Fortgeschrittene Funktionale Programmierung

Teil 1: Einführung

Steffen Jost

LFE Theoretische Informatik, Institut für Informatik,
Ludwig-Maximilians Universität, München

16. Oktober 2018

1. Organisation
   - Ziele
   - Ablauf
   - Literatur
2. Grundlagen funktionaler Programmierung
3. Haskell Grundlagen
   - Haskell Tool Stack
   - Glasgow Haskell Compiler
   - Fallstricke
   - Basistypen
   - Algebraische Datentypen
     - List-Comprehension
     - Data.Maybe
     - Data.Either
   - Funktionen

1. Polymorphismus
   - Motivation
   - Parametrisch
   - Ad-hoc
   - Typklassen
   - Unterklassen
2. Monoide
3. Funktoren
   - Beispiele
   - Kinds
   - Newtype
   - Zusammenfassung Funktoren
   - Zusammenfassung Polymorphismus
4. Module
   - Export

Modulbeschreibung

Aufbauend auf die Einführung in die funktionale Programmierung im Rahmen der Lehrveranstaltung “Programmierung und Modellierung” (Semester 2), studieren wir fortgeschrittene Techniken der funktionalen Programmierung.

Dies umfasst die Behandlung von I/O und Effekten, Nebenläufige und Parallele Programme, Testen und Verifikation, sowie die Entwicklung ereignisgesteuerter Anwendungen wie Webapplikationen und graphischer Benutzeroberflächen.

Kenntnisse der Programmiersprache Haskell werden vorausgesetzt. Solide Vorkenntnisse einer anderen funktionalen Sprache (z.B. SML) reichen ebenfalls aus, da zu Beginn des Kurses Haskell-Syntax und die wichtigsten Grundlagen kurz wiederholt werden.

Mögliche Inhalte der Vorlesung

Thema	Anz.Vorl.
Wdh: Typklassen, Funktoren, Module, Kinds	2
Laziness & Corekursion	2
Applikative Funktoren & Monaden (Wdh.?)	2
Records & Linsen	1
Paralleles Rechnen (GpH, Eval & Par)	1
Nebenläufigkeit (STM, Concurrent) & Ausnahmen	2
Webapplikation (Yesod), GADTs, Template Haskell	2
Datenbanken (Yesod & Persist)	1
Debugging & Testen (Quickcheck)	1
	14

Mögliche weitere Themen

Parser-generation (Parsec), Foreign Function Interface, Grafische Benutzeroberfläche (GTK oder FRP), Effiziente Funktionale Datenstrukturen, ...

Voraussetzungen

- Inhalt Vorlesung "Programmierung und Modellierung" wird vorausgesetzt, d.h. solide Grundkenntnisse *einer* funktionalen Programmiersprache wie z.B. Haskell, OCaml, SML, F#, ...
- Bereitschaft, jede Woche praktische Programmier-Übungen durchzuführen
- In der Lage zu sein, ein Haskell Programm zu schreiben & auszuführen
  - GHC oder Stack Installation auf eigenem Laptop
  - GHC auf den Rechnern im CIP-Pool der Informatik

Organisation

Vorlesung entfällt am 20.11.18

- Dienstags, 18-20 Uhr, B U101, Oettingenstr. 67
- Bitte in Uni2work zur Vorlesung anmelden!
- Vorlesungshomepage beachten: http://www.tcs.ifi.lmu.de/lehre/ws-2018-19/fun

Übungen ab 24.10.18

- Mittwoch 12-14 Uhr, A121, Edmund-Rumpler-Str. 9
- Mittwoch 14-16 Uhr, A121, Edmund-Rumpler-Str. 9
- Gruppenübung, ohne Vorbereitung. Laptop mitbringen!
- Abgabe und Lösungen per Uni2work!

Prüfung Programmierprojekt im Anschluß zur Vorlesung, belegt Verständnis von mind. 3 Vorlesungsthemen

Abschlussprüfung

Benotung der Veranstaltung durch Programmierprojekt:

- Durchführbar alleine oder in Gruppe mit bis zu 3 Teilnehmern
- Präsentation & Befragung zu Projekt (ca. 30min)
- Abgabe & Präsentation: Ende März/Anfang April (KW13/14)

Ziel

Projekt soll im Ersatz einer Klausur belegen, dass der Teilnehmer die Inhalte der Vorlesung verstanden hat und anwenden kann.

- Anhand des Codes sollte tieferes Verständnis für mindestens 3–4 Themen der Vorlesung demonstrieren werden können.
- Passende unbehandelte Themen können nach Absprache auch gelten, z.B. andere Kapitel aus den Büchern, andere Frameworks, etc.
- Code anderer Sprachen wird in der Regel nicht berücksichtigt (z.B. JavaScript in Web-Applikationen wird nicht bewertet)

ABSCHLUSSPÄSENTATION

Alle zur Teilnahme eingeladen! Ausschluß Öffentlichkeit möglich

Maximal 10–20 Minuten Präsentation, je nach Teilnehmern, inkl.

- Demonstration der Software
- Diskussion des Codes:
  - Wo wurden welche Techniken aus der Vorlesung eingesetzt?
  - Welche interessanten Probleme traten auf?
  - Welche Bibliotheken/Vorlagen wurden benutzt?
- bei 2–3 Teilnehmern: Wer hat was gemacht?

Anschließend Befragung aller Teilnehmer zu Projekt und Code.

Bewertet werden dabei auch:

- Eingesetzte Techniken
- Korrektheit
- Eleganz und Klarheit des Codes
- Umfang

Ideen

Abschlussprojekte der Vorjahre

www.tcs.ifi.lmu.de/lehre/ws-2015-16/fun/abschlussprojekt
www.tcs.ifi.lmu.de/lehre/ws-2014-15/fun/fun#projekttable

Weitere Möglichkeiten

- Kommandozeilen Tools wie im Buch "Real-World Haskell"
- Interpreter/Parser für eine andere Sprache / DSL
  Techniken aus Übungen zum Lambda-Kalkül verwendbar
- Parallele Rechenintensive Hilfstools
  per Kommandozeile, Web-Interface oder GUI
- Spiel mit GUI, eventuell auch mit (paralleler) AI
- Yesod-Webapp: Spiel, Planer/Verwaltung, ...

Häufige Missverständnisse

- Themen der Vorlesung sind wichtiger als Thema des Projekts!
  Z.B. ist eine umfassende, schöne, funktionierende Webapplikationen trotzdem problematisch, wenn diese nur einen minimalen Umfang von Yesod einsetzt!
  Tipp: andere Vorlesungsthemen wie z.B. Fehlerbehandlung nicht vernachlässigen!
- Thema „Monaden“ bedeutet mehr, als einmal irgendwo `runDB` und `do` hinzuschreiben!
  D.h. entweder essentielle eigene Monaden-Definition, oder essentieller Einsatz von Monaden-Transformern, oder essentieller Einsatz einer Zustandsmonade, usw.
- Thema „Paralleles Rechnen“: nicht vergessen zu demonstrieren, dass tatsächlich eine Beschleunigung erfolgt und wie gut diese ist! (Threadscope, etc.)
- Es sollte kompilieren!
- Lesbarkeit des Codes ist wirklich wichtig!

Literatur

Die Vorlesung richtet sich nach folgenden Quellen, welche fast alle kostenlos online verfügbar sind:

- *Real World Haskell*
  von Bryan O'Sullivan, Don Stewart, John Goerzen
- *Parallel and Concurrent Haskell* von Simon Marlow
- *Haskell and Yesod* von Michael Snoyman
- *Programming in Haskell* von Graham Hutton
- *Learn You a Haskell for Great Good!* von Miran Lipovača
- *Purely Functional Data Structures* von Chris Okasaki

so wie ältere Skripte des Lehrstuhls TCS und das Haskell-Wiki.

Links/ISBN der Quellen, dieses Skript ⇒ Vorlesungshomepage

Hilfe aus dem Web

Es gibt zahlreiche Webseiten zu Haskell. Ein zentrale Auskunft liefert das Haskell Wiki: http://www.haskell.org/haskellwiki

Es gibt aber auch spezialisierte Suchmaschinen:

Hoogle http://haskell.org/hoogle

- Durchsucht Paket-Dokumentation
- Kann auch nach Typen suchen
  `(a -> b -> a) -> a -> [b] -> b`
  findet all möglichen Varianten von der Faltungsfunktion
- Kann lokal laufen und in Editor integriert werden

Hayoo http://holumbus.fh-wedel.de/hayoo/hayoo.html

Wie Hoogle, sucht jedoch auch in nicht-standard Paketen

Hilfe aus der Gemeinschaft

Bei speziellen Fragen sind die Haskell-Mailinglisten eine gute Anlaufstelle, um Fragen zu stellen:

- haskell-cafe@haskell.org
- beginners@haskell.org
- haskell@haskell.org

Auch auf Code-Probleme spezialisierte Webseiten können Haskell-Programmierer helfen:
http://stackoverflow.com/questions/tagged/haskell

Warum Wieso Weshalb?

Warum funktionale Programmierung?

... deklarativer Ansatz ... "was" anstatt "wie" ...

... Korrektheit und Wartbarkeit wichtiger als Effizienz ...

...... Referentielle Transparenz ... Keine Seiteneffekte ...

... Gute Modularität ...

Warum Haskell?

... rein funktional ... sehr gute Dokumentation ...

... Compiler optimiert aggressiv ...

... wachsende komerzielle Unterstützung ...

Genauer ausgeführt: 1. Kapitel Folien ProMo Sommersemester 2018

Funktionaler Ansatz

Nachteile

- Völlig andere (mathematische) Denkweise ("was" statt "wie"); Effizienz bekannter Algorithmen ändert sich
- Compiler "nörgelt viel" herum, bevor man testen kann
- "Kontrollverlust": Maschinen-nahe Hand-Optimierung schwierig

Vorteile

- Programmierer bekommt Hilfe um Korrektheit sicherzustellen "Well-typed programs can't go wrong!", R.Milner
- Nachträgliche Optimierung/Refactoring oft unproblematisch
- Neues kommerzielle Interesse wegen Multi-Cores
  "The Downfall of Imperative Programming", Milewski

Funktionale Merkmale, bzw. Unterstützung Funktionaler Ansätze sind daher inzwischen in viele anderen Sprachen anzutreffen

z.B. Funktionen höherer Ordnung, Anonyme Funktionen, Java GC optimiert auf kurzlebige, statische Objekte, etc.

Referentielle Transparenz

Wichtige Eigenschaft deklarativer Sprachen:

Referentielle Transparenz

- Wert einer Variablen ist *unveränderlich*
- Wird ein Ausdruck zweimal ausgewertet, kommt der selbe Wert heraus. Funktionsaufrufe mit gleichen Argumenten liefern gleiches Ergebnis!
- Keine Seiteneffekte!

Konsequenzen:

- Programme lokal verständlich und kompositionell
- Testen/Verifikation reduziert sich auf Gleichheitsschließen
- Compiler kann aggressiv optimieren
- Parallele Berechnung einfacher

Vorteile referentieller Transparenz

- Berechnungen leicht parallelisierbar.
- Ermöglicht aggressive Optimierungen durch den Compiler, z.B. *common subexpression elimination*.
  `(x + y) * (x + y)` optimiert zu `let z = x + y in z * z`
  mögliche Seiteneffekte, wie z.B. Wert-Veränderung erschweren solche Optimierungen in anderen Sprachen:
  `(++x + y) * (++x + y)`
- Erleichtert Testen und Verifikation:
  Imperative Sprachen benötigen Hoare-Logik,
  Haskell benötigt nur Gleichungsrechnen.
  (Haskell Impl. "GOFER" = Good For Equational Reasoning)
- Unveränderliche Variablen gibt es auch in anderen Sprachen (z.B. in C mit Schlüsselwort `const`), aber die Verwendung kann nicht automatisch angenommen werden

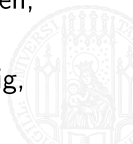
Unveränderliche Datenstrukturen

Konsequenz: Datenstrukturen sind auch unveränderlich!
Einfügen in eine Liste muss eine *neue* Liste erzeugen.
Dies ist eine Nachteil und ein Vorteil:

- *Persistenz:* Alte Liste existiert auch noch.
- *Sharing:* Unveränderte Restliste wird von alter und neuer Liste gemeinsam referenziert.
- *Garbage Collection:* Nicht mehr benötigte alte Versionen werden erst irgendwann aus dem Speicher entfernt.

Vorteile

- Typische Falle veränderlicher Strukturen entfällt, wie z.B. Veränderung während Iteration (`for` in Java).
- Backtracking (Rekursion): Alter Zustand ist noch verhanden, muss nicht wiederhergestellt werden.
- Nebenläufigkeit: Kein explizites Kopieren der Struktur nötig, wenn zwei Threads gleichzeitig manipulieren wollen.



Starke Typisierung

Funktionaler Sprachen haben fast immer ein **starkes Typsystem**.

Das Typsystem dient

- der Aufdeckung von Fehlern durch den Compiler
- der Auflösung von Überladung
- der Dokumentation von Funktionen
- der Suche passender Funktionen

Typen können oft automatisch inferiert werden (ML-Dialekte), oder zumindest zum größten Teil (Haskell, Scala).

Fortgeschrittene Typsysteme dienen der Verifikation (Agda).

Beispiel: Starke Typen

Beispiel: gegeben sei ein Typ einer Funkton mit drei Argumenten; erstes Argument ist eine Funktion, welche ein Paar `(a,b)` auf ein `c` abbildet; zweites Argument ist ein einzelnes `a`; drittes Argument ist ein `b`; Rückgabewert ist vom Typ `c`:

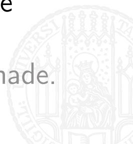
```
((a,b) -> c) -> a -> b -> c
```

Beobachtung: Es gibt nur *eine* totale Funktion mit diesen Typ!

...aber mehrere Möglichkeiten zur Implementierung

In einer Sprache mit Seiteneffekten kann dagegen alles mögliche passieren – der Typ allein reicht dort als Spezifikation nie aus.

Gilt natürlich auch Funktionen in der IO-Monade.



Abstraktion

Wichtiges Merkmal funktionaler Sprachen: **hohe Abstraktion.**

Extrahieren von gemeinsamen Codefragmenten wird durch *Funktionen höherer Ordnung* erleichtert:

```
factorial n = foldl (*) 1 [1..n]
```

Universell verwendbare Bausteine wie

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

sind gut verstehbar und leichter einsetzbar als Design-Patterns.

Hohe Wiederverwendbarkeit wird durch starken Einsatz von **Polymorphismus** erreicht.

Haskell

- Effektfreie **rein funktionale** Sprache
  mit verzögerter Auswertung
- Benannt nach Haskell Curry (1900-82), Logiker
- Standards: Haskell98 und Haskell2010
- Viele verschiedene Implementierung verfügbar; wichtigste:
  GHC: Glasgow/Glorious Haskell Compiler   Hammond, 1989
  aktuelle Versionen mehrmals pro Jahr
  Hauptentwickler erhielten 2011 ACM SIGPLAN Programming Languages Software Award:
  Simon Peyton-Jones   Microsoft Research Cambridge
  Simon Marlow   Facebook

Interessante Blog Posts:
„Haskell fights spam at Facebook“ by Facebook / by Wired

Stack

Vorlesung verwendet GHC; am besten über **Stack** installieren:

http://www.haskellstack.org/

Stack ist ein Plattform-übergreifendes Werkzeug zur Entwicklung und Verwaltung von Haskell Projekten seit Juni 2015

- Stack erzeugt für jedes Projekt Konfigurationsdateien zur Veröffentlichung und Verwaltung aber kein eigenes Revisioning
- Jedes Projekt behält seine Haskell- und Bibliotheks-Versionen; falls ein Update gewünscht ist, erfolgt dies pro Projekt!
- Benötigte Paket Versionen und auch Tools werden von Stack automatisch lokal installiert und verwaltet
- Keinerlei Admin-Rechte erforderlich, alles lokal!

Verwendung von Stack am Anfang der Vorlesung nicht wichtig; aber später für Yesod oder Gtk2Hs sehr hilfreich!

Verwendung von Stack

Verfügbare Templates anzeigen:

```
> stack templates
```

Neues Project *MyProject* aus Template *new-template* erstellen:

```
> stack new MyProject new-template
```

Projekt einmalig vorbereiten (installiert ggf. GHC lokal):

```
> cd MyProject
> stack setup
```

Projekt kompilieren und ausführen:

```
> stack build
...
> stack exec MyProject-exe
```

Manuelles Ausführen von `ghc` und `ghci` möglich mit:

```
> stack exec -- ghc Foo.hs
> stack ghci
```

GHC

In der Vorlesung arbeiten wir mit GHC. Die Dokumentation ist online verfügbar:
http://www.haskell.org/ghc/docs/latest/html/libraries/

Glasgow Haskell Compiler (GHC) kennt zwei Arbeitsweisen:

GHC Normaler Compiler. Ein Programm wird in mehreren Dateien geschrieben und mithilfe des GHC in ein ausführbares Programm übersetzt.

GHCi Interpreter Modus: Man gibt Ausdrücke und Definitionen ein und GHCI wertet diese sofort innerhalb der IO-Monade aus und zeigt den Wert an.

GHCI

```
> ghci +RTS -M1g

GHCi, version 8.2.2: http://www.haskell.org/ghc/
Prelude> 1 + 2
3
Prelude> 3 + 4 * 5
23
Prelude> (3 + 4) * 5
35
```

Der Text hinter dem Prompt `Prelude>` wurde vom Benutzer getätigt, alles andere sind Ausgaben von GHCi. Der Prompt gibt per Default alle geladenen Bibliotheken (**Module**) an.

GHCI

Der Interpreter wertet alle eingegebenen Ausdrücke aus. Mit den Pfeiltasten kann von vorherige Eingaben durchblättern

Zur Steuerung des Interpreters stehen Befehle zur Verfügung, welche alle mit einem Doppelpunkt beginnen, z.B. `:?` für die Hilfe.

Alle Befehle kann man abkürzen. So kann man den Interpreter sowohl mit `:quit` also auch mit `:q` verlassen. Für uneindeutige Abkürzungen gibt es voreingestellte Defaults.

Auch für GHCi macht es Sinn, Programmdefinitionen mit einem gewöhnlichen Texteditor in eine separate Datei speichern und dann in den Interpreter zu laden, um nicht immer alles neu eintippen zu müssen.

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

Fallstricke bei Haskell

I) Haskell beachtet Groß-/Kleinschreibung!

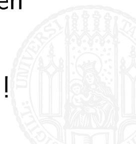
II) Haskell ist "whitespace"-sensitiv: Veränderungen an Leerzeichen, Tabulatoren und Zeilenumbruch können Fehler verursachen!
Grundsätzlich gilt: Beginnt die nächste Zeile in ...

- *Spalte weiter rechts:* vorheriger Zeile geht weiter
- *gleicher Spalte:* nächstes Element eines Blocks beginnt
- *Spalte weiter links:* Block beendet

Anstatt Einrückung können aber auch `{ }` und `;` benutzt werden.

Daraus folgt:

- Alle Top-level Definition müssen in gleicher Spalte beginnen
- Einrückung kann viele Klammern sparen
- Tabulatorweite in Editor und GHC muss übereinstimmen!!!



Editor und IDE

Zum Programmieren reicht ein gewöhnlicher Texteditor, sofern dieser Tabulatoren eliminieren kann!

Hinweise zur Verwendung von Editoren und IDEs findet man auf:
https://wiki.haskell.org/IDEs

Wer unbedingt eine IDE braucht, sollte das IntelliJ Plugin verwenden, welches recht gut funktioniert, aber bei großen Projekten viel Leistung verbraucht. Diverse Eclipse Plugins für Haskell hatten sich in der Vergangenheit immer als problematisch erwiesen.

Typen

Ein Typ ist eine Menge von Werten; so bezeichnet der Typ `Int` meistens die Menge der ganzen Zahlen von -2^{29} bis $2^{29} - 1$.

Haskell-Syntax:

- Typnamen beginnen immer mit Großbuchstaben
- Typvariablen beginnen immer mit Kleinbuchstaben

GHCI Befehl `:type` zeigt Typ eines Ausdrucks:

```
Prelude> :type 'a'
'a' :: Char
```

`e :: A` gelesen als "Ausdruck `e` hat Typ `A`"

Mit dem Befehl `:set +t` wird der Typ jedes ausgewerteten Ausdrucks angezeigt, mit `:unset +t` stellt man das wieder ab.

Wichtige Numerische Typen

Int Ganze Zahlen ("fixed precision integers"), maschinenabhängig, mindestens von -2^{29} bis $2^{29}-1$. Es wird nicht auf Überläufe geprüft.

Integer Ganze Zahlen beliebiger Größe
"arbitrary precision integers"

Float Fließkommazahlen mindestens nach IEEE standard
"single-precision floating point numbers"

Double Fließkommazahlen mit mindestens doppelter Genauigkeit nach IEEE standard
"double-precision floating point numbers"

Rational Rationale Zahlen beliebiger Genauigkeit, werden mit dem Prozentzeichen konstruiert: `1 % 5` ≈ 0.2
% nicht im Prelude-Modul enthalten

Haskell kennt viele weitere numerische Datentypen, z.B. komplexe Zahlen, Uhrzeiten oder Festkommazahlen.

Wichtige Typen

- **Bool** Boole'sche (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.

Haskell ist minimal definiert: Von diesen drei Typen ist lediglich `Char` wirklich speziell eingebaut, die beiden anderen kann man leicht selbst definieren:

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

Basistyp Bool

```
data Bool = True | False
```

Standardbibliothek **Prelude** definiert u.a. folgende Funktionen

bis auf `not` in Infix-Notation

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

```
Prelude> (True || False) && (not True)
False

Prelude> 3 == 4
False

Prelude> 3 /= 4
True
```

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 $A_1 \times, \ldots, \times A_n$ heißen allgemein ***n*-Tupel**, spezieller auch Paare, Tripel, Quadrupel, Quintupel, Sextupel,...

In Haskell schreiben wir Tupelausdrücke und Produkttypen mit runden Klammern und Kommas. $\mathbb{Z} \times \mathbb{Z}$ wird zu `(Int,Int)`.

```
Prelude> :t (True, 'a', 7)
(True, 'a', 7) :: (Bool, Char, Int)

Prelude> :t (4.5,"Hi!")
(4.5,"Hi!") :: (Double, String)
```

Listen

Einer der wichtigsten Typen in Haskell sind Listen, also geordnete Folgen von Werten. Listen bekannnter Länge schreiben wir mit eckigen Klammern und Kommas:

```
[1,2,3] :: [Int]

[1,2,2,3,3,3] :: [Int]

["Hello","World","!"] :: [[Char]]
```

Eine Liste kann sogar ganz leer sein, geschrieben `[]`.
Eigentlich ist `[1,2,3]` Kurzschreibweise für `1:2:3:[]`

Liste Anzahl Elemente unbekannt,
aber alle Elemente haben gleichen Typ

Tupel Anzahl Elemente bekannt,
aber Elemente können unterschiedliche Typen haben

Listen vs Tupel

Listen und Tupel kann 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], ())])
```

Achtung: `[]` und `[[]]` und `[[],[]]` und `[[[]]]` und `[[],[[]]]` sind alles verschiedene Werte.

Unit Typ Das 0-Tupel ist ebenfalls erlaubt: `() :: ()`.
Der Typ `()` wird als **Unit**-Typ bezeichnet, und hat nur den einzigen Wert `()`. Aus dem Kontext wird fast immer klar, ob mit `()` der Typ oder der Wert gemeint ist.

List-Comprehension

Mengen werden in der Mathematik oft intensional beschrieben:

$$\left\{ x^2 \,\middle|\, x \in \{1, 2, \ldots, 10\} \text{ und } x \text{ ist ungerade} \right\} = \{1, 9, 25, 49, 81\}$$

wird gelesen als "Menge aller x^2, so dass gilt..."

Haskell bietet diese Notation ganz analog für Listen:

```
[ x^2 | x <- [1..10], odd x ]  ==  [1,9,25,49,81]
```

"Liste aller x^2,
wobei x aus der Liste $[1, \ldots, 10]$ gezogen wird und x ungerade ist"

Haskell hat auch eine Bibliothek für echte (ungeordnete) Mengen, aber Listen sind in Haskell grundlegender.

List-Comprehension

```
[ x^2 | x <- [1..10], odd x ] == [1,9,25,49,81]
```

Rumpf: bestimmt wie ein Listenelement berechnet wird

List-Comprehension

```
[ x^2 | x <- [1..10], odd x ] == [1,9,25,49,81]
```

Rumpf: bestimmt wie ein Listenelement berechnet wird

Generator: weist Variablen nacheinander Elemente einer anderen Liste zu — hier die Liste `[1..10]`

List-Comprehension

```
[ x^2 | x <- [1..10], odd x ] == [1,9,25,49,81]
```

Rumpf: bestimmt wie ein Listenelement berechnet wird

Generator: weist Variablen nacheinander Elemente einer anderen Liste zu — hier die Liste `[1..10]`

Filter: Ausdruck von Typ `Bool` (**Bedingung**) entscheidet, ob dieser Wert in erzeugter Liste enthalten ist

List-Comprehension

```
[ x^2 | x <- [1..10], odd x ] == [1,9,25,49,81]
```

Rumpf: bestimmt wie ein Listenelement berechnet wird

Generator: weist Variablen nacheinander Elemente einer anderen Liste zu — hier die Liste `[1..10]`

Filter: Ausdruck von Typ `Bool` (**Bedingung**) entscheidet, ob dieser Wert in erzeugter Liste enthalten ist

Abkürzung: `let` erlaubt Abkürzungen zur Wiederverwendung

```
> [ z | x <- [1..10], let z = x^2, z>50]
[64,81,100]
```

List-Comprehension

```
[ x^2 | x <- [1..10], odd x ] == [1,9,25,49,81]
```

Rumpf: bestimmt wie ein Listenelement berechnet wird

Generator: weist Variablen nacheinander Elemente einer anderen Liste zu — hier die Liste `[1..10]`

Filter: Ausdruck von Typ `Bool` (**Bedingung**) entscheidet, ob dieser Wert in erzeugter Liste enthalten ist

Abkürzung: `let` erlaubt Abkürzungen zur Wiederverwendung

```
> [ z | x <- [1..10], let z = x^2, z>50]
[64,81,100]
```

- Beliebig viele Generatoren, Filter und Abkürzungen
- Definition können "weiter rechts" verwendet werden
- Verschachtelung von List-Comprehensions erlaubt

Beispiele

Beliebig viele Generatoren, Filter und Abkürzungen dürfen in beliebiger Reihenfolge in List-Comprehensions verwendet werden:

```
> [ (wert,name) | wert <-[1..3], name <- ['a'..'b']]
[(1,'a'),(1,'b'),(2,'a'),(2,'b'),(3,'a'),(3,'b')]
```

Die Reihenfolge der Generatoren bestimmt die Reihenfolge der Werte in der Ergebnisliste.

```
> [ (wert,name) | name <- ['a'..'b'], wert <-[1..3]]
[(1,'a'),(2,'a'),(3,'a'),(1,'b'),(2,'b'),(3,'b')]
```

Typabkürzungen

Der Typ String ist nur eine Abkürzung für eine Char-Liste:

```
type String = [Char]
```

Solche Abkürzungen darf man genau so auch selbst definieren:
Hinter dem Schlüsselwort type schreibt meinen einen frischen Namen, der mit einem Großbuchstaben beginnt und hinter dem Gleichheitszeichen folgt ein bekannter Typ, z.B.

```
type MyWeirdType = (Double,[(Bool,Integer)])
```

Für die Ausführung von Programmen ist dies unerheblich.
Typabkürzungen dienen primär zur Verbesserung der Lesbarkeit.
GHC/GHCi kann Typabkürzungen in Fehlermeldungen ignorieren, d.h. GHCi gibt dann [Char] anstelle von String aus.
Es gibt noch weitere zusammengesetzte Typen, z.B. Records, Funktionstypen, etc.

Vgl. 2.48: newtype

Algebraische Datentypen

Algebraische Datentypen werden üblicherweise aus Summen und Produkten gebildet:

Summe `data Bool = True | False`
Werte des Typs `Bool` sind entweder `True` oder `False`

Produkt `data MyTuple = MyTupleConstructor Int Int`
Wie Tupel ohne Klammern und ohne Kommas; stattdessen mit vorangestelltem **Konstruktor**.

Dies darf man natürlich auch mischen:

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

```
> :t Birne 3 0.8
Birne 3 0.8 :: Frucht
```

Algebraische Datentypen

Algebraische Datentypen dürfen auch rekursiv sein:

```
data IntList = LeereListe | ListKnoten Int IntList

data CharBaum = CharBlatt Char
              | CharKnoten CharBaum Char CharBaum
```

Es dürfen auch Typparameter verwendet werden:

```
data List a = Leer | Element a (List a)

data Baum a b =
      Blatt a | Knoten (Baum a b) b (Baum a b)
```

Maybe

`Maybe` ist ein wichtiger polymorpher Datentyp der Standardbibliothek

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

Damit können wir auf eine sichere Weise ausdrücken, dass eine Berechnungen fehlschlagen kann.

Beispiel

```
data Frucht = Apfel  { preis::Int, anzahl::Int}
            | Birne  { preis::Int, anzahl::Int}
            | Banane { preis::Int, anzahl::Int
                     , krümmung::Double       }

getKrümmung :: Frucht -> Maybe Double
getKrümmung Banane {krümmung=k} = Just k
getKrümmung _                   = Nothing
```

Maybe

Beispiele

```
data Maybe a = Nothing | Just a

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]
```

Either

Polymorphe Datentypen können auch mehrere Typparameter haben. Either ist ein wichtiges Beispiel:

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

Either ist Typ für disjunkte Vereinigungen:

$$A_1 \dot{\cup} A_2 = \left\{ (i, a) \;\middle|\; a \in A_i \right\}$$

Beispiel:

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

Für endliche Mengen gilt: $|A_1 \dot{\cup} A_2| = |A_1| + |A_2|$

Man spricht auch von "Summentypen" bei Alternativen

Modul Data.Either

```
module Data.Either where
  data Either a b  =  Left a | Right b

  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
```

Modul Data.Either

```
module Data.Either where
  data Either a b  =  Left a | Right b

  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) =
    let (ls,rs) = partitionEithers t
    in case h of  Left l  -> (l:ls,   rs)
                  Right r -> (  ls, r:rs)
```

Either für Fehler-Kommunikation

Wir haben `Maybe a` für Berechnungen verwendet, welche fehlschlagen können.
Stattdessen bietet sich auch `Either String a` an, wenn man noch Fehlermeldungen mitgeben möchte, z.B.:

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

firstDiff :: Eq a => [a] -> [a] -> Either String a
firstDiff [] [] = Left "No Difference found."
firstDiff (x:xs) (y:ys)
  | x /= y    = Right x
  | otherwise = firstDiff xs ys
```

Datentypdeklaration

Syntax der Datentypdeklaration:

```
data Typname par_1 ... par_m
  = Konstruktor1 arg_11 ... arg_1i
  | Konstruktor2 arg_21 ... arg_2j
  | Konstruktor3 arg_31 ... arg_3k
```

Datentypdeklaration

Syntax der Datentypdeklaration:

```
data Typname par_1 ... par_m
  = Konstruktor1 arg_11 ... arg_1i
  | Konstruktor2 arg_21 ... arg_2j
  | Konstruktor3 arg_31 ... arg_3k
```

- Schlüsselwort `data`

Datentypdeklaration

Syntax der Datentypdeklaration:

```
data Typname par_1 ... par_m
  = Konstruktor1 arg_11 ... arg_1i
  | Konstruktor2 arg_21 ... arg_2j
  | Konstruktor3 arg_31 ... arg_3k
```

- Schlüsselwort `data`
- frischer Typname – muss mit Großbuchstaben beginnen

Datentypdeklaration

Syntax der Datentypdeklaration:

```
data Typname par_1 ... par_m
  = Konstruktor1 arg_11 ... arg_1i
  | Konstruktor2 arg_21 ... arg_2j
  | Konstruktor3 arg_31 ... arg_3k
```

- Schlüsselwort `data`
- frischer Typname – muss mit Großbuchstaben beginnen
- optionale Typparameter

Datentypdeklaration

Syntax der Datentypdeklaration:

```
data Typname par_1 ... par_m
  = Konstruktor1 arg_11 ... arg_1i
  | Konstruktor2 arg_21 ... arg_2j
  | Konstruktor3 arg_31 ... arg_3k
```

- Schlüsselwort `data`
- frischer Typname – muss mit Großbuchstaben beginnen
- optionale Typparameter
- optionale Alternativen — lies | als "oder"

Datentypdeklaration

Syntax der Datentypdeklaration:

```
data Typname par_1 ... par_m
  = Konstruktor1 arg_11 ... arg_1i
  | Konstruktor2 arg_21 ... arg_2j
  | Konstruktor3 arg_31 ... arg_3k
```

- Schlüsselwort `data`
- frischer Typname – muss mit Großbuchstaben beginnen
- optionale Typparameter
- optionale Alternativen lies | als "oder"
- frischer Konstruktor – muss mit Großbuchstaben beginnen

Datentypdeklaration

Syntax der Datentypdeklaration:

```
data Typname par_1 ... par_m
  = Konstruktor1 arg_11 ... arg_1i
  | Konstruktor2 arg_21 ... arg_2j
  | Konstruktor3 arg_31 ... arg_3k
```

- Schlüsselwort `data`
- frischer Typname – muss mit Großbuchstaben beginnen
- optionale Typparameter
- optionale Alternativen — lies | als "oder"
- frischer Konstruktor – muss mit Großbuchstaben beginnen
- beliebige Zahl bekannte Typen oder Typparameter

Datentypdeklaration

Syntax der Datentypdeklaration:

```
data Typname par_1 ... par_m
  = Konstruktor1 arg_11 ... arg_1i
  | Konstruktor2 arg_21 ... arg_2j
  | Konstruktor3 arg_31 ... arg_3k
  deriving (class_1, ..., class_l)
```

- Schlüsselwort `data`
- frischer Typname – muss mit Großbuchstaben beginnen
- optionale Typparameter
- optionale Alternativen lies | als "oder"
- frischer Konstruktor – muss mit Großbuchstaben beginnen
- beliebige Zahl bekannte Typen oder Typparameter
- optionale `deriving`-Klausel mit Liste von Typklassen

Datentypdeklaration

Syntax der Datentypdeklaration:

```
data Typname par_1 ... par_m
  = Konstruktor1 arg_11 ... arg_1i
  | Konstruktor2 arg_21 ... arg_2j
  | Konstruktor3 arg_31 ... arg_3k
  deriving (class_1, ..., class_l)
```

- Schlüsselwort `data`
- frischer Typname – muss mit Großbuchstaben beginnen
- optionale Typparameter
- optionale Alternativen — lies | als "oder"
- frischer Konstruktor – muss mit Großbuchstaben beginnen
- beliebige Zahl bekannte Typen oder Typparameter
- optionale `deriving`-Klausel mit Liste von Typklassen

Zusammenfassung: Datentypen

- Ein Typ (oder **Datentyp**) ist eine Menge von Werten
- Typdeklarationen in Haskell:
  - `type` Typabkürzungen für Lesbarkeit
  - `data` Deklaration wirklich neuer Typen
  - `newtype` wie `data`, zur Optimierung falls genau 1 Konstruktor mit 1 Argument — später
- Datentypen können andere Typen als Parameter haben, **Konstruktoren** können als Funktionen betrachtetet werden
- Datentypen können (wechselseitig) rekursiv definiert werden
- Unter einer **Datenstruktur** versteht man einen Datentyp plus alle darauf verfügbaren Operationen
- **Polymorphe Funktionen** können mit gleichem Code verschiedene Typen verarbeiten
- Datentypen verhindern versehentliches Verwechseln

Funktionsdefinition

Funktionen werden durch Gleichungen definiert:

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

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

- Alle Top-Level Definitionen müssen in der gleichen Spalte stehen
- Funktionsanwendung durch Leerzeichen: `foo 1 2 3`
- Infix-zu-Prefix durch Klammerung: `(+) 1 2 == 1 + 2`
- Prefix-zu-Infix durch Backticks: `add 1 2 == 1 `add` 2`

Funktionstypen

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 $A = A'$.

$\Rightarrow$ möglichst totale Funktionen in Haskell verwenden, ggf. `error` verwenden!
Alle Funktionen bilden 1 Argumenttyp auf 1 Ergebnistyp ab, allerdings dürfen diese beiden Typen algebraische Datentypen oder auch Funktionstypen sein:

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

Funktionstypen

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 $A = A'$.

$\Rightarrow$ möglichst totale Funktionen in Haskell verwenden, ggf. `error` verwenden!

Alle Funktionen bilden 1 Argumenttyp auf 1 Ergebnistyp ab, allerdings dürfen diese beiden Typen algebraische Datentypen oder auch Funktionstypen sein:

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

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

Partielle Anwendung

Klammerkonvention

- Funktionstypen sind implizit rechtsgeklammert:
  `Int -> Int -> Int` wird gelesen
  als `Int -> (Int -> Int)`
- Entsprechend ist Funktionsanwendung implizit linksgeklammert:
  `bar 1 8` wird gelesen als `(bar 1) 8`

Das bedeutet: `(bar 1)` ist eine Funktion des Typs `Int -> Int`
Funktionen sind also normale Werte in einer funktionalen Sprache!

Funktionsdefinition in Haskell

```
foo :: Typ1 -> Typ2 -> Typ3 -> Ergebnistyp
foo var1  var2  var3 = expr1
```

- Typdeklaration (optional – aber gute Dokumentation)

Funktionsdefinition in Haskell

```
foo :: Typ1 -> Typ2 -> Typ3 -> Ergebnistyp
foo var1  var2  var3 = expr1
```

- Typdeklaration (optional – aber gute Dokumentation)
- Funktionsname (immer in gleicher Spalte beginnen)

Funktionsdefinition in Haskell

```
foo :: Typ1 -> Typ2 -> Typ3 -> Ergebnistyp
foo var1  var2  var3 = expr1
```

- Typdeklaration (optional – aber gute Dokumentation)
- Funktionsname (immer in gleicher Spalte beginnen)
- Argumente

Funktionsdefinition in Haskell

```
foo :: Typ1 -> ... -> Typ3 -> Ergebnistyp
foo var_1 ... var_n = expr1
```

- Typdeklaration (optional – aber gute Dokumentation)
- Funktionsname (immer in gleicher Spalte beginnen)
- Argumente

Funktionsdefinition in Haskell

```
foo :: Typ1 -> ... -> Typ3 -> Ergebnistyp
foo var_1 ... var_n = expr1
```

- Typdeklaration (optional – aber gute Dokumentation)
- Funktionsname (immer in gleicher Spalte beginnen)
- Argumente

Funktionsdefinition in Haskell

```
foo :: Typ1 -> ... -> Typ3 -> Ergebnistyp
foo var_1 ... var_n = expr1
```

- Typdeklaration (optional – aber gute Dokumentation)
- Funktionsname (immer in gleicher Spalte beginnen)
- Argumente
- Funktionsrumpf

Funktionsdefinition in Haskell

```
foo :: Typ1 -> ... -> Typ3 -> Ergebnistyp
foo pat_1 ... pat_n = expr1
foo pat21 ... pat2n = expr2
foo pat31 ... pat3n = expr3
```

- Typdeklaration (optional – aber gute Dokumentation)
- Funktionsname (immer in gleicher Spalte beginnen)
- Argumente
- Funktionsrumpf
- Fallunterscheidung mit Pattern-Match

Funktionsdefinition in Haskell

```
foo :: Typ1 -> ... -> Typ3 -> Ergebnistyp
foo pat_1 ... pat_n = expr1
foo pat21 ... pat2n
    | grd211, ..., grd21i = expr21
    | grd221, ..., grd22i = expr22
foo pat31 ... pat3n
```

- Typdeklaration (optional – aber gute Dokumentation)
- Funktionsname (immer in gleicher Spalte beginnen)
- Argumente
- Funktionsrumpf
- Fallunterscheidung mit Pattern-Match
- Verfeinerung des Pattern-Match durch Wächter :: Bool

Funktionsdefinition in Haskell

```
foo :: Typ1 -> ... -> Typ3 -> Ergebnistyp
foo pat_1 ... pat_n = expr1
foo pat21 ... pat2n
    | grd211, ..., grd21i = expr21
    | grd221, ..., grd22i = expr22
foo pat31 ... pat3n
    | grd311, ..., grd31k = expr31
```

- Typdeklaration (optional – aber gute Dokumentation)
- Funktionsname (immer in gleicher Spalte beginnen)
- Argumente
- Funktionsrumpf
- Fallunterscheidung mit Pattern-Match
- Verfeinerung des Pattern-Match durch Wächter :: Bool
- Erster zutreffender Match gilt (von oben nach unten)

Funktionsdefinition in Haskell

```
foo :: Typ1 -> ... -> Typ3 -> Ergebnistyp
foo pat_1 ... pat_n = expr1
foo pat21 ... pat2n
    | grd211, ..., grd21i = expr21
    | grd221, ..., grd22i = expr22
foo pat31 ... pat3n
    | grd311, ..., grd31k = expr31
  where idA = exprA
        idB = exprB
```

- Typdeklaration (optional – aber gute Dokumentation)
- Funktionsname (immer in gleicher Spalte beginnen)
- Argumente
- Funktionsrumpf
- Fallunterscheidung mit Pattern-Match
- Verfeinerung des Pattern-Match durch Wächter :: Bool
- Erster zutreffender Match gilt (von oben nach unten)
- Nachgeschobene lokale Definitionen

Zusammenfassung Funktionen

- Funktionen sind ganz normale Werte
- Funktionsanwendung darf partiell sein;
  partielle Funktionsanwendung liefert eine Funktion
- Funktionen höherer Ordnung nehmen Funktionen als Argumente (und können Funktionen zurück liefern)
- Funktionen höherer Ordnung abstrahieren auf einfache Weise häufig verwendete Berechnungsverfahren;
  viele wichtige in Standardbibliothek verfügbar
- Die durch Funktionen höherer Ordnung gewonnene Modularität erlaubt sehr viele Kombinationsmöglichkeiten

Beispiele

```
show_signed :: Integer -> String
show_signed 0                = " 0"
show_signed i | i>=0         = "+" ++ (show i)
              | otherwise    =        (show i)

printPercent :: Double -> String
printPercent  x  = lzero ++ (show rx) ++ "%"
 where
   rx :: Double
   rx = (fromIntegral (round' (1000.0*x))) / 10.0

   lzero = if rx < 10.0 then "0" else ""

   round' :: Double -> Int -- avoids annoying warning
   round' z = round z
```

Beispiele

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

preis :: Frucht -> Double
preis (Apfel (p,z)) = (fromIntegral z) * p
preis (Birne  z p ) = (fromIntegral z) * p
```

Beispiele

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

preis :: Frucht -> Double
preis (Apfel (p,z)) = (fromIntegral z) * p
preis (Birne  z p ) = (fromIntegral z) * p



drop :: Int -> [a] -> [a]
drop n xs      | n <= 0 =  xs
drop _ []               =  []
drop n (_:xs)           =  drop (n-1) xs

> drop 3 [1,2,3,4,5]
[4,5]
```

Anonyme Funktionen

Es ist oft praktisch, einmal verwendeten Funktionen keinen besonderen Namen zu geben.
Aus dem Lambda-Kalkül (Alonzo Church, 1936), der mathematischen Grundlage der funktionalen Programmierung, nehmen wir daher die λ-Notation für anonyme Funktionen:

Nachfolgerfunktion	$\lambda x \,.\, x + 1$
Wurzelfunktion	$\lambda y \,.\, \sqrt{y}$

In Haskell schreiben wir anstelle von λ einfach einen Rückstrich \

```
\x -> x + 1
\y -> sqrt y
\x y z -> x + y + z
```

Es darf Pattern-Matching verwendet werden,
aber es kann nur ein Fall behandelt werden
(also **Irrefutable Patterns** verwenden)

Haskell Ausdrücke

Ein Ausdruck pro Pattern/Guard-Zweig

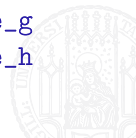
- Funktionsanwendung durch Leerzeichen: `f x`
- Anonyme Funktionsabstraktion: `\x y -> e`
- Konditional: `if b then x else y`
- Pattern-Match:

  Weitere Verfeinerung durch Wächter-Klauseln möglich

```
case e of
  p1      -> e1
  p2 | g1 -> e21
     | g2 -> e22
  p3      -> e3
```

- Lokale Definitionen:
  - erlaubt Funktionsdefinition
  - wechselseitig rekursiv
  - Begrenzung durch Einrückung

```
let f     = e_f
    g x   = e_g
    h y z = e_h
in e
```



Pattern-Guards

Pattern-Matching in Funktionsdefinitionen und Case-Ausdrücken darf mit Wächter-Klauseln (Guards) verfeinert werden.

Es dürfen mehrere, durch Kommata getrennte Wächter angegeben werden. Alle angegebenen Wächter erfüllt werden. Wird mindestens ein Wächter nicht erfüllt, so wird die nächste Wächter-Klausel bzw. nächstes Pattern geprüft.

Mögliche Wächter-Ausdrücke

- Ausdrücke des Typs `Bool` müssen zum Wert `True` auswerten
- Weitere Pattern-Matches mit speziellen Pattern-Guard Syntax `pattern <- expression` der Wächter ist nur erfüllt, wenn das Pattern-Matching erfolgreich war `-XPatternGuards`

Alle Wächter-Ausdrücke dürfen zuvor eingeführte lokale Variablen (z.B. aus dem Pattern-Match) verwenden.

Wechselseitig Rekursive Definitionen

```
data Foo = FNil | Foo Int    Bar
data Bar = BNil | Bar String Foo
```

... also eine Liste, welche Abwechselnd Werte der Typen `Int` und `String` enthält.

```
baz x = if x<=1 then 0 else 1 + qux x
quz x | even x    = baz (x `div` 2)
      | otherwise = baz (3 * x + 1)
```

... Funktionen rufen sich gegenseitig auf! (im Beispiel unnötig)

Fallstrick:

```
let x = 4 in
let x = x + 1 in x    -- <<loop>>
```

Terminiert nicht, da das innere `x` das äußere `überschattet`, es sich also rekursiv auf sich selbst bezieht. ⇒ Kapitel Zirkularität

Layout

Haskell ist “whitespace”-sensitiv: Einrückung ersetzt Klammerung

```
betragSumme :: Int -> Int -> Int
betragSumme x y =
  let x' = abs x
      y' = case y of 0 -> 0
                     x | x > 0     ->  x
                       | otherwise -> -x
  in x'+y'
```

Top-level Definition müssen in der ersten Spalte beginnen!
Erstes Zeichen nach `let`, `of`, etc. legt die Spalte fest:

WEITER RECHTS: gehört zu vorheriger Zeile

WEITER LINKS: Ausdruck beendet

Anstatt Einrückung können auch `{ }` und `;` benutzt werden.

$ Funktion

Was macht diese Infix-Funktion?

```
($)    :: (a -> b) -> a -> b
f $ x =  f x
```

$ Funktion

Was macht diese Infix-Funktion?

```
($)   :: (a -> b) -> a -> b
f $ x =  f x
```

ANTWORT: Funktionsanwendung / Klammern sparen!

Im Gegensatz zu dem Leerzeichen als Funktionsanwendung, hat `$` eine sehr niedrige Präzedenz (bindet schwach).

Merke: `$` ersetzt Klammer, welche so spät wie möglich schliesst

BEISPIEL:

```
sum (filter (> 10) (map (^2) (map (+1) [1..10])))
```

ist gleichwertig zu

```
sum $ filter (>10) $ map (^2) $ map (+1) [1..10]
```

SML vs. Haskell-Syntax: Funktionen

SML-Syntax

```
(* SML comment block *)


(* function abstraction *)
fn x => fn y => x
fn (x,y) => y
fn  []        => true
 |  (x::xs)   => false

(* function declaration *)
fun  iter f a 0 = a
  |  iter f a n =
       iter f (f a) (n-1)
```

Haskell-Syntax

```
{- Haskell comment block -}
-- Haskell one line comment

-- Lambda
\x y   -> x
\(x,y) -> y
\case         -- -XLambdaCase
  [] -> True
  (x:xs) -> False
-- just equations
iter f a 0 = a
iter f a n =
  iter f (f a) (n-1)
```

SML vs. Haskell-Syntax: Listen

Rollentausch: In Haskell ist `:` Listenkonstruktion (SML `::`) und `::` Typzuweisung (SML `:`).

SML		Haskell
`[]`	`(* empty list *)`	`[]`
`x::xs`	`(* cons *)`	`x:xs`
`[1,2,3]`	`(* literal *)`	`[1,2,3] {- or -} [1..3]`
`l @ l'`	`(* append *)`	`l ++ l'`

Typsignaturen `bezeichner :: typ` sind in Haskell optional.

```
fun  map f []           = []
  |  map f (x :: xs) =
       f x :: map f xs
```

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

SML vs. Haskell-Syntax: Datentypen

Datenkonstruktoren sind in Haskell ge**curry**te Funktionen.

```
datatype 'a tree
  = Leaf of 'a
  | Node of 'a tree * 'a tree

(* Leaf : 'a -> 'a tree *)
fun node l r = Node (l, r)


(* case distinction *)
case t
  of Leaf(a)     => [a]
   | Node(l,r)   => f l @ f r
```

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

-- Leaf :: a -> Tree a
-- Node :: Tree a -> Tree a ->
--                Tree a

-- similar to SML
case t of
  Leaf a   -> [a]
  Node l r -> f l ++ f r
```

Datentypen und Konstruktoren müssen in Haskell **Groß** geschrieben werden. In SML hat Großschreibung keine lexikalische Signifikanz.

SML vs. Haskell-Syntax: Lokale Definitionen

In Haskell können Definitionen mit `where` nachgestellt werden.

```
let   val   k    = 5
      fun   f 0 = k
        |   f n = f (n-1) * n
in    f k
end

(* no post-definition *)

local ... in ... end
```

```
let   k     = 5
      f 0   = k
      f n   = f (n-1) * n
in    f k

f k where
      f 0 = k
      f n = f (n-1) * n
      k   = 5

-- no local definitions for
-- multiple top-level decls.
```

SML bearbeitet Definitionsfolgen von vorne nach hinten. In Haskell ist die Reihenfolge der Definitionen unerheblich, da diese immer *wechselseitig rekursiv* sind.

SML vs. Haskell-Syntax: Rekursion

Rekursion muss in SML mit Schlüsselwort `rec` oder `fun...and...` angezeigt werden.
In Haskell sind alle Gleichungen wechselseitig rekursiv. Kann Fallstrick bei fehlerhafter Überschattung sein; besser frische Namen verwenden!

```
(* mutual recursion *)
fun   even 0 = true
  |   even n = odd (n-1)
and   odd  0 = false
  |   odd  n = even (n-1)

val rec f = fn x =>
  if x <= 1 then 1
    else x * f (x-1)
```

```
-- no need to mark mutual rec.
even  0 = true
even  n = odd (n-1)
odd   0 = false
odd   n = even (n-1)

f = \ x ->
  if x <= 1 then 1
    else x * f (x-1)
```

SML vs. Haskell-Syntax: Module

Haskell hat simples Modulsystem (keine Signaturen, keine Funktoren).

```
structure M = struct
  ...
end

open Text.Pretty
structure S = System.IO
```

```
module M where
...

import Text.Pretty
import qualified System.IO
          as S
```

- Pro Modul eine Datei mit gleich lautendem Namen
- Punkte in Modulnamen entsprechen Verzeichnisunterteilung: `ModA.SubA.MyMod` ist in Datei `ModA\SubA\MyMod.hs`
- Jedes Modul hat seinen eigenen Namensraum; Einschränkung bei Import/Export möglich
- Modulnamen werden immer groß geschrieben