Zusammenfassung/Überblick FPROG (WS 2021|2022)

3.0 VU Funktionale Programmierung 185.A03

20. Februar 2024

Teil 1 (Einführung)

Kapitel 0: Allgemeines

- Funktionen (Fakultät, Fibonacci)
- Programmaufbau: Abstützen von Fkt. auf anderen Fkt.
- zentral: Rechnen mit Funktionen (bspw. Funktionskomposition) (Argument und Resultat als Funktion)
- Unterschied imperativer (IP) vs. funktionaler Programmierung(FP) (Bsp. Fallunterscheidung → IP: Anweisung, FP:Ausdruck), allgemein:
  - **IP:** Bedeutung des Programms ist die Beziehung zwischen Anfangs- und Endzuständen, die bewirkte Zustandsänderung, Kontrollfluss, Seiteneffekte
  - **FP:** keine Seiteneffekte, Datenabhängigkeiten steuern die Auswertung(sreihenfolge), nur Ausdrücke (diese liefern Werte), Variablen sind Namen für Ausdrücke: Wert ist der Wert des Ausdrucks
    ⇒ **gleichungsbasiertes, ergebnisorientiertes Programmieren**

Kapitel 1: Motivation

- Taschenrechnerfunktionen (+,-,*, abs(), sqrt(), cos x, [-2..3], [n | n ← [-6..8]] , ...)
- Ausgabe:

```
main = putStrLn "Hello, World!"
putStrLn :: String -> IO
putStrLn "Hello, World!"
```

- **Fakultätsfunktion** (musterbasiert)
- **Euklidischer Algorithmus** (hierarchisches System von Funktionen, bewachte Ausdrücke)
- **Gerade/ungerade-Test für ganze Zahlen** (ggseitige Fkt.aufrufe)

- **Längenberechnung von Listen** (parametrisch polymorphe Fkt)
- **Umkehren von Zeichenreihen** (selbstgewählt, sprechende Typnamen, Zeichenreihe)
- **Transformieren von Listen** (Funktion höherer Ordnung = erstrangige Sprachelemente (engl. first class citizens))
- **Addieren von Zahlen** (Typklassen (instantiiert), eingeschränkte Polymorphie, überladene Funktionen)
- **Binomialkoezientenberechnung** (musterbasierte Funktionsdefinition mit hierarchischer Abstützung auf eine andere Funktion, musterbasierte (kaskaden- oder baumartig-) rekursive Funktionsdefinition, Uncurryfiziert versus Curryfiziert)
- **Sieb des Eratosthenes** (Programmierung mit Strömen - d.h. Listen als Argument bzw. Argumentstromtyp und Resultatstromtyp)

Zusammenfassung:

- **Funktionale Programme:** Systeme (wechselweise) rekursiver Funktionsvorschriften (oder Rechenvorschriften).
- **Funktionen:** sind zentrales Abstraktionsmittel in funktionalen Programmen (wie Prozeduren (Methoden) in prozeduralen (objektorientierten) Programmen).
- **Funktionale Programme:** werten Ausdrücke aus. Das Resultat dieser Auswertung ist ein Wert eines bestimmten Typs. Dieser Wert kann elementar oder funktional sein; er ist die Bedeutung, die Semantik des Ausdrucks

Teil 2 (Grundlagen)

Kapitel 2: Vordefinierte Datentypen

Zahlen, Zeichen, Wahrheitswerte, Tupel, Listen,...

- **Elementare Datentypen:**
  - **unstrukturierte Werte:** ganze Zahlen (Int, Integer, z.B. 42 :: Int ⇒ vom Typ Int), Gleitkommazahlen (Float, Double), Wahrheitswerte (Bool), Zeichen (Char) (mit typüblichen Operationen)
  - **strukturierte Werte:** Tupeltyp bzw. Kruezprodukttyp (z.B. ('m',2), () = Nulltupel), Listentyp (z.B. [2,3,4,5] :: [Int])
    - Tupellisten
    - Funktionslisten ([sin,cos,tan,sqrt] :: [Float → Float])
    - vordefinierte Funktionen auf Listen z.B. (:), (++), (!!), concat, reverse, head [1,2,3], tail [1,2,3]
    - Listenaufzählungsausdrücke z.B. [2..10], ['a','b'..'z']
    - Listenkomprehension
    - **Syntaktischer Zucker:** [1,2,3] ⇒ **Standarddstl.:** (1:(2:(3:[])))
    - vordefinierte Operatoren (++) und Relatoren (==)

Kapitel 3: Funktionen

Syntaxvarianten, curryfiziert, uncurryfiziert, Stelligkeit,...

- **Definition, Schreibweisen, Sprachkonstrukte**
  - if-then-else
  - Alterantive: wertbesierte Auswahl und musterbasierte Auswahl (bzw. Kombination aus beiden)

  - Lokale Deklaration mit *where, let-in*
  - Argumentfreie anonyme λ-*Abstraktion*

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

- **Funktionssignaturen, Funktionsterme, Funktionsstelligkeiten** (und Klammereinsparungsregeln)
  - (Funktions-) **Signaturen** sind **rechtsassoziativ** geklammert.
    (ersetze :: (Txt → (Vork → (Alt → (Neu → Txt))))) ... schrittweise Argumentkonsumation
  - (Funktions-) **Terme** sind **linksassoziativ** geklammert. (z.B.: ersetze "Ein alter Text"1 "alter" "neuer")
    ((((ersetze "Ein alter Text") 1) "alter") "neuer")
  - (Funktions-) **Stelligkeit** ist 1. (Fkt konsumiert nur ein Argument)

⇒ Assoziativität dient der Klammereinsparung

- **Funktionspfeilform** (binom :: **Integer** → **Integer** → Integer) (flexibler, weil ganze Funktion statt Tupel)
- **Kreuzproduktform** (binom' :: **(Integer,Integer)** → Integer) (partielle Auswertung nicht möglich)
- Argumente und Werte von Funktionen und Funktionstermen: können elementaren, zusammengesetzten oder funktionalen Typs sein.

• Curryfizierte, uncurryfizierte Funktionen

- Art der **Konsumation** der Argumente
  * Einzeln Argument für Argument: **curryfiziert** bzw. curryfizierte Funktionen
    ersetzt Kreuzproduktform durch Funktionspfeilform
    daher auch **partielle Auswertung möglich** (= Fkt liefert Fkt als Ergebnis)
  * Alle auf einmal als Tupel: **uncurryfiziert** bzw. uncurryfizierte Funktionen
    ersetzt Funktionspfeilform durch Kreuzproduktform
- *Prüfstein:* Entsteht nach Konsumation des **ersten Arguments** ein **(funkt.) Zwischenergebnis** und sind unter den Argumenten
  * keine Tupel? → curryfiziert

```
f1 :: Int → Int → Int: curryfziert
z.B. f1 42 :: Int → Int
```

  * auch Tupel? → nicht vollständig curryfiziert

```
f8 :: Int → (Int,Bool) → Int: curryfiziert, aber nicht vollständig.
z.B. f8 2 :: (Int, Bool) → Int: uncurryfiziertes Zwischenergebnis
```

- gilt der Prüfstein nicht, dann ist die Funktion uncurryfiziert, z.B.

```
f4 :: (Int → Int) → Int: uncurryfiziert.
f4 fac :: Int, f4 fib :: Int, Ergebnisse sind keine funkt. Zwischenergebnisse, nur Werte)
```

• Operatoren, Präfix- und Infixverwendung

- präfix (fac 5) oder (binom 45 6), infix (2 + 3) (45 'binom' 6 → mit Hochkommata als Infix möglich), postfix (5!)

• Operatorabschnitte (operator sections)

- Notationelle Abkürzungen ('syntaktischer Zucker') für **anonyme λ-Abstraktionen**

- **Partiell ausgewertete Binäroperatoren** heißen in Haskell ⇒ **Operatorabschnitte** - z.B.:
  dbl mit (*2), die Funktion, die ihr Argument verdoppelt (λx.x*2)
  oder
  inc mit (+1) bzw. (1+) wobei (λx.x+1) bzw. (λx.1+x)
- Operatorabschnitte können **selbstdefiniert** oder **vordefiniert** binären Operatoren gebildet werden
- **mit uncurryfizierten Fkt. können keine Operatorabschnitte definiert werden**
- Beispiel:

```
dbl :: Integer → Integer
dbl = (2*)

dbl 5 {-Aufruf mit Ergebnis 10-}
```

- **Angemessene, unangemessene Funktionsdefinitionen**
  - total und partiell definierte Fkten ⇒ **Sichtbarmachen der Partialität** durch **otherwise** = **error "undefined"**
    unangemessen: Fkt definieren ohne Ausnahmefall zu berücksichtigen

- **Funktions- und Programmformatierung, Abseitsregel**
  - Die Formatierung des Programmtexts trägt Bedeutung!
  - Einhaltung der Abseitsregel für Funktionsdefinitionen, d.h. bei bspw. Wächtern/bewachten Ausdrücken Einrückung nicht vergessen

Kapitel 4: Typsynonyme, Neue Typen, Typklassen

type, newtype, class, Uberladung,...

- **Typsynonyme**
  - **sprechende Funktionsnamen und Parameternamen**, Aliasnamen, keine neuen Typen bzw. keine eigene Typidentität
  - Bsp.: *type Dollar = Float*
  - type <Alias-Name> = <existierender Typname>
  - führen **nicht** zu höherer Typsicherheit

  - Zusammengesetzt: *type Student = (Vorname,Nachname,Email)*, wobei Vorname etc. = String ist
  - **Selektoren:** Student ⇒ (, ,) = "MaxMuxe123456@stud.tuw.ac.at"bzw. (v,n,e) ist n = Nachname
  - **Wild-card Selektoren:** (_,n,e)

- ## Neue Typen

  - **Erreichen von Typsicherheit**, neue eigene Typidentität (auch aufbauend auf existierenden Typen (z.B. *newtype Alter = A Int deriving Eq*)
  - **newtype**-Deklarationen: bspw. *newtype Dollar = USD Float*
    ((Datenwert-)Konstruktoren sind Funktionen, d.h. USD :: Float → Dollar)
  - newtype <freigewählter Typbezeichner> = <freigewälter (Datenwert-) Konstruktorbez.> <Bezeichner eines existierenden (!) Typs>
  - Beispiel:

```
newtype Dollar = USD Float
preis = USD 2.00 :: Dollar
```

  - **Nachteil:** vordefinierte Operationen und Relationen (hier auf Float) nicht mehr vorhanden ⇒ müssen selbst implementiert werden, z.B. so:

```
gleich_usd :: Dollar → Dollar → Bool
gleich_usd (USD x) (USD y) = x == y
```

  - **Typklassen -** ähnlich wie neue Typen, nur hier Definition neuer Typen und Instanzierung von Relationen und Operationen in einem

- ## Typklassen

  - **Hierarchie der Typklassen**

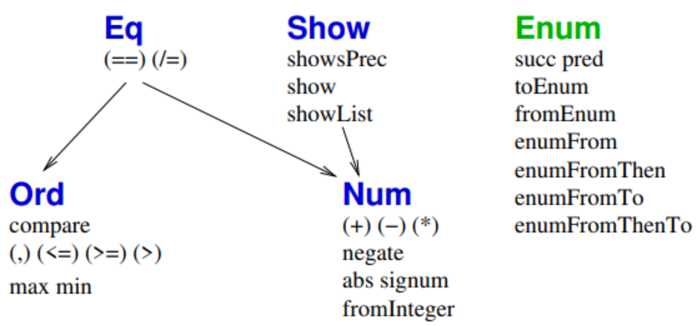


Abbildung 1: Kap. 4/121, Typklassen Eq, Show, Enum, Ord, Num

  - **Wozu:** Um Überladung von Relatoren, Operatoren zu organisieren
    Beispiel:

```
class Eq a where
   (==) :: a → a → Bool {-Funktionen d. Typklasse-}
   (/=) :: a → a → Bool {-mit ihrer Signatur-}
x /= y = not (x == y) {-Protoimplementierungen-}
x == y = not (x /= y) {-für (/=) und (==)-}
```

– **Instanzbildung** gibt **Wissensbereitstellung**

Instanzbildung eplizit über *instance*-Deklarationen:

```
instance Eq Dollar where
   (==) (USD x) (USD y) = x == y
   {- Impl. von (/=) vollautom. dank Proto.Impl.-}
```

Instanzbildung implizit und automatisch über *deriving*-Klausel:

```
newtype Dollar = USD Float deriving (Eq, Ord, Show, Num)
```

Vollstädniges Bsp.:

```
class Waehrung a where
   – 0-stellige Fkt., Konstanten
   ist_gesetzliches_Zahlungsmittel_in :: [Land]
   muenzen_im_Nennwert_von :: [Wert]
   anzahl_verschiedene_Muenzen :: Int
   – 1-stellige Funktionen
   betrag_in_Muenzen_verschieden_zahlbar :: a → Int
   kaufmaennisch_runden_auf_2_Kommastellen :: a → Float
   – Protoimplementierung
   anzahl_verschiedene_Muenzen x = length (muenzen_im_Nennwert_von x)


instance Waehrung Dollar where
   ist_gesetzliches_Zahlungsmittel_in = ["USA"]
   muenzen_im_Nennwert_von = [0.01,0.05,0.1,0.25,1.0,2.0]
   betrag_in_Muenzen_verschieden_zahlbar b = ...
   kaufmaennisch_runden_auf_2_Kommastellen b = ...


instance Waehrung Euro where...
```

Zusammenfassung:

* **Typklassen...**
  - ...dienen der Organisation und Verwaltung von Überladung,
  - ...sind Mengen von Typen, deren Werte typspezifisch mit Funktionen gleichen Namens bearbeitet werden können ((==),(>),(>=), (+),(*),(-), etc.).
  - ... erhalten Typen durch explizite Instanzbildung (instance-Deklaration) oder implizite automatische Instanzbildung (deriving-Klausel) als Elemente zugewiesen
* **direkte Überladung** (nichtleere Signatur, *(==) ::* ***Eq a*** $\Rightarrow$ $a \rightarrow a \rightarrow Bool$) und
  indirekte Überladung (sind unmittelbar in einer Typklasse eingeführt (*f ::* ***(Num a, Waehrung a)*** $\Rightarrow$ $a \rightarrow a \rightarrow a$) (Waehrung sind nicht selbst in Typklasse eingeführt, stütz sich aber auf solche Funktionen ab)
* **überladene Funktion** (leere Signaturkontext)
  - **Monomorpher Fall** (nur konkrete Typen, keine Typvariablen): (*fac :: Integer* $\rightarrow$ *Integer*)
  - **Parametrisch polymorpher Fall** (konkrete Typen, Typvariablen): (*length :: [a]* $\rightarrow$ *Int*)

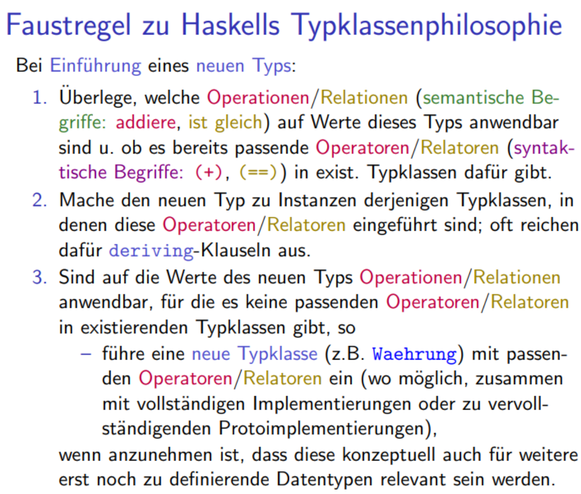


Abbildung 2: Kap. 4/137, Faustregel Typklassenphilosophie

Kapitel 5: Algebraische Datentypdeklarationen

data, Funktionen auf alg. Datentypen, Feldsyntax,...

- **Überblick, Orientierung**
  - **Aufgabe:** originär neue Datentypen und ihre Werte einzuführen ⇒ **Algebraischer Datentyp data**
  - **Summentyp**
  - **Produkttyp**
- **Summentyp**
  - Beispiel

```
type Info = String
data Baum = Blatt Info | Gabel Info Baum Baum deriving (Eq,Show)

-- Rekursive Traversierfunktion
rek :: Baum -> String
rek (Blatt a) = a
rek (Gabel a b1 b2) = a ++ rek b1 ++ rek b2
```

  - mit 1-, 2-, 3-, n-stelligen Konstruktoren (hier Blatt 1-stellig und Gabel 3-stellig)
  - Funktionen: rekursiv definiert (top-down)
    Datentypen: induktiv definiert (bottom-up)
  - **Möglicherweise-Typ (polymorph)**: *data Maybe a = Nothing | Just a deriving (Eq,Ord,Read,Show)*
  - **Entweder/Oder-Typ (polymorph)**: *data Either a b = Left a | Right b deriving (Eq,Ord,Read,Show)*
- **Produktyp**
  - Beispiel:

```
data Person = P Vorname Nachname Alter deriving(...)
```

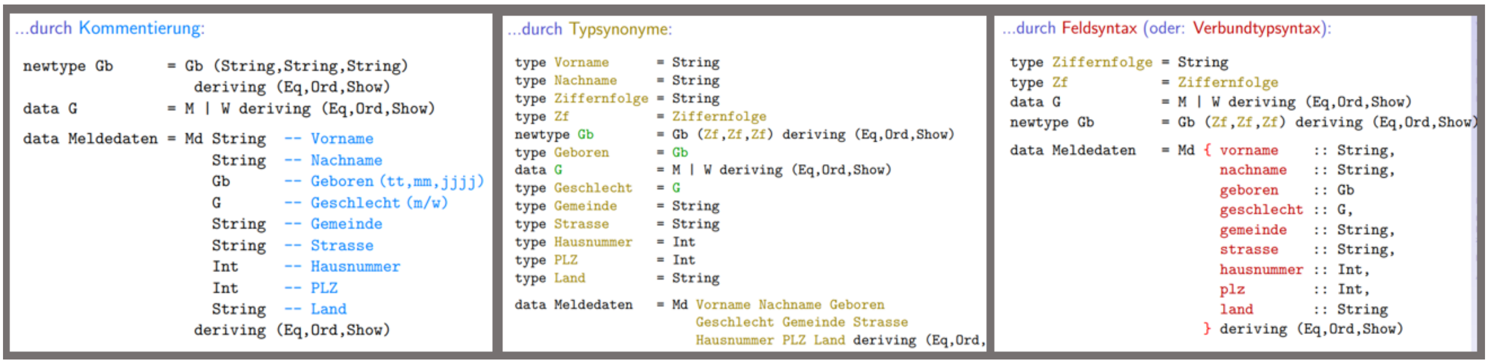
- **Aufzählungstypen**
  - Beispiel:

```
data Jahreszeit = Fruehling | Sommer | Herbst | Winter deriving (Eq,Ord,Bounded,
Enum,Read,Show)
```

– **Vordefinierte Aufzählungstypen:** Ordering, Bool, () (Nulltypen)

- **Feldsyntax**

  – Übersicht



...durch Kommentierung:

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

data Meldedaten = Md String  -- Vorname
                     String  -- Nachname
                     Gb      -- Geboren (tt,mm,jjjj)
                     G       -- Geschlecht (m/w)
                     String  -- Gemeinde
                     String  -- Strasse
                     Int     -- Hausnummer
                     Int     -- PLZ
                     String  -- Land
                    deriving (Eq,Ord,Show)
```

...durch Typsynonyme:

```
type Vorname       = String
type Nachname      = String
type Ziffernfolge = String
type Zf            = Ziffernfolge
newtype Gb         = Gb (Zf,Zf,Zf) deriving (Eq,Ord,Show)
type Geboren       = Gb
data G             = M | W deriving (Eq,Ord,Show)
type Geschlecht    = G
type Gemeinde      = String
type Strasse       = String
type Hausnummer    = Int
type PLZ           = Int
type Land          = String

data Meldedaten    = Md Vorname Nachname Geboren
                        Geschlecht Gemeinde Strasse
                        Hausnummer PLZ Land deriving (Eq,Ord,
```

...durch Feldsyntax (oder: Verbundtypsyntax):

```
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)
```

Abbildung 3: Kap. 5/174-176, Transparente, sprechende Typdeklarationen

Zusammenfassung:

- Typen
  – **type:** erlaubt existierenden Typen – zusätzliche, neue Namen zu geben (Synonyme, Aliase). Typ und Typsynonym sind ident; alle Funktionen auf dem Typ stehen daher auch auf jedem Typsynomym zur Verfügung; Typ und Typsynonyme können sich wechselweise vertreten.
  – **newtype:** erlaubt existierenden Typen – unverwechselbare, neue Identitäten zu verleihen. Typ und davon abgeleiteter Neuer Typ sind verschieden und unverwechselbar; keine auf dem Typ zur Verfügung stehende Funktion überträgt sich auf den abgeleiteten Neuen Typ; alle auf Werten des Neuen Typs benötigte Fkt. sind selbst zu implementieren; manchmal reicht eine deriving-Klausel dafür.
  – **data:** erlaubt – originär neue Typen und ihre Werte einzuführen. Alle auf Werten des neuen Typs benötigte Fkt. sind selbst zu implementieren; manchm. reicht eine deriving-Klausel dafür.
- Algebraische Datentypen
  a) **Echte Summentypen:** Mindestens zwei Konstruktoren, mindestens ein nicht nullstelliger Konstruktor.
  b) **Echte Produkttypen:** Exakt ein zwei- oder höherstelliger Konstruktor.
  c) **Aufzählungstypen:** Ausschließlich nullstellige Konstruktoren.

  d) **Randfall:** Ein algebraischer Datentyp mit genau einem einstelligen Konstruktor lässt sich in gleicher Weise als unechter Summen- wie als unechter Produkttyp ansehen.
- Feldsyntax

Kapitel 6: Muster und mehr

- **Muster, Musterpassung**
  - In Fkten verwendet und arbeiten bspw. mit Ausdrücken wie Wild-Cards, (n:ns), ...
  - **Unterscheidung:**
    * **Nichtabweisende Muster:** Variablen, Wild-Cards
    * **Abweisende Muster:** Konstanten, Strukturmuster (z.B. *(True,n,_)*, *(n:m:[]))*
- **Muster für Werte elementarer Datentypen**
  - Wenn also bspw. bei Fkt. ein Int mit einem Symbol 0 oder 1 oder n ersetzt wird
- **Muster für Werte von Tupeltypen**
  - Wenn also bspw. bei Fkt. Argumente als Tupel wie (Int, Int) **mit einem Tupel** ersetzt wird durch bspw. (_,0), (n,k)
- **Muster für Werte von Listentypen**
  - Wenn also bspw. bei Fkt. Argumente als Listen wie [Int] **mit einem Tupel** ersetzt wird durch bspw. (_,0), (n,k)
- **Muster für Werte algebraischer Datentypen**
  - Argument einer Fkt. ist dabei ein (wahrscheinlich selbstdefinierter) Alg. Datentyp - Abfrage in der Fkt mittels (Konstruktor Variable) z.B. (Blatt a), (Blatt _), (Wurzel _ l r)
- **Das als-Muster**
  - Lässt sich für Muster strukturierter Werte verwenden - Tupeltyp, Listentyp, alg. Datentyp
  - z.B.: *s@(_ :cs) = s : nichtleere_postfixe cs*, (nichtleere_postfixe ist dabei Fkt.name)
- **Listenkomprehension**
  - gebildet aus **Generatoren** (←), **Tests** (Fkt. innerhalb LK, hier "≥"oder "isPowOfTwo"), **Transformationen** (Fkt. ganz vorne, hier "id")
  - z.B. [id n|n ← ns2, isPowOfTwo n, n≥5]
- **Konstruktoren, Operatoren**

- **Konstruktor (:) :** z.B. [42,17,4] == (42:(17:(4:[])))
- **Operatoren (++) :** z.B. [42,17,4] == [42,17] ++ [] ++ [4]

Teil 3 (Applikative Programmierung)

Einleitung: Applikative Programmierung

- Applikatives Programmieren ist Programmieren und Rechnen mit Funktionen über elementaren Werten (Zahlen, Zeichen, Wahrheitswerte,...)
- Funktionen sind weder Argument noch Resultat v. Funktionen!
- Beispiel: (**Ausdruck**operand hier bspw. Nat0 bzw. n)

```
type Nat0 = Int
fac :: Nat0 → Nat0
fac n = prod [1..n]
```

Zusammenfassung:

- Appl. Progr. im stengen Sinn:
  - Programmieren und Rechnen auf dem Niveau elementarwertiger Ausdrücke und Funktionen mit elementaren Werten als Argument und Resultat.
  - Funktionen werden durch Abstraktion nach (unabhängig (variabel!) angesehenen) Ausdrucksoperanden gebildet.
  - Funktionen werden auf Ausdrücke aus Konstanten, Variablen u. Funktionstermen elementaren Werts appliziert.
  - **Summa summarum:** 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.**

Kapitel 7: Rekursion

Rekursionstypen, Aufrufgraphen, Komplexität,....

- ### Motivation

  - Rekursives Vorgehen bietet oft sehr elegante Lösungen
  - so wichtig, dass eine Klassifizierung von Rekursionstypen zweckmäßig ist
  - Beispiele: Quicksort und Türme von Hanoi

```
quickSort :: [Integer] → [Integer]
quickSort [] = []
quickSort (n:ns) = quicksort smaller ++ [n] ++ quicksort larger
   where smaller = [ m | m ← ns, m leq n ]
   larger = [ m | m ← ns, m > n ]
```

- **Rekursionstypen**
  - Rechenvorschrift heißt rekursiv, wenn sie in ihrem Rumpf (direkt oder indirekt) aufgerufen wird
  - Rekursion: mikroskopischer (einzelne Rechenvorschriften) und makroskopischer (Systeme von Rechenvorschriften) Ebene
  - **Mikroskopischer Ebene**:
    - **Repetitive (schlichte, endständige) Rekursion:** pro Zweig höchstens ein rekursiver Aufruf und diesen stets als äußerste Operation.
    - **Lineare Rekursion:** pro Zweig höchstens ein rekursiver Aufruf, davon mindestens einer nicht als äußerste Operation.
    - **Baumartige (kaskadenartige) Rekursion:** pro Zweig können mehrere rekursive Aufrufe nebeneinander vorkommen.
    - **Geschachtelte Rekursion:** rekursive Aufrufe enthalten rekursive Aufrufe als Argumente.
  - **Makroskopischer Ebene**:
    - **Indirekte (verschränkte, wechselweise) Rekursion:** zwei oder mehr Funktionen rufen sich wechselweise auf.
  - **Probleme:** Rekursion nicht immer effizient (siehe Mehrfachberechnung, Bsp.: Fibonacci-Zahlen mit baumartiger Rekursion → besser: *Rechnen auf Parameterposition*/Rückführung auf repetitive Rekursion), dabei gilt:
    - **repetitive Rekursion** am (kosten-) günstigsten.
    - **geschachtelte Rekursion** am ungünstigsten.
    - **daher Rückführung** baumartiger/lineare Rek. auf einfache/repetitive Rek. durch neue Fkt. mit mehr Parametern

- **Aufrufgraphen**
  - Aufrufgraphen erleichtern es, sich über die **Struktur** von **Systemen von Rechenvorschriften** Klarheit zu verschaffen.
  - Der Aufrufgraph von **System von Rechenvorschriften (S)** sei ein Graph. Rechenvorschriften seien **Knoten** (f,g,...) und der Aufruf von einer Rechenvorschrift f einer anderen Rechenvorschrift g seien **gerichtete Kanten**.

```
ggt :: Int -> Int -> Int
ggt m n
 | n == 0 = m
 | n > 0  = ggt n (mod m n)
mod :: Int -> Int -> Int
mod m n
 | m < n  = m
 | m >= n = mod (m-n) n
```

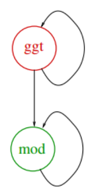


Abbildung 4: Kap. 7/71, System hierarchischer Rechenvorschriften der Funktionen ggt und mod

- **Interpretation von Aufrufgraphen:**

   - **Direkte Rekursivität** (Selbstkringel), **Wechselweise Rekursivität** (Kreise mit Kanten), **Direkte hierarchische Abstützung** (eine direkte Kante von f nach g, aber nicht umgekehrt)
   - **Indirekte hierarchische Abstützung** (nur Folge von Kanten, nicht direkt) , **Indirekte wechselweise Abstützung** (mit Kreisen mit mehr als zwei Kanten), **Unabhängigkeit/Isolation** (Knoten hat keine ausgehenden Kanten)

- **Komplexität, Komplexitätsklassen**
   - Obere, untere, einhüllende Schranken

Kapitel 8: Auswertung einfacher Ausdrücke

Ausdrücke ohne/mit einfachen Fkt.-Termen,...

Auswertung einfacher Ausdrücke

   - ...hat das Ziel, Ausdrücke soweit zu vereinfachen wie nur irgend möglich und so ihren Wert zu berechnen
   - Zusammenspiel von **Expandieren** (Funktionsterme, Funktionsaufrufe) und **Simplifizieren** (Funktionstermfreie Ausdrücke)

- **Auswertung einfache Ausdrücken ohne Funktionsterme**
   - Die sog. Church-Rosser- oder Diamant-/Rauteneigenschaft: Rechnen mit links- bzw rechtestmöglicher Stelle bzw. Mittelweg (schneller)

- **Auswertung einfacher Ausdrücke mit Funktionstermen**
   - **Applikativ:** Argumentauswertung **vor** Expansion (→ zuerst **Simplifizieren**, dann **Expandieren**)
   - **Normal:** Argumentauswertung **nach** Expansion (→ zuerst **Expandieren**, dann **Simplifizieren**)
   - Beispiel

```
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.
```

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

Abbildung 5: Kap. 8/111, links applikativ, rechts normal

Kapitel 9: Programmentwicklung, Programmverstehen

Vorgehensrichtlinien zu Programmentw., Programmverst.

- **Motivation**
  - Finden eines algorithmischen Lösungsverfahrens (nicht vollständig automatisierbar, aber Vorgehensweisen, Faustregeln)
  - **5 Entwicklungsschritte**:
    (1) Lege die (Daten-) Typen fest.
    (2) Führe alle relevanten Fälle auf.
    (3) Lege die Lösung für die einfachen (Basis-) Fälle fest.
    (4) Lege die Lösung für die übrigen Fälle fest.
    (5) Verallgemeinere und vereinfache das Lösungsverfahren.
- **Programmverstehen**
  - Strategien, um Programme zu lesen und zu verstehen
    * Verhaltenshypothesen
    * Ressourcenbedarfs verstehen (konzeptuelle Ebene: Analyse Zeit- und Speicherplatzverhalten)
    * zusätzlich eingestreuter Programmkommentare in Form von **Vor- und Nachbedingungen**

Teil 4 (Funktionale Programmierung)

Applikatives Programmieren ist das

- Programmieren und Rechnen mit Funktionen über
  - elementaren Werten.
- Argument und Resultat von Funktionen sind
  - elementare Werte (Zahlen, Zeichen, Wahrheitswerte,...)!

Funktionales Programmieren ist das

- Programmieren und Rechnen mit Funktionen über
  - funktionalen Werten (Funktionen).
- Argument und Resultat von Funktionen sind
  - funktionale Werte (Funktionen) (*sin*, *cos*, *tan*, !, *fib*, $\binom{n}{\cdot}$, $\binom{\cdot}{k}$ $\binom{\cdot}{\cdot}$, `length`, `quickSort`, `curry`,...)!

Abbildung 6: Kap. 10/4, applikatives versus funktionales Programmieren

Zwei Säulen: Rechnen mit Funktionen höherer Ordnung und Polymorphie

Leitsatz: Maximale Liberalität (Einschränkung von Werten bzgl.: Datentypen, Argumenten, Resultaten von Funktionen, Typ von Funktionen) ⇒ bspw. Typ von Listenelementen braucht nicht bekannt sein, wichtiger: Liste von irgendwas ⇒ d.h., **Strukturidentes soll polymorph ausgedrückt sein**

Curryfizierte (wenn Parameter einer Fkt ausschließlich eine Fkt ist) und **Uncurryfizierte** Funktionen

Beispiel Funktion höherer Ordnung mit polymorphen Parametertypen:

Beispiel:

```
flip :: (a → b → c) → (b → a → c)
flip f = g where g x y = f y x
```

Kapitel 10: Funktionen höherer Ordnung | Funktionen mit Funktionen als Argument und Resultat

- **Funktionen als Argument und Resultat von Funktionen: Motivation**
  - Funktionen als Argument und Resultat einer Funktion
  - Beispiel

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

Aufruf: map (*2) [2,4..10] ⇒ [4,8,..20] mit [2*2,4*2,..10*2] :: [Int → Int]
```

  - Man kann mit Resultatlisten sofort rechnen: **head (map (*) [2,4..10]) 10 ⇒ ((*) 2 10) ⇒ 20**

- **Curryfizierte** versus **uncurryfizierte** Funktionen

- **Funktionen als Resultat**
  - Beispiel

```
curry :: ((a,b) → c) → (a → b → c)
curry f = g where g xy= f (x,y)


uncurry :: (a → b → c) → ((a,b) → c)
uncurry f = g where g = y → f x y
```

- **Applikative vs. funktionale Berechnungsweise: Ein Beispiel**
  - Beispiel: *ggt_euklid_app* und *ggt_euklid_fkt*
  - **Ziele:**
    * Wiederverwendung von Programmcode.
    * Kürzere und meist einfacher zu verstehende Programme
    * Einfachere Herleitung, einfacherer Beweis von Programmeigenschaften (Stichwort: Programmverifikation).

Kapitel 11: Polymorphie

auf Funktionen, Datentypen; echt, unecht, direkt, indirekt

Generelle Unterscheidung: **Echte** versus **unechte (direkt/indirekt)** Polymorphie

- **Polymorphie auf Funktionen: Echte Polymorphie / parametrisch Polymorphie**
  - Beispiel Funktionen

```
id :: a → a
id x = x
```

  - Beispiel auf selbstdefinierte Datentypen

```
data Tree a = Leaf a | Node (Tree a) (Tree a)

depth :: (Tree a) → Int
depth (Leaf a) = 1
depth (Node t1 t2) = 1 + max (depth t1) (depth t2)
```

- **Ziele:**
  * **ermöglicht** die Wiederverwendung von Funktionsnamen, Funktionsimplementierungen
  * wird **synonym** bezeichnet als **parametrische Polymorphie**
  * ist **erkennbar** an **keiner typkontexteingeschränkten** Typvariablen der Funktionssignatur! ⇒ also wirklich rein polymorph wie a oder b (und nicht Num ⇒ a ...)

- **Unechte Polymorphie**
  - Beispiel an unecht polymorphen Funktionen (und Funktionssiganturen) und über selbstdefinierte Datentypen

```
Direkt unechte Polymorphie
(+) :: Num a => a -> a -> a

oder

change :: Eq a => (a -> b) -> a -> b -> (a -> b)
change f x y = g where g = -> if z==x then y else f z

oder über selbstdefinierte Datentypen

Indirekt unechte Polymorphie
data Tree a = Leaf a | Node (Tree a) (Tree a)
add :: Num a => (Tree a) -> a
add (Leaf x) = x
add (Node t1 t2) = add t1 + add t2
```

  - **Ziele:**
    * **ermöglicht** die Wiederverwendung von Funktionsnamen, **NICHT ABER VON Funktionsimplementierungen** (für jeden Typ ist eigene typspezifische Implementierung erforderlich)
    * wird **synonym** bezeichnet als **ad hoc Polymorphie** oder **Überladung**
    * ist **erkennbar** an: **eine oder mehrere** Typvariablen der Funktionssignatur sind **typkontexteingeschränkt**!

- **Direkt unechte Polymorphie (/ (Direkt) ad hoc P. / (Direkt) überladen P.)**
  - ...wenn sie Element einer Typklasse (Num, Eq,...) sind, in einer Typklasse eingeführt sind. (wie oben im Bsp.: *(+) :: Num a ⇒ a → a → a*)

- **Indirekt unechte Polymorphie (/ (Indirekt) ad hoc P. / (Indirekt) überladen P.)**

- ...wenn sie nicht Element einer Typklasse sind, sich aber auf ein solches Element einer Typklasse abstützen (add, zeige,...). Bsp.:

```
add :: Num a => (Tree a) -> a
add (Leaf x) = x
add (Node t1 t2) = add t1 + add t2 {- hier ist es die Addition, die die Abstützung auf Num dstl.
-}
```

- **Auflösen des indirekten Abstützen** durch eigene Definition oder bequemer durch automatische Instanzbildung mittels deriving-Klausel

- **Polymorphie auf Datentypen**

- Beispiel **polymorphe Typsynonyme**

```
type Paar a b = (a,b)
type Tripel abc = (a,b,c)
```

- Beispiel **polymorphe Neue Typen**

```
newtype Zweitupel a b = Z (a,b)
newtype Relation a b = R [(a,b)]
newtype Relationsmenge a b = RM [Relation a b]
```

- Beispiel **polymorphe algebraische Typen**

```
data BigTree abc = BigLeaf (a -> b) | BigNode [c] (BigTree a b c) (BigTree a b c)
```

- Beispiel **polymorphe Neue Typen mit Typkontexteinschränkung**

```
newtype (Num n, Num m) => NumerischeRelation n m = NR [(n,m)]
```

- Beispiel **polymorphe algebraische Typen mit Typkontexteinschränkung**

```
data (Ord sortierschluessel, Show info) => Kartei sortierschluessel info = Karteikarte info Unterkartei
sortierschluessel (Kartei sortierschluessel info) (Kartei sortierschluessel info)
```

- für **polymorphe Typsynonyme** gibt es keine Typeinschränkungen

Zusammenfassung:

* **Neue Typen , algebraische Typen :**
  - **Wiederverwendung von:**

(1) Datenstrukturnamen (Typ- und Konstruktornamen)

(2) Konstruktionsweise und strukturellem Aufbau für Datenwerte

(3) polymorphen Funktionen (Funktionsnamen und -implementierungen) auf diesen Datentypen.

  - **erkennbar an:**

(1) Typvariablenargumenten des Datentyps, die **typkontexteingeschränkt** bzw. **nicht typkontexteingeschränkt sein können**

* **Typsynonyme**
  (1) Polymorpher Neuer Typen
  (2) Polymorpher algebraischer Datentypen
  (3) Polymorpher Typsynonyme
  (4) Polymorpher Funktionen (Funktionsnamen und -implementierungen) auf diesen Datentypen
* **erkennbar an:**
  (1) Typvariablenargumenten des Typsynonyms, die alle **typkontextuneingeschränkt sein müssen**

Teil 5 (Fundierung funktionaler Programmierung)

Kapitel 12: λ-Kalkül

Berechenbar(keitstheorie), Churchsche These, einfachste funktionale (Programmier-) Sprache,...

- **Motivation**
  - Was ist berechenbar (Berechenbarkeitsmodelle)
  - '**Etwas**' ist intuitiv berechenbar, wenn es eine '**irgendwie machbare**' effektive mechanische Methode gibt, die für
    - gültige/nicht gültige Argumentwerte mit einem besonderen Fehlerwert oder nie abbricht. in endlich vielen Schritten den nicht gültige Argumentwerte mit einem besonderen Fehlerwert oder nie abbricht.
    - nicht gültige Argumentwerte mit einem besonderen **Fehlerwert** oder nie abbricht.
  - Jede solche Methode M definiert ein **formales Berechenbarkeitsmodell** mit einem formalen Berechenbarkeitsbegriff: Berechenbar mit M, M-berechenbar ('**Etwas**' ist intuitiv berechenbar gdw es ist M-berechenbar!) $\Rightarrow$ **Falsifizierbarkeit möglich**, Verifizierbarkeit unmöglich
  - **Berechnungsmodelle (z.B.):** Allgemein rekursive Funktionen, Turing-Maschinen, Registermaschinen, Markov-Algorithmen, µ-rekursive Funktionen, ...
  - **Zentrales Resultat der Berechenbarkeitstheorie:**
    - **Gleichmächtigkeit** (alle genannten formalen Berechnungsmodelle und zugehörige Methoden sind gleich mächtig)
    - **Korollar: Universalität des λ-Kalküls** (alles was in einem der vorher genannten Modelle berechenbar ist, ist im λ-Kalkül berechenbar (und umgekehrt!))
  - **Churchsche These ('λ-Kalkülthese'):** 'Etwas' ist intuitiv berechenbar gdw es ist im λ-Kalkül berechenbar.
  - **λ-Kalkül** = formales Berechnungsmodell (Berechnungsbegriff über Paaren, Listen, Bäumen, auch potentiell unendlichen, über Funktionen höherer Ordnung, etc), ist Grundlage aller funktionalen Programmiersprachen

- **Syntax des reinen λ-Kalküls**
  - Beispiel (zusätzlich **Bindungsbereich** und **Gültigkeitsbereich** der gebundenen Variablen einer λ-Abstraktion)

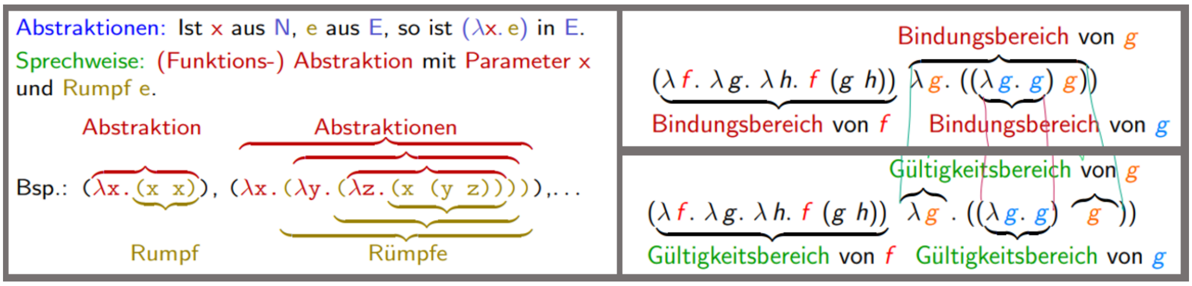


Abbildung 7: Kap. 12/38,42, Bildung, Bindungsbereich, Gültigkeitsbereich

– beachte **freies** (kein definierendes Vorkommen) **und gebundenes Variablenvorkommen**

- **Semantik des reinen λ-Kalküls**
  - Definition der Semantik von λ-Ausdrücke:
    * **Syntaktische Substitution:**

      e' [e/x] = denjenigen Ausdruck, der aus e' entsteht, indem jedes freie Vorkommen von x in e' durch e substituiert, ersetzt wird.
      z.B.: $\lambda x.\ (x\ y)\ [(a\ b)/y] = \lambda x.\ (x\ (a\ b))$
      Achtung Bindungsfehler: $\lambda x.\ (x\ y)\ [(x\ b)/y] = x.\ (x\ (x\ b)) \Rightarrow$ zuerst Umbennen von x in z: $\lambda z.\ (z\ y)\ [(x\ b)/y] = \lambda z.\ (z\ (x\ b))$

    * **Konversionsregeln/Reduktionsregeln:**

      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 $x \notin \mathit{frei}(e)$

      Abbildung 8: Kap. 12/54, Konversionsregeln

      Von **links nach rechts** angewendet: **Reduktion**. (Konversionen heißen auch β- und η-Reduktion)

      Von **rechts nach links** angewendet: **Abstraktion**. (Konversionen heißen auch β-Abstraktion)

    * **Reduktionsfolgen/Reduktionsstrategien**

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

      · **(Grund-) Reduktionsordnungen, -strategien:** Normale Reduktion(sordnung) (äußerst), Applikative Reduktion(sordnung) (innerst)

- Praktisch relevante Reduktionsordnungen, -strategien sind
  (1) **Linksnormale Reduktions(sordnung) (linkest-äußerst)** und
  (2) **Linksapplikative Reduktions(sordnung) (linkest-innerst)**
- λ-Ausdruck ist in **Normalform**, wenn er durch β-, η-Reduktionen nicht weiter reduzierbar ist
- Beispiel Applikative Ordnung und Normale Ordnung

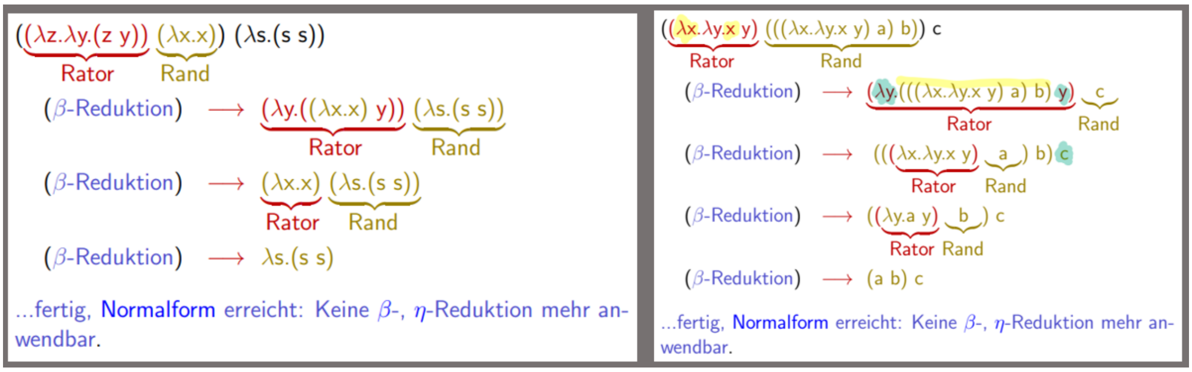


Abbildung 9: Kap. 12/59,61, Applikative Ordnung (links) und Normale Ordnung (rechts)

* **Normalformen (Existenz/Eindeutigkeit)**

  - ...existieren nicht notwendig; nicht jeder λ-Ausdruck besitzt eine Normalform, ist in Normalform konvertierbar (z.B. $\lambda x.(x\ x)\ \lambda x.(x\ x) \rightarrow\ \lambda x.(x\ x)\ \lambda x.(x\ x) \rightarrow\ \dots$ ...reproduziert sich durch fortgesetzte β-Reduktionen endlos: **Eine Normalform existiert nicht!**
  - manchmal terminiert **Normale Reduktion**, aber **Applikative Reuktion** nicht (obwohl Normalform existiert) (Kap.12/64)
  - **Church/Rosser-Theoreme:**

  (1) **Konfluenz-, Diamant-, Rauteneig.:** (Informell:) Wenn eine Normalform existiert, dann ist sie (bis auf α-Konversion) eindeutig bestimmt! ... Normalform eines λ-Ausdruck lässt sich (bis auf α-Ausdrücke) eindeutig bestimmen

  (2) **Standardisierung:** (Informell:) Normale Reduktion terminiert am häufigsten, so oft wie überhaupt nur möglich! ... normale Reduktionsordnung mit der Normalform terminiert

- Semantik von λ-Ausdrücken (Determiniertheit, Turingmächtigkeit)
- Rekursion vs. **Y-Kombinator** (Rekursion ist im reinen λ-Kalkülnicht vorgesehen $\Rightarrow$ sind ja anonym) $\Rightarrow$ Hilfe der Y-Kombinator (Kombinator = λ-Terme ohne freie Variable): Ermöglicht Ersetzung und Realisierung von Rekursion durch Kopieren (, Y-Kombinator haben Ausdrücke mit Selbstanwendung)

• **Angewandte λ-Kalkül**

- δ-Reduktionen für Terme (applikativ bzw. Abstraktionsterme) wie $\lambda x.(x+x)$
- Beispiel δ-Reduktionsfolge

$(\lambda x.\lambda.\, x*y)\ ((\lambda x.\lambda y.\, x+y)\ 9\ 5)\ 3$

$(\beta\text{-Reduktion, li}) \longrightarrow (\lambda x.\lambda y.\, x*y)\ ((\lambda y.\, 9+y)\ 5)\ 3$

$(\beta\text{-Reduktion, li}) \longrightarrow (\lambda x.\lambda y.\, x*y)\ (9+5)\ 3$

$(\beta\text{-Reduktion, li}) \longrightarrow (\lambda y.\, (9+5)*y)\ 3$

$(\beta\text{-Reduktion, li}) \longrightarrow (9+5)*3$

...keine β-, η-Reduktion mehr anwendbar; weiter mit δ-Reduktionen:

$\ldots\ (9+5)*3$

$(\delta\text{-Reduktion, li}) \longrightarrow 14*3$

$(\delta\text{-Reduktion, li}) \longrightarrow 42$

Anm.: Ratoren in rot, Randen in gold; li für linkest-innerst.

Abbildung 10: Kap. 12/79, δ-Reduktionsfolge

Kapitel 13: Auswertungsordnungen

normal, applikativ, faul, Church/Rosser-Resultate,...

- **Überblick, Orientierung**
  - **Applikativ:** Kennzeichen: Arg.Auswertung vor Expansion, d.h. sofortige, **frühe Arg.Auswertung** in Funktionstermen. (**linksapplikativ** ⇒ **eager evaluation**)
  - **Normal:** Kennzeichen: Arg.Auswertung nach Expansion, d.h. aufgeschobene, **späte Arg.Auswert.** in Funktionstermen. (**linksnormal** ⇒ **lazy evaluation**)
  - Expandierens (E) (von Funktionstermen) versus Simplifizierens (S) (von Ausdrücken verschieden von Funktionstermen)
- **Applikative, normale Funktionstermauswertung**
  - Applikative Funktionstermauswertung = **Argumentauswertung *vor* Expansion**
  - Normale Funktionstermauswertung = **Argumentauswertung *nach* Expansion:**
  - Beispiel Applikative versus Normale Funktionstermauswertung
- **Linksapplikative, linksnormale Auswertung**
  - ...neben der Art der Argumentauswertung von Funktionstermen (vor/nach Expansion) auch festlegt: **an welcher Stelle** in einem Ausdruck gerechnet wird (linkestmöglich)
  - (Funktions-) Argumenten
    - **Applikativ** (innerst): Ausgewertet übergeben.
    - **Normal** (äußerst): Unausgewertet übergeben.
  - Ausdruck

```
Applikativ:  2 * fac (2-1)                      (Arg.vereinf. zuerst)
    (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) ->> ...in diesem Stil fortfahren.

Normal:      2 * fac (2-1)                      (Expansion zuerst)
    (E) ->> 2 * (if (2-1) == 0 then 1
                     else ((2-1) * fac ((2-1)-1)))
    (S) ->> 2 * (if 1 == 0 then 1
                     else ((2-1) * fac ((2-1)-1)))
    (S) ->> 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) ->> ...in diesem Stil fortfahren.
```

Abbildung 11: Kap. 13/111, Applikative versus Normale Funktionstermauswertung

  * **Linksapplikativ** (linksinnerst): Linkestinnerste Stelle. -

    frühe, fleißige Auswertungsordn. (engl. eager evaluation)

  * **Linksnormal** (links äußerst): Linkest äußerste Stelle. -

    späte, faule Auswertungsordnung (engl. lazy evaluation)

- Welche Auswirkungen hat die Wahl von (links-) applikativer oder (links-) normaler Auswertungsordnung?
  * **Terminierungsgeschwindigkeit:** linksapplikativ und linksnormal untersch. sich je nach Situation
  * **Terminierungsverhalten/Terminierungshäufigkeit:** (Links-) normale Auswertung terminiert am häufigsten. ((links-) normale und (links-) applikative Auswertung untersch. sich in ihrem Terminieren)
    $\Rightarrow$ wichtig: wenn es applikativ terminiert, dann muss es auch normal terminieren, andersherum geht es nicht
  * **Ergebnis:** beide enden mit demselben Ergebnis
  * **Fazit:** lazy evaluation (Normale Auswertung) ist etwas besser, weil sie eine annähernde Effizienz, aber eine bessere/sicherere Terminierungshäufigkeit besitzt
  * **am besten: Späte, faule Auswertung** (Kap.13/150), braucht aber mehr Speicher als die beiden anderen

- **Parameterübergabemechanismenanalogien**
  - Applikative Auswertungsordnung entspricht: **Call-by-value**
  - Normale Auswertungsordnung entspricht: **Call-by-name**
  - Späte Auswertungsordnung entspricht: **Call-by-need**

- **Argumentauswertungshäufigkeit**

- Applikative Auswertungsordnung: **Jedes Argument wird genau einmal ausgewertet.**
- Normale Auswertungsordnung entspricht: **Jedes Argument wird so oft ausgewertet, wie es benutzt wird.**
- Späte, faule Auswertungsordnung: **Jedes Argument wird höchstens einmal ausgewertet.** (⇒ Implementierung linksnormaler Auswertung und dargestellt mittels Graph)
- **...späte, faule Auswertung ist damit am effizientesten**

• Striktheit, Terminierung, Ergebnisneutralität

- strikte Argumente = wenn diese Argumente einer Fkt nicht eingehalten werden, endet die Fkt undefined
- **Striktheit, Terminierung:** Für strikte Funktionen stimmen die Terminierungsverhalten von früher und später Auswertungsordnung für die strikten Argumente überein.
- **Striktheit, Ergebnisneutralität:** Durch den Übergang von später auf frühe Auswertung für strikte Argumente einer Funktion gehen keine Ergebnisse verloren

• Andere Bezeichnungen für applikative, normale Auswertung

- Zusammengefasst:

Applikative Auswertungsordnung (engl. applicative order eval.)
- ▶ Verwandte Bezeichnungen: Wertparameter-, innerste, strikte Auswertung (engl. call-by-value, innermost or strict evaluation).
- ▶ Operationalisierung: Linksapplikative, linkestinnerste, frühe, fleißige Auswertung (engl. leftmost- innermost or eager evaluation).

Normale Auswertungsordnung (engl. normal order evaluation)
- ▶ Verwandte Bezeichnungen: Namensparameter-, äußerste Auswertung (engl. call-by-name, outermost evaluation).
- ▶ Operationalisierung: Linksnormale, linkestäußerste Auswertung (engl. leftmost-outermost evaluation).
- ▶ Effiziente Operationalisierung mit Ausdrucksteilung: Späte, faule Auswertung (engl. lazy evaluation).
  - – Verwandte Bezeichnung: Bedarfsparameter-Auswertung (engl. call-by-need evaluation).

Abbildung 12: Kap. 13/159, Andere Namensgebung/Aliase für applikative und normale Ordnung

- **In Haskell späte, faule Argumentauswertung (engl. lazy evaluation)**
- Vor- und Nachteile früher versus später Auswertung (dabei wichtiger Aspekt, dass linksnormale eher terminiert)
- Welche Auswertung soll man nehmen - Aspekt d. Zahl der Argumentauswertungen:
  - ∗ **Normale Auswertung** ist theorie-relevant, nicht jedoch implementierungspraktisch (sicher aber nicht straightforward) (dabei wird kein Argument wird ausgewertet, dessen Wert nicht benötigt wird)

    - **Applikative Auswertungsordnung** ist eher straightforward, aber terminiert weniger als Normale Auswertung oder (Jedes Argument wird exakt einmal ausgewertet, auch die, die nicht benötigt werden)

  - Fazit:
    Beste beider Welten: Früh, fleißig, applikativ, wo möglich; spät, faul, wo nicht.

- **Steuerung: Früh(artig)e und späte Auswertung in Haskell**
  - Standardverfahren mit: *fac (2*(3+5))*
  - **Frühartigkeit** erzwingbar mithilfe des zweistelligen Operators ($!): *fac $! (2*(3+5))* ⇒ fac $! 16 ⇒ ...

Kapitel 14: Typprüfung, Typinferenz

monomorph, polymorph, mit Typklassen, Unifikation,...

- **Motivation**
  - **Aufteilung typisierte Programmiersprachen:**
    - schwach getypte Sprachen (Typprüfung zur Laufzeit)
    - stark getypte Sprachen (Typprüfung/-inferenz zur Übersetzungszeit)
    - In **Haskell** gilt für Typen wohlgetypter Ausdrücke:
      - **Monomorph**: *fac :: Int → Int*
      - **Parametrisch polymorph** (uneingeschränkt polymorph) *length :: [a] → Int*
      - **Ad hoc polymorph** (eingeschränkt polymorph durch Typkontexte): *elem :: Eq a ⇒ a → [a] → Bool*
    - **Automatische Typinferenz mit Hugs**

- **Monomorphe Typprüfung**
  - Am Ende monomorpher Typprüfung steht **als Ergebnis ein Ausdruck**, der **wohlgetypt** (eindeutig bestimmten konkreten Typ) bzw. **nicht wohlgetypt** (hat überhaupt keinen Typ) ist.
  - Typprüfung durch Kontextauswertung

- **Polymorphe Typprüfung**
  - Am Ende monomorpher Typprüfung steht **als Ergebnis ein Ausdruck**, der **wohlgetypt** (**einen, mehrere, möglicherweise unendlich viele konkrete Typen.**) bzw. **nicht wohlgetypt** (hat überhaupt keinen Typ) ist.

- Typprüfung durch **Lösen von Typkontextsystemen** unter **Unifikation** von Typausdrücken (⇒ Aufrufkontexten ergeben konkreteren Typ, also die Bestimmung der allgemeinst möglichen Typausdrücke)
- **Unifikation** bei Typprüfung v. Fkt.-Termen = informell gesprochen, dass zwei Ausdrücke unifiziert werden, d.h. sie müssen nicht direkt gleichen Typs sein, sondern nur ihre Auswertung muss letztendlich den gleichen Typ ergeben.

- **Typsysteme, Typinferenz**

  - **Typsysteme**: Sind **logische Systeme**, die uns erlauben, **Aussagen** der Form '*exp ist Ausdruck vom Typ t*' zu **formalisieren** und sie mithilfe von **Axiomen** und **Regeln des Typsystems** zu beweisen.
    Anders gesprochen: Ich habe ein logisches System an Axiomen und Regeln, mit denen ich Aussagen formalisieren kann (auf bewiesener Grundlage innerhalb dieses Systems).
  - **Typinferenz**: Bezeichnet den Prozess, den Typ eines Ausdrucks automatisch mithilfe der Axiome und Regeln des Typsystems abzuleiten.

  - **Haskell ist stark typisiert** (Fehler zur Laufzeit aufgrund von Typfehlern sind deshalb ausgeschlossen.)

Teil 6 (Weiterführende Konzepte)

Kapitel 15: Interaktive Programme: Ein-/Ausgabe

- ### Motivation
  - **Problem:** ...Dialog, Interaktion zwischen Benutzer und Programm findet (bisher) nicht statt ⇒ Achtung: Seiteneffekte
  - Achtung: Verlust von **referentieller Transparenz** (= ein Ausdruck kann mit seinem Wert ersetzt werden, ohne das Verhalten des Programms zu ändern.)
    ⇒ wichtiges Prinzip funktionaler (allg. deklarativer) Programmierung: **Die Betonung des 'was' (Ergebnisse) statt des 'wie' (die Art ihrer Berechnung)**

- ### Haskells Lösung
  - Konzeptuell wird in Haskell ein Programm geteilt in einen **rein funktionalen Berechnungskern** und einen **imperativähnliche Dialog- und Interaktionsschale.**
  - Umsetzung:
    (1) (vordef.) polymorpher Datentyp für Ein-/Ausgabe: **data IO a = ...**
    IO-Werte bleiben in der Schale und können nicht den rein funktionalen Kern "kontaminieren"
    (2) Vordefinierte primitive E/A-Operationen:
    getInt :: IO Int etc. bzw.
    putInt :: Int → IO () etc.
    (3) Operator zur Komposition von E/A-Operationen: **(»=) :: IO a → (a → IO b) → IO b**
    (4) Zwei Vermittlungs'operatoren' zwischen Schale (←, IO) und Kern (return, a):
    return :: a → IO a (**a** = **Kern** nach **IO a** = **Schale**) **Kontamination reiner Werte nach außen**
    '← :: IO a → a' (informell) **Dekontamination reiner Werte nach innen**
  - Aktionen: **Lesen** einer (Zeichenreihen-) Zeile vom Bildschirm (**getLine**), **Schreiben** einer Zeichenreihe auf den Bildschirm (**putStr**): Beipiel:

```
getLine :: IO String
putStr :: String →  IO ()
```

– statt IO-Kompositionsoperatoren (»=) und (») ist in Haskell die **do-Notation** praktischer (Aktionen kann man mit Hilfe von (»=) kombinieren, dann liegt eine komponierte Aktion vor)
kurz: IO-Komposition (»=) heißt ... getLine »= putLine Str

– dabei gilt:
  * **binde-dann-Operator (engl. bind oder then):**(»=) :: IO a → (a → IO b) → IO b
  * **dann-Operator (engl. sequence):** (») :: IO a → IO b → IO b
    (insgesamt ist (») nur von (»=) abgeleitet, wobei mit (aktion1 » aktion2) es ausdrückt, das Ergebnis von aktion1 in aktion2 nicht als Argument genutzt wird/aktion2 ignoriert Ergebnis von aktion1)

```
statt:
putStr "hello world"

kann man schreiben:
do {putStr "hello"; putStr "world"}

komplexer, statt (»=)-Komposition: (hier mit geschachtelten λ-Ausdrücken)
action1 »= ( x1 → action2 »= ( x2 → mk_action3 x1 x2 ))

einfacher mit do-Notation:
do { x1 ← action1
   ; x2 ← action2
   ; mk_action3 x1 x2 }
```

– Aktion = (1) E/A-Operation (prozedural) + (2) Wertlieferung (funktional) = **wertliefernde E/A-Operation**

– (Informell:) 'do' entspricht '(»=) plus anonyme λ-Abstraktion'.

– weiteres Beispiel (in ghci direkt programmiert)

```
direkt in ghci:
C:\Users\Name> ghci
GHCi, version 9.0.1: https://www.haskell.org/ghc/ :? for help
ghci> :{
   putStrLn_4mal :: String → IO ()
   putStrLn_4mal str = do putStrLn str; putStrLn str; putStrLn str; putStrLn str
:}
putStrLn_4mal "hallo"

Ausgabe: hallo hallo hallo hallo
Anmerkung :{ ... :} für Multiline-Anweisungen
```

  - **Leseaktionen mit (IO a)** und **Schreibaktionen mit (IO ()) mit ()** also Nulltupeltyp

- **'Iteration' vs. Rekursion**
  - **Einmal-Wertvereinbarungsoperator :** mit $\leftarrow$ (über das gesamte Programm) dauerhaft
  - **Mehrfach-Wertzuweisungsoperator:** mit := (temporäre Wertzuweisung, voriger Wert wird zerstört und so überschrieben $\Rightarrow$ destruktive Zuweisung)

Kapitel 16: Robuste Programme: Fehlerbehandlung

- **Überblick, Orientierung**
  - Typische Fehlersituationen und Sonderfälle: Division durch 0 (*div 1 0*), Zugriff auf erstes Element einer leeren Liste (*head* [ ])
  - typischer Sonderfall: Auseinanderfallen von intendiertem und implementiertem Definitionsbereich einer Funktion $\Rightarrow$ welcher Umgang
  - 3 Möglichkeiten eines **sukzessive systematisch(er)en Umgangs**
    - Panikmodus
    - Auffangwerte (Funktionsspezifisch, Aufrufspezifisch)
    - Fehlertypen, Fehlerwerte, Fehlerfunktionen

- **Panikmodus**
  - Ziel: **Fehler und Fehlerursache melden** und **Fehlerhafte Programmauswertung stoppen.**
  - *otherwise = error "Ungültige Eingabe."*

- **Auffangwerte**
  - Ziel: **Panikmodus vermeiden** und **Programmlauf nicht zur Gänze abbrechen, sondern Berechnung möglichst sinnvoll fortführen.**
  - **Funktionsspezifische** (bspw. Rückgabewert -1)
  - **Aufrufspezifische** (Argumentwert einfach wieder ausgeben)
    - **Erweiterung:** Fehlerwert bei Funktionsaufruf mitgeben (**Erweiterung der Signatur**)
    - **fehlerbehandelnde Hüllfunktion** als eine Art Wrapper, in der sicher dann die eigentliche Funktion aufgerufen wird
  - **Auffangwerte (engl. default values) zur Weiterrechnung im Fehlerfall.**

- **Fehlertypen, Fehlerwerte, Fehlerfunktionen**

  - Ziel: systematisch **Erkennen-Anzeigen-Behandeln** von Fehlersituationen
  - **Werkzeuge:** Fehlertypen, Fehlerwerte, Fehlerfunktionen (statt schlichter Auffangwerte)
  - Werkzeuge bspw.:
    * Typ a zum (Fehler-) Datentyp Maybe a: **data Maybe a = Just a | Nothing deriving (...)** (Nothing hier expliziter Fehlerwert)
    * Hüllfunktion mit (Maybe b)-Datentyp als Ergebnis, so kann man auch Fehlerwert **Nothing** produzieren und gewünschten Ergebnistyp explizit definieren (**Just** (f u))

    * Was mit **Nothing**-Fehlertyp machen ⇒ weiterreichen an eine Fehlerfunktion (Signatur: *map_ Maybe :: (a → b) → Maybe a → Maybe b*)

Kapitel 17: Programmierung im Großen: Module

- **Überblick, Orientierung**

  - **Modularisierung:** Zerlegung von Programmen in überschaubare, (oft) getrennt übersetzbare Programmeinheiten als wichtige programmiersprachliche Unterstützung der Programmierung im Großen.
  - Zwei wichtige Eigenschaften (für gute Modularisierung):
    * **(starke) Kohäsion:** modullokal, intramodular, inneren Zusammenhang von Modulen, Art und Typ der in einem Modul zusammengefassten Funktionen. (Funktionale Kohäsion: Fkt gleicher Funktionalität zsmfassen) (Datenkohäsion: Fkt, die auf gleichen Datenstrukturen arbeiten)
    * **(schwache) Koppelung:** modulübergreifend, intermodular, äußeren Zusammenhang von Modulen (Export/Import), Datenaustausch (Schwache funktionale Koppelung)(Feste Datenkoppelung: Ergebnisse einer Fkt. werden Argumente anderer Fkt.)
  - Unterstützung des Geheimnisprinzips (Schnittstelle Import/Export)

- **Haskells Modulkonzept**

  - **Moduldateien:** ***module M where*** gefolgt von **Deklaration von Typen/Typklassen/Funktionen**
  - **Import:** Modul M2 importiert aus Modul M1 alle (global sichtbaren) Bezeichner und Definitionen, die danach in M2 verwendet werden können. Bzw. ausschließlich die explizit genannten Bezeichner und Definitionen (D_1 von M1, ...):

```
module M2 where
import M1 (D_1 (..), D_2, T_1, C_1 (..), C_2, f_5)

hier wird alles importiert außer das explizit Genannte:
module M3 where
import M1 hiding (D_1, T_2, f_1)
```

- **Export:** Nicht selektiver Export (einfach „normal“, ohne irgendetwas anzugeben) und selektiver Export wie folgt:

```
hier wird alles exportiert:
module M1 where
...

hier wird nur das Genannte exportiert:
module M1 (D_1 (..), D_2, D_3 (Dc_1,...,Dc_k), C_1 (..) where

data D_1 ... = ...
...
```

- **Reexport:** wenn M2 ← M1 (M2 import aus M1) und M3 ← M2, so erhält M3 aber nicht die von M2 aus M1 importierten Namen (kein automatischer Reexport)
  händischer Reexport: *module M2 (module M1,f_M2_j) where ...*

- **Namenskonflikte, Umbenennungen, Konventionen**
  - **qualifizierter Import: import qualified M1** und Verwendung **M1.f**
  - **Umbenennung von Modulnamen:** *import qualified M1 as MyLocalNameForM1*

- **Modul-Anwendung: Abstrakte Datentypen**
  - **Konkrete Datentypen (KDT)** (z.B. Algebr. Datentypen) versus **Abstrakte Datentypen (ADT)** (Geheimnisprinzip)
  - ADT (grundlegende Idee): Schnittstellenfestlegung (öffentlich), Verhaltensfestlegung (öffentlich), Implementierung (nicht öffentlich, hier KDT)
  - Ziel ADT (Datenabstraktion) = Kapselung von Daten + Geheimnisprinzip/information hiding (ADT-Schnittstelle bekannt, KDT-Implementierung verborgen)

Kapitel 18: Allgemeine und fkt. Programmierprinzipien

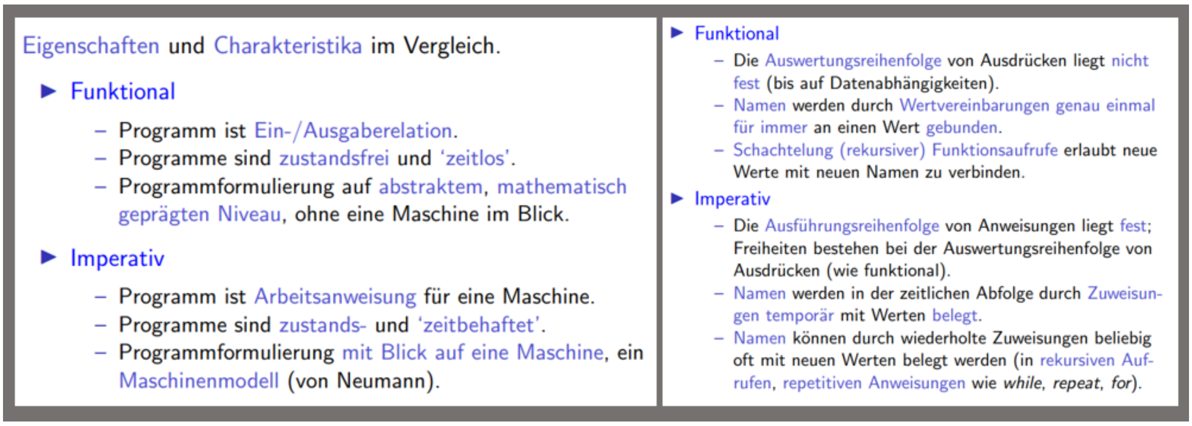
- **Überblick, Orientierung**
  - **Allgemein:** Stetes Hinterfragen u. Anpassen seines Tuns
  - **Fkt. Programmierprinzipien:**
    - Kapseln algorithmischen Vorgehens (Fkt. höherer Ordnung): Divide et impera (top-down Problemzerstückelung)...Probleme auf Fkt. aufteilen und in einer Synthesefkt. wieder zusammenfügen (Fkt. höherer Ordnung)
    - Problemorientiertes Modularisieren (späte Auswertung):
  - **Generatormodule:** ermöglicht neue, problemorientierte Modularisierungen
    (Generiere/selektiere- (G/S-) Modularisierungen,
    Generiere/filtere- (G/F-) Modularisierungen,
    Generiere/transformiere- (G/T-) Modularisierungen, ...)
    d.h., dass man eine Generator-Fkt hat und danach Fkt., die selektieren/filtern/transformieren/...
  - **Rechnen mit Strömen** (fibs = map fib [0..]) bzw.:

    Generator der Fibonacci-Zahlen:
    ```
    fibs :: [Integer]
    fibs = 0 : 1 : zipWith (+) fibs (tail fibs)
    ```
  - **Generatoranwendungen und Hilfsfunktionen**

Teil 7 (Abschluss)

Kapitel 19: Rückschau, Ausschau

- ### Rückschau, Rückblick
  - Funktionale versus imperative Programmierung



Eigenschaften und Charakteristika im Vergleich.

- Funktional
  - Programm ist Ein-/Ausgaberelation.
  - Programme sind zustandsfrei und 'zeitlos'.
  - Programmformulierung auf abstraktem, mathematisch geprägten Niveau, ohne eine Maschine im Blick.
- Imperativ
  - Programm ist Arbeitsanweisung für eine Maschine.
  - Programme sind zustands- und 'zeitbehaftet'.
  - Programmformulierung mit Blick auf eine Maschine, ein Maschinenmodell (von Neumann).

- Funktional
  - Die Auswertungsreihenfolge von Ausdrücken liegt nicht fest (bis auf Datenabhängigkeiten).
  - Namen werden durch Wertvereinbarungen genau einmal für immer an einen Wert gebunden.
  - Schachtelung (rekursiver) Funktionsaufrufe erlaubt neue Werte mit neuen Namen zu verbinden.
- Imperativ
  - Die Ausführungsreihenfolge von Anweisungen liegt fest; Freiheiten bestehen bei der Auswertungsreihenfolge von Ausdrücken (wie funktional).
  - Namen werden in der zeitlichen Abfolge durch Zuweisungen temporär mit Werten belegt.
  - Namen können durch wiederholte Zuweisungen beliebig oft mit neuen Werten belegt werden (in rekursiven Aufrufen, repetitiven Anweisungen wie *while*, *repeat*, *for*).

Abbildung 13: Kap. 19/46,47, Funktionale versus imperative Programmierung