Fortgeschrittene Funktionale Programmierung mit Haskell

Einführung in Haskell

Steffen Jost

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

10. Oktober 2014

Organisation

Vorlesung
- Freitags, 10-12 Uhr, U151, Oettingenstr. 67
- Bitte in UniworX zur Vorlesung anmelden!
- Vorlesungshomepage beachten:

https://www.tcs.ifi.lmu.de/lehre/ws-2014-15/fun

Übung
- Mittwochs, 18-20 Uhr, 027, Oettingenstr. 67

ab 15.10.

- Gruppenübung. Laptop mitbringen!
- Abgabe und Lösungen per UniworX.

Prüfung Bevorzugt durch Mini-Programmierprojekt, ggf. durch Klausur — wird noch entscheiden.

Kommunikation per eMail oder im TCS-Forum mit Kürzel FUN:
http://www.die-informatiker.net/forum/TCS

Voraussetzungen

- Vorlesung "Programmierung und Modellierung" wird vorausgesetzt, d.h. solide Grundkenntnisse einer *funktionalen* Programmiersprache wie z.B. Haskell, SML, OCaml, F#, ...
- Bereitschaft, jede Woche praktische Programmier-Übungen durchzuführen
- Umgang mit einem Editor
- In der Lage zu sein, ein Haskell Programm auszuführen
  - lokale GHC Installation auf eigenem Laptop
  - GHC auf den Rechnern im CIP-Pool der Informatik
  - im Webbrowser auf `https://www.fpcomplete.com/new-project`

Inhalte der Vorlesung

Wiederholung

Grundlagen funktionaler Programmierung in Haskell, d.h. Syntax, Rekursion, Datentypen, Funktionen höherer Ordnung,...

Vertiefung fortgeschrittener Konzepte

Überladung durch Typklassen, Monaden und Monadentransformatoren, Verzögerte Auswertung und unendliche Strukturen, GADTs, ...

Angewandte funktionale Programmierung

Parallelität und Nebenläufigkeit, Webapplikationen, Grafische Benutzeroberflächen, Parsec, Quickcheck, Foreign Function Interface, ...

Die Gewichtung der Inhalte werden im Verlauf der Veranstaltung mit den Teilnehmern abgestimmt.

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

so wie ältere Skripte des Lehrstuhls TCS
Links/ISBN der Quellen, dieses Skript $\Rightarrow$ Vorlesungshomepage

Weitere wichtige Online-Informationensquellen:

Haskell-Wiki und Hoogle Suchmaschine auf `www.haskell.org`

Warum funktionale Programmierung?

Zwei Stimmen aus dem Feedback zur ProMo 2014:

> *Evtl. etwas "realitäts-/anwendungsbezogenere" Aufgaben [...] – die meisten Studenten werden einige sehr ausführlich behandelte Themenbereich, z.B. die Typinferenz, nie in dem Umfang brauchen, wie in der Vorlesung behandelt.*

> *Mehr Realismus – Haskell [...] ist ein Abenteuer, meines Wissens ist in der Industrie heute nix in Haskell programmiert; Hintergrund > 15 Jahre IT bayerischer Automobilhersteller in München.*

Man könnte antworten: Wenn in der IT doch alles rund läuft, warum überhaupt etwas Neues lernen?

Aber: λ-Kalkül aus 1936, Lisp aus 1958, also nicht neu! Warum heute relevant?

Warum funktionale Programmierung?

Zwei Stimmen aus dem Feedback zur ProMo 2014:

> *Evtl. etwas "realitäts-/anwendungsbezogenere" Aufgaben [...] – die meisten Studenten werden einige sehr ausführlich behandelte Themenbereich, z.B. die Typinferenz, nie in dem Umfang brauchen, wie in der Vorlesung behandelt.*

> *Mehr Realismus – Haskell [...] ist ein Abenteuer, meines Wissens ist in der Industrie heute nix in Haskell programmiert; Hintergrund > 15 Jahre IT bayerischer Automobilhersteller in München.*

Man könnte antworten: Wenn in der IT doch alles rund läuft,
warum überhaupt etwas Neues lernen?

Aber: λ-Kalkül aus 1936, Lisp aus 1958, also nicht neu!
Warum heute relevant?

Imperativer Ansatz

Imperative Sprachen (Assembler, C, Java, Javascript, etc.) orientieren sich an den Fähigkeiten der Maschine.
Programmierung wird durch Abstraktion vieler Einzelschritte vereinfacht (z.B. Schleifen).

Vorteile

Programmierer hat sehr viel Kontrolle, insbesondere in ungetypten Skriptsprachen kann man einfach loslegen und mal machen.
Programmierer kann sehr gut optimieren.

Nachteile

With great power comes great responsibility.
Außer Testen hat der Programmierer nur wenig Hilfe, um Korrektheit sicherzustellen.
Nachträgliche Optimierung oft problematisch, da refactoring oft Korrektheit beeinflusst.

Deklarativer Ansatz

Charakteristische Eigenschaft **deklarativer** Sprachen ist die **Church-Rosser Eigenschaft**:

> *Die Reihenfolge der Auswertung ist unerheblich für das Ergebnis der Berechnung ist.*

Spezifiziere **was** berechnet werden soll,
nicht wie es berechnet wird!

Deklarative Sprachen oft durch die Mathematik motiviert:

Prädikaten-Logik	$\Rightarrow$	Prolog
Relationale Algebra	$\Rightarrow$	SQL
λ-Kalkül	$\Rightarrow$	Haskell

Vorteil Korrektheit ist oberstes Ziel
Nachteil Programmierer ist eingeschränkt und hat kaum Möglichkeiten zur Optimierung

Deklaratives Prinzip: Harte Landung

Das reine deklarative Prinzip scheitert an der Praxis:
Ressourcenverbrauch hängt kritisch vom **wie** der Berechnung ab.

- PROLOG:
  - *Prinzip:* Prädikatenlogik.
  - Nichtdeterminismus: Lösungen der Problemspezifikation werden (blind) **gesucht**.
  - Das **fifth generation computing project** (Japan 1980er) scheiterte.
  - PROLOG benötigt extralogische Konzepte (z.B. **cuts**).
- Erfolgreich: SQL
  - *Prinzip:* Relationale Algebra.
  - Rein deklarative Anfragen (**cross join** und **where**-Klauseln) zu ineffizient.
  - Explizite **join**-Klauseln effizienter.

Funktionaler Ansatz

Funktionale Programmierung ist auch ein deklarativer Ansatz, der dennoch einen Mittelweg geht:

Die Berechnung bleibt **deterministisch**

d.h. wir können vorhersagen, wie die Berechnung ablaufen wird und den Ressourcenverbrauch abschätzen.

Allerdings müssen wir uns nicht so sehr darum kümmern, wie die Berechnung abläuft, wenn wir das nicht wollen.

Funktionaler Ansatz

Nachteile

- Völlig andere (mathematische) Denkweise
- Kompiler "nörgelt viel" herum, bevor man testen kann
- Verlust von "Kontrolle" gegenüber herkömmlichen Sprachen, Maschinen-nahe Hand-Optimierung schwierig durchführbar
- Geringe kommerzielle Unterstützung IDE, Debugger,...

Vorteile

- Programmierer bekommt Hilfe um Korrektheit sicherzustellen "Well-typed programs can't go wrong!", R.Milner
- Nachträgliche Optimierung/Refactoring ist 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 kompositional
- Testen/Verifikation reduziert sich auf Gleichheitsschließen
- Kompiler kann sehr aggressiv optimieren
- Parallele Berechnung einfacher

Vorteile referentieller Transparenz

- Berechnungen leicht parallelisierbar.
- Ermöglicht Optimierungen durch den Kompiler, z.B. **common subexpression elimination**.
  `(x + y) * (x + y)` optimiert zu `let z = x + y in z * z`
  Seiteneffekte wie Wert-Veränderung verhindern solche Optimierungen: `(++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 eines Elementes in eine Liste muss eine **neue** Liste erzeugen.

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

Vorteile

- Typische Falle veränderlicher Strukturen: Veränderung während Iteration (`for`).
- Backtracking (Rekursion): Alter Zustand ist noch verhanden, muss nicht wiederhergestellt werden.
- Nebenläufigkeit: Kein explizites Kopieren der Struktur nötig, wenn 2 Threads gleichzeitig manipulieren wollen.

Abstraktion

Ein wichtiges Merkmal funktionaler Sprachen ist die **hohe Abstraktion.**

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

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

Die Entwicklung von universell verwendbaren Bausteinen wie

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

sind gut verstehbar und leichter einsetzbar als Designpatterns.

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

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).

Haskell

- Effektfreie **rein funktionale** Sprache
  mit verzögerter Auswertung
- Benannt nach Haskell Curry (1900-82), Logiker
- Standards: Haskell98 und Haskell2010
- Viele verschiedene Implementierung verfügbar; wichtigste ist die **Haskell Platform** mit "batteries included":
  Kompiler, Interpreter und zahlreichen Bibliotheken

GHC : Glasgow/Glorious Haskell Compiler Hammond, 1989
akutelle Versionen mehrmals pro Jahr
Hauptentwickler erhielten 2011 ACM SIGPLAN Programming Languages Software Award:
Simon Peyton-Jones Microsoft Research Cambridge
Simon Marlow Facebook

GHC

In der Vorlesung arbeiten wir mit der **Haskell Platform**, welche aus GHC, Bibliotheken und Dokumentation besteht. Die meiste Dokumentation ist aber auch online verfügbar, wie zum Beispiel
`http://www.haskell.org/ghc/docs/latest/html/libraries/`

Glasgow Haskell Kompiler (GHC) kennt zwei Arbeitsweisen:

GHC Normaler Kompiler. 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 7.6.3: http://www.haskell.org/ghc/
Loading package ghc-prim ... linking ... done.
Loading package integer-gmp ... linking ... done.
Loading package base ... linking ... done.
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 GHC

Haskell beachtet Groß-/Kleinschreibung!

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

Daraus folgt:

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

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

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 `:t` zeigt Typ eines Ausdrucks:

```
Prelude> :t '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]
data Bool = True | False
```

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.
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

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

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

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

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

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. Leider ignoriert GHC/GHCi Typabkürzungen meistens, d.h. GHCi gibt fast immer `[Char]` anstelle von `String` aus.

Es gibt noch weitere zusammengesetzte Typen, z.B. Records, Funktionstypen, etc.

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)
```

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
  - 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
```

- 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
```

- 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
```

- 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
```

- 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
```

- 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
```

- 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

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:

  `data` Deklaration wirklich neuer Typen
  `type` Typabkürzungen für Lesbarkeit

- 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 Definitionen müssen in der gleichen Spalte beginnen
- 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 (Partielle 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**.

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)
```

Funktionstypen

Definition (Partielle 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**.

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)
- 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

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
- Funktionsrumpf
- Fallunterscheidung mit Pattern-Match
- Verfeinerung des Pattern-Match durch Wächter :: Bool
- Erster zutreffender Match gilt (von oben nach unten)
- Nachgeschobene lokale Definitionen

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
- Funktionsrumpf
- Fallunterscheidung mit Pattern-Match
- Verfeinerung des Pattern-Match durch Wächter :: Bool
- Erster zutreffender Match gilt (von oben nach unten)
- Nachgeschobene lokale Definitionen

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
- Fallunterscheidung mit Pattern-Match
- Verfeinerung des Pattern-Match durch Wächter :: Bool
- Erster zutreffender Match gilt (von oben nach unten)
- Nachgeschobene lokale Definitionen

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
- Fallunterscheidung mit Pattern-Match
- Verfeinerung des Pattern-Match durch Wächter :: Bool
- Erster zutreffender Match gilt (von oben nach unten)
- Nachgeschobene lokale Definitionen

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
- Fallunterscheidung mit Pattern-Match
- Verfeinerung des Pattern-Match durch Wächter :: Bool
- Erster zutreffender Match gilt (von oben nach unten)
- Nachgeschobene lokale Definitionen

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
- Verfeinerung des Pattern-Match durch Wächter :: Bool
- Erster zutreffender Match gilt (von oben nach unten)
- Nachgeschobene lokale Definitionen

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
- Erster zutreffender Match gilt (von oben nach unten)
- Nachgeschobene lokale Definitionen

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)
- Nachgeschobene lokale Definitionen

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
```

```
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]
```

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]
```

Haskell Ausdrücke

Ein Ausdruck pro Pattern/Guard-Zweig

- Funktionsanwendung durch Leerzeichen:
- Anonyme Funktionsabstraktion:
- Konditional:
- Pattern-Match:

  Weitere Verfeinerung durch Wächter-Klauseln möglich

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

```
f x
```

```
\x y -> e
```

```
if b then x else y
```

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

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

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) [1..10]))
```

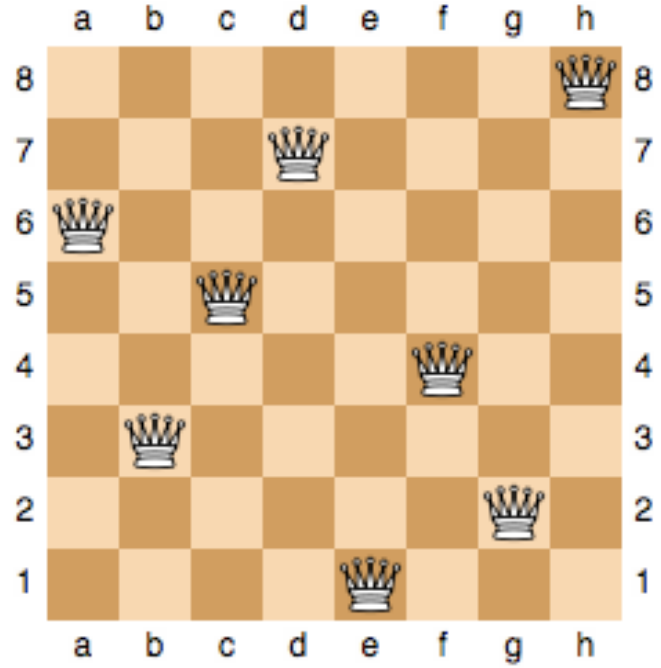
ist gleichwertig zu

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

Musterbeispiel deklarativer Programmierung: Die acht Damen

Plaziere 8 Damen auf einem Schachbrett, so dass keine eine andere bedroht.

- nur eine pro Spalte
- Lösung ist 8-elementige Liste von Zeilenpositionen. Hier: `[6,3,5,7,1,4,2,8]`



Statt $\binom{64}{8} \approx 4$ Milliarden brauchen wir nur $8^8 \approx 16$ Millionen Damenplazierungen betrachten.

Zu deklarativ!

```
import Data.List

queens :: Int -> [[Int]]
queens n = [ l
  | l <- lists n [1..n]   -- n columns with row positions in 1..n
  , sort l == [1..n]      -- permutation of 1..n (horiz. threats)
  , noDiagThreats l
  ]

-- | Enumerate all lists of length @n@ with elements in @l@.
lists n l   | n > 0       = [ x:xs | x <- l, xs <- lists (n-1) l ]
            | otherwise   = [[]]

noDiagThreats []          = True
noDiagThreats (x:ys)      =
  all (\ (y,n) -> abs(x-y) /= n) (zip ys [1..])
  && noDiagThreats ys
```

Zu deklarativ!

- Verwendete Listenfunktionen:

```
sort  :: Ord a => [a] -> [a]
zip   :: [a] -> [b] -> [(a,b)]
all   :: (a -> Bool) -> [a] -> Bool
```

- Program zu langsam!
- Pro Zeile darf auch nur eine Dame plaziert werden!
- `queens` erzeugt viele Kandidaten, die keine Permutationen von $1..n$ sind, die dann wieder aussortiert werden müssen.
- Besser: betrachte nur die $8! = 40320$ Permutationen von $1..8$.

Betrachte nur Permutationen

```
import Data.List

queens2 n = [ l
  | l <- choose n [1..n] -- permutation of [1..n]
  , noDiagThreats l
  ]

-- | Choose @n@ distinct elements from list @l@.
choose n l
  | n > 0     = [ x:xs   | x   <- l
                         , xs  <- choose (n-1) (delete x l) ]
  | otherwise = [[]]
```

Viel schnellere Ausführung!

Betrachte nur gültige Positionen

- Wenn Dame plaziert wird, entferne **alle, auch die diagonal** bedrohten Positionen vom Brett.
- Eliminiert frühzeitig weitere Kandidaten.
- choose benötigt nun ein Brett mit noch verfügbaren Positionen.
- Weitere Listenfunktionen:

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

Betrachte nur gültige Positionen

```
-- | Produce all queen positionings on a board of size @n*n@.
queens3 n = choose3 (board n)

-- | Initially, for each row, all columns are possible.
board n = take n (repeat [1..n])

choose3 []       = [[]]
choose3 (r:rs)   = [ c:cs | c  <- r   -- pick an available column
                          , cs <- choose3 (removeThreatened c rs) ]

-- | Delete column positions in remaining rows @rs@
--   threatened by a queen in column @c@ of the current row.
removeThreatened c rs = zipWith (del c) [1..] rs
  where del c n = filter (\ y ->      let  d = abs(c-y)
                                      in   d /= 0 && d /= n)
```

Noch deutlich besser!

Zusammenfassung

- Bisherige Nische für funktionale Sprachen: Compiler, Softwareanalyse und -verifikation, akademische Landschaft.
- Funktionale Programmierung erlebt eine Renaissance:
  - Closures in C#, C++, Java.
  - Funktionale OO-Sprache Scala.
  - Unveränderliche Datenstrukturen für Nebenläufigkeit.
  - Ocaml/F# und Haskell erobern Nische in Finanzbranche.
- Effektfreiheit ermöglicht Optimierung durch Compiler und Parallelisierung.
- Nachteil: Werkzeugkette meist noch nicht komplett oder ausgereift (Bibliotheken, IDE, Debugger, Monitoring...).

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


-- \ for λ
\ x y     -> x
\ (x,y)   -> y
{- no multi-clause anonymous
   functions -}

-- 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 `:`).

```
[]        (* empty list
*)
x::xs     (* cons
*)
[1,2,3]   (* literal
*)
l @ l'    (* append
*)
```

```
[]
x:xs
[1,2,3] {- or -} [1..3]
l ++ l'
```

Typsignaturen `bezeichner :: typ` sind 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

-- nothing corresponding
```

SML bearbeitet Definitionsfolgen von vorne nach hinten. In Haskell ist die Reihenfolge der Definitionen unerheblich.

SML vs. Haskell-Syntax: Rekursion

In Haskell sind alle Gleichungen wechselseitig rekursiv.

```
(* 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)
```

In SML benötigt Rekursion das Schlüsselwort **rec** oder **fun...and...**.

SML vs. Haskell-Syntax: Module

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

```
structure M = struct          module M where
  ...                         ...
end

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