Hochschule RheinMain

Funktionale Programmierung

03 Haskell

Prof. Dr. David Sabel
Sommersemester 2025

Stand der Folien: 20. Mai 2025

Ziele des Kapitels

Hochschule RheinMain

- Übersicht über die wesentliche Konstrukte von Haskell
- Werkzeuge und Haskell, Typen, Zahlen und einfache Typen, Datentypen: Aufzählungstypen, Produkttypen, Summentypen, Record-Syntax, Funktionen, Module

Hochschule RheinMain

HASKELL UND WERKZEUGE

Haskell und Werkzeuge

Hochschule RheinMain

Haskell `http://www.haskell.org`

- ist eine rein funktionale Sprache mit verzögerter Auswertung.
- benannt nach dem US-amerikanischen Logiker Haskell B. Curry (1900-1982).
- Standards: Haskell 98 und Haskell 2010
- Interpreter / Compiler: GHCi bzw. GHC (Glasgow / Glorious Haskell Compiler)
- Dokumentation des GHC:
  `http://downloads.haskell.org/ghc/latest/docs/html/users_guide/`
- Dokumentation der Standardbibliotheken:
  `http://www.haskell.org/ghc/docs/latest/html/libraries/index.html`

GHC / GHCi installieren

Hochschule RheinMain

Verschiedene Varianten zur Installation:

- Am einfachsten die Ratschläge und Anweisungen unter `https://www.haskell.org/get-started` befolgen
- Einfachste Installation mit ghcup und damit dann ghc, cabal, stack installieren
- Stack: Projektbasiertes Build-Tool, automatisches Management von Abhängigkeiten

GHCi verwenden

Hochschule RheinMain

Interpreter starten und Ausdrücke eingeben und auswerten lassen:

```
> ghci
GHCi, version 9.4.8: https://www.haskell.org/ghc/  :? for help
ghci> 1+2
3
ghci> 10 * (2 + 4)
60
ghci> 14 `mod` 6
2
```

Mit den Pfeiltasten  ,  kann man die vorherigen Eingaben durchblättern

GHCi verwenden

Hochschule RheinMain

Steuerung des Interpreters:

- Befehle, die mit `:` beginnen, z.B. `:?` für die Hilfe.
- Befehle abkürzen, z.B. statt `:quit` auch `:q`, um den Interpreter zu verlassen

Quelltexte laden:

```
:l datei.hs -- lade Definition aus Datei datei.hs
:r          -- erneut alle offenen Dateien einlesen
```


Quellcode-Dateien

Hochschule RheinMain

Datei-Endung:

- Normale Haskell-Quellcodedateien enden mit `.hs`
- Literate-Haskell-Dateien mit `.lhs`

Normale Haskell-Quellcodedateien:

- Alles ist Quellcode, außer Kommentare
- `--␣` leitet Kommentar bis zum Zeilenende ein
- Mehrzeiliger (geklammerter) Kommentar mit `{-` und `-}`

Literate-Haskell-Programm

- Alles ist Kommentar, außer es ist als Quellcode gekennzeichnet
- Bird-Stil (nach Richard Bird): Code-Zeilen beginnen mit `>␣` und Code- und Kommentarzeilen sind durch Leerzeile getrennt.
- LaTeX-Stil: Code steht in `\begin{code} ... \end{code}`-Umgebung.

Beispiel: Normales Haskell

Hochschule RheinMain

```
{- Hier steht beliebiger Kommentar, der
   sich auch über mehrere Zeile erstrecken
   kann.
-}
meineFunktion [] = putStrLn "Hallo Welt"
meineFunktion (_:rs) = do
                       putStrLn "Hallo Welt"
                       meineFunktion rs

-- Hier steht erneut ein Kommentar.
```


Beispiel: Literate-Haskell im Bird-Stil

Hochschule RheinMain

```
Hier steht beliebiger Kommentar, der
sich auch über mehrere Zeile erstrecken
kann.

> meineFunktion [] = putStrLn "Hallo Welt"
> meineFunktion (_:rs) = do
>                        putStrLn "Hallo Welt"
>                        meineFunktion rs

Hier steht erneut ein Kommentar.
```


Beispiel: Literate-Haskell im LaTeX-Stil

Hochschule RheinMain

```
Hier steht beliebiger Kommentar, der
sich auch über mehrere Zeile erstrecken
kann.
\begin{code}
meineFunktion [] = putStrLn "Hallo Welt"
meineFunktion (_:rs) = do
                       putStrLn "Hallo Welt"
                       meineFunktion rs
\end{code}
Hier steht erneut ein Kommentar.
```


Allgemeine Syntaxregeln

Hochschule RheinMain

- Haskell beachtet die Groß-/Kleinschreibung
- Haskell ist „whitespace“-sensitiv: Einrückungen werden beachtet
- Veränderungen an Leerzeichen, Tabulatoren und Zeilenumbrüchen können Fehler verursachen.
- Statt Einrückung kann man auch geschweifte Klammern `{ }` und Semikolons `;` verwenden.

Dringende Empfehlung:

Keine Tabulatoren verwenden, sondern (am besten durch den Text-Editor) durch Leerzeichen ersetzen.

Editoren und IDEs: `https://wiki.haskell.org/IDEs`.

Allgemeine Syntaxregeln (2)

Hochschule RheinMain

Daumenregel für Einrückungen:

Beginnt die nächste Zeile in einer Spalte

- weiter rechts als die vorherige, dann geht die vorherige Zeile weiter
- in der gleichen Spalte wie die vorherige, dann kommt das nächste Element des Blocks
- weiter links als die vorherige, dann ist der Block beendet.

Falsch:

```
fun x = let x = 2
          y = 3
        in x+y
<interactive>: parse error on input ‘y’
```

Falsch:

```
fun x = let x = 2
              y = 3
         in x+y
<interactive>: parse error on input ‘=’
```

Richtig:

```
fun x = let x = 2
            y = 3
        in x+y
```


TYPEN IN HASKELL

Hochschule RheinMain

Typen

Hochschule RheinMain

- **Typ** = Menge von Werten
- Z.B. `Int` = Menge der ganzen Zahlen von -2^{63} bis $2^{63}-1$.
- In Haskell:
  - Typnamen beginnen mit einem Großbuchstaben
  - Typvariablen beginnen mit einem Kleinbuchstaben.
- Im GHCi: `:type` *Ausdruck* zeigt Typ an

```
ghci> :type 'a'
'a' :: Char
```


Typen: Sprechweisen

Hochschule RheinMain

Gilt e `::` T, so sagt man

- „Ausdruck e hat Typ T“
  oder alternativ
- „Ausdruck e ist vom Typ T“

Syntax von Typen

Hochschule RheinMain

Die Syntax von Typen (ohne Typklassenbeschränkungen) in Haskell ist

$$\mathbf{T} ::= TV \mid TC\ \mathbf{T}_1\ \ldots\ \mathbf{T}_n \mid \mathbf{T}_1\ \texttt{->}\ \mathbf{T}_2$$

wobei TV für eine Typvariable steht und TC ein Typkonstruktor mit Stelligkeit n ist

- Typkonstruktoren beginnen mit einem Großbuchstaben
  (wenn sie infix verwendet werden, mit einem Doppelpunkt)
- $\mathbf{T}_1$ -> $\mathbf{T}_2$ ist ein **Funktionstyp**, d.h. Typ jener Funktionen,
  - deren Eingabe Werte des Typs $\mathbf{T}_1$ sind, und
  - die als Ausgabe Werte des Typs $\mathbf{T}_2$ liefern.

Sprechweisen für verschiedene Typen

Hochschule RheinMain

Basistypen: nullstellige Typkonstruktoren
(Basistypen haben keine Typvariablen und sind nicht zusammengesetzt aus anderen Typen)

Z.B. `Int`, `Bool`

Grundtypen / **monomorphe Typen**: Typen, die keine Typvariablen enthalten.
zusammengesetzte Typen sind erlaubt

Z.B. `Baum Int`, `[(Int,Bool)]`, `Int -> Bool`

Polymorphe Typen: Typen, die auch Typvariablen enthalten dürfen

Z.B. `[a]`, `a -> b -> b`

Es gilt: Basistypen $\subseteq$ Grundtypen $\subseteq$ polymorphe Typen

Hochschule RheinMain

ZAHLEN UND EINFACHE TYPEN IN HASKELL

Haskell: Zahlen

Hochschule RheinMain

Eingebaut sind:

- Ganze Zahlen beschränkter Größe: `Int` (mindestens von -2^{29} bis $2^{29}-1$)
- Ganze Zahlen beliebiger Größe: `Integer`
- Gleitkommazahlen: `Float`
- Gleitkommazahlen mit doppelter Genauigkeit: `Double`
- Rationale Zahlen: `Rational`

  (verallgemeinert `Ratio` α, wobei `Rational = Ratio Integer`)

  Darstellung x `%` y (Modul `Data.Ratio` importieren)

Arithmetische Operationen

Hochschule RheinMain

Rechenoperationen:

- `+` für die Addition
- `-` für die Subtraktion
- `*` für die Multiplikation
- `/` für die Division
- `mod` , `div`

Die Operatoren sind überladen. Dafür gibt es Typklassen.

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

Genaue Behandlung von Typklassen: später

Funktionsnamen / Operatoren

Hochschule RheinMain

- Namen von **Funktionen** müssen in Haskell mit einem Kleinbuchstaben oder _ beginnen. (z.B. `mod`, `div`)
- **Operatoren** bestehen nur aus Symbolen, die keine alphanumerischen Zeichen sind.
  (z.B. +, $, <*>)
- Operatoren werden standardmäßig infix verwendet,
  während Funktionen standardmäßig präfix verwendet werden.
- Operatoren, die mit : beginnen, spielen eine Sonderrolle, da sie Datenkonstruktoren sind

Präfix / Infix

Hochschule RheinMain

Anmerkung zum Minuszeichen:

- Mehr Klammern als man denkt: `5 + -6` geht nicht,
  richtig: `5 + (-6)`
- In Haskell können Funktionen durch Einfassen in Backquotes ‘ auch infix benutzt werden:
  `mod 5 6` ; infix durch Backquotes: `5 ‘mod‘ 6`
- Umgekehrt können infix-Operatoren durch Einklammern in () auch präfix benutzt werden:
  `5 + 6` ; präfix durch Klammern: `(+) 5 6`

Weitere Typen

Hochschule RheinMain

`Bool` Boolesche (logische) Wahrheitswerte: `True` und `False`

`Char` Unicode Zeichen, z.B. `'q'`. Diese werden immer in Apostrophen eingeschlossen.

`String` Zeichenketten, z.B. `"Hallo!"`. Diese werden immmer in Anführungszeichen eingeschlossen.

Davon ist lediglich `Char` eingebaut, denn:

```
type String = [Char]      -- Typsynonym
data Bool = True | False
```

Einige vordefinierte Funktionen

Hochschule RheinMain

Die Standardbibliothek (die sogenannte Prelude) definiert u.a. folgende Funktionen

- `&&` Konjunktion, logisches Und
- `||` Disjunktion, logisches Oder
- `not` Negation, Verneinung
- `==` Test auf Gleichheit
- `/=` Test auf Ungleichheit

```
ghci> (True || False) && (not True)
False
ghci> True == False
False
ghci> False /= True
True
```


Vergleichsoperatoren

Hochschule RheinMain

- `==` für den Gleichheitstest
  `(==) :: (Eq a) => a -> a -> Bool`
- `/=` für den Ungleichheitstest
- `<`, `<=`, `>`, `>=`, für kleiner, kleiner gleich, größer und größer gleich
  (der Typ ist `(Ord a) => a -> a -> Bool`.

Assoziativitäten und Prioritäten

Hochschule RheinMain

```
infixr 9  ∘
infixr 8  ^, ^^, **
infixl 7  *, /, ‘quot‘, ‘rem‘, ‘div‘, ‘mod‘
infixl 6  +, -

-- The (:) operator is built-in syntax, and cannot
-- legally be given a fixity declaration; but its
-- fixity is given by:
--   infixr 5  :

infix  4  ==, /=, <, ≤, >=, >
infixr 3  &&
infixr 2  ||
infixl 1  >>, >>=
infixr 1  =≪
infixr 0  $, $!, ‘seq‘
```


Hochschule RheinMain

FUNKTIONEN IN HASKELL

Funktionen

Hochschule RheinMain

Definition (Funktion)

Eine partielle Funktion $f : A \to B$ ordnet einer Teilmenge $A' \subset A$ einen Wert aus B zu, und ist ansonsten undefiniert.
Wir bezeichnen A als Quellbereich, A' als Definitionsbereich und B als Zielbereich. Für totale Funktionen gilt $A = A'$.

Im allgemeinen: Programmiere totale Funktionen, z.B. durch Verwendung von `error` für die undefinierten Fälle.

Funktionen in Haskell

Hochschule RheinMain

Funktionen werden durch Gleichungen definiert:

```
double1 x = x + x              -- Funktion mit 1 Argument
fun x y z = x + y * double1 z          -- mit 3 Argumenten
add       = λx y -> x + y      -- Anonyme Fkt. 2 Argumente

twice f x = f x x              -- Higher-order function
double2 y = twice (+) y
double3   = twice (+)          -- Partielle Applikation
```

- In Quellcode-Datei: Alle Top-Level Definition in gleicher Spalte
- Funktionsanwendung durch Leerzeichen: `foo 1 2 3`.
  Klammerung ist links-assoziativ: `foo 1 2 3` = `(((foo 1) 2) 3)`

Funktionstypen

Hochschule RheinMain

```
foo :: (Int,Int) -> (Int,Int)
foo (x,y) = (x+y,x-y)

bar :: Int -> (Int -> (Int,Int))
bar x y = (x+y,x-y)
```

- Klammerkonvention: Funktionstypen implizit rechtsgeklammert.
  `Int -> Int -> Int` äquivalent zu `Int -> (Int -> Int)` und verschieden zu `(Int -> Int) -> Int`.
- In Haskell darf man partiell anwenden.
  Z.B. ist `(bar 1)` eine Funktion des Typs `Int -> (Int,Int)`
- Funktionen sind also normale Werte in einer funktionalen Sprache.

Funktionsdefinition in Haskell

Hochschule RheinMain

funktionsName :: Typ_1 -> Typ_2 -> ... -> Typ_n -> *Ergebnistyp*
funktionsName var_1 var_2 ... var_n = *expr*

- *funktionsName* beginnt mit einem Kleinbuchstabe oder _
- Typdeklaration ist optional
- var_1, var_2, ..., var_n sind die formalen Parameter
- *expr* ist der Funktionsrumpf.
- Anstelle von Variablen können auch Pattern verwendet werden.
- Mehrere Definitionszeilen sind möglich, erste passende wird genommen, Abarbeitung von oben nach unten (analog bei Guards).

Beispiele

Hochschule RheinMain

Test, ob Zahl gerade ist

```
even 0 = True
even 1 = False
even x = if x < 0 then even (abs x) else even (x-2)
```

Falsch:

```
even' x = even' (x-2)
even' 0 = True
even' 1 = False
```

Richtig:

```
even'' 0 = True
even'' 1 = False
even'' x
  | x > 1 = even'' (x-2)
even'' x = even'' (abs x)
```


Anonyme Funktionen

Hochschule RheinMain

- Anonyme Funktion = Funktion ohne Namen (auch Abstraktion genannt)
- Herkunft aus dem Lambda-Kalkül (Alonzo Church, 1936)
- Abstraktion im Lambda-Kalkül $\lambda x_1, \ldots, x_n.e$
- In Haskell: \ statt λ, Leerzeichen statt , und -> statt .

Beispiele:

	Lambda-Kalkül	Haskell
Identitätsfunktion	$\lambda x.x$	`\x -> x`
Nachfolgerfunktion	$\lambda x.x+1$	`\x -> x+1`
Wurzelfunktion	$\lambda y.\sqrt{y}$	`\y -> sqrt y`
Dreistelliges Plus	$\lambda x, y, z.x+y+z$	`\x y z -> x + y + z`

Ausdrücke

Hochschule RheinMain

Ausdrücke werden gebildet durch:

- Anwendungen
- `if-then-else`-Ausdrücke:
  `if` *Ausdruck* `then` *Ausdruck* `else` *Ausdruck*
- `case`-Ausdrücke zum Zerlegen von Daten:
  `case` *Ausdruck* `of` *Alternativen*
  erläutern wir später genauer!

Ausdrücke (2)

Hochschule RheinMain

- `let`-Ausdrücke der Form
  `let` *Bindungen* `in` *Ausdruck*
  wobei
  - *Bindungen* sind Bindungen der Form
    Funktionsname $par_1 \ldots par_n$ = *Ausdruck*
  - Die Bindungen in `let`-Ausdrücken sind rekursiv: Gültigkeitsbereich der Funktionsnamen sind alle rechten Seiten der Bindungen sowie der `in`-Ausdruck.
- `do`-Notation: Diese erläutern wir später
- `seq`-Ausdrücke: `seq` e_1 e_2 Semantik: Wenn e_1 terminiert ist der Wert e_2. Operational: Erst e_1, dann e_2 auswerten.
- Datenkonstruktoren, Listen, List-Comprehensions

where-Klauseln

Hochschule RheinMain

Beispiel:

```
sumprod' 1 = (1,1)
sumprod' n = (s'+n,p'*n)
  where (s',p') = sumprod' (n-1)
```

Das where ist gültig für die Funktionsdefinition kann und daher für alle Guards wirken, z.B.

```
f x
 | x == 0    = a
 | x == 1    = a*a
 | otherwise = a*f (x-1)
 where a = 10
```

Falsch:

```
f x = λy -> mul
   where mul = x * y
```


Anmerkung zum $-Operator

Hochschule RheinMain

Definition:

```
f $ x = f x
```

wobei Priorität ganz niedrig, z.B.

```
f1 5 $ f2 10 $ f4 (5*3)
```

wird als

```
f1 5 (f2 10 (f4 (5*3)))
```

geklammert

Hochschule RheinMain

ALGEBRAISCHE DATENTYPEN

Aufzählungstypen

Hochschule RheinMain

Aufzählungstyp = Aufzählung verschiedener Werte

data Typname = Konstante$_1$ | Konstante$_2$ | ... | Konstante$_N$

Beispiele:

```
data Bool      = True | False

data Wochentag = Montag | Dienstag | Mittwoch | Donnerstag
               | Freitag | Samstag | Sonntag
  deriving(Show)
```

deriving(Show) erzeugt Instanz der Typklasse Show,
damit der Datentyp angezeigt werden kann.

Aufzählungstypen (2)

Hochschule RheinMain

```
istMontag :: Wochentag -> Bool
istMontag x = case x of
                Montag -> True
                Dienstag -> False
                Mittwoch -> False
                Donnerstag -> False
                Freitag -> False
                Samstag -> False
                Sonntag -> False
```

Mit Default-Alternative:

```
istMontag' :: Wochentag -> Bool
istMontag' x = case x of
                Montag -> True
                y      -> False
```


Aufzählungstypen (3)

Hochschule RheinMain

In Haskell:

Pattern-matching in den linken Seiten der Funktionsdefinition:

```
istMontag'' :: Wochentag -> Bool
istMontag'' Montag = True
istMontag'' _      = False
```


Produkttypen

Hochschule RheinMain

Produkttyp = Zusammenfassung verschiedener Werte

Bekanntes Beispiel: Tupel

Definition (Kartesisches Produkt)

Sind $A_1, \ldots A_n$ Mengen, so ist das kartesische Produkt definiert als
$A_1 \times \cdots \times A_n = \{(a_1, \ldots, a_n) \mid a_i \in A_i \text{ für } i = 1 \ldots n\}$

Die Elemente von $A_1 \times \cdots \times A_n$ heißen allgemein n-*Tupel*, spezieller auch Paare, Tripel, Quadrupel, ...

Produkttypen (2)

Hochschule RheinMain

In Haskell: Tupelausdrücke und -typen mit runden Klammern und Kommas, z.B. $\mathbb{Z} \times \mathbb{Z}$ wird zu `(Int,Int)`.

```
ghci> :t (True, 'a', 7)
(True, 'a', 7) :: (Bool, Char, Int)
ghci> :t (4.5,"Hi!")
(4.5,"Hi!") :: (Double, String)
```

Das 0-Tupel ist ebenfalls in Haskell erlaubt: `() :: ()`.
Der Typ `()` wird als Unit-Typ bezeichnet und hat nur den einzigen Wert `()`.

Produkttypen (3)

Hochschule RheinMain

Mit `data` können auch Produkttypen definiert werden (die keine Tupelsyntax verwenden).

Die Syntax hierfür ist

`data Typname = KonstruktorName` $\text{Typ}_1\ \text{Typ}_2 \ldots \text{Typ}_N$

Beispiel:

```
data Student = Student
                 String  -- Name
                 String  -- Vorname
                 Int     -- Matrikelnummer
```

Beachte: `Student` taucht hier auf als Typ und als Datenkonstruktor

Produkttypen (3)

Hochschule RheinMain

```
setzeName :: Student -> String -> Student
setzeName x name' =
 case x of
  (Student name vorname mnr)
           -> Student name' vorname mnr
```

Alternativ mit Pattern auf der linken Seite der Funktionsdefinition:

```
setzeName :: Student -> String -> Student
setzeName (Student name vorname mnr) name' =
  Student name' vorname mnr
```


Produkttypen und Aufzählungstypen

Hochschule RheinMain

Man kann beides mischen, Z.B.

```
data DreiDObjekt =
      Wuerfel Int
    | Quader Int Int Int
    | Kugel Int
```

Früchte:

```
data Frucht = Apfel (Int,Double)
            | Birne Int Double
            | Banane Int Int Double
```

Allgemein nennt man dies auch **Summentyp**, allgemein

`data Typ =` Typ_1 `|` ... `|` Typ_n

wobei $\text{Typ}_1,\ldots,\text{Typ}_n$ komplexe Typen sind

Record-Syntax: Einführung

Hochschule RheinMain

```
data Student = Student
                 String  -- Vorname
                 String  -- Name
                 Int     -- Matrikelnummer
```

Nachteil: Nur die Kommentare verraten, was die Komponenten darstellen.

Außerdem mühsam: Zugriffsfunktionen erstellen:

```
vorname :: Student -> String
vorname (Student vorname name mnr) = vorname
```

Record-Syntax: Einführung (2)

Hochschule RheinMain

Änderung am Datentyp:

```
data Student = Student
                 String  -- Vorname
                 String  -- Name
                 Int     -- Matrikelnummer
                 Int     -- Hochschulsemester
```

muss für Zugriffsfunktionen nachgezogen werden

```
vorname :: Student -> String
vorname (Student vorname name mnr hsem) = vorname
```

Abhilfe verschafft die Record-Syntax

Record-Syntax in Haskell

Hochschule RheinMain

Student mit Record-Syntax:

```
data Student =  Student {
                          vorname        :: String,
                          name           :: String,
                          matrikelnummer :: Int
                        }
```

Zur Erinnerung: Ohne Record-Syntax:

```
data Student = Student String String Int
```

⇒ Die Komponenten werden mit Namen markiert

Beispiel

Hochschule RheinMain

Beispiel: `Student "Hans" "Mueller" 1234567`

kann man schreiben als

```
Student{vorname="Hans",
        name="Mueller",
        matrikelnummer=1234567}
```

Reihenfolge der Komponenten egal:

```
Student{matrikelnummer=1234567,
        vorname="Hans",
        name="Mueller"}
```


Record-Syntax

Hochschule RheinMain

Zugriffsfunktionen sind automatisch verfügbar, z.B.

```
*> matrikelnummer x
1234567
```

- Record-Syntax ist in den Pattern erlaubt
- Nicht alle Felder müssen abgedeckt werden
- bei Erweiterung der Datenstrukturen, daher kein Problem

```
nameMitA Student{name = 'A':xs} = True
nameMitA _ = False
```

Record-Syntax: Update

Hochschule RheinMain

```
setzeName :: Student -> String -> Student
setzeName student neuername =
        student {name = neuername}
```

ist äquivalent zu

```
setzeName :: Student -> String -> Student
setzeName student neuername =
  Student {vorname       = vorname student,
           name          = neuername,
           matrikelnummer = matrikelnummer student}
```


Parametrisierte Datentypen

Hochschule RheinMain

Datentypen in Haskell dürfen polymorph parametrisiert sein:

`data` *Typname* $par_1 \ldots par_m$ = *Konstruktor*$_1$ $arg_{1,1} \ldots arg_{1,i_1} | \ldots |$ *Konstruktor*$_n$ $arg_{n,1} \ldots arg_{n,i_n}$
`deriving` ($class_1, \ldots, class_l$)

- Typname muss mit einem Großbuchstaben beginnen
- Typparameter sind optional, ebenso Alternativen
- Konstruktoren müssen mit Großbuchstaben beginnen
- Konstruktoren erwarten als Argumente eine beliebige Anzahl von Argumenten: Argument = bekannter Typ oder Typparameter
- Optionale `deriving`-Klausel mit einer Liste von Typklassen

Der Datentyp Maybe

Hochschule RheinMain

```
data Maybe a = Nothing | Just a
```

- ist polymorph über dem Parameter a.
- Konkrete Instanz ist z.B. `Maybe Int`.
- Verwendung: Undefinierte Fälle partieller Funktionen mit `Nothing` abbilden, dadurch totale Funktionen Erstellen

Beispiel:

```
getKruemmung :: Frucht -> Maybe Double
getKruemmung (Banane _ _ k)  = Just k
getKruemmung _               = Nothing
```


Einige vordefinierte Funktionen für Maybe

Hochschule RheinMain

```
isJust         :: Maybe a -> Bool
isJust Nothing = False
isJust _       = True

fromMaybe :: a -> Maybe a -> a
fromMaybe standardWert Nothing  = standardWert
fromMaybe      _       (Just x) = x

catMaybes :: [Maybe a] -> [a]
catMaybes ls = [x | Just x <- ls]
```

Der Datentyp Either

Hochschule RheinMain

Der Datentyp `Either` hat mehrere Typparameter:

```
data Either a b  =  Left a | Right b
```

Z.B.: Verwende `Either a String` anstelle von `Maybe a` für die Rückgabe von Fehlermeldungen.

```
getKruemmung' :: Frucht -> Either String Double
getKruemmung' (Banane _ _ k)  = Right k
getKruemmung' _               = Left "Eingabe ist keine Banane."

myDiv :: Double -> Double -> Either String Double
myDiv x 0 = Left "Error: Division by 0"
myDiv x y = Right $ x / y
```


Der Datentyp Either (2)

Hochschule RheinMain

`Either` ist der Typ für disjunkte Vereinigungen:

$$A_1 \dot\cup A_2 = \{(i, a) \mid a \in A_i\}$$

Zum Beispiel:

$$\{\diamondsuit, \heartsuit\} \dot\cup \{\heartsuit, \clubsuit, \spadesuit\} = \{(1, \diamondsuit), (1, \heartsuit), (2, \heartsuit), (2, \clubsuit), (2, \spadesuit)\}$$

Hilfreiche Funktion für Either

Hochschule RheinMain

```
isRight :: Either a b -> Bool
isRight (Left  _) = False
isRight (Right _) = True

lefts  :: [Either a b] -> [a]
lefts x = [a | Left a <- x]

partitionEithers :: [Either a b] -> ([a],[b])
partitionEithers []      = ([],[])
partitionEithers (h : t)
   | Left  l <- h = (l:ls,   rs)
   | Right r <- h = (  ls, r:rs)
  where (ls,rs) = partitionEithers t
```


Rekursive Datentypen

Hochschule RheinMain

Rekursiver Datentyp: Der definierte Typ kommt rechts vom = wieder vor

```
data Typ = ... Konstruktor Typ ...
```

Listen könnte man definieren als:

```
data List a = Nil | Cons a (List a)
```

In Haskell, eher Spezialsyntax:

```
data [a] = [] | a:[a]
```

In Haskell: $[a_1, \ldots, a_n]$ als Abkürzung für $a_1 : (a_2 : (\ldots : []) \ldots)$

Beispiele:

```
[1,2,3] :: [Int]
[1,2,2,3,3,3] :: [Int]
["Hello","World","!"] :: [[Char]]
```

Listen vs. Tupel

Hochschule RheinMain

	Listen	Tupel
Anzahl an Elementen	variabel	fest
Typ der Elemente	identisch	auch verschieden erlaubt

Man kann Listen und Tupel man beliebig ineinander verschachteln:

```
[(1,'a'),(2,'z'),(-4,'w')] :: [(Integer, Char)]

[[1,2,3],[],[4]] :: [[Integer]]

(4.5,[(True,'a',[5,7],())]) :: (Double,[(Bool,Char,[Integer],())])
```

Beachte: `[]` `[[]]` `[[],[]]` `[[[]]]` `[[],[[]]]`

sind alles **verschiedene** Werte

Haskell: Geschachtelte Pattern

Hochschule RheinMain

Haskell erlaubt geschachtelte Pattern, damit kann man u.a. Listen zerlegen:

```
viertesElement (x1:(x2:(x3:(x4:xs)))) = Just x4
viertesElement _                      = Nothing
```


Case-Ausdrücke

Hochschule RheinMain

Syntax hatten wir bereits definiert als

`case` *Ausdruck* `of` *Alternativen*

Dabei sind *Alternativen* case-Alternativen der Form
Pattern `->` *Ausdruck* oder *Pattern Match*

- *Pattern* z.B. `((x,y):xs)`
- *Match* bezeichnet eine Folge der Form
  | $Guard_1$ -> $Ausdruck_1$
  | ...
  | $Guard_n$ -> $Ausdruck_n$
- Wert des case: Erstes *Pattern*, das passt und erster Guard, der zu `True` auswertet

Beispiel

Hochschule RheinMain

```
myFun frucht p personendb =
  case frucht of
    Banane l h kruemmung
      | Just x <- lookup p personendb -> Right x
    _  -> Left "Fehler"
```

Aufrufe:

```
*Main> myFun (Banane 1 2 3) 1 [(1,"Horst")]
Right "Horst"
*Main> myFun (Apfel (1,2)) 1 [(1,"Horst")]
Left "Fehler"
*Main> myFun (Banane 1 2 3) 10 [(1,"Horst")]
Left "Fehler"
```

Typdefinitionen in Haskell

Hochschule RheinMain

Drei syntaktische Möglichkeiten in Haskell

- `data`
- `type`
- `newtype`

Verwendung von `data` haben wir bereits ausgiebig gesehen

Typdefinitionen in Haskell (2)

Hochschule RheinMain

`type`; Variante von Typdefinitionen.

Mit `type` definiert man Typsynonyme, d.h:

Neuer Name für bekannten Typ

Beispiele:

```
type IntCharPaar = (Int,Char)
type Studenten = [Student]
type MyList a = [a]
```

Sinn davon: Verständlicher, z.B.

```
alleStudentenMitA :: Studenten -> Studenten
alleStudentenMitA = map nameMitA
```


Typdefinitionen in Haskell (3)

Hochschule RheinMain

Typdefinition mit `newtype`:

- `newtype` ist sehr ähnlich zu `type`
- Mit `newtype`-definierte Typen dürfen eigene Klasseninstanz für Typklassen haben
- Mit `type`-definierte Typen aber nicht.
- Mit `newtype`-definierte Typen haben einen neuen Konstruktor
- case und pattern match für Objekte von einem mit `newtype`-definierten Typ sind immer erfolgreich.

Typdefinitionen in Haskell (4)

Hochschule RheinMain

Beispiel für *newtype*:

```
newtype Studenten' = St [Student]
```

Diese Definition kann man sich vorstellen als

```
data Studenten'    = St [Student]
```

Ist aber nicht semantisch äquivalent dazu, da Terminierungsverhalten anders

Vorteil `newtype` vs. `data`: Der Compiler weiß, dass es nur ein Typsynonym ist und kann optimieren: case-Ausdrücke dazu werden eliminiert und durch direkte Zugriffe ersetzt.

Beispiel für newtype vs. data

Hochschule RheinMain

```
newtype Studenten' = St [Student]
data  Studenten'' = St'' [Student]
```

Compiler entfernt `St` überall, Effekt:

```
*Main> case undefined of {St _ -> 1}
1
*Main> case undefined of {St'' _ -> 1}
*** Exception: Prelude.undefined
```


HASKELLS MODULSYSTEM

Hochschule RheinMain

Haskells hierarchisches Modulsystem

Hochschule RheinMain

Aufgaben von Modulen

- Strukturierung / Hierarchisierung
- Kapselung
- Wiederverwendbarkeit

Moduldefinition in Haskell

Hochschule RheinMain

$$\texttt{module}\ \mathit{Modulname}(\mathit{Exportliste})\ \texttt{where}\ \left.\begin{array}{l}\mathit{Modulimporte},\\ \mathit{Datentypdefinitionen},\\ \mathit{Funktionsdefinitionen},\ldots\end{array}\right\}\ \mathit{Modulrumpf}$$

- Name des Moduls muss mit Großbuchstaben beginnen
- Exportliste enthält die nach außen sichtbaren Funktionen und Datentypen
- Dateiname = Modulname.hs (bei hierarchischen Modulen Pfad+Dateiname = Modulname.hs)
- Ausgezeichnetes Modul `Main` muss Funktion `main :: IO ()` exportieren

Import / Export

Beispiel:

```
module Spiel where
  data Ergebnis = Sieg | Niederlage | Unentschieden
  berechneErgebnis a b = if a > b then Sieg
                         else if a < b then Niederlage
                              else Unentschieden
  istSieg Sieg = True
  istSieg _    = False
  istNiederlage Niederlage = True
  istNiederlage _          = False
```

- Da keine Exportliste: Es werden alle Funktionen und Datentypen exportiert
- Außer importierte

Exporte

Exportliste kann enthalten

- Funktionsname (in Präfix-Schreibweise) Z.B.: ausschließlich `berechneErgebnis` wird exportiert

  ```
  module Spiel(berechneErgebnis) where
  ```

- Datentypen mittels `data` oder `newtype`, drei Möglichkeiten
  - Nur `Ergebnis` in die Exportliste:

    ```
    module Spiel(Ergebnis) where
    ```

    Typ `Ergebnis` wird exportiert, die Datenkonstruktoren, d.h. `Sieg`, `Niederlage`, `Unentschieden` jedoch nicht
  - ```
    module Spiel(Ergebnis(Sieg, Niederlage))
    ```

    Typ `Ergebnis` und die Konstruktoren `Sieg` und `Niederlage` werden exportiert, nicht jedoch `Unentschieden`.
  - Durch den Eintrag `Ergebnis(..)`, wird der Typ mit sämtlichen Konstruktoren exportiert.

Exporte (2)

Exportliste kann enthalten

- Typsynonyme, die mit `type` definiert wurden

  ```
  module Spiel(Result) where
    ◦ .. wie vorher ◦ ..
    type Result = Ergebnis
  ```

- Importierte Module:

  ```
  module Game(module Spiel, Result) where
    import Spiel
    type Result = Ergebnis
  ```

  `Game` exportiert alle Funktionen, Datentypen und Konstruktoren, die auch `Spiel` exportiert sowie zusätzlich noch den Typ `Result`.

Importe

Einfachste Form: Importiert alle Einträge der Exportliste

`import` *Modulname*

Andere Möglichkeiten:

- Explizites Auflisten der zu importierenden Einträge:

  ```
  module Game where
    import Spiel(berechneErgebnis, Ergebnis(..))
    ◦ ..
  ```

- Explizites Ausschließen einzelner Einträge:

  ```
  module Game where
    import Spiel hiding(istSieg,istNiederlage)
    ◦ ..
  ```
  ```

Importe (2)

Hochschule RheinMain

So importierte Funktionen und Datentypen sind mit ihrem unqualifizierten und ihrem qualifizierten Namen ansprechbar.
Qualifizierter Name: *Modulname*.*unqualifizierter Name*

```
module A(f) where
    f a b = a + b
module B(f) where
    f a b = a * b
module C where
    import A
    import B
    g = f 1 2 + f 3 4 -- funktioniert nicht!

ghci> :l C.hs
ERROR C.hs:4 - Ambiguous variable occurrence "f"
*** Could refer to: B.f A.f
```


Importe (3)

Hochschule RheinMain

Mit qualifizierten Namen funktioniert es:

```
module C where
  import A
  import B
  g = A.f 1 2 + B.f 3 4
```

Mit Schlüsselwort `qualified` kann man nur die qualifizierten Namen importieren:

```
module C where
  import qualified A
  g = f 1 2   -- f ist nicht sichtbar

ghci> :l C.hs
ERROR C.hs:3 - Undefined variable "f"
```


Importe (4)

Hochschule RheinMain

Lokale Aliase: Schlüsselwort as:

```
import LangerModulName as C
```


Importe (5)

Hochschule RheinMain

Übersicht:

Import-Deklaration	definierte Namen
`import M`	`f, g, M.f, M.g`
`import M()`	keine
`import M(f)`	`f, M.f`
`import qualified M`	`M.f, M.g`
`import qualified M()`	keine
`import qualified M(f)`	`M.f`
`import M hiding ()`	`f, g, M.f, M.g`
`import M hiding (f)`	`g, M.g`
`import qualified M hiding ()`	`M.f, M.g`
`import qualified M hiding (f)`	`M.g`
`import M as N`	`f, g, N.f, N.g`
`import M as N(f)`	`f, N.f`
`import qualified M as N`	`N.f, N.g`

Hierarchische Modulstruktur

Hochschule RheinMain

- Modulnamen mit Punkten versehen geben Hierarchie an: z.B.

  ```
  module A.B.C
  ```

- Allerdings ist das rein syntaktisch
  (es gibt keine Beziehung zwischen Modul `A.B` und `A.B.C`)

- Beim Import:

  ```
  import A.B.C
  ```

  sucht der Compiler nach dem File namens `C.hs` im Pfad `A/B/`