read or write operation and yielding an a-value x as result; subsequently q is applied to x, thereby performing a read or write operation and yielding a b-value y as result.
- return: Lifts an a-value to an IO a-value w/out performing any input or output operation.

17. functional logic programming?

Answer:

Logic Programming Functionally

Declarative programming

- Characterizing: Programs are declarative assertions about a problem rather than imperative solution procedures.
- Hence: Emphasizes the 'what,' rather than the 'how.'
- Important styles: Functional and logic programming.

If each of these two styles is appealing for itself

- (features of) functional and logic programming

uniformly combined in just one language should be even more appealing.

Question

- Can and shall (features of) functional and logic programming be uniformly combined?

Functional vs. Logic Languages

Functional languages

- are based on the notion of mathematical function.
- programs are sets of functions that operate on data structures and are defined by equations using case distinction and recursion.
- provide efficient, demand-driven evaluation strategies that support infinite structures.

Logic languages

- are based on predicate logic.
- programs are sets of predicates defined by restricted forms of logic formulas, such as Horn clauses (implications).
- provide non-determinism and predicates with multiple input/output modes that offer code reuse.

Combining Functional and Logic Programming

...some principal approaches for combining their features:

- Ambitious: Designing a new programming language enjoying features of both programming styles (e.g., Curry, Mercury, etc.).
- Less ambitious: Implementing an interpreter for one style using the other style.
- Even less ambitious: Developing a combinator library allowing us to write logic programs in Haskell.

Current functional logic languages aim at balancing

- generality (in terms of paradigm integration).
- efficiency of implementations.

Functional logic programming offers

- support of specification, prototyping, and application programming within a single language.
- terse, yet clear, support for rapid development by avoiding some tedious tasks, and allowance of incremental refinements to improve efficiency.

Overall: Functional logic programming is

- an emerging paradigm with appealing features.

18. parallel functional programming?

    Answer: Functional languages are promising candidates for effective parallel programming, because of their high level of abstraction and, in particular, because oftheir referential transparency. In principle, any subexpression could be evaluated in parallel. As this implicit parallelism[23] would lead to too much overhead, modern parallel functional languages allow the programmers to specify parallelism explicitly[24] [25].

19. functional programming?

    Answer: (Wikipedia) Programming paradigm where programs are constructed by applying and composing functions. It is a declarative programming paradigm in which function definitions are trees of expressions that map values to other values, rather than a sequence of imperative statements which update the running state of the program.

    In functional programming, functions are treated as first-class citizens, meaning that they can be bound to names (including local identifiers), passed as arguments, and returned from other functions, just as any other data type can. This allows programs to be written in a declarative and composable style, where small functions are combined in a modular manner.

    Functional programming is sometimes treated as synonymous with purely functional programming, a subset of functional programming which treats all functions as deterministic mathematical functions, or pure functions. When a pure function is called with some given arguments, it will always return the same result, and cannot be affected by any mutable state or other side effects. This is in contrast with impure procedures, common in imperative programming, which can have side effects (such as modifying the program's state or taking input from a user). Proponents of purely functional programming claim that by restricting side effects, programs can have fewer bugs, be easier to debug and test, and be more suited to formal verification.

    Functional programming has its roots in academia, evolving from the lambda calculus, a formal system of computation based only on functions. For more: https://en.wikipedia.org/wiki/Functional_programming

20. Haskell?

    Answer: a general-purpose, statically typed, purely functional programming language with type inference and lazy evaluation[26].

[23] Also: expression parallelism. Idea: If $f(e_1, ..., e_n)$ is a functional expression, then arguments (and functions) can be evaluated in parallel.

[24] Idea: Introducing and using meta-statements (e.g., for controlling the data and load distribution, communication).

[25] Some people distinguish between the words parallelism and concurrency. A parallel computation is a computation in which some number of tasks occur simultaneously. Parallelism improves throughput by using a number of processors (or cores or machines) to execute a set of tasks. A concurrent computation is a computation in which several tasks access a shared, mutable resource. The example given above of the functions f and g reading and writing from the mutable variable x is an example of a concurrent computation – the shared resource is the mutable cell x. See https://www.cs.princeton.edu/~dpw/courses/cos326-12/notes/parallel.php.

[26] Also: call-by-need, is an evaluation strategy which delays the evaluation of an expression until its value is needed (non-strict evaluation) and which also avoids repeated evaluations (sharing)

21. equational reasoning?

Answer: Equational reasoning is a well-known mathematical means for reasoning about and proving the validity of e.g. arithmetical statements: $(a+b)\cdot(a-b) = a^2 - b^2$.

This carries over to functional programming because in functional programming the equality symbol = means: »equal by definition«. An equation of the form `f x y = x+y` as (part of the) definition of a function `f` is a genuine mathematical equation, i.e the expression on the left hand side and the right hand side of = have the same value.

Example: The functions `f` and `g` denote the same function.[27] where

```
f :: Int -> Int -> Int
f a b = (a+b) * (a-b)
g :: Int -> Int -> Int
g a b = a^2 - b^2
```

In imperative programming an expression of the form `x = x+y` does not represent a mathematical equation meaning that `x` and `x+y` have the same value.

Extra: Proven equality of functions can be used e.g. for optimization by replacing a less efficient implementation (called initial algorithm, ini- tial program) by a more efficient one (called final algorithm, final program). E.g.

- Initial program

  ```
  reverse :: [a] -> [a]
  reverse []
  = []
  reverse (x:xs) = reverse xs ++ [x]
  ```

- Final program

  ```
  fast_reverse :: [a] -> [a]
  fast_reverse xs = fr xs [] where
  fr [] ys = ys fr (x:xs) ys = fr xs (x:ys)
  ```

Theorem 1. *The functions reverse and fast reverse denote the same function, i.e.,* $\forall$ `ls` $\in$ `a-List` . `reverse ls` = `fast_reverse ls` *(see Functional Pearl)*

22. program calculation?

Answer: Snippet from the lecture notes — In principle, every proof technique can be made use of by approaches aiming at correctness by construction, among these (1) (inductive) proof principles and (2) equational reasoning, sometimes also called proof by program calculation. (See equational reasoning).

[27] Careful: Haskell semantics implicitly imposes an ordering on the equations. E.g. `isZero :: Int --> Bool` where `isZero 0 = True` (is a logical property) and `isZero n = false` (can only be applied if `n` is different from `0`.)

23. a monoid?

Answer:

...monoids are instances of type class `Monoid` obeying the monoid laws.

Type Class `Monoid`

```
class Monoid m where
  mempty  :: m
  mappend :: m -> m -> m
  mconcat :: [m] -> m
  -- Default implementation
  mconcat  = foldr mappend mempty
```

Monoid Laws

```
mempty `mappend` x           = x           (MonoL1)
x `mappend` mempty           = x           (MonoL2)
(x `mappend` y) `mappend` z  =
  x `mappend` (y `mappend` z)              (MonoL3)
```

Monoids are types with

- a binary operation `mappend`.
- a value `mempty`.
- a unary operation `mconcat` reducing a list of monoid values to a single monoid value using `mappend`.

The monoid laws

- MonoL1 and MonoL2 require that `mempty` is a left-unit and a right-unit of `mappend`, hence a unit.
- MonoL3 requires that `mappend` is associative.

Programmer obligation:

- Programmers must prove that their instances of `Monoid` satisfy the monoid laws.

Example: List Monoid, Numerical Monoids (`Sum` and `Product` monoids), Boolean Monoids (`All` and `Any` monoids), Ordering Monoid

Note: For some monoids, commutativity of `mappend` holds, e.g., `sum`, `product`, `any`, `all` monoids. For other instances it does not hold, e.g., `list`, `ordering` monoids.

Monoids are most useful for defining folds over values of structured data since folding requires an associative operation. Folding seems obviousand natural for lists but is possible, too, for the values of many other structured data, e.g., trees. This motivates the introduction of the type (constructor) class `Foldable` as collection of all type constructors whose values can be folded.

...type classes of a new kind:

```
class Foldable f where
  foldr   :: (a -> b -> b) -> b -> f a -> b
  foldl   :: (a -> b -> a) -> a -> f b -> a
  foldMap :: (Monoid m, Foldable t) =>
                              (a -> m) -> t a -> m
  ...
```

Note:

- `f` and `t` are applied to type variables, here `a` and `b`. This means, `f` and `t` are (1-ary) type constructors, not types.
- `Foldable` is thus a type constructor class, a special type class.
- The `foldl`, `foldr` operations of `Foldable` extend folding of lists to folding of values of other 'foldable' structured data while allowing to reuse the operation names.

Looking ahead: The List Type Constructor `[]`

...is one important instance of `Foldable`:

```
foldr :: (a -> b -> b) -> b -> [] a -> b
foldl :: (a -> b -> a) -> a -> [] b -> a
```

where `Data.Foldable.foldl` and `Data.Foldable.foldr` are defined in terms of their counterparts `foldl` and `foldr` introduced in Chapter 10.5, LVA 185.A03 Funktionale Programmierung.

`Foldable` is the first example of this new kind of higher-order type classes called type constructor classes of which we consider more examples next: `Functor`, `Applicative`, `Monad`, and `Arrow` (cf. Chapters 10, 11, 12, and 13).

24. a functor?

Answer: A functor is a representative of a new kind of type classes, a higher-order type class, a so-called: *type constructor class*. Functors are instances of the type constructor class `Functor`

```
class Functor f where
  fmap :: (a -> b) -> f a -> f b
```

obeying the functor laws:

```
fmap id      = id                    (FL1)
fmap (h . g) = fmap h . fmap g       (FL2)
```

Instances of Functor (like of other type constructor classes) are type constructors, no types.

Type classes and type constructor classes are conceptually equal but differ in the type of their instances:

- Type constructor classes (`Foldable`, `Functor`, `Monad`, `Arrow`,...) have type constructors as instances:
  - `Tree`, `[]`, `(,)`, `(->)`,...
- Type classes (`Eq`, `Ord`, `Num`, `Monoid`,...) have types as instances:
  - `Tree a`, `[] a`, `(,) a b`, `(->) a b`,...

Type constructors are maps

- constructing new types from given ones.

  Examples: Tuple constructors `(,)`, `(,,)`, `(,,,)`; list constructor `[]`; map constructor `(->)`; input/output constructor `IO`,...

Example: Types, whose values can be mapped over compositionally, with a neutral element, like e.g. Lists with `mapL` and `id`

```
g :: a -> b, h :: b -> c
mapL g []       = []
mapL g (x:xs)   = (g x) : mapL g xs
mapL (h . g) xs = mapL h (mapL g xs) (compositional)
mapL id xs      = xs (neutral element)
```

should be made instances of type constructor class Functor,

```
instance Functor [] where
fmap g []     = []
fmap g (l:ls) = g l : fmap g ls
```

Examples (Functors): Identity Functor, List Functor, Maybe Functor, Either Functor, Map Functor, Input/Output Functor.

25. parsing?

Answer: Parsing is the basic task of a compiler. Umbrella term for the lexical and syntactical analysis of the structure of text, e.g., source code text of programs.

Example:

Informally

...the parsing problem is the following:

1. Read a sequence of objects/values of a type a.
2. Yield an object/value or a sequence of objects/values of a type b.

Illustration:

1. Read a sequence of values of type Char:
   ⟨`if n mod = 0 then 2*n else 2*n+1 fi`⟩
2. Yield a sequence of pairs of tokens and strings:
   ⟨`(if_token,""),(var_token,"n"),(op_token,"mod"),`
   `(rel_token,"="),(cst_token,"0"),(then_token,"")`
   `(cst_token,"2"),(op_token,"*"),(var_token,"n"),`
   `(else_token""),...,(fi_token,"")`⟩

...a parser p for arithmetic expressions could be assumed to

1. read strings representing well-formed arithmetic expression
2. yield the Exp values matching the strings read with:

```
data Exp = Lit Int | Var Char | Op Ops Exp Exp
data Ops = Add | Sub | Mul | Div | Mod
```

Example:

```
p "((2+b)*5)"
  ->> Op Mul (Op Add (Lit 2) (Var 'b')) (Lit 5)
```

Functionally speaking, there are two different but conceptually related approaches:

(a) *Combinator parsing.*

In addition to primitive parsers, e.g., `isDigit :: Char --> Bool` there are parser combinators (re-usable higher-order polymorphic functions) to write more complex and powerful parser functions.

(b) *Monadic parsing.*

Making use of `newtype Parser a = Parse (String --> [(a,String)])` one can write parsing functions by means of Monadic operations.

In Conclusion

...combinator and monadic parsing rely (in part) on different language features but are quite similar in spirit as becomes obvious when opposing their primitives and combinators:

	Combinator Parsing	Monadic Parsing

Primitive Parsers	`none`
	`succeed`
	`token`
	`spot`
		`mzero`
		`return`
		`char`
		`sat`

Parser Combinators	`alt`
	`(>*>)`
	`build`
		`mplus`
		`(>>=)`
		`mbuild`

Invaluable

...for combinator (as well as monadic) parsing are:

- Higher-order functions: `Parse a b` (like `Parser a`) is of a functional type; all parser combinators are thus higher-order functions.
- Polymorphism: The type `Parse a b` is polymorphic: We do need to be specific about either the input or the output type of the parsers we build. Hence, the parser combinators mentioned above can immediately be reused for tokens of any other data type (in the examples, these were lists and pairs, characters, and expressions).
- Lazy evaluation: 'On demand' generation of the possible parses, automatical backtracking (the parsers will backtrack through the different options until a successful one is found).

»And how many hours a day did you do lessons?« said Alice, in a hurry to change the subject.

»Ten hours the first day,« said the Mock Turtle: »nine the next, and so on.«

»What a curious plan!« exclaimed Alice.

»That's the reason they're called lessons,« the Gryphon remarked: »because they lessen from day to day.«